Automatic SQL Tuning in Oracle 10g

Benoit Dageville, Dinesh Das, Karl Dias,

Khaled Yagoub, Mohamed Zait, Mohamed Ziauddin

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

{Benoit.Dageville, Dinesh.Das, Karl.Dias, Khaled.Yagoub, Mohamed.Zait, Mohamed.Ziauddin}@oracle.com

Abstract

SQL tuning is a very critical aspect of database performance tuning. It is an inherently complex activity requiring a high level of expertise in several domains: query optimization, to improve the execution plan selected by the query optimizer; access design, to identify missing access structures; and SQL design, to restructure and simplify the text of a badly written SQL statement. Furthermore, SQL tuning is a time consuming task due to the large volume and evolving nature of the SQL workload and its underlying data.

In this paper we present the new Automatic SQL Tuning feature of Oracle 10g. This technology is implemented as a core enhancement of the Oracle query optimizer and offers a comprehensive solution to the SQL tuning challenges mentioned above. Automatic SQL Tuning introduces the concept of SQL profiling to transparently improve execution plans. It also generates SQL tuning recommendations by performing cost-based access path and SQL structure "what-if" analyses.

This feature is exposed to the user through both graphical and command line interfaces. The Automatic SQL Tuning is an integral part of the Oracle's framework for self-managing databases. The superiority of this new technology is demonstrated by comparing the results of Automatic SQL Tuning to manual tuning using a real customer workload.

**Proceedings of the 30th VLDB Conference,
Toronto, Canada, 2004**

1. Introduction

Over the past decade two clear trends have emerged: (a) database systems have been deployed in new areas, such as electronic commerce, bringing a new set of database requirements, and, (b) database applications have become increasingly complex with support for very large numbers of concurrent users. As a result, the performance of database systems has become highly visible and thus critical to the success of the businesses running these applications.

One important part of database system performance tuning is the tuning of SQL statements. SQL tuning involves three basic steps:

1. Identify high load or top SQL statements that are responsible for a large share of the application workload and system resources, by looking at the past SQL execution history available in the system;
2. Attempt to find ways to improve execution plans produced by the query optimizer for these statements; and
3. Implement possible corrective actions to generate better execution plans for poorly performing SQL statements.

The three steps outlined above are repeated in that order until the overall system performance reaches a satisfactory level or no more statements can be tuned. The corrective actions include one or more of the following:

1. Enable the query optimizer to find a better plan by:
   a. Gathering or refreshing the data statistics it uses to build an execution plan. For example, by creating a histogram on a column that contains skewed data.
   b. Changing the value of configuration parameters that will affect the optimizer behavior. For

example, by changing the optimization mode for the SQL statement from "all rows" to "first <n> rows" to produce an execution plan minimizing the time to produce the first <n> rows.

c. Adding optimizer hints to the statement. For example, by using an access path hint to replace a full table scan by an index range scan.

2. Manually rewrite the SQL statement, not necessarily into a semantically equivalent form, to enable more efficient data processing. For example, by replacing UNION operator by UNION ALL.
3. Create or drop a data access structure on a table. For example, by creating an index or a materialized view.

Typically the database administrator (DBA) or an application developer performs the tuning process. However, it is often a challenging task even for a tuning expert. First, it requires a high level of expertise in several complex areas: query optimization, access design, and SQL design. Second, it is a time consuming process because each statement is unique and needs to be tuned individually. Third, it requires an intimate knowledge of the database (i.e., view definitions, indexes, table sizes, etc.) as well as the application (e.g. process flow, system load). Finally, the SQL tuning activity is a continuous task because the SQL workload and the database are always changing.

To help the DBA and the application developer overcome these challenges, several software companies have developed diagnostics tools that help identify SQL performance issues and suggest actions to fix them [3, 4]. However, these tools are not integrated with the query optimizer, the system component that is most responsible for SQL performance. Indeed, these tools interpret the optimization information outside of the database to perform the tuning, so their tuning results are less robust and limited in scope. Moreover, they cannot directly tackle the internal challenges faced by the query optimizer in producing an optimal execution plan. Finally, the recommended actions often require modification of the SQL text in the application source code, making the recommendations hard to implement by the DBA.

In Oracle 10g, the SQL tuning process has been automated by introducing a new manageability feature called **Automatic SQL Tuning**. This feature is designed to work equally well for OLTP and Data Warehouse workloads. Unlike existing tools, Automatic SQL Tuning is performed in the database server by the Oracle query optimizer itself, running in a special mode. When running in this mode, the Oracle query optimizer is referred to as the **Automatic Tuning Optimizer**.

It is important to point out that the Automatic Tuning Optimizer is a natural extension of the Oracle query optimizer. In fact, the goal for both modes of the optimizer (i.e., regular optimization mode and tuning mode) is to find the best possible execution plan for a given SQL statement. The main difference is that the Automatic Tuning Optimizer is allowed to run for a much longer period of time, generally minutes versus a sub-second during the regular optimization mode. The Automatic Tuning Optimizer takes advantage of this extra time to profile the SQL statement and validate the statistics and estimates used in the process of building an execution plan. In addition, the Automatic Tuning Optimizer can also explore execution plans that are outside the search space of the regular optimizer. This is because these execution plans are only valid if some external changes made by the DBA (e.g. create a new index) or by the application developer (e.g. rewrite the SQL statement, possibly into a semantically non-equivalent form) are assumed. The Automatic Tuning Optimizer uses this what-if capability for access path and SQL structure analysis.

Among all the above aspects, SQL profiling is probably the most novel one. The main goal of SQL profiling is to build custom information (a SQL Profile) for a given SQL statement to help the query optimizer produce a better execution plan. SQL profiles are stored persistently in the database and are transparently used every time their associated SQL statements are optimized. This new technology allows tuning SQL statements without altering their text, a key advantage for users of packaged applications. As such, SQL profiling can be considered an integral part of the optimization process. Given that it is a resource-intensive process, we believe it works best when it is limited to a subset of SQL statements, namely those that have the highest impact on the system resources or whose performance is most critical to the database application.

