Oracle 10g Expert SQL Tuning Techniques

This book excerpt (http://www.rampant-books.com/book_2005_1_awr_proactive_tuning.htm) will focus on real-world techniques for improving the speed of SQL queries with a focus on the new Oracle10g features. The topics will include the new Oracle parameters that affect SQL performance, the use of hints to change SQL execution plans, re-writing SQL queries in more efficient forms and the use of advanced techniques such as Materialized Views, replacing SQL with PL/SQL, the new automated CBO statistics collection, and using the new Oracle10g CPU costing approach.

This is an excerpt from the bestselling book "Oracle Tuning: The Definitive Reference" (http://www.rampant-books.com/book_2005_1_awr_proactive_tuning.htm) by Alexey Danchenkov and Donald Burleson, technical editor Mladen Gogala. To supplement the script

Understanding Oracle SQL Tuning

Before relational databases were introduced, database queries required knowledge of the internal structures and developers needed to build in the tuning as a part of writing the database query. However, the SQL standard imposed a declarative solution to database queries where the database optimizer determines important data access methods such as what indexes to use and the optimal sequence to join multiple tables together.

"I think we need to tune your SQL"

Today, it is not enough for a developer to write an SQL statement that provides the correct answer. SQL is declarative, so there are many ways to formulate a query, each with identical results but with far different execution times.

Oracle SQL tuning is a phenomenally complex subject, and entire books have been devoted to the nuances of Oracle SQL tuning, most notably the Kimberly Floss book *Oracle SQL & CBO Internals* by Rampant TechPress. This chapter provides a review the following areas of SQL tuning:

- The goals of SQL tuning
- Simplifying complex SQL
- SQL Optimization instance parameters
- Statistics and SQL optimization (http://www.dba-oracle.com/art_orafaq_cbo_stats.htm)
- Oracle10g and CBO statistics (http://www.dba-oracle.com/art_otn_cbo_p2.htm)
- Oracle tuning with hints (http://www.dba-oracle.com/art_sql_tune.htm)
- Oracle10g SQL profiles (http://www.dba-oracle.com/oracle10g_tuning/t_dbms_sqltune_tasks.htm)
- AWR and SQL tuning (http://www.dba-oracle.com/art_orafaq_awr_sql_tuning.htm)
- ADDM and SQL tuning (http://www.dba-oracle.com/s_addm_asm_awr_assm.htm)

The first three sections will be an overview of general Oracle10g tuning concepts, so that the basic tools and techniques for tuning SQL optimization are clearly introduced. The focus will then shift to an exploration of the new Oracle10g SQL Profiles, and will eventually delve into the internals of AWR and explore how the SQLTuning and SQLAccess advisor use time-series metadata.

Optimizing Oracle SQL Execution

The key to success with the Oracle Cost-based Optimizer (CBO) is stability, and ensuring success with the CBO involves the consideration of several important infrastructure issues.

- **Ensure static execution plans:** Whenever an object is re-analyzed, the execution plan for thousands of SQL statements may be changed. Most successful Oracle sites will choose to lock down their SQL execution plans by carefully controlling CBO statistics, using stored outlines (optimizer plan stability), adding detailed hints to their SQL, or by using Oracle10g SQL Profiles. Again, there are exceptions to this rule such as LIMS databases, and for these databases, the DBA will choose to use dynamic sampling and allow the SQL execution plans to change as the data changes.
- **Reanalyze statistics only when necessary:** One of the most common mistakes made by Oracle DBAs is to frequently re-analyze the schema. The sole purpose of doing that is to change the execution plans for its SQL, and if it isn't broken, don't fix it. If

the DBA is satisfied with current SQL performance, re-analyzing a schema could cause significant performance problems and undo the tuning efforts of the development staff. In practice, very few shops are sufficiently dynamic to require periodic schema re-analysis.

