GlobalLogic®

SQL Performance Tuning

Shalabh Mehrotra,
Senior Solutions Architect

Noida, India

Database performance is one of the most challenging aspects of an organization's database operations, and SQL tuning can help signficantly improve a system's health and performance. This white paper demonstrates how to improve the performance an organization's structured query language.

www.globallogic.com

Table of Contents

Introduction

Database performance is one of the most challenging aspects of an organization's database operations. A well-designed application may still experience performance problems if the SQL it uses is poorly constructed. It is much harder to write efficient SQL than it is to write functionally correct SQL. As such, SQL tuning can help signficantly improve a system's health and performance. The key to tuning SQL is to minimize the search path that the database uses to find the data.

The target audience of this whitepaper includes developers and database administrators who want to improve the performance of their SQL queries.

General SQL Tuning Guidelines

The goals of writing any SQL statement include delivering quick response times, using the least CPU resources, and achieving the fewest number of I/O operations. The following content provides best practices for optimizing SQL performance.

Write Sensible Queries

Identify SQL statements that are taking a long time to execute. Also identify SQL statements that involve the joining of a large number of big tables and outer joins. The simplest way to do this usually involves running the individual statements using SQLPlus and timing them (SET TIMING ON). Use EXPLAIN to look at the execution plan of the statement. Look for any full table accesses that look dubious. Remember, a full table scan of a small table is often more efficient than access by rowid.

Check to see if there are any indexes that may help performance. A quick way to do this is to run the statement using the Rule Based Optimizer (RBO) (SELECT /*+ RULE */). Under the RBO, if an index is present, it will be used. The resulting execution plan may give you some ideas as to which indexes to play around with. You can then remove the RULE hint and replace it with the specific index hints you require. This way, the Cost Based Optimizer (CBO) will still be used for table accesses where hints aren't present. Remember, if data volumes change over time, the hint that helped may become a hindrance! For this reason, hints should be avoided if possible, especially the /*+ RULE */ hint.

Try adding new indexes to the system to reduce excessive full table scans. Typically, foreign key columns should be indexed, as these are regularly used in join conditions. On occasion it may be necessary to add composite (concatenated) indexes that will only aid individual queries. Remember, excessive indexing can reduce INSERT, UPDATE and DELETE performance.

Tuning Subqueries

If the SQL contains subqueries, tune them. In fact, tune them first. The main query will not perform well if the subqueries can't perform well themselves. If a join will provide you with the functionality of the subquery, try the join method first before trying the subquery method. Pay attention to correlated subqueries, as they tend to be very costly and CPU- insentive.

Limit the Number of Tables in a Join

There are several instances when the processing time can be reduced several times by breaking the SQL statement into smaller statements and writing a PL/SQL block to reduce database calls. Also, packages reduce I/O since all related functions and procedures are cached together. Use the DBMS_SHARED_POOL package to pin a SQL or PL/SQL area. To pin a set of packages to the SQL area, start up the database and make a reference to the objects that cause them to be loaded. Use DBMS_SHARED_POOL.KEEP to pin it. Pinning prevents memory fragmentation. It also helps to reserve memory for specific programs.

Recursive Calls Should be Kept to a Minimum

Recursive calls are SQL statements that are triggered by Oracle itself. Large amount of recursive SQL executed by SYS could indicate space management activities such as extent allocations taking place. This is not scalable and impacts user response time. Recursive SQL executed under another user ID is probably SQL and PL/SQL, and this is not a problem.

The Oracle trace utility tkprof provides information about recursive calls. This value should be taken into consideration when calculating resource requirement for a process. Tkprof also provides library cache misses and provides the username of the individual who executed the SQL statement. Tkprof-generated statistics can be stored in a table tkprof_table to be queried later.

Improving Parse Speed

Execution plans for SELECT statements are cached by the server, but unless the exact same statement is repeated, the stored execution plan details will not be reused. Even differing spaces in the statement will cause this lookup to fail. Use of bind variables allows you to repeatedly use the same statements while changing the WHERE clause criteria. Assuming the statement does not have a cached execution plan, it must be parsed before execution. The parse phase for statements can be decreased by efficient use of aliasing.

Alias Usage

If an alias is not present, the engine must resolve which tables own the specified columns. A short alias is parsed more quickly than a long table name or alias. If possible, reduce the alias to a single letter. The following is an example:

Bad Statement
```
SELECT first_name, last_name, country FROM employee, countries
WHERE country_id = id
AND last_name = 'HALL';
```

Good Statement
```
SELECT e.first_name, e.last_name, c.country FROM employee e, countries c
WHERE e.country_id = c.id
AND e.last_name = 'HALL';
```

Driving Tables

The structure of the FROM and WHERE clauses of DML statements can be tailored to improve the performance of the statement. The rules vary depending on whether the database engine is using the Rule or Cost-based optimizer. The situation is further complicated by the fact that the engine may perform a Merge Join or a Nested Loop join to retrieve the data. Despite this challenge, there are a few rules you can use to improve the performance of your SQL.

