Optimizing the Optimizer: Essential SQL Tuning Tips and Techniques

An Oracle White Paper
September 2005

ORACLE

Optimizing the Optimizer: Essential SQL Tuning Tips and Techniques

Optimizing the Optimizer: Essential SQL Tuning Tips and Techniques

INTRODUCTION

In the real world, SQL tuning considerations and techniques vary depending on type of system being tuned, i.e., whether it is a production, test, or development system, as well as on other factors such as the availability of hardware resources, type of application, characteristics of the workload, etc. Another serious concern in the realm of SQL tuning is the avoidance of SQL performance degradation after database upgrades. In this paper, we will look at SQL tuning issues that arise in these three specific scenarios: 1) tuning of production systems, 2) tuning of development systems, and 3) prevention of SQL execution plan regressions after database upgrades. We will begin with a discussion of the various capabilities of the Oracle database that facilitate general SQL tuning, followed by specific techniques on how to effectively tune development and production systems, and finally we will outline a step-by-step approach on how to ensure that SQL execution plan regressions after database upgrades are systematically detected and fixed in a timely manner.

CAUSES OF POOR SQL PERFORMANCE

SQL statements can perform poorly for a variety of reasons. The problem may be with SQL optimization, i.e., sub-optimal execution plan, or it may have to do with inadequate resources such I/O, memory or CPU bandwidth. In this paper, we will focus on the first type of problems only, those that have to do with sub-optimal SQL execution plans.

Stale Optimizer Statistics

SQL execution plans are generated by Oracle's cost based query optimizer (CBO). CBO explores hundreds of different candidates for execution plans and then generates one with the least cost. However, for the CBO to effectively choose the most efficient or cheapest plan, it needs accurate information on the data volume and distribution of the tables and indexes referenced in the queries. Stale optimizer

statistics that do not accurately represent the current status of the data in objects can easily mislead the optimizer to generate sub-optimal plans.

Although the importance of up-to-date optimizer statistics for proper SQL optimization cannot be overstated, there really is no easy way to determine whether optimizer statistics are up-to-date or stale, and even if they are stale, whether they are causing the optimizer to generate a poor execution plan or not. For this reason it is best to err on the side of caution and ensure that the optimizer always has the most up to date information available by regularly refreshing statistics on all their objects. Oracle Database 10g has significantly diminished this problem by automating the collection of optimizer statistics. A new job "GATHER_STATS_JOB" is created-out-of-the-box that runs DBMS_STATS.GATHER_DATABASE_STATS_JOB_PROC procedure. This procedure runs during the maintenance window every night and gathers statistics for all database objects that are either stale or missing. However, a user may knowingly or unknowingly disable this job or change its default configuration that can lead to inaccurate or partial statistics collection. For these and other reasons, the problem of stale statistics continues to be encountered by many users and therefore is an issue that still needs to be considered.

Missing Access Structures

Absence of the appropriate access structures like indexes and materialized views is a common source of poor SQL performance. The right set of indexes and materialized views can improve SQL performance by several orders of magnitudes. However, identifying the right combination access structures is a non-trivial task and assessing the degradation they can cause on DML portion of the SQL workload is even more challenging.

Sub-optimal Execution Plan Selection

The CBO can sometimes choose a sub-optimal execution plan for a SQL statement when an alternative, superior plan could have been generated. This happens for the most part because of incorrect estimates of some attribute of a SQL statement such as its cost, cardinality or predicate selectivity. There are many factors that can cause large errors in the estimates and lead the optimizer to generate a sub-optimal plan. Some of the important factors are: a) the use of internal default predicate selectivity when statistics are missing (e.g., table is not analyzed), b) when the predicate is complex so the query optimizer has no idea how much data will be filtered by this predicate, c) presence of data correlation between columns of a table which means the optimizer's general assumption of no correlation is wrong, d) skewed or sparse join relationship between tables which is very difficult to capture in the form of statistics, (e) existence of data correlation between columns of two or more tables (for example, join relationship between product, location, and sales tables wherein the #sales of snow shoes is very large in New York but it is very small in Arizona). For the query optimizer to generate a good plan it is important that the estimates it

makes do not contain large errors. When such errors exist, users may see poor SQL performance due to the selection of a sub-optimal plan.

Typically, users use query hints to remedy this problem, but this requires that the users have significant expertise in SQL optimization, particularly as it relates to the Oracle database. Even if users possess such expertise, it is still a very time consuming process. A user may have to try several different hints and run Explain Plan on the SQL to see if a better execution plan is generated. Furthermore, because the application environment of a development system where tuning is most likely being performed is generally not identical to the live production system, the Explain Plans generated in such an environment may not necessarily match the actual execution plan that would be seen on the production system. All these factors make manual plan tuning a complex and expensive task. However, for packaged applications, manual plan tuning is not even an option since it requires changes to the application code and users of packaged applications generally do not have this privilege. For them the only recourse is to log a bug with the application vendor and wait for them to fix the problem, something that can easily take weeks, months, and sometimes even years.

Bad SQL Design

If a SQL statement is designed poorly, there is nothing much that can be done by the optimizer or indexes or materialized views to improve its performance. A missing join condition leading to a Cartesian join, or the use of the more expensive SQL constructs like UNION in pace of UNION ALL when there was no possibility of getting any duplicates in the result row set, are just a couple of examples of inefficient SQL design. In such cases, the only remedy is to rewrite the SQL statements in a more efficient fashion but this, once again, can be a daunting and time consuming task as it requires deep knowledge about the data properties (e.g., there are no nulls in a column) as well as a very good understanding of the semantics of SQL constructs.