- **Pre-tune the SQL before deploying:** Many Oracle systems developers assume that their sole goal is to write SQL statements that deliver the correct data from Oracle. In reality, writing the SQL is only half their job and successful Oracle sites require all developers to ensure that their SQL accesses Oracle in an optimal fashion. Many DBAs will export their production CBO statistics into their test databases so that their developers can see how their SQL will execute when it is placed into the production system. DBAs and staff should be trained to use the AUTOTRACE and TKPROF utilities and to interpret SQL execution results.
- **Manage schema statistics:** All Oracle DBAs should carefully manage the CBO statistics to ensure that the CBO works the same in their test and production environments. A savvy DBA knows how to collect high quality statistics and migrate their production statistics into their test environments. This approach ensures that all SQL migrating into production has the same execution plan as it did in the test database.
- **Tune the overall system first:** The CBO parameters are very powerful because a single parameter change could improve the performance of thousands of SQL statements. Changes to critical CBO parameters such as *optimizer_mode*, *optimizer_index_cost_adj*, and *optimizer_index_caching* should be done before tuning individual SQL statements. This reduces the number of suboptimal statements that require manual tuning.

Prior to Oracle10g, it was an important job of the Oracle DBA to properly gather and distribute statistics for the CBO. The goal of the DBA was to keep the most accurate production statistics for the current processing. In some cases, there may be more than one set of optimal statistics.

For example, the best statistics for OLTP processing may not be the best statistics for the data warehouse processing that occurs each evening. In this case, the DBA will keep two sets of statistics and import them into the schema when processing modes change.

The following section provides a quick, simple review of the goals of SQL tuning.

Goals of SQL Tuning

There are many approaches to SQL tuning and this paper describes a fast, holistic method of SQL tuning where we optimize the SGA, the all-important optimizer parameters, and adjust CBO statistics, all based on current system load. Once the "best" overall optimization is achieved, we drill-down into the specific cases of sub-optimal SQL, and change their execution plans with SQL profiles, specialized CBO stats, or hints.

Despite the inherent complexity of tuning SQL, there are general guidelines that every Oracle DBA follows in order to improve the overall performance of their Oracle systems. The goals of SQL tuning are simple:

- Replace unnecessary large-table full-table scans with index scans.
- Cache small-table full table scans
- Verify optimal index usage
- Verify optimal JOIN techniques
- Tune complex subqueries to remove redundant access

These goals may seem deceptively simple, but these tasks comprise 90 percent of SQL tuning. They do not require a thorough understanding of the internals of Oracle SQL. This venture will begin with an overview of the Oracle SQL optimizers.

Of course, the SQL can be tuned to one's heart's content, but if the optimizer is not fed with the correct statistics, the optimizer may not make the correct decisions. Before tuning, it is important to ensure that statistics are available and that they are current.

The following section will provide a closer look at the goals listed above as well as how they simplify SQL tuning.

Remove unnecessary large-table full table scans

Unnecessary full table scans (FTS) are an important symptom of sub-optimal SQL and cause unnecessary I/O that can drag down an entire database. The tuning expert first evaluates the SQL based on the number of rows returned by the query. Oracle says that if the query returns less than 40 percent of the table rows in an ordered table or seven percent of the rows in an unordered table, based on the index key value described by *clustering_factor* in *dba_indexes*, the query can be tuned to use an index in lieu of the full-table scan. However, it's not that simple. The speed of a FTS versus an index scan depends on many factors:

- Missing indexes, especially function-based indexes
- Bad/stale CBO statistics
- Missing CBO Histograms
- Clustering of the table rows to the used index
- System ability to optimize multiblock I/O

    (e.g. db_file_multiblock_read_count)

The most common tuning tool for addressing unnecessary full table scans is the addition of indexes, especially function-based indexes. The decision about removing a full-table scan should be based on a careful examination of the amount of logical I/O (consistent gets) of the index scan versus the costs of the full table scan. This decision should be made while factoring in the multiblock reads and possible parallel full-table scan

execution. In some cases, an unnecessary full-table scan can be forced to use an index by adding an index hint to the SQL statement.