Except for SQL profiling, all other aspects of SQL tuning require interaction with the end user (the DBA or the application developer). As a result, the Automatic SQL Tuning feature is exposed to the end user via an advisor called the **SQL Tuning Advisor**. The SQL Tuning Advisor takes one or more SQL statements, and produces statement-specific tuning advices to help produce well-tuned execution plans. Here, the term "advisor" should not confuse the reader. It is important to remember that the SQL Tuning Advisor is neither a tuning tool nor a utility but rather an Oracle server interface that exposes a comprehensive tuning solution implemented inside the Oracle optimizer.

Finally, the Automatic SQL Tuning feature is fully integrated in the Oracle 10g manageability framework making it an end-to-end solution to the SQL tuning challenges [7].

The rest of the paper is organized as follows. In Section 2, we present the architecture of the Automatic SQL Tuning. In Section 3, we give details about SQL profiling. In Section 4, we describe the access path analysis. Section 5 details SQL structure analysis. The manageability framework is discussed in Section 6.

Section 7 contains a real case study comparing manual to automatic tuning. Related work is summarized in Section 8. Finally, we conclude the paper in Section 9.

2. Automatic SQL Tuning Architecture

The Automatic SQL Tuning is based on an extension of the Oracle query optimizer called the Automatic Tuning Optimizer. The Automatic Tuning Optimizer performs additional tasks such as SQL profiling and what-if tuning analyses while building an execution plan for a SQL statement being tuned. The result of SQL profiling and tuning analyses is a set of tuning recommendations. The tuning output is presented to the user via the SQL Tuning Advisor.

The Automatic Tuning Optimizer and the SQL Tuning Advisor constitute the Automatic SQL Tuning component in Oracle 10g. Figure 1 shows the Automatic SQL Tuning architecture and the functional relationship between its two sub-components.

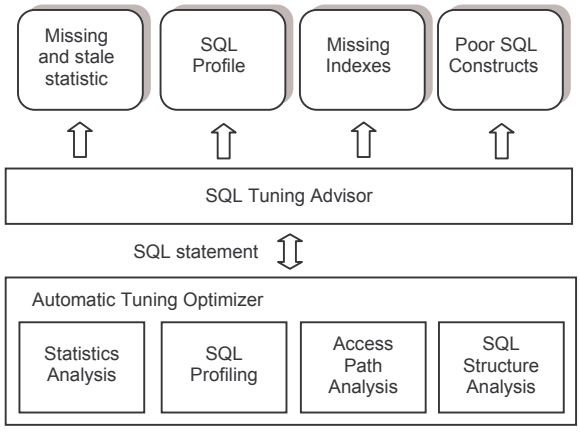

Figure 1. Automatic SQL Tuning Architecture

There are several advantages from using the Oracle query optimizer as the basis for the Automatic SQL Tuning:

- The tuning is done by the same component that is responsible for picking the execution plan, and knows best what additional information help produce a better plan.
- Future enhancements to the query optimizer are automatically taken into account in the tuning process.
- The tuning process uses the execution history of a SQL statement and customizes the optimizer settings for that SQL statement because it knows the effect of a particular setting on the query performance.

The Oracle query optimizer normally has stringent constraints on the amount of time and system resources it can use to find a good execution plan for a given SQL statement. For example, it is allotted an optimization budget in the form of a number of join permutations. Therefore, it uses a combination of cost-based and heuristics-based techniques to reduce the optimization time. Furthermore, it cannot validate the size estimates of intermediate results when standard estimation methods based on data independence assumption are known to cause large errors. Most validation techniques require running part of the query on a sample of the input data, which can be time consuming. As a consequence of these constraints, a sub-optimal plan can be generated.

In contrast, the Automatic Tuning Optimizer is given a larger time budget, e.g., several minutes, to perform necessary investigation and verification steps as part of the tuning process. It uses the extra time mainly to profile the SQL statement, verify data statistics, and perform what-if tuning analyses. The output of SQL profiling and verification gives it a much better chance to generate a well-tuned plan. The Automatic Tuning Optimizer uses dynamic sampling and partial execution (i.e. execute fragments of the SQL statement) techniques to verify its standard estimates of cardinality, cost, etc. It also uses the past execution history of the SQL statement to determine appropriate settings of the optimization parameters.

The SQL Tuning Advisor accepts a SQL statement and passes it to the Automatic Tuning Optimizer along with other input parameters, such as a time limit. The Automatic Tuning Optimizer then performs SQL profiling and what-if tuning analyses while building a query plan, which may produce one or more tuning recommendations as output.

The Automatic Tuning Optimizer results are relayed to the user via the SQL Tuning Advisor in the form of tuning advices. An advice consists of one or more recommendations, each with a rationale and an estimate of the benefit. The user is given an option to accept one or more recommendations, thus completing the tuning of the corresponding SQL statement.

3. SQL Profiling

The query optimizer relies on data and system statistics to function properly. For example, it uses the number of blocks and number of rows to estimate the cost for a full scan of a table. From these base statistics, the query optimizer derives, using probabilistic models, various data size estimates such as the table cardinalities, the join cardinalities, and the distinct cardinalities (e.g., the number of rows resulting from applying aggregate or duplicate elimination operation).

Some of the factors that lead the query optimizer to generate a sub-optimal plan are:

- Missing or stale base statistics. Missing statistics cause the optimizer to apply guesses. For example, the optimizer assumes uniform data distribution

even though the column contains skewed data when there is no histogram.

- Wrong estimation of intermediate result sizes. For example, the predicate (filter or join) is too complex, such as *(a*b)/c > 10*) to apply standard statistical methods to derive the number of rows.
- Inappropriate optimization parameter settings. For example, the user may set a parameter that tells the query optimizer that he intends to fetch the complete query result but actually fetches only few rows. In this case, the query optimizer will favor plans that return the complete result fast, while a better plan would be the one that returns first few rows fast.

