Migrating to the Cost-Based Optimizer

An Oracle White Paper
April 2005

ORACLE

EXECUTIVE OVERVIEW

In Oracle Database 10*g*, only one query optimizer is supported: the Cost-Based Optimizer (CBO). Oracle's legacy optimizer, the Rule-Based Optimizer (RBO), is no longer supported. While the majority of applications running on Oracle today use the CBO, some applications continue to use the RBO. This paper describes the steps in migrated an application from the RBO to the CBO.

INTRODUCTION

Oracle introduced the Cost-Based Optimizer (CBO) over a decade ago, with the release of Oracle7 in 1992. Prior to that, only the Rule-Based Optimizer (RBO) was available. Over the years, the CBO has been substantially improved and it supports all of Oracle's new features. In contrast, the RBO has not been enhanced since the introduction of the CBO and lacks support for many fundamental features that have been introduced since Oracle7, such as bitmap indexes, function-based indexes, hash joins, index-organized tables, and partitioning. While major packaged applications, such as SAP, Oracle E-Business Suite, and PeopleSoft use the CBO, there remain some applications that use the RBO for historical reasons. The purpose of this document is to describe how to migrate an existing application from the RBO to the CBO, a necessary pre-cursor for migrating RBO-based application to Oracle Database 10*g*.

WHY IS THE RBO NO LONGER SUPPORTED?

The RBO is being de-supported primarily for two reasons:

- The existence of the RBO prevents Oracle from making key enhancements to its query-processing engine. The removal of the RBO will permit Oracle to improve performance and reliability of the query-processing components of the database engine.
- The use of the RBO prevents applications from leveraging many of the key features and enhancements introduced since Oracle7. Oracle is committed to improving the performance, reliability, and manageability of applications in every upgrade, and the use of the CBO will enable Oracle to deliver these benefits much more effectively.

INTRODUCTION TO THE CBO

For readers who may not be familiar with the CBO, this section provides a brief introduction to both the general topic of query optimization as well as the operation of Oracle's CBO. While detailed knowledge of the CBO is not necessary to administer a database, a basic understanding of the principles behind the CBO can helpful during the migration process.

Query optimization is of great importance for the performance of a relational database, especially for the execution of complex SQL statements. A query optimizer determines the best strategy for performing each query. The query optimizer chooses, for example, whether or not to use indexes for a given query, and which join techniques to use when joining multiple tables. These decisions have a key effect on SQL performance, and thus query optimization is an important technology for every application, from operational systems to data-warehouse systems to content-management systems.

However, while the query optimizer has an important role in database performance, it is a large invisible feature from the perspective of the application and the end-user. The query optimizer is entirely transparent to the SQL application.

Because applications may generate very complex SQL, query optimizers must be extremely sophisticated and robust to ensure good performance. Determining whether or not to use an index, or choosing a specific join technique for joining two tables are the most basic operations of a query optimizer. However, a query optimizer is also capable of much more sophisticated operations. For example, query optimizers may transform SQL statements, so that view definitions are 'merged' into the text of the query, subqueries may be 'flattened' and converted into joins, and predicates may be 'pushed' into other portions of the SQL tables. These sophisticated transformations enable a query optimizer to find more efficient ways to execute complex queries.

Both the RBO and CBO of the Oracle Database are query optimizers. The inputs to both optimizers are SQL statements and the outputs from the optimizers are strategies for efficiently executing those SQL statements. These strategies for executing the SQL statements are called 'execution plans', and they can be viewed using Oracle's EXPLAIN PLAN facility.

As their names imply, the RBO is a rule-based (heuristic) query optimizer, while the CBO is a cost-based query optimizer. The RBO chooses its plans based on a set of fixed 'rules'. For example, if one has a query of the form 'select * from EMP where EMPNO < 5000', and if there is an index on EMPNO, then one of the RBO's rules specifies that this query will always be executed using the index. The behavior of the RBO is entirely fixed; the execution plan for the sample query remains the same regardless of whether the EMP tables is 10 rows or 10 million rows and regardless of whether the 'EMPNO < 5000' predicate will return 2 rows or 2 million rows.

The CBO, which is Oracle's sole query optimizer starting with Oracle Database 10*g*, uses a cost-based optimization strategy, in which multiple execution plans are generated for a given query, and then an estimated cost is computed for each plan. The query optimizer chooses the plan with the lowest estimated cost.

The CBO has built-in knowledge about the cost properties of all of the different access and join methods within the Oracle database. The CBO uses this knowledge in conjunction with statistical information about the database and the objects in this database. There are three primary areas of statistics used as input to the CBO:

- Statistics which describe the database objects involved in the query, e.g., the number of rows in a table, the number of distinct values in a column, and the number of leaf blocks of an index.
- Statistics on the relative performance of the hardware platform (so-called 'CPU statistics'). These statistics help the CBO to understand how efficiently the underlying hardware platform can process cpu-intensive and io-intensive operations. Every combination of operating system, hardware server, and storage is different, so that CPU statistics allow the CBO to automatically compensate for the strengths and weaknesses of each configuration.
- Statistics on the buffer cache, which describe whether a given table or database object is typically cached or not.