Cache small-table full table scans

For cases in which a full table scan is the fastest access method, the tuning professional should ensure that a dedicated data buffer is available for the rows. In Oracle7, an *alter table xxx cache* command can be issued. In Oracle8 and beyond, the small-table can be cached by forcing it into the KEEP pool.

Logical reads (consistent gets) are often 100x faster than a disk read and small, frequently referenced objects such as tables, clusters and indexes should be fully cached in the KEEP pool. Most DBA's check *x$bh* periodically and move any table that has 80% or more of its blocks in the buffer into the KEEP pool. In addition, *dba_hist_sqlstat* should be checked for tables that experience frequent small-table full-table scans.

Verify optimal index usage

Determining the index usage is especially important for improving the speed of queries with multiple WHERE clause predicates. Oracle sometimes has a choice of indexes, and the tuning professional must examine each index and ensure that Oracle is using the best index, meaning the one that returns the result with the least consistent gets.

Verify optimal JOIN techniques

Some queries will perform faster with NESTED LOOP joins, some with HASH joins, while others favor sort-merge joins. It is difficult to predict what join technique will be fastest, so many Oracle tuning experts will test run the SQL with each different table join method.

Tuning by Simplifying SQL Syntax

There are several methods for simplifying complex SQL statements, and Oracle10g will sometimes automatically rewrite SQL to make it more efficient.

- Rewrite the query into a more efficient form
- Use the WITH clause
- Use Global Temporary Tables
- Use Materialized Views

The following example shows how SQL can be rewritten. For a simple example of SQL syntax and execution speed, the following queries can be used. All of these SQL statements produce the same results, but they have widely varying execution plans and execution performance.

```
-- A non-correlated sub-query
```

```
select
  book_title
from
  book
where
  book_key not in (select book_key from sales);
```

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=1 Card=1 Bytes=64)
   1    0   FILTER
   2    1    TABLE ACCESS (FULL) OF 'BOOK' (Cost=1 Card=1 Bytes=64)
   3    1    TABLE ACCESS (FULL) OF 'SALES' (Cost=1 Card=5 Bytes=25)
```

```
-- An outer join
```

```
select
  book_title
from
  book   b,
  sales  s
where
  b.book_key = s.book_key(+)
and
  quantity is null;
```

```
Execution Plan
----------------------------------------------------------
0   SELECT STATEMENT Optimizer=CHOOSE (Cost=3 Card=100 Bytes=8200)

1  0 FILTER
2  1   FILTER
3  2    HASH JOIN (OUTER)
4  3     TABLE ACCESS (FULL) OF 'BOOK' (Cost=1 Card=20 Bytes=1280)
5  3     TABLE ACCESS (FULL) OF 'SALES' (Cost=1 Card=100 Bytes=1800)
```

```
-- A Correlated sub-query
```

```
select
  book_title
from
  book
where
  book_title not in (
                select
                distinct
                  book_title
                from
                  book,
                  sales
                where
                  book.book_key = sales.book_key
                and
                  quantity > 0);
```