These are the four main causes of poor SQL optimization. They can have a drastic impact on performance and each one of them poses its set of challenges that can be difficult to address or remedy in a comprehensive, quick and efficient manner.

OVERVIEW OF SQL ADVISORS

SQL Tuning Advisor and SQL Access Advisor were introduced in Oracle Database 10g. These advisors automate the SQL tuning process and help eliminate the need for manual tuning by comprehensively addressing all the main causes of poor SQL optimization.

SQL Tuning Advisor

The SQL Tuning Advisor takes one of more SQL statements as input and tunes it by performing the following four types of analyses on them:

- **Statistics Analysis**: The SQL Tuning Advisor checks each object referenced in a SQL statement for missing or stale statistics, and makes appropriate recommendations to gather relevant statistics. It also collects auxiliary information to supply missing statistics or correct stale statistics in case recommendations are not implemented and the new statistics help produce a better execution plan.
- **Access Path Analysis**: The SQL Tuning Advisor explores whether a new index can be used to significantly improve access to each table in the query, and when appropriate makes recommendations to create new indexes.
- **Plan Tuning (SQL Profiling)**: In this analysis, the SQL Tuning Advisor addresses the sub-optimal plan selection problem. It interfaces with the query optimizer in a special tuning mode, which then validates its own estimates and collects auxiliary information to remove estimation errors. In this analysis, past execution history of the SQL statements is also used to determine correct settings for the optimizer. For example, if the execution history reveals that a SQL statement is only partially executed majority of times then appropriate setting will be to set optimizer setting to FIRST_ROWS. This will be a customized setting for the SQL statement being tuned. All the customized information collected in this analysis is stored a new object called a SQL Profile. When a SQL Profile is created it enables the CBO to use the optimizer statistics in conjunction with the new information collected in the SQL Profile to generate a well-tuned plan. This eliminates the need for manually adding query hints to SQL statements to tune them and because it does not make any direct changes to the application code, it can also be used for tuning packaged applications.
- **SQL Structure Analysis**: Here the SQL Tuning Advisor performs SQL structure analysis to detect poorly written SQL statements and recommends, subject to user verification, possible alternative ways of rewriting the SQL statement to improve its performance. There are various reasons related to the structure of the SQL statement that can cause poor performance. Some reasons are syntax-based, some are semantics-based, and some are purely design issues.

  i. **Semantic-based Constructs**: A construct such as NOT IN subquery, when replaced by a corresponding but not semantically equivalent NOT EXISTS subquery can result in a significant performance boost. However, this replacement is possible only if NULL values are not present in the related join columns, thus ensuring that same result is produced by either of these operators. Another example is the replacement of UNION with UNION ALL provided there is no possibility of getting duplicate rows in the result.

  ii. **Syntax-based Constructs**: Many of these are related to how predicates are specified in a SQL statement. For example, if a predicate such as col

= :bnd is used with col and :bnd having different types, then such a predicate cannot be used as an index driver. Similarly, a predicate involving a function or expression (e.g. func(col) = :bnd, col + 1 = :bnd) will not be used as an index driver unless there is a functional index on the expression or function itself.

iii. **Design Issues**: An accidental use of a Cartesian product, for example, is a common problem occurring when one of the tables is not joined to any of the other tables in a SQL statement. This can happen especially when the query involves a large number of tables.

Thus, the SQL structure analysis helps both novice and experts alike to write more efficient SQL statements by giving them precise advise on exactly what to change and how to change it to get better SQL performance.

SQL Access Advisor

The SQL Access Advisor complements SQL Tuning Advisor functionality by focusing in the identification of indexes (bitmap, b-tree-, functional), materialized views, indexes on materialized views, and materialized view logs to improve the performance of the entire SQL workload. It considers the impact the new access structures will have on the DML statements in the SQL workload provided as input. Only if new indexes or materialized views will improve the performance of the entire workload, are they recommenced.

The main difference between the SQL Tuning Advisor's access path analysis and the analysis performed by the SQL Access Advisor is as follows:

- SQL Tuning Advisor looks at each SQL statement in the input and tunes it individually, while the SQL Access Advisor looks at the entire SQL workload given as input before generating recommendations. This means that the SQL Access Advisor takes into account the impact of DML operations, if any are present in the input, before producing any recommendations.
- SQL Tuning Advisor access analysis is limited to b-tree indexes while the SQL Access Advisor analysis includes b-tree, bitmap, and functional indexes as well as materialized views.
- SQL Tuning Advisor will only generate b-tree index recommendation if it will improve the SQL performance by at least 90%. SQL Access Advisor has no such preset threshold for recommendations.

For all the reasons cited above, SQL Access Advisor is the primary solution for identifying missing access structures. Limited access analysis capabilities were added to the SQL Tuning Advisor to detect any important index dropped by mistake, a not so uncommon occurrence, and because it tunes each statement individually and does not consider the impact of DML type operations in the workload on the SQL being tuned, a high index recommendation threshold was set

to ensure that performance improvement realized as a result of the recommendation more than compensated for any negative impact on the DML operations. For this reason, any time a recommendation is made from the SQL Tuning Advisor to create an index, an auxiliary recommendation to verify it via the SQL Access Advisor is also made.