The CBO requires accurate statistics in order to deliver good query performance. Hence, it is crucial that the statistics the CBO uses are available and representative. The section of Statistics Management, below, discusses how Oracle automates statistics collections as well as how DBA's can customize statistics collection.

The use of statistics implies that the CBO by its nature is more dynamic than the RBO. Collecting new statistics could lead to some changes in execution plan if the new statistics are sufficiently different from the old. This behavior is desirable since, as a database grows over time, the optimal execution plans for an application may change as well. Moreover, for a packaged application, different installations may have different properties: a packaged application installed in a 50,000-employee company will have significantly different data characteristics than the same packaged application installed in a 200-employee company. There may not exist a set of execution plans that is optimal for all installations. With the CBO, the particular properties of the installation will determine the execution plans through the use of optimizer statistics.

In contrast to the dynamic properties of the CBO, the RBO has no notion of cost or cardinality. The RBO has no way of distinguishing a small table from a large one or a highly selective condition from a nonselective one. Consequently, the RBO is unable to generate an execution plan based on such properties or adjust execution plans over time as the properties of the database change. As a result, even RBO

plans that were hand tuned to be optimal when the application was first implemented may become inefficient as data sets grow or change.

A more in-depth description of the features and functionality of the CBO is available in Oracle's documentation, in particular in the following books: Database Concepts, Database Performance Tuning Guide and Reference, and (for data-warehouse environments) Data Warehousing Guide.

APPLICATION MIGRATION

The process of migrating from the RBO to the CBO involves many of the same issues as any other database migration. We assume that the reader is familiar with the recommended practices, such as conducting performance testing on a test system before migrating the production system. General documentation about these practices can be found in Database Migration book of Oracle's documentation. This white paper adds a more in-depth discussion of the issues related to optimizer migration, and specifically the issues of understanding and addressing changes in execution plans, than is provided in the Database Migration manual.

Changes in execution plans due to different optimizer behavior always carry a certain element of risk. Let us say that an application generates 100,000 SQL-statements and 1,000 of them change as a result of the migration. Even if 999 out of those 1,000 plan changes resulted in improved performance, the single query that deteriorated could well outweigh the improvements if it is a critical query where the response time now is unacceptable. In other words, a small number of queries where the performance goes from "acceptable" to "unacceptable" may well overshadow a larger number of queries where performance goes from "acceptable" to "even better." Hence, it is prudent to conduct sufficient testing before migrating a production system to ensure that there are no performance issues causing disruptions.

Rather than suggest a single migration strategy, this white paper suggests a range of approaches for moving to the CBO depending upon the nature of the current RBO-based application. This section will briefly discuss three strategies for migrating to the RBO, based on three profiles:

Type A:

- Description: High-profile, mission-critical application. Performance requirements: Any performance degradations could potentially have severe consequences on company operations.
- Migration goal: Zero performance degradations for all commonly executed SQL statements.

Type B:

- Description: Application used by large number of users, and significant performance changes would be serious. However, minor or temporary performance changes would only be an inconvenience.
- Migration goal: Minor performance degradations for <10% of the queries would be acceptable. Minimize risk of severe performance degradations, but balance migration effort with risk.

Type C:

- Description: Application's performance is not mission-critical. The application has a small number of users and/or the usage of it is peripheral to the end-user's job function. Minor performance degradations would likely be unnoticed.
- Migration goal: Minor performance degradations are acceptable. Minimize the effort required for migration.

The following provides a high-level migration strategy for each of these three types of applications, starting with the simplest scenario (Type C). This is only a high-level outline of the migration strategy. The subsequent sections provide more details on each step.

Migration strategy for a Type C application:

1. Do this application have a test system?

(yes) Go to step 2.

(no) Go to step 4.

2. Using the test system, gather statistics (see 'Managing Statistics' below) and enable the CBO (see 'Relevant Initialization Parameters' below). Run the application and validate performance.

Is performance acceptable?

(yes) Go to step 4.

(no) Go to step 3.

3. Troubleshooting: Identify queries with performance issues, analyze their execution plans. Devise resolution strategy. (See 'Comparing Execution Plans' below). Test resolution strategy by repeating step 2.

4. Test CBO performance on production instance, while other users continue to use RBO (see 'Relevant Initialization Parameters' below).

Is performance acceptable?

(yes) Go to step 5.

(no) Go to step 3.

5. Statistics-gathering: Enable dml-monitoring, and use 'GATHER AUTO' to gather statistics (see 'Gathering statistics' below).

6. Convert production users to the CBO (either all users at once, or in phases).

Migration Strategy for a Type B application:

1. Identify key SQL statements (e.g. statements which are most commonly executed and/or statements whose performance is most critical). See 'Comparing Execution Plans' below.