```
Execution Plan
----------------------------------------------------------
0   SELECT STATEMENT Optimizer=CHOOSE (Cost=1 Card=1 Bytes=59)
1  0  FILTER
2  1   TABLE ACCESS (FULL) OF 'BOOK' (Cost=1 Card=1 Bytes=59)
3  1   FILTER
4  3     NESTED LOOPS (Cost=6 Card=1 Bytes=82)
5  4       TABLE ACCESS (FULL) OF 'SALES' (Cost=1 Card=5 Bytes=90)
6  4       TABLE ACCESS (BY INDEX ROWID) OF 'BOOK' (Cost=1 Card=1)
7  6         INDEX (UNIQUE SCAN) OF 'PK_BOOK' (UNIQUE)
```

The formulation of the SQL query has a dramatic impact on the execution plan for the SQL, and the order of the WHERE clause predicates can make a difference. Savvy Oracle developers know the most efficient way to code Oracle SQL for optimal execution plans, and savvy Oracle shops train their developers to formulate efficient SQL.

The following section will show how the WITH clause can help simplify complex queries.

Using the WITH clause to simplify complex SQL

Oracle SQL can run faster when complex subqueries are replaced with global temporary tables. Starting in Oracle9i release 2, there was an incorporation of a subquery factoring utility implemented the SQL-99 WITH clause. The WITH clause is a tool for materializing subqueries to save Oracle from having to recompute them multiple times.

Use of the SQL WITH clause is very similar to the use of Global Temporary Tables (GTT), a technique that is often employed to improve query speed for complex subqueries. The following are some important notes about the Oracle WITH clause:

- The SQL WITH clause only works on Oracle 9i release 2 and beyond.
- Formally, the WITH clause was called subquery factoring.
- The SQL WITH clause is used when a subquery is executed multiple times.
- The ANSI WITH clause is also useful for recursive queries, but this feature has not yet been implemented in Oracle SQL.

The following example shows how the Oracle SQL WITH clause works and see how the WITH clause and Global temporary tables can be used to speed up Oracle queries.

All Stores with above average sales

To keep it simple, the following example only references the aggregations once, where the SQL WITH clause is normally used when an aggregation is referenced multiple times in a query.

The following is an example of a request to see the names of all stores with above average sales. For each store, the average sales must be compared to the average sales for all stores as shown in Figure 15.1.

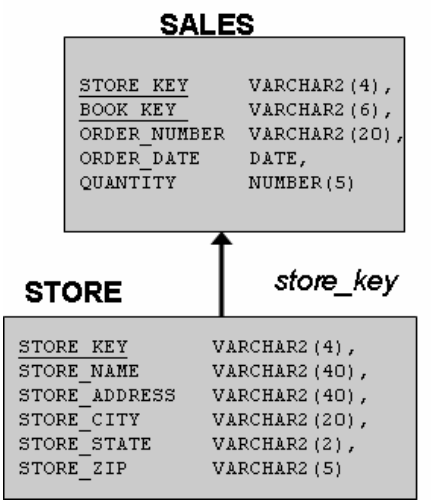


Figure 15.1: *The relationship between STORE and SALES*

Essentially, the query below accesses the STORE and SALES tables, comparing the sales for each store with the average sales for all stores. To answer this query, the following information must be available:

- The total sales for all stores.
- The number of stores.
- The sum of sales for each store.

To answer this in a single SQL statement, inline views will be employed along with a subquery inside a HAVING clause:

```
select
   store_name,
   sum(quantity)                                                    store_sales,
   (select sum(quantity) from sales)/(select count(*) from store) avg_sales
from
   store  s,
   sales  sl
where
   s.store_key = sl.store_key
having
   sum(quantity) > (select sum(quantity) from sales)/(select count(*) from store)
group by
   store_name
;
```

While this query provides the correct answer, it is difficult to read and complex to execute as it is recomputing the sum of sales multiple times.

To prevent the unnecessary re-execution of the aggregation (sum(sales)), temporary tables could be created and used to simplify the query. The following steps should be followed:

1. Create a table named T1 to hold the total sales for all stores.
2. Create a table named T2 to hold the number of stores.

3. Create a table named T3 to hold the store name and the sum of sales for each store.

A fourth SQL statement that uses tables T1, T2, and T3 to replicate the output from the original query should then be written. The final result will look like this:

```
create table t1 as
select sum(quantity) all_sales from stores;

create table t2 as
select count(*) nbr_stores from stores;

create table t3 as
select store_name, sum(quantity) store_sales from store natural join sales;


select
   store_name
from
   t1,
   t2,
   t3
where
   store_sales > (all_sales / nbr_stores)
;
```

While this is a very elegant solution and easy to understand and has a faster execution time, the SQL-99 WITH clause can be used instead of temporary tables. The Oracle SQL WITH clause will compute the aggregation once, give it a name, and allow it to be referenced, perhaps multiple times, later in the query.

The SQL-99 WITH clause is very confusing at first because the SQL statement does not begin with the word SELECT. Instead, the WITH clause is used to start the SQL query, defining the aggregations, which can then be named in the main query as if they were real tables:

```
WITH
   subquery_name
AS
  (the aggregation SQL statement)
SELECT
  (query naming subquery_name);
```

Retuning to the oversimplified example, the temporary tables should be replaced with the SQL WITH clause:

```
WITH
  sum_sales      AS
      select /*+ materialize */
      sum(quantity) all_sales from stores
  number_stores  AS
      select /*+ materialize */
      count(*) nbr_stores from stores;
  sales_by_store AS
      select /*+ materialize */
      store_name, sum(quantity) store_sales from
      store natural join sales
SELECT
   store_name
FROM
```

```
   store,
   sum_sales,
   number_stores,
   sales_by_store
WHERE
   store_sales > (all_sales / nbr_stores)
;
```

Note the use of the Oracle undocumented *materialize* hint in the WITH clause. The Oracle *materialize* hint is used to ensure that the Oracle CBO materializes the temporary tables that are created inside the WITH clause, and its opposite is the undocumented *inline* hint. This is not necessary in Oracle10g, but it helps ensure that the tables are only created one time.

Tip! Depending on the release of Oracle in use, the global temporary tables (GTT) might be a better solution than the WITH clause because indexes can be created on the GTT for faster performance.

Future enhancement to the WITH clause

Even though it is part of the ANSI SQL standard, as of Oracle 10g, it should be noted that the WITH clause is not yet fully functional within Oracle SQL, and it does not yet support the use of WITH clause replacement for CONNECT BY when performing recursive queries.

To show how the WITH clause is used in ANSI SQL-99 syntax, the following is an excerpt from Jonathan Gennick's work *Understanding the WITH Clause* showing the use of the SQL-99 WITH clause to traverse a recursive bill of materials hierarchy.

NOTE: This ANSI SQL syntax does NOT work (yet) with Oracle SQL