SQL TUNING BEST PRACTICES

Now that we have given a brief overview of the SQL tuning solutions and the types of problems they address, we will now discuss best practices of using these solutions. We will focus on three specific SQL tuning use cases:

1. Techniques for tuning SQL in live production system,
2. Techniques for tuning SQL in development systems, and
3. Tips on how to avoid plans regressions during upgrade from Oracle Database 10.1 to 10.2.

Identifying Problem SQL

Before we can discuss how to tune SQL statements, we first have to address the issue of identifying problem SQL statements. In some cases it may be easy, for example, when a user runs a query in SQL*Plus and it performs poorly. In this case, it is obvious which is the offending SQL statement. In most cases, however, it is not that easy. Generally users are running some application and a particular module within that may start running slowly. They may not know exactly which SQL statements within that module are causing the slowdown. To deal with these situations Oracle offers two key sources for identifying problem SQL statements:

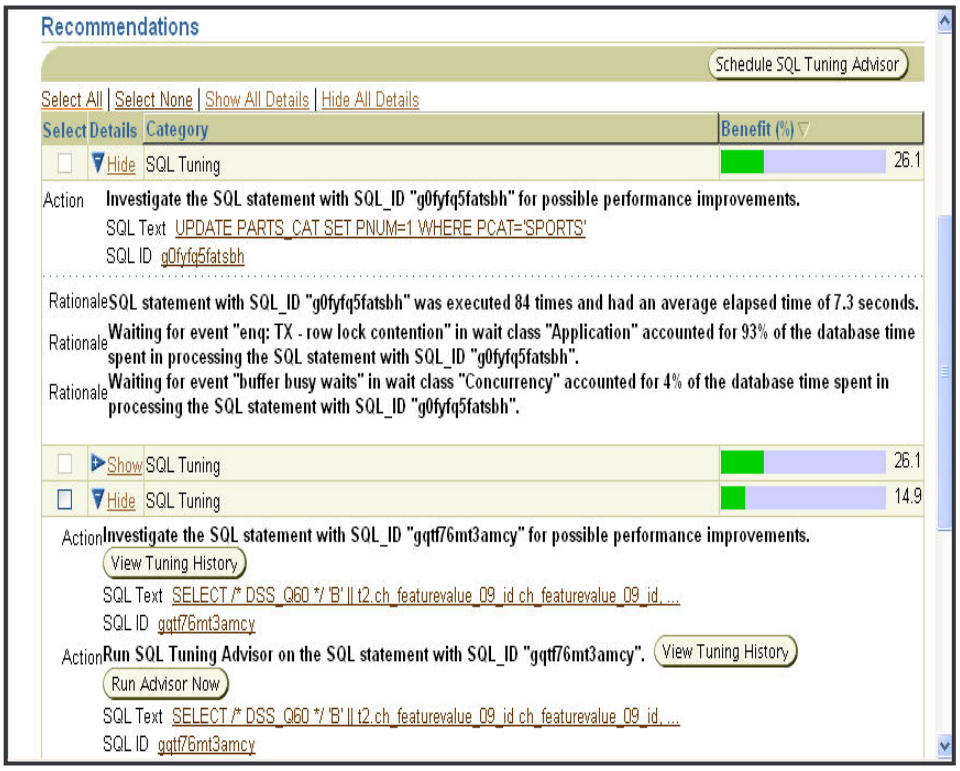
1. Automatic Database Diagnostic Monitor (ADDM)

   ADDM is the self-diagnostic engine of the Oracle database that proactively identifies all kinds of performance problems in the database including high-load SQL statements. By default, it runs every hour analyzing the database for any performance bottlenecks. If bottlenecks are found, it identifies its root cause and proposes a remedy for it.

   ADDM is the primary source of identifying high-load SQL statements. Every run of ADDM produces a report containing its analysis and recommendations. These reports are stored, by default, for 30 days in the database before being purged. You can therefore go to the report produced in the period when performance problems was being experienced on the system to see if they were caused by SQL statements. If high-load SQL statements are found, ADDM would recommend running SQL advisors on them. It should be noted that not all high-load SQL statements are good candidates for SQL advisors. For example, if a SQL statement was taking too long to run because of high-watermark enqueue (HWM) waits, this is a space configuration problem and

there is not anything the optimizer or SQL advisors can do about it. ADDM is cognizant of these facts and does not recommend the running of SQL advisors in such cases. Figure 1 below shows a sample ADDM report. Potential problem SQL are identified and the recommendation to run SQL Tuning Advisor is not made for all SQL but for only those where applicable