2. Do you have a full-sized test system?

(yes) Go to step 3.

(no) Gather statistics on the production system (after setting OPTIMIZER_MODE to RULE) and export statistics to test system (see 'Gathering Statistics' below).

3. Measure and compare the performance of key SQL statements with RBO and CBO on test system.

4. Tune any queries with degraded performance. Revalidate performance of all key queries on test system.

5. Devise and test statistics-gathering plan. Old statistics should be saved before gathering new statistics. (see 'Gathering Statistics' below).

6. Roll out test-system changes and CBO into production system (in phases, if possible).

Migration strategy for a Type A application:

1. Gather all SQL statements executed by the application (see 'Comparing Execution Plans' below).

2. Do you have a full-sized test system?

(yes) Go to step 3.

(no) Gather statistics on the production system (after setting OPTIMIZER_MODE to RULE) and export statistics to test system (see 'Gathering Statistics' below).

3. Validate that RBO execution plans on production system are identical to RBO execution plans on test system (See 'Comparing Execution Plans' below)

4. Gather the execution plans with CBO using two or more different OPTIMIZER_MODE settings and with RBO for every SQL statement. (See 'Comparing Execution Plans' below). If your application uses bind variables, compare CBO execution plans with and without peeking (see 'Bind Variables' below).

5. Compare the plans. Identify the optimizer mode and bind variable setting with the least number of plan differences.

6. For all queries with different execution plans, compare the performance results.

7. Tune all queries with degraded performance (see 'Comparing Execution Plans' below). Revalidate performance on test system.

8. Devise and test statistics-gathering plan. Recommended approach is to enable dml-monitoring and use 'GATHER AUTO'. Old statistics should be saved before gathering new statistics. (see 'Gathering Statistics' below).

9. Roll out test-system changes and CBO into production system (in phases, if possible).

Migration Strategies Summary

The goal of these proposed strategies is to balance the amount of effort required for a migration with the risks associated with a migration. Given the large number of applications of all sorts using the CBO today, most migrations should move smoothly and the key to these migration strategies to manage an optimizer migration using the best-practice principles of any other migration. One consideration to keep in mind is that the RBO is no longer supported in Oracle Database 10*g*, so that an RBO-based application must be migrated to the CBO before upgrading to 10*g*.

Finally, when an application is migrated from the CBO to the RBO, a wealth of new features and capabilities is available to that application. Some of these enhancements are implicitly enabled (such as more sophisticated handling of views and subqueries), while other improvements would occur when the DBA enhances the schema (such as partitioning tables or creating new types of indexes).

The following sections will provide details on various aspects of the CBO and its migration.

RELEVANT PARAMETERS

There are a few initialization parameters that are of specific interest when migrating from the RBO to the CBO.

Optimizer_mode

When migrating from the RBO to the CBO, the *optimizer_mode* parameter is of crucial importance. It can take the following values in Oracle9*i*:

Choose. This value is the default and has the following meaning: If any object (table, index, etc.) referenced in the SQL-statement being optimized has associated optimizer statistics, the CBO will be used. Otherwise, the RBO will be used. If the CBO is used, it will try to optimize the statement in "all_rows" mode (see below) so that it uses as little resources as possible to complete, thereby maximizing the throughput on the system.

All_rows. This mode forces the CBO to optimize for minimal resource utilization ("best throughput") regardless of whether optimizer statistics have been collected for the objects in the query.

Rule. This mode forces the use of the RBO regardless of whether optimizer statistics exist. It should be noted that if certain types of objects, such as partitioned or index-organized tables, are referenced in a statement, the CBO will always be used regardless of the setting of optimizer_mode or the existence of optimizer statistics. Also, the existence of a hint inside the query will trigger the CBO, with some exceptions, such as a RULE hint. Note that the hint need not necessarily occur in the query text itself, but could reside the definition of a view referenced in the query.

First_rows. This mode is meant to optimize for response time, i.e., the time it takes before the first result row is returned to the screen of the user. In Oracle9*i* (and later releases), we recommend using *first_rows_n* instead.

First_rows_n. Legal values for *n* are 1, 10, 100, and 1000. This mode is meant to optimize for response time, i.e., the time it takes before the first batch of *n* rows is returned to the user. The size of the batch could be a single row or a screen full depending on the application; hence, the support for different values for *n*.

The optimizer mode can also be set at the session level with *alter session*, or at the statement level using a hint. The setting at the session level overrides the initialization setting, and the setting at the statement level, in turn, overrides the setting for the session.