```
WITH recursiveBOM
   (assembly_id, assembly_name, parent_assembly) AS
(SELECT parent.assembly_id,
        parent.assembly_name,
        parent.parent_assembly
FROM bill_of_materials parent
WHERE parent.assembly_id=100
UNION ALL
SELECT child.assembly_id,
       child.assembly_name,
       child.parent_assembly
FROM recursiveBOM parent, bill_of_materials child
WHERE child.parent_assembly = parent.assembly_id)
SELECT assembly_id, parent_assembly, assembly_name
FROM recursiveBOM;
```

The WITH clause allows one to pre-materialize components of a complex query, making the entire query run faster. This same technique can also be used with Global temporary tables.

Tuning SQL with Temporary Tables

It has long been understood that materializing a subquery component can greatly improve SQL performance. Oracle global temporary tables are a great way to accomplish this.

With global temporary tables Oracle can allow hundreds of end-users to create their own copies of intermediate SQL result sets, independent of other users in the system. A silver bullet has been included at the end of this chapter with more details on tuning SQL with temporary tables.

Before delving into the tuning of individual SQL statements, it will be useful to examine how global Oracle parameters and features influence SQL execution. When tuning SQL, it is critical to optimize the instance as a whole before tuning individual SQL statements. This is especially important for proactive SQL tuning where the SQL may change execution plans based on changes in object statistics.

Oracle SQL Performance Parameters

Making the cost-based SQL optimizer (CBO) one of the most sophisticated tools ever created has cost Oracle Corporation millions of dollars. While the job of the CBO is to always choose the most optimal execution plan for any SQL statement, there are some things that the CBO cannot detect. This is when the DBA's expertise is needed.

The best execution plan for an SQL statement is affected by the types of SQL statements, the speed of the disks, and the load on the CPUs. For instance, the best execution plan resulting from a query run at 4:00 a.m. when 16 CPUs are idle may be quite different from the same query at 3:00 p.m. when the system is 90 percent utilized.

The CBO is not psychic, despite the literal definition of Oracle. Oracle can never know, the exact load on the Oracle system; therefore, the Oracle professional must adjust the CBO behavior periodically.

Most Oracle professionals make these behavior adjustments using the instance wide CBO behavior parameters such as *optimizer_index_cost_adj* and *optimizer_index_caching*. However, Oracle does not advise altering the default values for many of these CBO settings because the changes can affect the execution plans for thousands of SQL statements.

Some major adjustable parameters that influence the behavior of the CBO are shown below:

- **parallel_automatic_tuning:** Full-table scans are parallelized when set to ON. Since parallel full-table scans are extremely quick, the CBO will give a higher cost to index access and will be friendlier to full-table scans.
- **hash_area_size** (if not using *pga_aggregate_target*): The setting for the *hash_area_size* parameter governs the propensity of the CBO to favor hash joins over nested loops and sort merge table joins.

- **db_file_multiblock_read_count:** The CBO, when set to a high value, recognizes that scattered (multi-block) reads may be less expensive than sequential reads. This makes the CBO friendlier to full-table scans.
- **optimizer_index_cost_adj:** This parameter changes the costing algorithm for access paths involving indexes. The smaller the value, the cheaper the cost of index access.
- **optimizer_index_caching:** This is parameter tells Oracle the amount the index is likely to be in the RAM data buffer cache. The setting for *optimizer_index_caching* affects the CBO's decision to use an index for a table join (nested loops) or to favor a full-table scan.
- **optimizer_max_permutations:** This controls the maximum number of table join permutations allowed before the CBO is forced to pick a table join order. For a six-way table join, Oracle must evaluate six factorial (6!), or 720 possible join orders for the tables. This parameter has been deprecated in Oracle10g.
- **sort_area_size** (if not using *pga_aggregate_target*): The *sort_area_size* influences the CBO when deciding whether to perform an index access or a sort of the result set. The higher the value for *sort_area_size*, the more likely a sort will be performed in RAM, and the more likely that the CBO will favor a sort over pre-sorted index retrieval. Note that *sort_area_size* is ignored when *pga_aggregate_target* is set and when *workarea_size_policy =auto*, unless you are using a specialized feature such as the MTS (shared servers). If dedicated server connections are used, the *sort_area_size* parameter is ignored.

Note: Oracle10g release 2 has a new sorting algorithm which claims to use less server resources (specifically CPU and RAM resources). A hidden parameter called *_newsort_enabled* (set to TRUE or FALSE} is used to turn-on the new sorting method.

Remember, the *optimizer_index_cost_adj* parameter controls the CBO's propensity to favor index scans over full-table scans. In a dynamic system, as the type of SQL and load on the database changes, the ideal value for *optimizer_index_cost_adj* may change radically in just a few minutes.

Using *optimizer_index_cost_adj*Error! Bookmark not defined.

The most important parameter is the *optimizer_index_cost_adj*, and the default setting of 100 is incorrect for most Oracle systems. For OLTP systems, resetting this parameter to a smaller value (between 10 and 30) may result in huge performance gains as SQL statements change from large-table full-table scans to index range scans. The Oracle environment can be queried so that the optimal setting for *optimizer_index_cost_adj* can be intelligently estimated.

The *optimizer_index_cost_adj* parameter defaults to a value of 100, but it can range in value from one to 10,000. A value of 100 means that equal weight is given to index versus multiblock reads. In other words, *optimizer_index_cost_adj* can be thought of as a "how much do I like full-table scans?" parameter.

With a value of 100, the CBO likes full-table scans and index scans equally, and a number lower than 100 tells the CBO index scans are faster than full-table scans. Although, with a super low setting such as *optimizer_index_cost_adj*=1, the CBO will still choose full-table scans for no-brainers such as tiny tables that reside on two blocks.

The following *optimizer_index_cost_adj.sql* script illustrates the suggested initial setting for the *optimizer_index_cost_adj*.

optimizer_index_cost_adj Error! Bookmark not defined.**.sql**