Figure 1: Using ADDM to identify high-load SQL

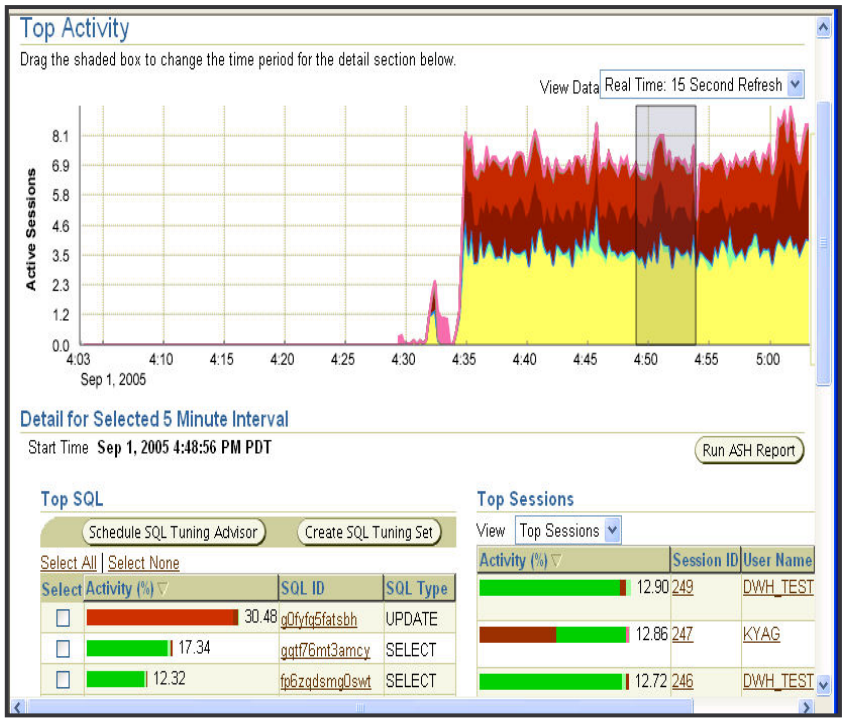


2. Top Activity Enterprise Manager Screens

Oracle Enterprise Manager (EM) 10g facilitates detection of problem SQL by showing the high-load SQL statements of the system. These screens can be viewed in two modes, real-time and historical.

- Real-time Mode: This is the default mode. From the Performance page, the ***Top Activity*** link takes you to the screen showing the high-load SQL statements currently on the system. Figure 2 shows this screen. The source of this data is the view V$ACTIVE_SESSION_HISTORY, or ASH for short. ASH displays active sessions for the last one hour on the system and the SQL being run by them. This mode is very helpful in detecting problem SQL statements currently affecting the system.

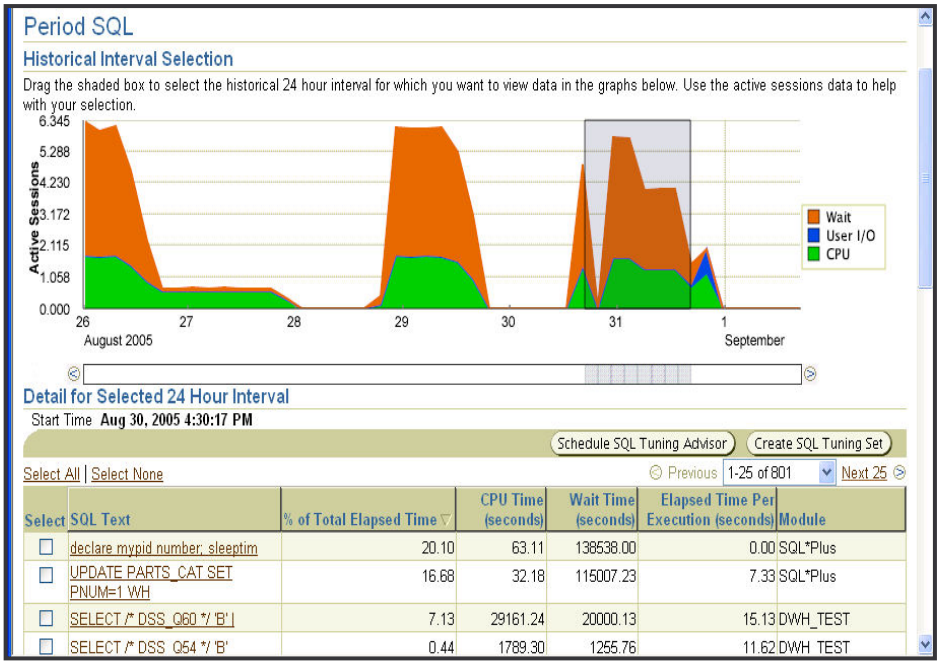
Figure 2: Top Activity EM screen. Data source is ASH.



- Historical Mode: The EM Performance page can also be viewed in historical mode. By default, last seven day's data is retained in the database for historical analysis. In this mode, the ***Period SQL*** link will take you to the page showing historical high-load SQL statements. You can select the 24-hour period that to want to analyze based on the input from users about when they were experiencing performance issues, and the corresponding SQL statements will then be shown for that period. Figure 3 shows the historical top SQL screen. This source for this data are the Automatic Workload Repository (AWR) views. The amount of data available in this mode is dependant on the retention and snapshot interval of the AWR. The default configuration is one snapshot every hour with a retention period of 7 days.

Identification of the SQL statements that are actually the cause of performance problems is the first key step to SQL tuning. ADDM, ASH and AWR along with their EM screens enable the quick and precise identification of such SQL.

Figure 3: Historical top SQL screen. Data source is AWR.



Tuning SQL of a Production System

Tuning actions such as running SQL advisors do consume resources. CPU, memory, and I/O are all consumed to varying degrees by the SQL advisors. Before launching SQL advisor to tune problem SQL statements on a production system, one must evaluate whether the system can spare the resources needed for tuning in order to ensure that performance does not suffer even more as a result of these actions.