When migrating from the RBO to the CBO, the optimizer_mode parameter is of interest for a number of different reasons:

- It is possible to optimize for response time rather than throughput (the default) using *first_rows_n* if it is appropriate for the application. ***Most OLTP applications should use the first_rows_n behavior when migrating to the CBO.*** This mode will provide the best performance for most OLTP applications and moreover will have the most similar execution plans to the RBO.
- In order for the CBO to generate the same plan on the test system as it would on the production system, it is important that the optimizer statistics used on the test system are the same as they would be on the production system. To ensure that this is the case, it might be necessary to collect the optimizer statistics on the production system and export them to the test system. If the *optimizer_mode* is set to *rule* on an RBO-baed production system, then statistics can safely be gathered on the production system without impacting the end-users optimizer behavior. However, in order to prevent the existence of statistics from switching the production system to the CBO prematurely, it may be necessary to run the production system using the *rule* mode.

- The *optimizer_mode* parameter can be set on a per-session basis. This will enable a DBA to set *optimizer_mode* to rule at the instance level, but selectively enabled specific users or session to use other setting of *optimizer_mode*. This technique can be used for testing, so that the behavior of different settings of *optimizer_mode* can be evaluated directly on the productions. This technique can also be used to migrate users individually (by setting this parameter on a per-session basis using a log-on trigger) to the CBO, rather than migrating all users at one time.
- After the migration of the production system to the CBO, the *rule* mode can be used to switch back to the RBO in case of an emergency.

Other Parameters

In addition to the two parameters discussed so far, other optimizer parameter can also be of interest for the purpose of performance tuning. See the Oracle documentation manuals Database Performance Tuning Guide and Reference and Database Reference for further details.

COMPARING EXECUTION PLANS

Finding those SQL-statements that are likely to get different execution plans after the migration requires several steps. The statements need to be captured. Plans have to be generated and compared. Finally, the statements that have different execution plans need further investigation.

In this document, we specifically focus on using execution plan differences ("plan diffs") as a means of focusing performance testing, diagnosing problems, and finding remedies for performance deterioration. A plan diff is the comparison of execution plans between the RBO and the CBO (or between two versions of the CBO) for the same application SQL-statements. The general process is to capture the SQL-statements and their execution plans on the production system and to compare them to the plans generated on the test system, which has already been migrated. For SQL-statements that generate the same plan, there should not be any performance differences caused by different optimizer behavior. Those particular SQL-statements that have different plans are obvious candidates for further performance testing. Hence, plan diffs can be used as a means of focusing performance testing. Moreover, if there is a performance deterioration, the plan diff can be useful in order to determine its cause and possible remedies.

It should be noted, however, that plan diffs are no substitute from benchmarking the migrated test system with realistic workloads. If the migration is between two different versions of Oracle, there are numerous factors other than changes in execution plans that could give rise to performance issues. In addition, such upgrades usually require other types of testing than for mere performance, such as functionality and compatibility testing.

Capturing a Workload

In order to analyze differences in execution plans, the SQL-statements of the relevant workload must be extracted. If the source code for the application is available, it may be possible to extract the SQL-text directly from that code. However, in many cases, extracting the SQL text is not an option. For example, the application source code may not be accessible or the SQL-text may be generated dynamically as the application is executed. A more general approach is to capture the statements by monitoring the contents of the cursor cache. The text of the cursors in the cache can be accessed using the V$SQL or V$SQLAREA views. (If the statement is longer than 1000 characters, the view V$SQLTEXT is needed in order to retrieve the full text of the statement.) In Oracle9*i*, the execution plan for each cursor is also available as well as statistics about the execution. This information is available through V$SQL_PLAN and V$SQL_PLAN_STATISTICS, which can be joined to V$SQL or V$SQLAREA to match up the SQL-text with the plan. Being able to get both the SQL-text and the corresponding execution plan simultaneously has several advantages. Firstly, it may obviate the extra effort needed to generate the plan using EXPLAIN PLAN. Secondly, it shows the actual plan and is therefore more accurate than EXPLAIN PLAN. This issue will be discussed in the section on Bind Variables.

Oracle's Statspack provides the functionality for monitoring these V$-views and taking a snapshot of their contents during regular intervals during a workload. See Oracle9*i* Database Performance Tuning Guide and Reference Release 2 (9.2). In order to capture the plans as well as the SQL-text, the statistics level needs to be set to 6 or higher. Moreover, in order to capture all the SQL-statements in the cursor cache, some threshold needs to be set to a level that is low enough. For example, the threshold for the number of executions could be set to 0. The interval between snapshots needs to be sufficiently small to minimize the risk that a statement is aged out of the cursor cache without being captured. How long that would be depends on the size of the cache and the arrival rate of SQL-statements that are not in the cache. In addition, the period during which snapshots are taken needs to be long enough to ensure that all interesting aspects of the application have been exercised.

It is also useful to be able to store the captured statements and execution plans in a repository for later use. Statspack records the snapshots in tables, and the data in these tables can be the foundation of such a repository. Ideally, a unique key should be assigned to each distinct SQL-statement. Such a key could also be used to identify each plan, e.g., through the STATEMENT_ID column in the standard PLAN_TABLE. This mechanism would support doing EXPLAIN PLAN SET STATEMENT_ID = … INTO… FOR … for each SQL-statement if the queries cannot actually be executed on the test system. By creating a table with plans for each optimizer version that is tested, plan diffs can be performed through SQL-queries or through a simple PL/SQL script that compares the plans for each distinct SQL-statement to see if they are the same. See Appendix A for examples of such scripts.