To cope with the factors mentioned above, we provide a SQL profiling capability inside the optimizer, to collect auxiliary information specific to a SQL statement. A SQL Profile is built from the auxiliary information generated during 1) statistics analysis (e.g., provide missing statistics for an object), 2) estimates analysis (e.g., validation and correction of intermediate result estimates), and 3) parameters settings analysis. When it is built, the Automatic Tuning Optimizer generates a recommendation for the user to either accept or reject the SQL profile.

In the remainder of this section we provide more details about the three tasks outlined above, and then describe the content of the SQL profile.

3.1 Statistics Analysis

The goal of statistics analysis is to verify that statistics are neither missing nor stale. The query optimizer logs the types of statistics that are actually used or needed during the plan generation process, in preparation for the verification process. For example, when a SQL statement contains an equality predicate, it logs its use of the number of distinct values statistic of predicate column.

Once the statistics logging is complete, the Automatic Tuning Optimizer checks if each of these statistics is available on the associated query object (i.e. table, index or materialized view). If the statistic is available then it samples data from the corresponding query object and compares its result to the stored statistic to check its accuracy (or staleness). However, the sampling result must be sufficiently accurate before it can be used to verify the stored statistic. Iterative sampling with increasing sample size is used to meet this objective.

If a statistic is found to be missing, auxiliary information is generated to supply the missing statistic. If a statistic is available but found to be stale, auxiliary information is generated to compensate for staleness.

Note that the statistics analysis phase produces two kinds of output:

- Recommendations to gather statistics for the objects that are found to have either no statistics or stale statistics,
- Auxiliary information to supply missing statistics or correct stale statistics.

It is preferable to implement the recommendation to gather statistics and re-run the SQL Tuning Advisor. The auxiliary information is used in case the recommendation to gather statistics is not accepted by the user.

3.2 Estimates Analysis

One of the main features of a cost-based query optimizer is its ability to derive the size of intermediate results. For example, the query optimizer estimates the number of rows from applying table filters when deciding which join algorithm to pick. One of the main factors causing the optimizer to generate a sub-optimal plan is the presence of error in its estimates. Wrong estimates can be caused by a combination of the following factors: a) standard statistical methods cannot be used to derive the number of rows because the predicate (filter or join) is too complex, b) assuming uniform data distribution in the absence of a histogram when, in fact, there is skewed data, and c) data in different columns is correlated but the query optimizer is not aware of it, causing it to assume data independence.

During SQL profiling, various standard estimates are validated, and when errors are found, auxiliary information is generated to compensate for errors. The validation process may involve running part of the query on a sample of the input dataset, or on the entire input dataset when efficient access paths are available.

3.3 Parameter Settings Analysis

The Automatic Tuning Optimizer uses the past execution history of a SQL statement to determine the correct optimizer settings. For example, if the execution history shows that the output of a SQL statement is often partially consumed, the appropriate setting is to optimize it to quickly produce the first *n* rows, where *n* is derived from this execution history. This constitutes a customized parameter setting for the SQL statement. Note that the past execution statistics for SQL statements are automatically collected and stored in the Automatic Workload Repository (AWR) presented in section 6.

3.4 SQL Profile

The result of the above three analyses is stored in a SQL Profile. It is a collection of customized information for the SQL statement that is being tuned. Thus SQL Profile is to a SQL statement what statistics is to a table or index object. Once a SQL Profile is created, it is used in conjunction with the existing statistics by the Oracle query optimizer to produce a well-tuned plan for the corresponding SQL statement.

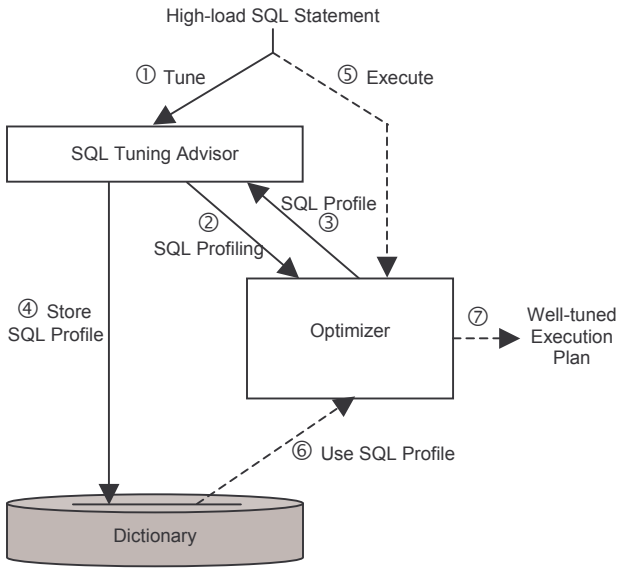

Figure 2. Creation and Use of SQL Profile

Figure 2 shows the process flow of the creation and use of a SQL Profile. The process consists of two separate phases: an Automatic SQL Tuning phase, and a regular optimization phase. In figure 2, the solid arrows are used to show the Automatic SQL Tuning process flow, while the broken arrows are used to show the regular optimization process flow.

During the Automatic SQL Tuning phase, a DBA selects a SQL statement and runs the SQL Tuning Advisor using either the Oracle Enterprise Manager (Oracle's database server GUI), or the command-line interface (step 1). The SQL Tuning Advisor invokes the Automatic Tuning Optimizer to perform SQL profiling and what-if analyses on the SQL statement (step 2). The Automatic Tuning Optimizer generates tuning recommendations possibly with a SQL Profile (step 3). Assuming a SQL Profile is built, it is stored in the data dictionary once it is accepted by the DBA (step 4).

Later, during the regular optimization phase, a user submits the same SQL statement for execution (step 5). The Oracle query optimizer finds the matching SQL Profile from the data dictionary (step 6), and uses it together with other statistics to build a well-tuned execution plan (step 7). The use of SQL Profile remains completely transparent to the user.