Another important consideration when tuning a live production system is that a tuning recommendation may impact a system negatively in some, albeit rare, cases. For example, if the SQL Tuning Advisor detects stale statistics it will recommend that they be refreshed. This refreshing of statistics may help tune the SQL under consideration but it may negatively impact another statement going against the same set of objects. Since this is always a possibility, even though it may be an uncommon one, it is a good practice to always test tuning actions in a controlled environment before making them public. Consequently, before initiating tuning on a live production system, one must first ask the following questions:

1. How much system resources will be consumed by the tuning activity?
2. Can the system spare resources for tuning?

3. Can the tuning recommendations be tested in a controlled environment before making them public in order to shield the system from any potential negative impact?

Once these questions have been answered, we can then decide which technique to employ for tuning the problem SQL.

Resource Consumption of SQL Tuning and Access Advisors

The SQL Tuning Advisor has two modes of operation, *limited* and *comprehensive*. In limited mode, only three of the four types of analyses described earlier are performed. These are statistics analysis, access analysis, and SQL structure analysis. SQL Profiling (plan tuning) is not performed in this mode. The resource consumption is minimal in this mode. On average it takes less than 1 second to analyze one SQL statement.

When the SQL Tuning Advisor is run in comprehensive mode, SQL Profiling is performed in addition to the statistics, access and SQL structure analysis. SQL Profiling can possibly be an expensive operation. It takes roughly the same amount of time and resources to profile a SQL statement as it takes to run it. This means that long running SQL statements will proportionately consume more resources than a SQL statement of short duration. Thus, if SQL statements being tuned are very resource intensive, or the SQL themselves are light but a large number of them are being tuned such that the cumulative execution time is significant, then **remote** tuning should be performed, provided of course that the system is running close to capacity and cannot spare resources. If, on the other hand, the system can spare the needed resources for tuning or if the system is performing so badly then additional degradation will not matter, then one can tune the live production system **directly**.

For the SQL Access Advisor, resource consumption becomes significant only when a large number of SQL statements are being tuned. Typically, on a production system, we expect only a handful of SQL statements to require tuning. For such SQL, the resource consumption overhead of running SQL Access Advisor would normally be quite small and it would be acceptable to tune the system directly as opposed to performing remote tuning.

Tuning Directly on a Live Production System

The process to tune a live production system directly can be divided into three main steps:

1. After determining that the production system has sufficient spare resources for tuning, the first step is to ensure that all tables and indexes references in the problem SQL being tuned have up-to-date optimizer statistics. This is necessary for the obvious reason that the CBO needs up-to-date statistics for proper optimization but also because our next step is to run the SQL Access Advisor, and it relies on the presence of statistics and their accuracy for its

recommendations. If you are unsure of about the status of the optimizer statistics, you can run SQL Tuning Advisor in limited mode for verification. If the objects have stale statistics, a recommendation to refresh them will be generated.

2. After refreshing any stale optimizer statistics, the next step is to run the SQL Access Advisor in *comprehensive* mode. This will identify any missing access structures than can improve SQL performance. If the input to the advisor is just a few SQL statements and is not a complete workload, then you should set the "**Recommendation Type**" option to indexes only. This excludes materialized views recommendation. This is because for a single SQL statement, a materialized view will always give the best performance. However, if the input to the SQL Access Advisor is a complete (or almost complete) workload then materialized view recommendations would be appropriate because the advisor will take into account the negative impact the new materialized view may have on any DML statements in the workload. Thus, for tuning workloads, the "Recommendation Type" option should be set to indexes and materialized views and while tuning just one-off SQL statements, the option should be set to indexes only.

   After running the advisor with the appropriate option described above, implement its recommendations.

3. The third and the last step is to run the SQL Tuning Advisor on the SQL statements in comprehensive mode in order to tune the execution plan. If the execution plan can be tuned, a SQL Profile will be recommended. It is highly recommended that the SQL Profile be tested in a private session before making it public. In order to do this the category of the SQL Profile should be set to a non-default value when accepting it.

```
DBMS_SQLTUNE.ACCEPT_SQL_PROFILE(
                task_name => '<tuning task name>',
                name => '<SQL Profile Name>',
                category => 'MY_CATEGORY') ;
```

   This will enable the profile only in the session whose category is set to "MY_CATEGORY". You can then alter your session and set it to the category of the profile:

```
ALTER SESSION SET SQLTUNE_CATEGORY = 'MY_CATEGORY';
```

   You can test the SQL statement in this session and once satisfied with its performance set the profile category to DEFAULT as shown below.

```
DBMS_SQLTUNE.ALTER_SQL_PROFILE (
                name => '<SQL Profile Name>',
                attribute_name => 'category',
                value => 'DEFAULT');
```

These are some basic techniques on how to directly tune live production systems. Oracle Database 10g offers some very helpful solutions that enable the safe tuning of these systems.

Remote SQL Tuning Techniques

The main goal of remote SQL tuning is to protect a live system from the performance degradation caused by extra resources being consumed by the tuning actions. As discussed earlier, depending on the number and type of SQL statements being tuned, the production system may not have resources to spare and hence remote tuning becomes the only suitable option.

