Oracle's Self-Tuning Architecture and Solutions

Benoit Dageville and Karl Dias
Oracle USA
{Benoit.Dageville,Karl.Dias}@oracle.com

Abstract

Performance tuning in modern database systems requires a lot of expertise, is very time consuming and often misdirected. Tuning attempts often lack a methodology that has a holistic view of the database. The absence of historical diagnostic information to investigate performance issues at first occurrence exacerbates the whole tuning process often requiring that problems be reproduced before they can be correctly diagnosed. Even when the problem root cause is identified, fixing it often requires a very high level of expertise that very few DBA possess. This is especially true for the inherently complex activity of SQL Tuning, requiring a high level of expertise in several domains: query optimization, access design, and SQL design.

In this paper we describe how Oracle overcomes these challenges and provides a way to perform automatic performance diagnosis and tuning. The ability to self-tune is a critical aspect towards building a self-managed database, which was one of the key objectives for the latest version of Oracle, Oracle10g, that was released in early 2004.

1 Introduction

In today's around-the-clock economy, the importance of an efficient and reliable IT infrastructure for the success of an enterprise hardly needs any explanation. As businesses increasingly rely on this infrastructure, system performance becomes more important than ever before. Businesses are building more and bigger databases, and database administrators (DBAs) are expected to take on this ever-increasing load. Hiring highly skilled administrative staff to manage such complex environments results in spiraling management costs, making self-managing technologies a must-have for modern database systems [4].

In this context, being able to effectively analyze system performance is crucial for ensuring good quality of service. Database systems traditionally expose a plethora of measurements and statistics about their operation and it can be hard to get an overall view of what is happening in the system. Identification of the root cause of a performance problem is not easy [10, 3, 2]. It is not uncommon for DBAs to spend large amounts of time and resources fixing performance *symptoms*, only to find that this has marginal effect on system performance. Lack of a holistic view of the database leads to incorrect diagnosis, misdirected tuning efforts and over-configured systems, increasing the total cost of ownership. [9, 3, 8].

Even when the proper methodology for analysis is followed, it is often found that the available data stops short of what is required to fully diagnose the root cause. Lack of adequate statistics is a very common issue

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

because collecting appropriate ones is prohibitively expensive, especially since a very broad class of statistics is required to address a very large spectrum of potential issues. Worse, to be effective, statistics collection must be continuous and enabled by default, since a performance problem can strike any time. Additionaly, statistics need to be persisted since the analysis of a performance issue is often performed long after this issue has occurred.

When appropriate statistics are not available, an option is to reproduce the problem while collecting a larger set of targeted statistics, in the hope that this would be enough to complete the performance diagnosis. In real world, this solution is rarely feasible because it requires a full-scale test system and a way to simulate/reproduce a full-scale workload. This is either impossible to do or far too expensive to be practical.

Recognizing these challenging demands, Oracle 10g introduces a sophisticated self-managing database that automatically monitors, adapts, and fixes itself. This paper provides a overview of Oracle's self-tuning architecture along with a more detailed presentation of two automatic tuning solutions: *Automatic Database Diagnostic Monitor* (ADDM) which automatically diagnoses the bottlenecks affecting the total database throughput and provides actionable recommendations to alleviate them; and the *Automatic SQL Tuning Advisor* which provides comprehensive tuning recommendations for a SQL workload that span query optimization, access path analysis and statement restructuring.

2 Self-Tuning Architecture

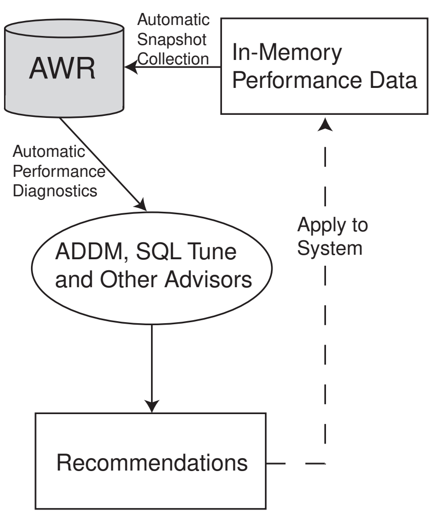

Figure 1: The Self-Managing Database Framework.