Maintaining a mechanism that maps each distinct SQL-statement to a unique key can be somewhat cumbersome, so it may be easier to use the SQL hash value instead. This value is directly available in the V$-views and also stored by Statspack for each statement in the snapshot. However, the hash value is not guaranteed to be unique and there is a slight risk of collisions and those would make plan comparisons more difficult for those particular statements where collisions occurred. For an application that generates several hundred thousand distinct statements, there may be a small number of such collisions. Dealing with a very small number of collisions when analyzing plan diffs may well be less time consuming than generating a mechanism that maintains keys that are guaranteed to be unique.

When the Production and Test Systems Are Different

Sometimes, the test system cannot hold all the data of the production system or its hardware characteristics may be completely different. If so, the test system might not provide for performance experiments that would be representative of the production system. A test system that is inadequate for reliable performance experiments adds risk to the migration, but there are still some things that can be done using the test system by analyzing execution plans.

Firstly, it is crucial to extract the optimizer statistics from the production system and install them on the test system rather than gathering them on the data that is on the test system. In order to find out what plans the optimizer would generate on the production system, the statistics must be based on the production system data and no other data set. The dbms_stats package contains procedures for extracting the optimizer statistics from one system and installing them on another. See Supplied PL/SQL Packages and Types Reference of the Oracle documentation.

Secondly, if the migration is from the RBO to the CBO on the same version of Oracle, it may be possible to carry out some of the performance testing on the actual production system during off-peak hours or by a subset of the applications users acting as guinea pigs. By changing the optimizer mode to rule for the regular production users and collecting optimizer statistics, it is possible to switch the users involved in the testing to the CBO at the session level (e.g. using logon triggers) and thereby allowing performance comparisons on the same system. Theoretically, the use of a test system could be completely bypassed in this scenario provided that the production system has enough bandwidth to accommodate all the steps of the plan diff process.

Using EXPLAIN PLAN

The ideal plan diff would be based on running complete workloads on both the production and test system and capturing the plans based on the V$SQL_PLAN view using Statspack. However, it may not always be possible to generate a full workload on a test system. If so, the plan diff may have to be carried out using the EXPLAIN PLAN facility. Unfortunately, such a plan diff is not completely reliable,

but it may still be useful in many situations. This unreliability is a separate issue from that of being able to conduct performance experiments on the test system and mainly concerns getting the correct execution plans for queries with bind variables.

Bind Variables

The Oracle EXPLAIN PLAN facility has never been able to fully guarantee that the execution plan shown for a SQL-statement involving bind variables would be the same plan that would actually be used when executing the statement as opposed to merely "explaining" it. The reason for the potential discrepancy is that EXPLAIN PLAN is allowed without any actual bindings for the bind variables and, therefore, Oracle is unaware of the actual data types or values of the execution time bindings. Prior to Oracle9*i*, the potential discrepancy would only concern statements where Oracle would insert a type conversion operator in some expression based on the actual data type of the value supplied as the binding for the bind variable. The plan generated by EXPLAIN PLAN would be unaware of this conversion operator and could show an index scan when the actual execution would use a full table scan. This kind of discrepancy should be relatively rare, but the introduction of the Bind Variable Peeking feature in Oralce9*i* increases the likelihood that such discrepancies will occur since the value of the bind variable may affect the plan as well. We will discuss this feature in the next section. However, it should be pointed out that Oracle9*i* also introduced the V$SQL_PLAN view, which is the only interface that is guaranteed to show the actual plan that a SQL-statement being executed is using.

Bind Variable Peeking

Bind variables are used to increase cursor sharing and, hence, improve resource utilization for systems with large numbers of concurrent users. However, bind variables have traditionally been a problem for the query optimizer since lack of knowledge about the actual bindings have meant that the optimizer has to make its decision without some very crucial information. Consider the following query:

```
select sum(sales_price) from sales
where sales_date between :1 and :2
```

This query computes the total sum of sales between two dates. There are essentially three cases:

1. The range of dates is very narrow, in which case it may make sense to use an index on sales_date to access the sales data.
2. The range of dates is very wide, in which case accessing the sales table using a full table scan may be the most efficient alternative.
3. Some users provide a narrow date range for the bindings of :1 and :2 while others provide a wide range.