It is important to note that the creation and use of a SQL Profile doesn't require changes to the application source code. Therefore, SQL profiling is the only way to tune SQL statements issued from packaged applications, such as Oracle E-Business Suite, where the users have no control over the application source code.

4. Access Path Analysis

Creating suitable indexes is a well-known tuning technique that can significantly improve the performance of SQL statements because the amount of data fetched from an object is typically a small fraction of the data stored on disk. The Automatic Tuning Optimizer recommends the creation of indexes based on what-if analysis of various predicates and clauses present in the SQL statement being tuned. It recommends an index only when the query performance can be improved by a large factor because it is based on tuning of a single statement without knowing the workload characteristics.

The Automatic Tuning Optimizer determines the candidate indexes that could potentially improve the performance of the statement were they to exist. This what-if analysis can result in the discovery of several promising indexes. The following are some examples of the techniques used to identify index candidates:

- Equality predicate on a column, e.g., State='CA'. In this case, an index with *State* as a leading column will help to access only the relevant rows from the table and avoid a full scan,
- Predicates on several columns, e.g., State='CA' AND Age > 33. In this case, a multi-column index on *State* and *Age*, in that order, is considered a candidate,
- The query contains an ORDER BY on a column, and creating an index on that column could eliminate an expensive sort operation.

Once candidate indexes are identified, the next step is to verify their effectiveness. To do that, the Automatic Tuning Optimizer derives statistics for each candidate index based on the statistics of its table and relevant columns. It then optimizes the SQL statement pretending that these indexes actually exist. If the cost of a plan that uses one or more candidate indexes is cheaper by a large factor compared to the cost of best plan using no candidate indexes, then the Automatic Tuning Optimizer alerts the user that these critical indexes are missing, and recommends to add them.

Since Automatic SQL Optimizer does not perform an analysis of how its index recommendations are going to affect the entire SQL workload, it also recommends running the SQL Access Advisor [6] on the SQL statement along with a representative SQL workload. The SQL Access Advisor is a workload-based server-side tuning solution of the Oracle 10g database. The SQL Access Advisor collects tuning advice given on each

statement of a SQL workload, and consolidates them into a global advice for the entire SQL workload. The SQL Access Advisor takes into account the level of DML activity on the related objects in its global recommendations. It also recommends other types of access structures like materialized views, as well as indexes on the recommended materialized views.

5. SQL Structure Analysis

Often a SQL statement can be a high load SQL statement simply because it is badly written. This usually happens when there are different, but not semantically equivalent, ways to write a statement to produce same result. Knowing which of these alternate forms is most efficient is a difficult and daunting task for application developers since it requires both a deep knowledge about the properties of data they are querying as well as a very good understanding of the semantics and performance of SQL constructs. Besides, during the development cycle of an application, developers are generally more focused on how to write SQL statements that produce desired results than improving their performance.

It is important to note that the Oracle query optimizer performs extensive query transformations while preserving the semantics of the original query. Some of the transformations are based on heuristics (i.e. internal rules), but many others are based on cost-based selection. Examples of query transformations include subquery unnesting, materialized view (MV) rewrite, simple and complex view merging, rewrite of grouping sets into UNIONs, and other types of transformations. SQL profiling improves the outcome of this process by reducing the errors in various cost estimates, thereby improving the cost-based selection of query transformations.

However, the query optimizer applies a transformation only when the query can be rewritten into a semantically equivalent form. Semantic equivalence can be established when certain conditions are met; for example, a particular column in a table has the non-null property. However, these conditions may not exist in the database but enforced by the application. The Automatic Tuning Optimizer performs what-if analysis to recognize missed query rewrite opportunities and makes recommendations for the user to undertake.

There are various reasons related to the structure of a SQL statement that can cause poor performance. Some reasons are syntax-based, some are semantics-based, and some are purely design issues.

1. **Syntax-based constructs**: Most of these are related to how predicates are specified in a SQL statement. For example, a predicate involving a function or expression (e.g. func(col) = :bnd, col1 + col2 = :bnd) on an indexed column prevents the query optimizer from using an index as an access path. Therefore, rewriting the statement by simplifying such complex predicates can enable index access paths leading to a better execution plan.
2. **Semantic-based constructs**: A SQL construct such as UNION, when replaced by a corresponding but not semantically equivalent UNION-ALL construct can result in a significant performance improvement. However, this replacement is possible only if there is no possibility of duplicate rows (e.g., a unique constraint is maintained in the application), or duplicate rows when produced do not matter to the application. If this is the case, it is better to use UNION-ALL instead thus eliminating an expensive duplicate elimination operation from the execution plan. Another example is the use of NOT IN sub-query while a NOT EXIST sub-query could have produced same result much more efficiently.
3. **Design issues**: An accidental use of a cartesian product, for example, occurs 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 and the application developer is not very careful in checking all join conditions. Another example is the use of an outer-join instead of an inner-join when the referential integrity together with non-null property of the join key is maintained in the application.

The SQL structure what-if analysis is performed by the Automatic Tuning Optimizer to detect poor SQL constructs falling in one or more categories listed above. This analysis is performed in two steps.

In the first step, the Automatic Tuning Optimizer generates internal annotations to remember the reasons why a particular rewrite was not possible. The annotations include the necessary conditions that were not met, as well as various choices that were available at that time. For example, when the Automatic Tuning Optimizer explores the possibility of merging a view, it will check necessary conditions to see if it is logically possible to merge the view. If not possible, it will record the reasons for not merging the view. It will also record other alternatives that were available, such as pushing join predicates inside of the view to make it into a LATERAL view.