SQL Tuning Sets (STS) were introduced in Oracle Database 10g Release 1 for capturing and managing SQL workloads. In release 2, STS have been enhanced and are now transportable between databases. Similarly SQL Profiles have also been enhanced and are now transportable as well. These two enhancements of Transportable STS and Transportable SQL Profiles now makes it possible to perform remote tuning.

Described below are the key steps for performing remote tuning:

1. Setup test system

   A test system needs to be created where remote tuning will be performed. This system should be identical to the production system. It should have the same:

   - Schema.
   - Data distribution, and
   - Data volume.

   It is important to have the same data distribution in particular because plan tuning or SQL Profiling involves adjusting optimizer estimates of key statistics. For these adjustments of statistics to be valid on the source or production system, data distribution and volume should be identical on both systems.

2. Create STS on source or production system containing problem SQL

   EM can be used to create an STS containing the problem SQL identified earlier. By choosing the ***Period SQL*** link on the EM performance page (under historical mode), the appropriate SQL can be selected and then using the ***Create SQL Tuning Set*** option, an STS containing the problem SQL will be created.

3. Export STS to target or test system

   The Transportable STS feature of Oracle Database 10g Release 2 makes the export of STS to the test system possible. To export an STS:

- Create a staging table. This table will be used to download the contents of the STS into it. The command to create the staging table is shown below. The name of the table created in this example is STAGING_TABLE.

  ```
  DBMS_SQLTUNE.CREATE_STGTAB_SQLSET (
  table_name => 'STAGING_TABLE' );
  ```

- Next the contents of the STS need to be downloaded into the staging table. This can be done as follows:

  ```
  DBMS_SQLTUNE.PACK_STGTAB_SQLSET(
  sqlset_name => '<STS Name>',
  staging_table_name => 'STAGING_TABLE');
  ```

- Now that the staging table has been uploaded with the contents of the STS, it can now be exported to the target database using any data transfer utilities such a Data Pump, export/import, or SQL*Loader, etc. Alternatively you can also copy the table using DB Links.

- Once the staging table has been moved to the test system, the final step is to unload the contents of the STS from the staging table. This can be done be using the following command:

  ```
  DBMS_SQLTUNE.UNPACK_STGTAB_SQLSET(
     sqlset_name => '%', replace => TRUE,
     staging_table_name => 'STAGING_TABLE');
  ```

  This will create an STS on the test system with the same name as that on the production system.

4. Tune STS using SQL Tuning Advisor on test system

   Now that the problem SQL statements have been moved to the remote system, the SQL Tuning Advisor can be used for tuning in comprehensive mode without worrying about the negative performance impact on production system. Run the SQL Tuning Advisor in comprehensive mode on the STS.

5. Implement advisor recommendations and verify results

   Review advisor recommendations and implement them to verify results. The most likely recommendations, if any, are expected to be SQL Profile recommendations. This is because one would expect that the SQL Tuning Advisor would have already been run on the production system in limited mode, since it is quite inexpensive in terms of resources consumption. In that case, index, statistics refresh, and SQL restructure recommendations would already have been reviewed and implemented as appropriate on the production system.

6. If SQL Profile is recommended, set profile category to non-default value. After all the SQL Profiles recommended have been verified on the test system,

their category should be changed to a non-default value. This is because the profile should first be tested in a private session on the production system before it is made public to all users. The profile category can be changed as follows:

```
DBMS_SQLTUNE.ALTER_SQL_PROFILE(
name => '<SQL Profile Name>',
attribute_name => 'category',
value => 'MY_CATEGORY');
```

This sets the profile category to 'MY_CATEGORY'.

7. Export SQL Profile from test to production system

   The Transportable SQL Profile feature of Oracle Database 10g Release 2 makes this possible. The procedure for exporting a SQL Profile is identical to export an STS. It also consists of 4 steps:

   - Create staging table. This staging table will be used to store the contents of the SQL Profile.

     ```
     DBMS_SQLTUNE.CREATE_STGTAB_SQLPROF(
     table_name => 'PROFILE_STGTAB');
     ```

     The name of the table created is PROFILE_STGTAB.

   - Load the contents of the SQL Profile in the staging table.

     ```
     DBMS_SQLTUNE.PACK_STGTAB_SQLPROF (
     profile_name => 'MY_PROFILE',
     staging_table_name => 'PROFILE_STGTAB');
     ```

     The name of the SQL Profile being loaded is MY_PROFILE.

   - Export the staging table, PROFILE_STGTAB to the production system using Data Pump, export/import, or DB Links.

   - Download SQL Profile content from staging table to create profile in production.

     ```
     DBMS_SQLTUNE.UNPACK_STGTAB_SQLPROF(
     replace => FALSE,
     staging_table_name => 'PROFILE_STGTAB');
     ```

8. Test profile in private session and verify results

   A SQL Profile can be tested in a private session by setting the session and profile category to the same value, other than DEFAULT. To do this, set the session category value to MY_CATEGORY, which is the same as SQL Profile category set in step 6.

   ```
   ALTER SESSION SET SQLTUNE_CATEGORY='MY_CATEGORY';
   ```

9. If SQL performance okay, set profile category to DEFAULT

Once the performance of the SQL has been tested and is found satisfactory, the profile category can be set to DEFAULT so that it is now available to all.

```
DBMS_SQLTUNE.ALTER_SQL_PROFILE(
name => '<SQL Profile Name>',
attribute_name => 'category',
value => 'DEFAULT');
```

Using the remote tuning techniques outlined above, we can now tune poorly performing SQL statements on production systems without being worried about their impact on system performance and can also thoroughly test and verify any remedies before their implementation.