Case 3 would seem to indicate that the bind variable mechanism is being misused by the application since some users will invariably get a highly suboptimal execution

plan for the query and the overhead of the suboptimal plans will likely more than outweigh the benefits of cursor sharing. However, assuming that the bind variable mechanism is used correctly, it would be very beneficial for the optimizer to know which of cases 1 and 2 is the one that applies to the application. Oracle9*i* introduced the notion of *bind variable peeking* to deal with this issue. Bind variable peeking means that the optimizer will "peek" at the bind variable values submitted during a "hard parse" (that is, a compilation of a query that is not found in the cursor cache) and use those values to determine whether the range is narrow or wide and, hence, determine the optimal plan. Subsequent invocations of the same cursor while the original one is still cached will get the same plan based on the assumption that the use of bind variables means that cursor sharing is desired. The use of bind peeking could result in the execution plan being different from the plan that EXPLAIN PLAN will generate if the actual values that the optimizer peeks at affect the optimizer's decisions. If the optimizer does not care about the actual bindings for a bind variable, it will not even bother peeking, but even if it does peek, it may still end up generating the same plan as EXPLAIN PLAN.

There are two cases where the optimizer would peek at the actual bindings of a bind variable and where the actual bindings therefore could make a difference for what plan would get generated.

Range predicates. Example:

```
sales_date between :1 and :2 and price > :3.
```

Equality predicates when the column has histograms. Example:

```
order_status = :4
```

assuming that `order_status` has histograms.

In contrast, a condition like `order_id = :1` will not trigger bind peeking assuming that `order_id` does not have histograms. (It may, for instance, be a primary key column, in which case histograms are not beneficial.)

Obviously, the usefulness of a plan generated using EXPLAIN PLAN is limited if it is known that bind variable peeking would be invoked for the actual execution. However, the criteria, above, may help determine whether bind variable peeking would actually be an issue for a given query. Moreover, in many cases, examining the plan generated without peeking in conjunction with the query might be sufficient to determine if it would make a difference even under the assumption that it will actually take place.

When Are Plans Different?

When comparing two plans for the same statement, it is necessary to decide the criteria for when they would be considered different. For example, when comparing an RBO plan with a CBO plan, the CBO plan will have cost estimates, whereas the entries in the cost column of the plan table for the RBO plan are all NULL. This difference is irrelevant for deciding whether the two plans should be considered the

same for performance purposes. Rather, the distinction should be made on the basis of the actual factors that affect the performance of the plan, namely join order and access paths. For the RBO, the interesting plan table columns are relatively straightforward.

When comparing two CBO plans for different versions of Oracle, there are more factors that can be included in the comparison. For example, for partitioned objects, the plan will contain information about how keys are set up to access specific partitions. That information could reasonably be included in the criteria for determining whether two plans are the same. So could the placements of predicates. Note that newer versions of Oracle tend to include more types of information in the plan table and any plan comparison between versions would by necessity have to be based only on the types of information that is present in both versions. In Appendix A, we give a sample script for comparing plans, but such a script may need to be modified over time to capture the specific information that is relevant when comparing two specific versions of Oracle.

What to Do When Plans Are Different

Let us assume that the plan diff has resulted in a number of statements that have different execution plans. The next step is to drill down on these statements in order to understand what may have caused the difference and whether the difference should be a cause for concern. First, it should be noted that even if the number of statements that have different plans is large, it does not necessarily mean that the number of underlying issues causing the differences is large. For example, the access path for a commonly used table may have switched from one particular index to another in a large number of statements. If so, the underlying issue may be the same in all those plan changes. Hence, the first thing to do if the number of plan changes is very large is to select a random subset of these changes in order to look for common patterns.

The next step is to try to estimate the performance impact of the changed plans. There is no real substitute for actual performance testing at this stage since it is often hard and time consuming to determine the impact just by looking at the plans.

If the performance impact is good overall and no major deterioration is found, this result should increase the confidence in the success of the migration. If, on the other hand, some of the new plans show unacceptable performance, a variety of steps may need to be taken depending on the exact nature of the plan changes. There are numerous techniques for query tuning in Oracle that can be applied to address underperforming queries, like creating histograms, changing the values of tuning parameters, using hints, etc. See Oracle Database Performance Tuning Guide. In addition, since the plan deterioration is in the context of a migration from the RBO to the CBO or from one version of the CBO to the next, it may be possible to use various features that could make Oracle revert to the old plan. We have already described the parameters optimizer_mode and

optimizer_features_enable in an earlier section. There is also reason to consider the use of the Plan Stability feature which we will describe next.

Using Plan Stability

Oracle's Plan Stability feature allows Oracle to capture descriptions of the execution plans and store them as plan *outlines*. An outline is implemented as a set of optimizer hints that are associated with the SQL-statement. If the use of the outline is enabled for the statement, Oracle will automatically consider the stored hints and try to generate an execution plan in accordance with those hints. See Oracle Database Performance Tuning Guide.

While there are classes of SQL-statements and features where an exact reproduction of the original execution plan is not guaranteed, Plan Stability can still be a highly useful part of the migration process as follows. Before the migration, outline capturing of execution plans should be turned on until all or most of the applications SQL-statements have been covered. If, after the migration, there are performance problems for some specific SQL-statement, the use of the stored outline for that statement can be turned on as a way of restoring the old behavior. The use of stored outlines is not always the best way of resolving a migration related performance problem since it prevents plans from adapting to changing data properties, but it adds to the arsenal of techniques that can be used to address such problems.