The second step of the analysis takes place after the best execution plan has been built. The Automatic Tuning Optimizer examines the annotations associated with costly operators in the execution plan. A costly operator can be defined as one whose individual cost is more than 10% of the total plan cost. Using the annotations associated to expensive plan operators, it produces appropriate recommendations. For example, if it was not possible to merge a view because of *rownum* predicate (i.e., a limit to clause) present in the view, the recommendation would be to move *rownum* predicate outside of the view. With each recommendation, a rationale is given in terms of cost improvement.

Since the implementation of SQL structure recommendations requires rewriting the problematic SQL statements, the SQL structure analysis is much more suited for SQL statements that are being developed but not yet deployed into a production system or packaged application. Another important benefit of the SQL structure recommendations is that it can help educate the developers in writing well-formed SQL.

6. Automatic SQL Tuning in the Oracle10g Self Managing Database

As shown in the previous sections, the main focus of the automatic SQL Tuning feature is to tune a SQL statement by profiling it and by recommending other tuning actions to the end user. However, the scope of SQL tuning goes far beyond tuning a single statement. Indeed, the SQL tuning task usually starts by identifying high-load SQL. High-load SQL typically represents a small subset of SQL statements (generally a small fraction) that are either consuming a large share of system resources (e.g., more than 80 percent) or account for a large portion of the time spent by a database application to perform one of its essential functions.

In Oracle10g, a substantial amount of development effort and focus has been put into making the database self-managing. Automatic SQL Tuning is an integral part of the manageability framework that was developed for this purpose. The goal is to provide an end-to-end solution to the many SQL tuning challenges faced by the database administrators and application developers. Figure 3 represents a typical illustration of the SQL tuning life cycle as it is now performed in Oracle10g. It includes four key manageability components: AWR (Automatic Workload Repository), ADDM (Automatic Database Diagnostic Monitor), STS (SQL Tuning Set), and STB (SQL Tuning Base). These components are described in detail below.

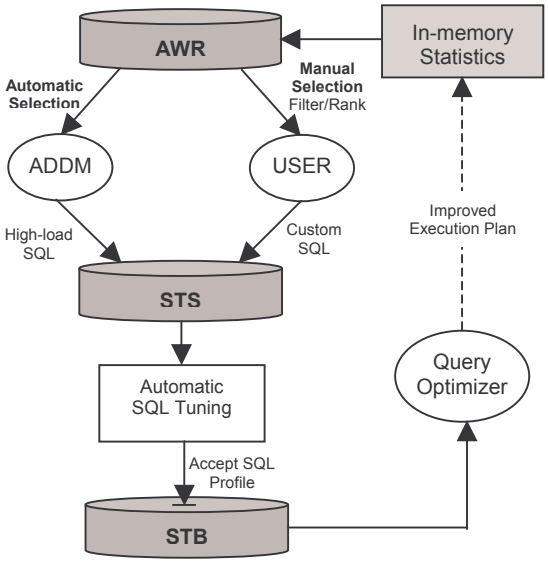

Figure 3. SQL Tuning Life Cycle in Oracle10g

The SQL tuning life cycle follows the three phases of the Oracle10g self-managing loop: **Observe, Diagnose,** and **Resolve.** Each of the components of the self-managing framework (labeled AWR, ADDM, STS and STB in Figure 3) plays a key role in one or more of these three phases.

6.1 Observe Phase

This phase is automatic and continuous in Oracle10g. It provides the data needed for analysis. To enable accurate system performance monitoring and tuning, it is imperative that the system under consideration expose relevant performance measurements. The manageability framework allows for instrumentation of the code to obtain precise timing information, and provides a lightweight comprehensive data collection mechanism to store these measurements for further online or offline analysis.

The chief component of the observe phase is the **Automatic Workload Repository** (AWR). The AWR is a persistent store of performance and system data for Oracle10g. The database collects performance data from in-memory views every hour and stores it in AWR. Each collection is referred to as a snapshot. A snapshot provides a consistent view of the system for its respective time period. For example, among other things, the AWR identifies and captures top SQL statements that are resource intensive in terms of CPU consumption, disk reads, parse calls, memory usage, etc. for each time interval.

AWR is self-managing, and based on internal measurements its overhead is less than 2 percent of the system load. AWR has standard policy for data retention but also accepts user input and, if required, proactively purges data should it encounter space pressure.

6.2 Diagnose Phase

The activities in this phase refer to the analyses of various parts of the database system using the data in AWR or in-memory performance views. Oracle10g introduces a framework for analyzing and optimizing the performance of its respective sub-components, such as the buffer cache, SQL execution, undo management, etc.

At the heart of the diagnose phase is the **Automatic Database Diagnostic Monitor** (ADDM). ADDM is a central database-wide performance diagnostic engine that optimizes for system throughput by taking a holistic view of the entire database system for a given analysis period. It runs automatically and identifies the root causes of the top performance bottlenecks and excessive resource consumption along with the exact impact on the workload in terms of time. It also provides a set of recommendations to alleviate the problems detected.

In the case of SQL statements consuming excessive resources, ADDM will recommend the invocation of the SQL Tuning Advisor for those high-load SQL statements. Besides the automatic selection performed by

ADDM, Oracle10g also provides a user driven mechanism to manually select the set of SQL statements to tune. This manual path (illustrated by downward arrows on the right side of Figure 3 exists because the user - generally the application developer or the DBA - might have to tune the response time of a subset of SQL statements involved in a critical function of the database application, even if that function accounts for a small percentage of the overall load.

The **SQL Tuning Set** (STS) feature is introduced in Oracle10g for the user to create and manage the SQL workload to tune. A SQL Tuning Set is a database object that persistently stores one or more SQL statements along with their execution statistics and execution context. The execution context stored with each SQL statement includes the parsing schema name, application module name, list of bind values, and compilation parameters. This enables the system to replicate the runtime environment under which the SQL statement was detected. The execution statistics include elapsed time, CPU time, disk reads, rows processed, statement fetches, etc.