Oracle processes result sets one table at a time. It starts by retrieving all the data for the first (driving) table. Once this data is retrieved, it is used to limit the number of rows processed for subsequent (driven) tables. In the case of multiple table joins, the driving table limits the rows processed for the first driven table.

Once processed, this combined set of data is the driving set for the second driven table, etc. Roughly translated, this means that it is best to process tables that will retrieve a small number of rows first. The optimizer will do this to the best of its ability, regardless of the structure of the DML, but the following factors may help.

Both the Rule and Cost-based optimizers select a driving table for each DML statement. If a decision cannot be made, the order of processing is from the end of the FROM clause to the start. Therefore, you should always place your driving table at the end of the FROM clause. Always choose the table with less number of records as the driving table. If three tables are being joined, select the intersection tables as the driving table.

The intersection table is the table that has many tables dependent on it. Subsequent driven tables should be placed in order so that those retrieving the most rows are nearer to the start of the FROM clause. However, the WHERE clause should be written in the opposite order, with the driving tables conditions first and the final driven table last (example below).

```
FROM d, c, b, a
WHERE a.join_column = 12345
AND a.join_column = b.join_column AND b.join_column = c.join_column AND c.join_column = d.join_column;
```

If we now want to limit the rows brought back from the "D" table, we may write the following:

```
FROM d, c, b, a
WHERE a.join_column = 12345 AND a.join_column = b.join_column AND b.join_column = c.join_column AND c.join_column = d.join_column AND d.name = 'JONES';
```

Depending on the number of rows and the presence of indexes, Oracle may now pick "D" as the driving table. Since "D" now has two limiting factors (join_column and name), it may be a better candidate as a driving table. The statement may be better written as:

```
FROM c, b, a, d
WHERE d.name = 'JONES'
AND d.join_column = 12345
AND d.join_column = a.join_column AND a.join_column =
b.join_column AND b.join_column = c.join_column
```

This grouping of limiting factors will guide the optimizer more efficiently, making table "D" return relatively few rows, and so making it a more efficient driving table. Remember, the order of the items in both the FROM and WHERE clause will not force the optimizer to pick a specific table as a driving table, but it may influence the optimizer's decision. The grouping of limiting conditions onto a single table will reduce the number of rows returned from that table, which will therefore make it a stronger candidate for becoming the driving table. Also, you can have control over which table will drive the query through the use of the ORDERED hint. No matter what order the optimizer is from, that order can be overridden by the ORDERED hint. The key is to use the ORDERED hint and vary the order of the tables to get the correct order from a performance standpoint.

Caching Tables

Queries will execute much faster if the data they reference is already cached. For small, frequently used tables, performance may be improved by caching tables. Normally, when full table scans occur, the cached data is placed on the Least Recently Used (LRU) end of the buffer cache. This means that it is the first data to be paged out when more buffer space is required.

If the table is cached (ALTER TABLE employees CACHE;), the data is placed on the Most Recently Used (MRU) end of the buffer, and so it is less likely to be paged out before it is re-queried. Caching tables may alter the CBO's path through the data and should not be used without careful consideration.

Avoid Using Select * Clauses

The dynamic SQL column reference (*) gives you a way to refer to all of the columns of a table. Do not use the * feature because it is very inefficient -- the * has to be converted to each column in turn. The SQL parser handles all the field references by obtaining the names of valid columns from the data dictionary and substitutes them on the command line, which is time consuming.

Exists vs. In

The EXISTS function searches for the presence of a single row that meets the stated criteria, as opposed to the IN statement that looks for all occurrences. For example:

```
PRODUCT - 1000 rows
ITEMS - 1000 rows

(A)
SELECT p.product_id
FROM products p
WHERE p.item_no IN (SELECT i.item_no
            FROM items i);

(B)
SELECT p.product_id
FROM products p
WHERE EXISTS (SELECT '1'
            FROM items i
            WHERE i.item_no = p.item_no)
```

For query A, all rows in ITEMS will be read for every row in PRODUCTS. The effect will be 1,000,000 rows read from ITEMS. In the case of query B, a maximum of 1 row from ITEMS will be read for each row of PRODUCTS, thus reducing the processing overhead of the statement.

Not Exists vs. Not In

In subquery statements such as the following, the NOT IN clause causes an internal sort/ merge.

```
SELECT * FROM student
WHERE student_num NOT IN (SELECT student_num
FROM class)
```

Instead, use:
SELECT * FROM student c
WHERE NOT EXISTS
(SELECT 1 FROM class a WHERE a.student_num = c.student_num)

In with Minus vs. Not In for Non-Indexed Columns

In subquery statements such as the following, the NOT IN clause causes an internal sort/ merge.