```
col c1 heading 'Average Waits for|Full Scan Read I/O' format 9999.999
col c2 heading 'Average Waits for|Index Read I/O'     format 9999.999
col c3 heading 'Percent of| I/O Waits|for Full Scans' format 9.99
col c4 heading 'Percent of| I/O Waits|for Index Scans' format 9.99
col c5 heading 'Starting|Value|for|optimizer|index|cost|adj' format 999


select
   a.average_wait                                    c1,
   b.average_wait                                     c2,
   a.total_waits /(a.total_waits + b.total_waits)  c3,
   b.total_waits /(a.total_waits + b.total_waits)  c4,
   (b.average_wait / a.average_wait)*100           c5
from
   v$system_event  a,
   v$system_event  b
where
   a.event = 'db file scattered read'
and
   b.event = 'db file sequential read'
;
```

The following is the output from the script.

```
                                                                      Starting
                                                                      Value
                                                                        for
                                                                    optimizer
                                      Percent of      Percent of      index
Average waits for   Average waits for     I/O waits       I/O waits      cost
full scan read I/O   index read I/O   for full scans   for index scans   adj

------------------ ------------------ --------------- --------------- -----
      1.473              .289                .02             .98          20
```

In this example, the suggested starting value of 20 for *optimizer_index_cost_adj* may be too high because 98 percent of the data waits are on index (sequential) block access. Weighting this starting value for *optimizer_index_cost_adj* to reflect the reality that this system has only two percent waits on full-table scan reads, a typical OLTP system with few full-table scans, is a practical matter. It is not desirable to have an automated value for *optimizer_index_cost_adj* to be less than one or more than 100.

This same script may give a very different result at a different time of the day because these values change constantly, as the I/O waits accumulate and access patterns change. Oracle10g now has the *dba_hist_sysmetric_summary* table for time-series analysis of this behavior.

Setting the SQL Optimizer Cost Model

Starting with Oracle9i, DBAs have the ability to view the estimated CPU, TEMP and I/O costs for every SQL execution plan step. Oracle Corporation has noted that typical OLTP databases are becomingly increasingly CPU-bound and has provided the ability for the DBA to make the optimizer consider the CPU costs associated with each SQL execution step.

The developers of Oracle10g recognized this trend toward CPU-based optimization by providing the ability to choose CPU-based or I/O-based costing during SQL optimization with the 10g default being CPU-costing. In Oracle10g, system stats are gathered by default, and in Oracle9i the DBA must manually execute the *dbms_stat.gather_system_stats* package to get CBO statistics.

```
alter session set "_optimizer_cost_model"=choose;
alter session set "_optimizer_cost_model"=io;
alter session set "_optimizer_cost_model"=cpu;
```

This parameter can be used to choose the best optimizer costing model for a particular database, based on the I/O and CPU load.

The choice of relative weighting for these factors depends upon the existing state of the database. Databases using 32-bit technology and the corresponding 1.7 gigabyte limit on SGA RAM size tend to be I/O-bound with the top timed events being those performing disk reads:

```
Top 5 Timed Events
~~~~~~~~~~~~~~~~~~                                      % Total
Event                                   Waits    Time (s) Ela Time
----------------------------------- ------------ ----------- --------
db file sequential read                  xxxx       xxxx      30
db file scattered read                   xxxx       xxxx      40
```