SQL statements can be loaded into a SQL Tuning Set from different SQL sources. The SQL sources include the Automatic Workload Repository, the statement cache, and custom SQL statements supplied by the user. The capability to specify complex filters and rankings for the SQL statements are provided while loading into or reading data from the STS. For example, the user can create a STS storing the top N SQL statements issued by application module "order entry", where top N is based on the cumulative elapsed time of each statement.

Once created and populated, a SQL Tuning Set becomes the main input of the SQL Tuning Advisor.

6.3 Resolve Phase

The various advisors, after having performed their analyses, provide as output a set of recommendations that need to be implemented or applied to the database. The recommendations may be automatically applied by the database itself or be initiated manually. This is referred to as the Resolve Phase.

In the context of SQL tuning, the action part includes accepting SQL Profiles recommended by the SQL Tuning Advisor. When a SQL Profile is accepted, it is stored in the **SQL Tuning Base** (STB). The SQL Tuning Base is an extension of the Oracle dictionary that stores and manages all the tuning actions targeting specific SQL statements.

Accepting SQL profile recommendations closes an iteration of the SQL tuning loop; SQL Profiles will most likely improve the execution plan of the targeted set of SQL statements, hence reducing their overall performance impact on the system. This will be reflected in the performance measurements being collected. The next tuning cycle can then begin with a different set of high-load SQL statements. The process can be repeated several times until the desired performance level is achieved.

7. Experimental Results

The Automatic SQL Tuning feature was evaluated using a decision support workload obtained from one of Oracle's customers, a market research firm. Even though we do not demonstrate it in this paper, Automatic SQL Tuning can also tune OLTP queries equally well. In fact, we used it successfully on several queries from our internal OLTP systems. It is commonly assumed that OLTP queries are very simple with obvious execution plans, and thus do not offer many optimization opportunities. However, this is generally not true. Some OLTP queries can be very complex, joining more than 20 tables with multiple sub-queries and predicates. In this type of environment, the optimizer can fail to find optimal execution plans. Additionally, most OLTP applications run complex batch and reporting queries. Our SQL tuning methodology can be very effective in tuning OLTP queries.

For this experiment, we chose 73 decision support queries that had the highest impact on the performance of the customer's database system. As a result, the customer and an Oracle consulting team spent a significant amount of time to manually tune each of these queries. Figure 4 shows the response time of all 73 queries prior to tuning. Throughout this section, graphs show response time in ascending order (i.e. from fastest to slowest) using a logarithmic scale to improve readability. One can observe that without tuning, most queries perform very poorly. The worst response time for a query is almost 2 hours (5,751s) with an average response time of 817s and a cumulative response time (time to run the entire workload sequentially) close to 16 hours. This was unacceptable to the customer who had to resort to manual tuning of these SQL statements.

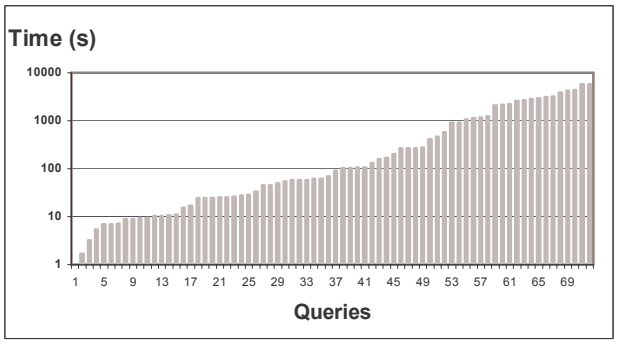

Figure 4. Response Time Without Tuning

Most statements were manually tuned using optimizer hints to improve their execution plans. In this particular instance, the Oracle query optimizer was unable to find the most optimal execution plan because these SQL statements used complex join predicates (e.g. inequality join predicates like "T1.C1 between T2.C1

and T2.C2") and had filters on highly correlated columns originating from different tables being joined (i.e. inter-table correlation). The combination of these two factors made it very hard for the Oracle query optimizer to properly estimate the cardinality of some intermediate joins. Hence, the optimizer would sometime fail to produce a good join order, leading to a sub-optimal execution plan and poor query performance.

Figure 5 below graphs the response time after performing manual tuning. As one can see, manual tuning was able to dramatically improve the response time of most queries in the set. The worst response time was reduced to 275s - instead of the initial 5,751s - with an average response time of 30s - instead of the initial 817s - and a cumulative response time of 2131s, instead of the initial 16 hours.

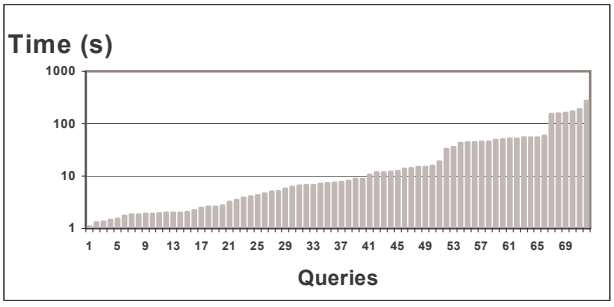

Figure 5. Response Time after Manual Tuning

The initial set of 73 queries was then stored in a SQL Tuning Set, which was then tuned using Oracle 10g Automatic SQL Tuning feature. For the purpose of this particular test, we decided to implement only SQL profile recommendations since our goal was to show how the execution plans for these statements could be improved without performing any SQL rewrite (i.e., altering SQL source code), and without modifying the underlying database schema. Figure 6 presents the new response time after the SQL Profiles recommended by the Automatic SQL Tuning were all accepted.

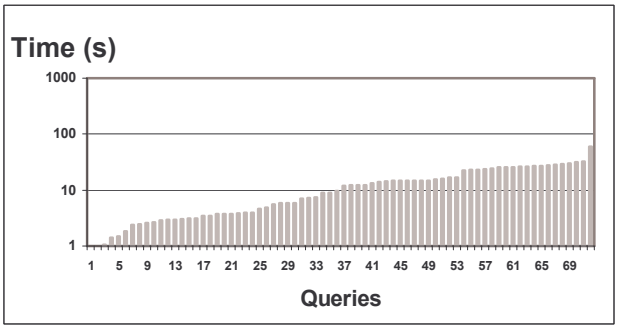

Figure 6. Response Time after Automatic Tuning

Overall, the results show dramatic improvements over manual tuning. The maximum response time was reduced from 275s to 59s. The average response time reduced from 30s to 13s. The cumulative SQL workload response time was less than 15 minutes instead of the 16 hours before tuning or the 35 minutes after manual tuning. Table 1 below summarizes these results.