SELECT * FROM system_user
WHERE su_user_id NOT IN
(SELECT ac_user FROM account)

Instead, use:
SELECT * FROM system_user
WHERE su_user_id IN
(SELECT su_user_id FROM system_user
MINUS
SELECT ac_user FROM account)

Correlated Subqueries vs. Inline Views

Do not use code correlated subqueries in your applications, as they will adversely impact system performance. Instead, use inline views (subqueries in the from clause of your select statements), which perform orders of magnitude faster and are much more scalable. The query below displays all employees who make more than the average salary of the department in which they work.

Before:

```
SELECT outer.*
        FROM emp outer
        WHERE outer.salary >
            (SELECT avg(salary)
                FROM emp inner
                WHERE inner.dept_id = outer.dept_id);
```

The preceding query contains a correlated subquery, which is extremely inefficient and is very CPU- intensive. The subquery will be run for every employee record in the EMP table. As the number of records in EMP increase, the performance can degrade exponentially. When rewritten with inline views, the query is not only functionally equivalent, but it is also significantly more scalable and is guaranteed to outperform its predecessor.

After:

```
SELECT e1.*
         FROM e1, (SELECT e2.dept_id dept_id, avg(e2.
salary) avg_sal
                FROM emp e2
                GROUP BY dept_id) dept_avg_sal
WHERE e1.dept_id = dept_avg_sal.dept_id
        AND e1.salary > dept_avg_sal.avg_sal;
```

Views Usage

Beware of SQL statements with views in them. Odd as it may seem, Oracle does not necessarily execute a view the same way by itself as it does in a complex SQL statement containing tables. Consider including the view definition in the main query by including its code without the actual view name. Views can cause potential performance problems when they have outer joins (CBO goes haywire with them, even in 9I) or are considered non-mergable views by Oracle.

Use Decode to Reduce Processing

Use DECODE when you want to scan same rows repetitively or join the same table repetitively.

```
SELECT count(*) , sum(sal) FROM emp
WHERE deptno = 10
AND ename LIKE 'MILLER';

SELECT count(*) , sum(sal)
FROM emp
WHERE deptno = 20
AND ename LIKE 'MILLER';
```

The same result can be achieved using a single query as follows:

```
SELECT count(decode(deptno,20,'x')) dept20_count,
    count(decode(deptno,10,'x')) dept10_count,
    sum(decode(deptno,20,sal)) dept20_sal,
    sum(decode(deptno,10,sal)) dept10_sal
FROM emp
WHERE ename LIKE 'MILLER' ;
```

Inequalities

If a query uses inequalities (item_no > 100), the optimizer must estimate the number of rows returned before it can decide the best way to retrieve the data. This estimation is prone to errors. If you are aware of the data and its distribution, then you can use optimizer hints to encourage or discourage full table scans to improve performance.

If an index is being used for a range scan on the column in question, the performance can be improved by substituting >= for >. In this case, item_no > 100 becomes item_no >= 101. In the first case, a full scan of the index will occur. In the second case, Oracle jumps straight to the first index entry with an item_no of 101 and range scans from this point. For large indexes, this may significantly reduce the number of blocks read.

Using Union in Place of Or

In general, always consider the UNION verb instead of OR verb in the WHERE clauses. Using OR on an indexed column causes the optimizer to perform a full-table scan rather than an indexed retrieval.

Using Union All Instead of Union

The SORT operation is very expensive in terms of CPU consumption. The UNION operation sorts the result set to eliminate any rows that are within the sub-queries. UNION ALL includes duplicate rows and does not require a sort. Unless you require that these duplicate rows be eliminated, use UNION ALL.

Influencing the Optimizer Using Hints

Hints are special instructions to the optimizer. You can change the optimization goal for an individual statement by using Hint. Some commonly used Hints are CHOOSE, RULE, FULL(table_name), INDEX(table_ name index_name), USE_NL, USE_HASH(table_ name), PARALLEL(table_name parallelism), etc.

```
SELECT /*+rule*/ name,
acct_allocation_percentage
FROM accounts WHERE account_id = 1200
```

The above SQL statement will be processed using the RULE-based optimizer.

```
SELECT /*+ index(a, acct_id_ind) */ name, acct_
allocation_percentage
FROM accounts a
WHERE account_id = :acct_id AND client_id= :client_id
```

In the above SQL statement, an Index Hint has been used to force the use of a particular index.