Once 64-bit became popular, Oracle SGA sizes increased, more frequently referenced data was cached, and databases became increasingly CPU-bound. Also, solid-state disk (RAM SAN) has removed disk I/O as a source of waits:

```
Top 5 Timed Events
~~~~~~~~~~~~~~~~~~                                      % Total
Event                                   Waits    Time (s) Ela Time
----------------------------------- ------------ ----------- --------
CPU time                                 xxxx       xxxx      55.76
db file sequential read                  xxxx       xxxx      27.55
```

The gathered statistics are captured via the *dbms_stats* package in 9.2 and above, and the following CPU statistics are captured automatically in 10g and stored in the *sys.aux_stat$* view.

- single block disk read time, in microseconds
- multiblock disk read-time, in microseconds)
- CPU speed in mhz

- average *multiblock_read_count* in number of blocks

A database where CPU is the top timed event may benefit from a change in the SQL optimizer to consider the CPU costs associated with each execution plan.

Using CPU costing may not be good for databases that are I/O-bound. Also, changing to CPU-based optimizer costing will change the predicate evaluation order of the query. MetaLink bulletin 276877.1 provides additional information on this.

Turning on CPU Costing

The default setting for the optimizer cost model is CHOOSE, meaning that the presence of CBO statistics will influence whether or not CPU costs are considered. According to the documentation, CPU costs are considered when SQL optimizer schema statistics are gathered with the *dbms_stat.gather_system_stats* package, which is the default behavior in Oracle10g, and CPU costs will be considered in all SQL optimization.

It gets tricky because of Bug 2820066 where CPU cost is computed whenever *optimizer_index_cost_adj* is set to a non-default value. Unless the 9.2.0.6 server patch set has been applied, the Oracle9i database may be generating CPU statistics, regardless of the CBO stats collection method.

To ensure that CPU costing is in use:

- In Oracle9i, use *dbms_stats.gather_system_stats* to collect statistics
- Set the undocumented parameter *_optimizer_cost_model=cpu*;

Turning on I/O Costing

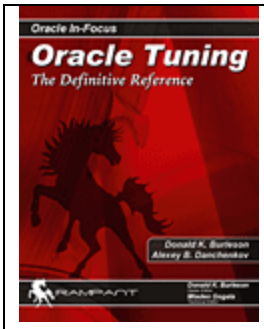
I/O-bound databases, especially 32-bit databases, may want to utilize I/O-based SQL costing. The default optimizer costing in Oracle10g is CPU, and it can be changed to IO costing by using these techniques:

- Ensure that *optimizer_index_cost_adj* is set to the default value (Oracle9i bug 2820066)
- Add a *no_cpu_costing* hint in the SQL
- alter session set *"_optimizer_cost_model"=io*;
- Set *init.ora* hidden parameter *_optimizer_cost_model=io*

Notes on Bug 2820066:

CPU cost is computed when *optimizer_index_cost_adj* is set to a non-default value. If *optimizer_index_cost_adj* is set to a non-default value, CPU costs are calculated regardless of the optimizer cost model used. If *optimizer_index_cost_adj* is set and the optimizer CPU cost model is not in use, but the explain plan shows that for queries not using domain indexes CPU costs are being calculated, this bug is likely in play.

In sum, CPU cost is always computed regardless of optimizer mode when *optimizer_index_cost_adj* is set in unpatched Oracle versions less than 10.1.0.2.



This is an excerpt from the bestselling book "Oracle Tuning: The Definitive Reference" (http://www.rampant-books.com/book_2005_1_awr_proactive_tuning.htm) by Alexey Danchenkov (http://www.wise-oracle.com/) and Donald Burleson (http://www.dba-oracle.com/books.htm), technical editor Mladen Gogala.

Incorporating the principles of artificial intelligence, Oracle10g has developed a sophisticated mechanism for capturing and tracking database performance over time periods. This new complexity has introduced dozens of new v$ and DBA views, plus dozens of Automatic Workload Repository (AWR) tables.

The AWR and its interaction with the Automatic Database Diagnostic Monitor (ADDM) is a revolution in database tuning. By understanding the internal workings of the AWR tables, the senior DBA can develop time-series tuning models to predict upcoming outages and dynamically change the instance to accommodate the impending resource changes.

This is not a book for beginners. Targeted at the senior Oracle DBA, this book dives deep into the internals of the v$ views, the AWR table structures and the new DBA history views. Packed with ready-to-run scripts, you can quickly monitor and identify the most challenging performance issues.