Tuning SQL of a Development System

Types of SQL tuning issues and resource constraints of a development system are not the same as that of a production system. Generally, there are not severe restrictions on how much resources can be consumed by tuning actions. Types of problems being addressed are also different in that they are more focused around SQL and schema design issues. This necessitates a different approach than that used in tuning a production system. The steps for this approach are described below:

1. Capture SQL workload

   As mentioned earlier, SQL Tuning Sets (STS) are new objects introduced in Oracle Database 10g for capturing and managing SQL workloads. To capture the systems SQL workload, first create an STS:

   ```
   DBMS_SQLTUNE.CREATE_SQLSET(
   sqlset_name => 'MY_STS',
   sqlset_owner => own);
   ```

   Next run the test workload on the system while simultaneously capturing it in the STS. STS have been enhanced in Oracle Database 10g Release 2 to capture workload directly from the cursor cache.

   ```
   DBMS_SQLTUNE.CAPTURE_CURSOR_CACHE_SQLSET (
   sqlset_name => 'MY_STS',
   time_limit => 3600,
   repeat_interval => 60,
   sqlset_owner => own);
   ```

   The variable time limit should be set to the time needed in seconds for the workload to complete and repeat interval is the time in seconds that is paused between sampling the cursor cache. In the example above, the sampling interval is 60 seconds. This is the most efficient way of capturing SQL workloads in an STS.

2. Run SQL Tuning Advisor on the workload to identify and restructure poorly written SQL.

   Give the STS captured in step 1 as input to the SQL Tuning Advisor and run the advisor. The SQL structure analysis performed by the advisor helps redesign poorly written SQL by recommending more optimal SQL constructs as alternatives and identifying possible errors, such as missing join conditions. It does this by generating extensive annotations and diagnostics during the plan building process and associates them to the execution plans. The annotations include the decisions made by the optimizer and the reasons for making those decisions. Using the reasons associated with costly operators in the execution plan the advisor gives recommendations either on how to rewrite the SQL statement or to make necessary schema changes to improve the performance. After reviewing the recommendations and their rationale given by the advisor, implement them as appropriate.

3. Run SQL Access Advisor on the workload to identify appropriate access structures.

   The last step is to optimize schema design by identifying the right combination of indexes and materialized views that will most significantly improve the performance of the entire workload. To do this run the SQL Access Advisor with the STS created in step 1 as input. Set the "Recommendation Type" option to both indexes and materialized views. Review the advisor recommendations and implement them as appropriate.

Using the STS workload capture capability along with the SQL Tuning and Access Advisors described above, even novice application developers can easily improve the design of their SQL statements and tune their application for best performance.

AVOIDING PLAN REGRESSIONS AFTER DATABASE UPGRADES

One of the major concerns when performing a database upgrade is the possibility of unexpected SQL execution plan regressions. A very high percentage of all SQL statements would normally perform the same or better on an upgraded system. However, even if a tiny percentage, e.g., less than 0.1%, of SQL statements have plan regressions, result can potentially be very serious. For this reason, systematically detecting plan regressions, if any, in a newly upgraded system and remedying them in a timely manner is very important. Please note that in this paper we will restrict the discussion to upgrades from Oracle Database 10.1 to 10.2 only.

Outlined below is a step-by-step method for detecting and avoiding execution plan regressions after upgrades.

1. Increase AWR retention on the 10.1 system to be upgraded in order to capture entire workload cycle

Starting with Oracle Database 10.1, high-load SQL are automatically captured into the AWR and, by default, are kept for 7 days before being purged. If 7 days is not long enough to capture all the different SQL statements on the system because the workload cycle is longer, then increase AWR retention accordingly to accommodate the entire cycle. For example, many companies have workloads that repeat every quarter. In this case the AWR retention would be set to 3 months. This will ensure that at al times, the high-load SQL from the entire workload are captured in the AWR.

Here the assumption is that the system being upgraded is the actual/production system or one that is its exact replica (e.g., the system was created using RMAN backup). If, however, the upgrade is first being tested on a smaller system, then the optimizer statistics, at a minimum, from the actual/production system should be exported to this system. Also, the test system should have the exact same configuration settings for AWR, optimizer etc., as the actual system.

2. Allow normal workload to run on the system for a complete workload cycle

   Run the entire workload on the system. The high-load SQL will automatically be captured in AWR. Because AWR retention has already been adjusted to accommodate the entire workload, at the end of this step all the high-load SQL statements of interest should be in AWR.

3. Upgrade system to Oracle Database 10.2

   Follow the normal upgrade process and upgrade the system to 10.2.

4. Create baseline STS

   In this step all the high-load SQL captured in AWR in 10.1 are loaded into an STS. This will serve as the baseline STS and its contents, including SQL statements, execution plans, etc., are that of the 10.1 system. The only reason we create the baseline STS in 10.2 and not in 10.1 is that the STS in 10.2 also store SQL execution plans while those in 10.1 did not. This capture of execution plans generated by 10.1 in the STS will allow us to compare them to those generated by 10.2 at a later stage.