Presence Checking

If processing is conditional on the presence of certain records in a table, you may use code such as:

```
SELECT count(*)
INTO v_count
FROM items
WHERE item_size = 'SMALL';

IF v_count = 0 THEN
-- Do processing related to no small items present
END IF;
```

If there are many small items, time and processing will be lost retrieving multiple records that are not needed. This would be better written as one of the following:

```
SELECT count(*)
INTO v_count
FROM items
WHERE item_size = 'SMALL'
AND rownum = 1;

IF v_count = 0 THEN
  -- Do processing related to no small items present
END IF;
OR
BEGIN
  SELECT '1'
  INTO v_dummy
  FROM items
  WHERE item_size = 'SMALL'
  AND rownum = 1;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    -- Do processing related to no small items present
END;
```

In these examples, only a single record is retrieved in the presence/absence check.

Using Indexes to Improve Performance

Indexes primarily exist to enhance performance. But they do not come without a cost. Indexes must be updated during INSERT, UPDATE and DELETE operations, which may slow down performance. Some factors to consider when using indexes include:

- Choose and use indexes appropriately. Indexes should have high selectivity. Bitmapped indexes improve performance when the index has fewer distinct values like Male or Female.
- Avoid using functions like “UPPER” or “LOWER” on the column that has an index. In case there is no way that the function can be avoided, use Functional Indexes.
- Index partitioning should be considered if the table on which the index is based is partitioned. Furthermore, all foreign keys must have indexes or should form the leading part of Primary Key.
- Occasionally you may want to use a concatenated index with the SELECT column. This is the most favored solution when the index not only has all the columns of the WHERE clause, but also the columns of the SELECT clause. In this case there is no need to access the table. You may also want to use a concatenated index when all the columns of the WHERE clause form the leading columns of the index.
- When using 9i, you can take advantage of skip scans. Index skip scans remove the limitation posed by column positioning, as column order does not restrict the use of the index.
- Large indexes should be rebuilt at regular intervals to avoid data fragmentation. The frequency of rebuilding depends on the extents of table inserts.

Why Indexes are Not Used

The presence of an index on a column does not guarantee it will be used. The following is a list of factors that may prevent an index from being used:

- The optimizer decides it would be more efficient not to use the index. As a rough rule of thumb, an index will be used on evenly distributed data if it restricts the number of rows returned to 5% or less of the total number of rows. In the case of randomly distributed data, an index will be used if it restricts the number of rows returned to 25% or less of the total number of rows.
- You perform mathematical operations on the indexed column, i.e. WHERE salary + 1 = 10001
- You concatenate a column, i.e. WHERE firstname || ‘ ‘ || lastname = ‘JOHN JONES’
- You do not include the first column of a concatenated index in the WHERE clause of your statement. For the index to be used in a partial match, the first column (leading- edge) must be used.
- The use of OR statements confuses the CBO. It will rarely choose to use an index on a column referenced using an OR statement. It will even ignore optimizer hints in this situation. The only way to guarantee the use of indexes in these situations is to use the /*+ RULE */ hint.
- You use the is null operator on a column that is indexed. In this situation, the optimizer will ignore the index.
- You mix and match values with column data types. This practice will cause the optimizer to ignore the index. For example, if the column data type is a number, do not use single quotes around the value in the WHERE clause. Likewise, do not fail to use

single quotes around a value when it is defined as an alphanumeric column. For example, if a column is defined as a varchar2(10), and if there is an index built on that column, reference the column values within single quotes. Even if you only store numbers in it, you still need to use single quotes around your values in the WHERE clause, as not doing so will result in full table scan.

- When Oracle encounters a NOT, it will choose not to use an index and will perform a full-table scan instead. Remember, indexes are built on what is in a table, not what isn't in a table.

Conclusion

Eighty percent of your database performance problems arise from bad SQL. Designing and developing optimal SQL is quintessential to achieving scalable system performance and consistent response times. The key to tuning often comes down to how effectively you can tune those single problem queries.

To tune effectively, you must know your data. Your system is unique, so you must adjust your methods to suit your system. A single index or a single query can bring an entire system to a near standstill. Get those bad SQL and fix them. Make it a habit...and stick with it.

References

1. Oracle 9I Documentation

2. Oracle Performance Tuning 101 by Gaja Krishna Vaidyanatha, Kirtikumar Deshpande and John Kostelac

3. http://www.dba-village.com/dba/village/dvp_papers.Main?CatA=45 by Sumit Popli and Puneet Goenka.

About GlobalLogic Inc.

GlobalLogic is a full-lifecycle product development services leader that combines deep domain expertise and cross-industry experience to connect makers with markets worldwide.Using insight gained from working on innovative products and disruptive technologies, we collaborate with customers to show them how strategic research and development can become a tool for managing their future. We build partnerships with market-defining business and technology leaders who want to make amazing products, discover new revenue opportunities, and accelerate time to market.

For more information, visit www.globallogic.com

GlobalLogic®

Contact

Emily Younger
+1.512.394.7745
emily.younger@globallogic.com