	Average Response Time	Maximum Response Time	Cumulative Response Time
No Tuning	817s	5,751s	58,821s
Manual Tuning	30s	275s	2131s
Auto Tuning	13s	59s	929s

Table 1. Result Summary

These first results are very encouraging and demonstrate that SQL profiling represents a very effective way to empower the query optimizer in finding better execution plans. Overall, SQL profiling even surpassed manual tuning.

The last aspect of the benchmark is the performance of the Automatic SQL Tuning process itself. Internally, the goal of SQL profiling is to regulate its time such that, at worst, the time spent to tune a query is no more than the response time of that query before tuning. To achieve this goal, a cost-based and bottom-up tuning approach is used to determine which internal optimizer estimates are worth verifying. This, combined with the use of dynamic and iterative sampling techniques, makes Automatic SQL Tuning very efficient.

Figure 7 validates this goal. On average, the time to tune a query ranged from less than a minute to a maximum of less than two minutes. The entire workload was tuned in a little more than an hour (74 minutes) versus 16 hours to run the set of queries before tuning. This should be contrasted with the significant man-hours spent by domain experts to perform the manual tuning task, making Automatic SQL Tuning a very cost-effective solution.

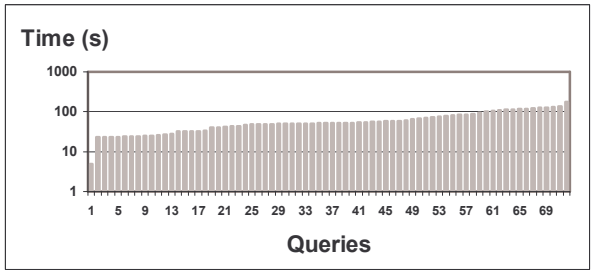

Figure 7. Automatic Tuning Time

8. Related Work

Several research groups, commercial databases and tools vendors have tried to solve the SQL tuning problem. Their solutions have concentrated on one of many areas. These include improving the optimizer itself; providing novel data statistics; rewriting the SQL statements into semantically equivalent forms; and making recommendations for new indexes to improve performance. However, none of the commercially

available query optimizers exploit a selective body of knowledge built by a learning component to influence future query plan generation. In Oracle10g, this body of knowledge is encapsulated by SQL profiles built by the Automatic Tuning Optimizer, our learning component.

The LEO (LEarning Optimizer) research project at IBM [1] [2] corrects errors in cardinality estimates made by the query optimizer by comparing them with the actual values measured at each step of the execution plan. The corrections are computed as adjustments to the query optimizer estimates and stored in dictionary tables. When a SQL statement is compiled, the query optimizer first checks whether any adjustments are available as a result of previous executions of a related query and if so, it applies them. The idea of correcting errors in optimizer estimates to produce a better execution plan is similar to SQL Profiling. However, the two approaches differ in several ways:

1. LEO detects cardinality estimate errors only in the final plan selected by the optimizer. In contrast, SQL profiling error detection is performed when the Oracle optimizer is searching for an optimal plan. As a result, SQL profiling guides the optimizer in its plan search algorithm so that the true optimal plan can be found. On the other hand, LEO tries to find an optimal plan through several iterations, each of which requires a new execution of the SQL statement by the application. Several issues with LEO's iterative error correction model are detailed in [2] and outlined here:
   a. Performance of intermediate execution plans is not guaranteed to improve because of partial error corrections. Indeed, during the learning phase, performance can even degrade making this method hard to use in production systems. By contrast, the Automatic Tuning Optimizer produces a SQL profile in a single iteration, without impacting the application.
   b. The process of converging to an optimal plan can extend over a long period of time since a single estimate error can be the source of many other estimate errors. For example, an error in the cardinality estimate of a join between tables A and B, will cascade to every join permutation that includes A and B (e.g., C, A, D, B). Hence, before finding the optimal join permutation, LEO might have to correct many join permutations, each correction requiring a full execution of the SQL statement by the application. This issue does not exist for SQL profiling since all relevant estimates are validated during the plan search process, using partial query execution techniques on data samples.
   c. Finally, there is no guarantee that LEO will be able to find the optimal plan. For example, two errors can cancel each other out, making the real error impossible to detect. Also, it may not be possible to pinpoint the source of an estimate error in a combination of predicates (e.g., C1<10 and C2>50) because the optimizer chose to evaluate them together, e.g. during a full table scan. Identifying the exact source of an error is important since it could enable a different access path (e.g., an index range scan). On the other hand, the Auto Tuning Optimizer judiciously verifies cardinality estimates during the plan search process. In the above example, it will verify in isolation the selectivity of predicates that are access path enablers.
2. Another difference between LEO's approach and our approach is the usage model. SQL Profiling can target a small subset of SQL statements, generally the ones that have the highest impact on the performance of the system. By contrast, LEO will potentially gather corrections for every statement executed in the system and what is learned could impact many other statements. This could be viewed as an advantage for LEO since it could learn even for queries executed only once (e.g. purely ad-hoc queries) while SQL profiling cannot do this. On the other hand, the corrections gathered by LEO can be overwhelming, both in terms of storage requirements and time spent in managing them. SQL Profiling maximizes the benefit/overhead ratio since it can be used in a very selective way to focus on a small subset of important statements, while not disturbing the performance of other statements. In addition, the impact of SQL profiling is easier to understand and evaluate, making this feature probably less risky to deploy in a real world production system.
3. The feedback mechanism used by LEO is defined in very general terms, simply by saying that predicate corrections are stored in the dictionary. In our opinion, this aspect is one of the most challenging. That is, how to represent, store, lookup, and manage feedback information. In LEO's approach, it is not clearly explained how cardinality corrections can be applied in cases other than for single table estimates, e.g., join, aggregate, set operations. By contrast, SQL Profiles allow correction of any type of estimate made by the Oracle optimizer during the plan search process. Also, lookup of a SQL Profile is simply done by computing a signature on the text of the SQL statement, making the feedback retrieval process very efficient.
4. A SQL Profile is a general feedback mechanism to deliver any type of information to the optimizer to influence query plan generation. For instance, in addition to correcting optimizer estimates, we use it to customize the optimization mode of a SQL

statement. As far as we know, LEO has no provision for this.