STATISTICS MANAGEMENT

Since the CBO is dependent on accurate optimizer statistics in order to make good decisions, it is crucial to have a process in place that recollects the statistics at regular intervals to ensure that they are representative of the data. Oracle provides an Auto Gathering feature that significantly simplifies the task of keeping the optimizer statistics current in a production environment. We also discuss some of the underlying issues that may have to be addressed during a migration or when doing plan diffs.

Different Types of Optimizer Statistics

The optimizer statistics consist of *object-level statistics* that represent properties of tables and indexes and *system statistics* that represent properties of the system, such as CPU-speed. While we recommend that system statistics be used, we will simply refer the reader to Oracle Database PL/SQL Packages and Types Reference, and Oracle Database Performance Tuning Guide for information about how to collect them.

Object-level optimizer statistics measure properties of tables and indexes that are of concern to the optimizer. For example, the size of a table or an index may affect the optimizer's decision about access path. Hence, the optimizer statistics include measures like the number of blocks and the number of rows for a table and the number of leaf blocks and distinct keys for an index. Another key statistic is the

properties of a column. Whenever a column is used in a condition in the WHERE-clause of a query, it is important for the optimizer to know how selective the condition is. This information is used to estimate how many rows from the table will match the condition, something that could be of crucial importance when determining the best join order, for instance. Normally, a column is assumed to have a uniform distribution of values between its minimum and maximum values. If so, information is kept about the minimum and maximum and the number of distinct values for the column. However, in some cases, the distribution of values is not uniform, but highly skewed. For example, a column for a status flag may take on a small number of distinct values, but 95 percent of the rows may have the value for "normal." In order to be able to determine the selectivity of a condition involving a highly skewed column, Oracle supports histograms as a representation of the column's properties. By building a histogram with up to 254 buckets representing the distribution of values of the column, the optimizer can make accurate selectivity estimates even if the data distribution is highly skewed. However, the histogram representation of the column statistics uses up more space than the regular one.

Hence, the task of managing the optimizer statistics has two dimensions. One is to make sure that all the tables and indexes have up-to-date statistics. The second is to make sure that all the columns that need histograms have them.

Statistics management is done through the *dbms_stats* package. See Oracle Database PL/SQL Packages and Types Reference. This package contains routines for gathering (or deleting) statistics at various levels of granularity. For example, statistics can be gathered for all tables and indexes in a schema; statistics can be gathered for a specific table; or histograms can be created for a specific column. In addition, the package supports exporting and importing statistics between different systems, as well as storing a copy of the old statistics before starting a new collection of statistics. The package also supports sampling the objects in order to generate the statistics based on a subset of the data, and parallel execution of the statistics collection, both of which help speed up the statistics collection significantly.

On a production system with tens of thousands of tables and hundreds of thousands of columns, it may be very hard to know what objects are in need of new statistics or what columns need histograms. Gathering statistics can require significant resources, so it should be avoided if the changes to the data of the object have been insignificant since the last time statistics were gathered. Likewise, computing histograms for columns where it is not needed is also a waste of resources. The *auto gathering* option for dbms_stats addresses these issues through the use of *monitoring* and automatic sizing histogram calculations.

Monitoring

There are two types of monitoring, *DML-monitoring* and *column-usage monitoring*. DML-monitoring means that Oracle keeps track of the approximate number of

rows that have been modified in a table since last time the optimizer statistics were gathered. This feature will help dbms_stats determine whether the optimizer statistics for a given table should be considered stale and in need of being recollected. Column usage monitoring means that Oracle marks those columns that are used in a condition in the WHERE-clause of a query. This type of monitoring is used to help dbms_stats determine whether there is any point of creating histograms for that column. Over time, precisely those columns that occur in WHERE-clauses will be marked. If a column never occurs in a WHERE-clause, there is no point in creating histograms for it no matter how skewed its data distribution is. Note that DML-monitoring has to be turned on explicitly since incurs a small amount of overhead for DML statements (typically less than 0.5 percent). The dbms_stats package supports turning on monitoring for all objects in a schema with a single command.

Auto Gathering

Assuming that monitoring is used, the following single dbms_stats command can be used to gather the relevant statistics:

```
dbms_stats.gather_schema_stats(<schema_name>, options
=> 'GATHER AUTO');
```

This command tells dbms_stats to gather optimizer statistics for those objects for which the current statistics are considered stale. It will also create histograms for columns based on two criteria: The column must have been marked as occurring in some WHERE-clause, and the data distribution must be non-uniform. The first condition is determined through column usage monitoring. The second condition is determined during the actual computation of the column statistics. Moreover, the statistics collection will use sampling and use the minimum sample size (and hence the least amount of resources) required in order to generate accurate statistics. Note that the required sample size can be different for different objects and is determined dynamically based on the data distribution for each object. Finally, Oracle will automatically determine the degree of parallelism to use during statistics collection based on the current parameters setting relating to parallel queries.