   The baseline STS can be created using EM or command-line interface. If using EM, from the STS screen, choose ***Create SQL Tuning Sets from Snapshots*** option and give the beginning and end snapshot ID's of AWR. You can also use DBMS_SQLTUNE.CREATE_SQLSET and DBMS_SQLTUNE.LOAD_SQLSET PL/SQL procedures to create the baseline STS. Shown below is an example for creating STS consisting of all SQL statements in AWR. The baseline STS created in this example is called '10.1 sts'.

```
declare
  own VARCHAR2(30) := '&owner';
  bid NUMBER := '&begin_snap';
  eid NUMBER := '&end_snap';
  sts_cur dbms_sqltune.sqlset_cursor;
begin
  dbms_sqltune.create_sqlset(sqlset_name => '10.1 sts',
     sqlset_owner => own);
   open sts_cur for
    select value(P) from
     table(dbms_sqltune.select_workload_repository(bid,
     eid,  null, null, null, null, null, 1, null,
     'ALL')) P;

  dbms_sqltune.load_sqlset(sqlset_name => '10.1 sts',
                           populate_cursor => sts_cur,
                           load_option => 'MERGE');
end;
/
```

5. Run workload on 10.2 and capture results in new STS

   Next run the workload on the upgraded 10.2 system and capture it in STS simultaneously using the CAPTURE_CURSOR_CACHE_SQLSET procedure of the DBMS_SQLTUNE package described earlier. This is shown in the example below. The new STS created is called '10.2 sts'.

```
declare
  own      VARCHAR2(30) := '&owner';
  time_lim NUMBER := &time;
begin
  dbms_sqltune.create_sqlset(sqlset_name => '10.2 sts',
                             sqlset_owner => own);
  dbms_sqltune.capture_cursor_cache_sqlset (
       sqlset_name => '10.2 sts', time_limit => time_lim,
       repeat_interval => 60, sqlset_owner => own);
end;
/
```

6. Compare contents of two STS to identify plan changes

   Now that we have two the STS, one with SQL statements and their execution plans from 10.1 database and the other from its upgraded version, we can compare their contents to see if any execution plans have changed and whether there are new high-load SQL statements in 10.2 that were absent in 10.1. This would be a clear indication that SQL performance has regressed.

   The query below compares the top 100 SQL statements in '10.2 sts' with those in '10.1 sts'. If the statement is new to 10.2 or if the plans have changed, then the OLD_PLAN_HASH column will be empty and the NEW_PLAN column

will have a 1. This statement is good candidate for plan regression investigation. For statements that are in both STS's and the execution plans have not changed the NEW_PLAN column will be 0. These statements have had no plan regressions after the upgrade and should normally represent the bulk of statements in the output.

```
select * from (
select r2_sts.sql_id,
       r1_sts.plan_hash_value old_plan_hash,
       r2_sts.plan_hash_value new_plan_hash,
       r2_sts.elapsed_time/r2_sts.executions elap_time,
       nvl2(r1_sts.plan_hash_value, 0, 1) new_plan,
       r2_sts.sql_text
from dba_sqlset_statements r1_sts,
     dba_sqlset_statements r2_sts
where r1_sts.sqlset_name (+)  = '10.1 sts' and
      r1_sts.sqlset_owner (+) = '&own' and
      r1_sts.sql_id (+) = r2_sts.sql_id  and
      r1_sts.plan_hash_value (+) = r2_sts.plan_hash_value
      and
      r2_sts.sqlset_name = '10.2 sts' and
      r2_sts.sqlset_owner = '&own'
order by elap_time desc)
where rownum < 100 ;
```

Sample output for the query is shown below.

```
SQL_ID        OLD_PLAN_HASH NEW_PLAN_HASH   ELAP_TIME
------------- ------------- -------------   ----------
NEW_PLAN      SQL_TEXT
----------    ----------------------------
fyrbzparfb4gb     403312434     403312434           23
0               select * from scott.dept
```

As shown above, the contents of the two STS can be compared by querying the DBA_SQLSET_STATEMENTS view and any plan regression can systematically be detected.

7. Examine plan changes to determine if any plans have regressed and tune them using SQL Advisors.

   The final step is to examine the new execution plans to see if they have actually regressed or improved. The DBMS_XPLAN.DISPLAY_SQLSET function can be used for this purpose as shown below:

```
select * from table(dbms_xplan.display_sqlset(
&sqlset_name, &sql_id, &plan_hash_value,
'TYPICAL', &sqlset_owner));
```

For those SQL statements whose plans have regressed, tune them using techniques discussed in the SQL tuning section of the paper.

CONCLUSION

Oracle Database 10g offers many new capabilities that drastically simplify SQL tuning. The automatic tuning capabilities of the SQL advisors eliminate the need for complex, repetitive manual tuning and the workload capture and management functionality of AWR and SQL Tuning Sets minimize the chances of encountering unexpected performance problems after upgrades. Database administrators and application developers can significantly reduce the risk of performance problems by utilizing these new capabilities to the fullest and at the same time improve system reliability and availability.

White Paper Title: Optimizing the Optimizer: Essential SQL Tuning Tips and Techniques
Septemnber 2005
Author: Mughees A. Minhas
Contributing Authors: Benoit Dageville, Khaled Yagoub, Pete Belknap

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
oracle.com