Microsoft SQL Server offers an Index Wizard [5] to provide recommendations to the DBA on the indexes that can potentially improve the query execution plans. This approach is similar to the DB2 Advisor [10], and SQL Access Advisor [6] component in Oracle 10g manageability framework. However, Index Wizard is limited to access path recommendations and cannot be used to improve the quality of execution plans, unlike what SQL Profiles can do.

There are a number of commercial tools that assist a DBA in some aspects of tuning inefficient SQL statements. None of them, however, provides a complete tuning solution, partly because it is not integrated with the Oracle database server. Quest Software's SQLab Vision [3], provides a mechanism for identifying high load SQL based on several measures of resource utilization. It also can rewrite SQL statements into semantically equivalent, but potentially more efficient, alternative forms and suggests creation of indexes to offer more efficient access path to the data. Since the product resides outside of the Oracle RDBMS, the actual benefit of these recommendations is unknown until they are actually implemented and executed by the user.

LeccoTech's SQLExpert [4] is a toolkit that scans new applications for problematic SQL statements as well as high load SQL statements in the system. It generates alternative execution plans for a SQL statement by rewriting it into all possible semantically equivalent forms. There are three problems with this approach. First, it cannot identify all forms of rewriting a SQL statement (which is normally the domain of a query optimizer). Second, equivalent forms of a SQL statement do not guarantee that the query optimizer will find an efficient execution plan if the bad plan is a result of errors in the optimizer estimates, such as cardinality of intermediate results. Third, all the alternative plans will have to be executed to actually determine which, if any, is superior to the original execution plan found by the optimizer.

9. Conclusion

In this paper, we have described the Automatic SQL Tuning feature introduced in Oracle10g. It is tightly integrated with the Oracle query optimizer, and is an integral part of the manageability framework for self-managing databases introduced in Oracle10g. The Automatic SQL Tuning is based on the Automatic Tuning Optimizer, the new generation Oracle query optimizer. The SQL Tuning Advisor tunes SQL statements and produces a set of comprehensive tuning recommendations including SQL Profiles. The user decides whether to accept the recommendations. Once a SQL Profile is created, the Oracle query optimizer will use it to generate a well-tuned plan for the corresponding SQL statement. A tuning object called the SQL Tuning Set is also introduced that enables a user to create a customized SQL workload, e.g., in order to tune it. The interface to the Automatic SQL Tuning is provided primarily through Oracle Enterprise Manager but is also accessible via a programmatic interface**.**

Many of the techniques we have described in this paper have been proposed before in different contexts [1], [2], [5], [10], [11]. But SQL Profiling is a novel technique that we have described here. Also, we have shown how these techniques have been combined together in order to offer an innovative end-to-end SQL tuning solution in Oracle 10g.

Finally, we have illustrated the feature using a real customer workload. It works equally well for OLTP and DSS workloads, because it helps the query optimizer cope with query complexity by improving its estimates. Although the feature is in its first production release, initial case studies have demonstrated the superiority of Automatic SQL Tuning over manual tuning. This position is further cemented by the fact that Automatic SQL Tuning results can scale over a large number of queries, and they can evolve over time with changes in the application workload and the underlying data. Automatic SQL Tuning is also far cheaper option than manual tuning. Together, these reasons position Automatic SQL Tuning as an effective and economical alternative to manual tuning.

References

[1] Michael Stillger, Guy M. Lohman, Volker Markl, Mokhtar Kandil: LEO – DB2's Learning Optimizer, *The VLDB Journal*, 2001.

[2] V. Markl, G.M. Lohman, V. Raman: LEO: An autonomic query optimizer for DB2, *IBM Systems Journal, Vol 42, No 1, 2003*.

[3] Quest Software, Quest Central for Oracle: SQLab Vision, http://www.quest.com, 2003.

[4] Leccotech, LECCOTECH Performance Optimization Solutions for Oracle, *White Paper*, http://www.leccotech.com/, 2003.

[5] S. Chaudhuri, V. Narasayya: An Efficient, Cost-driven Index Tuning Wizard for Microsoft SQL Server, *23rd International Conference on Very Large Data Bases*, 1997.

[6] Oracle Corporation: Performance Tuning using the SQL Access Advisor, *Oracle White Paper*, http://otn.oracle.com, 2003.

[7] Oracle Corporation: Database 10g: The Self-Managing Database, *Oracle White Paper*, http://otn.oracle.com, 2003.

[8] Oracle Corporation: The Self-Managing Database: Automatic Performance Diagnosis, *Oracle White Paper*, http://otn.oracle.com, 2003.

[9] Oracle Corporation: The Self-Managing Database: Guided Application and SQL Tuning, *Oracle White Paper*, http://otn.oracle.com, 2003.

[10] Gary Valentin, Michael Zuliani, Daniel Zilio, Guy Lohman, Alan Skelley: DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes, *16th International Conference on Data Engineering, 2000.*

[11] Hamid Pirahesh, Joseph Hellerstein, Waqar Hasan: Extensible/Rule Based Query Rewrite Optimization in Starburst, *ACM SIGMOD Conference, 1992*.