The only task that is not fully automated is the determination when it is appropriate to run the dbms_stats job to auto-gather statistics. Even though auto gathering seeks to use up as little resources as possible, it could still have a significant impact on the system if a large number of large tables need new statistics. Hence, the DBA still has the responsibility to schedule the statistics-gathering job at point in time when the load and usage pattern on the system permit it.

We also strongly recommend that the old statistics be saved away prior to collecting the new statistics. Normally, the old set of statistics is obsolete once the new set has been generated, but in the rare event that something goes wrong during the collection of the new statistics, the ability to fall back to the previous set of statistics can significantly reduce the negative impact in terms of performance and availability.

Enhancements in Oracle Database 10*g* for Statistics

Oracle Database 10*g* entirely automates the statistics gathering process, so that it requires no DBA intervention to gather statistics. The Oracle database gathers statistics using the same mechanisms described above, except that the Oracle database implicitly schedules the statistics-gathering during periods of light workloads.

CONCLUSION

This white paper provided a basic strategy for migrating from the RBO to the CBO. By migrating to the CBO, RBO-based applications will be prepared to upgrade to Oracle Database 10*g*. Moreover, RBO-based applications should gain both short-term and long-term performance and manageability gains via the CBO.

APPENDIX A

Below are some simple scripts intended to illustrate the process of performing plan diffs. First we give a script for generating plans for a set of statements using EXPLAIN PLAN rather then V$SQL_PLAN. This kind of script can be used to generate plans on the test system if the SQL-statements cannot actually be run there, e.g., due to lack of information about bind variable bindings.

In this script, we assume that the text of the SQL-statements is stored in a table called `sql_rep1` and that this table includes the columns `sql_text` and `statement_id`. We use `statement_id` as a way to match the plans with statements. We also need a plan table. In the script it is called `st_plan` and has the same definition as a standard PLAN_TABLE. In case `statement_id` is not a unique key for the SQL-statements, such as may happen if the SQL hash value is used, we give each plan a unique identifier in the remarks column. To get good performance for this operation, an index should be created on the `statement_id` column. We also use a table `failed` for those statements that give rise to errors during EXPLAIN PLAN, e.g., due to a permission problem.

```
create table failed(id varchar2(30));
```

The following PL/SQL script performs the explain plans and populates `st_plan` and `failed`.

```
declare
  id varchar2(30);
  stmnt long;
  c2 number;
  i integer := 0;
  dummy number;
  cursor c1 is select statement_id, sql_text from sql_rep1
                    order by statement_id;
begin
  open c1;
  c2 := dbms_sql.open_cursor;
  loop
    fetch c1 into id, stmnt;
    exit when c1%notfound;
    begin
      dbms_sql.parse(c2, 'explain plan set statement_id = '''
                    || id  || ''' into st_plan for ' ||
                    stmnt, dbms_sql.native);
      dummy := dbms_sql.execute(c2);
      update st_plan set remarks = to_char(i)
      where statement_id = id;
      i:= i + 1;
      commit;
```

```
    exception
      when others then insert into failed values(id);
      commit;
    end;
  end loop;
  dbms_sql.close_cursor(c2);
  close c1;
end;
/
```

The next script performs a plan diffs between two different tables with plans, plan_1 and plan_2. We assume that they have the same format as a standard PLAN_TABLE. We assume that plans with the same statement_id in the two tables correspond to the same SQL-statement, and we want to find those plans that are different for the same statement. First we create a table to store the result;

```
create table plan_diff(statement_id varchar2(30));
```

We make two assumptions about this particular plan diff. We are only interested in the basic operations, like join orders, join methods, and access paths. This particular diff does not compare PLAN_TABLE columns relating to partitioning keys, parallel query information, or predicate usage. Such information can be included in the diff by adding predicates to the query below. We also assume that there can be some statements for which there are plans in one of the tables but not in the other and that we do not want such plans to register in the diff. For this purpose, we intersect the result with the `statement_id` columns of the two tables.

```
insert into plan_diff
  select  coalesce(p1.statement_id, p2.statement_id)
  from plan_1 p1 full outer join plan_2 p2
  on
  (p1.statement_id = p2.statement_id
  and p1.id = p2.id
  and nvl(p1.operation, '0') = nvl(p2.operation, '0')
  and nvl(p1.object_name, '0') = nvl(p2.object_name, '0')
  and nvl(p1.options, '0') = nvl(p2.options, '0'))
  where (p1.id is null or p2.id is null)
intersect
  select statement_id from plan_1
intersect
  select statement_id from plan_2;
```

White Paper Title
April 2005
Author: George Lumpkin
Contributing Authors: Hakan Jakobsson, Mark Van de Wiel

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