Oracle's tuning framework developed in Oracle10g for self-managing databases is centered around the three phases of the self-managing loop: *Observe, Diagnose*, and *Resolve*. This framework enables a comprehensive tuning solution by providing the necessary components. Each component provided by the framework plays a key role in one or more of these phases, and can be broadly classified into two categories: *Statistics Collection and Storage (observe)* which includes components that measure and collect interesting statistics and performance data for current as well as historical analysis and alerting; and *Advisor (diagnose and resolve)* which includes the components that carry out a targeted analysis of the data and work towards optimizing the performance for a given area.

The phases in the self-tuning loop refer to a particular tuning cycle (e.g. total database tuning cycle via ADDM, or a SQL Tuning cycle), and there could be many such tuning cycles occurring concurrently each in different phases. A tuning cycle could contain other tuning cycles. In fact the system is designed with precisely such a hierarchical model in mind; a system-wide top-down throughput based tuning methodology is used wherein ADDM acts as the central advisor that directs further tuning activity in the system by invoking other subsystem specific advisors based on top issues affecting overall throughput. Figure 1 illustrates the relationship between the Statistics Collection and Storage components and the Advisors.

Before we briefly explore each stage in the self-tuning loop, we would like to introduce the key concept of *Database Time* that has enabled us to successfully tackle inter-component database wide tuning.

2.1 Database Time

Traditionally, performance of various subsystems of the database is measured using different metrics. For example, the efficiency of the data-block buffer cache is expressed as a percentage in buffer hit-ratio; the I/O subsystem is measured using average read and write latencies. Using such disparate metrics to find the performance impact of a particular component over the total database throughput is extremely hard, if not infeasible. We addressed this issue in Oracle10g by introducing the concept of *Database Time* or simply *DbTime* in this paper, a new time based measure.

DbTime is defined as the sum of the time spent inside the database processing user requests. It is only a portion of the response time perceived by the user since it does not include time spent in the intervening layers like the network or the middle tiers. It is directly proportional to the number and duration of user requests, and can be higher or lower that the corresponding wall-clock time. It is a measurement of the total amount of work done by the database, and the rate at which the database time is consumed can be thought of as the database load average, similar to the OS load average.

DbTime serves as a common currency for the measurement of a subsystem's performance impact. For example, the performance impact of an under-sized buffer cache would be measured as the total database time spent in performing additional I/O requests that could have been avoided if the buffer cache was larger.

2.2 Observe Phase

This phase is automatic, enabled by default and continuous in Oracle10g. It's reponsibility is to collect and store an extensive set of statistics. Oracle10g has been extensively instrumented to obtain precise timing information, both CPU and wait times, for a wide range of database operations. In addition, the observe phase records samples of database sessions activity at a frequency of one every second, to allow for fine grain analysis of user activity; it collects various statistics on resource usage, both at database and OS level, to help identifying any resource bottlenecks; finally it maintains statistics for highly used database entities, like high-load SQL statements and on often accessed objects like tables and indices.

Statistics collected by the observe phase are stored in the *Automatic Workload Repository* (AWR). AWR is a persistent store of performance data for Oracle10g and can be thought of as the Oracle performance datawarehouse. Statistics in AWR are organized chronologically, using hourly delta snapshots of in-memory statistics. The AWR is self-managed; it accepts policies for data retention and proactively purges data should it encounter space pressure. The same data is also used for feedback analysis, i.e. to analyze the result of tuning actions undertaken as part of previous analysis.

2.3 Diagnose Phase

Activities in this phase refer to the analysis of various parts of the database system using data in AWR or in in-memory views. The analysis is performed by a set of *Advisors*. Oracle10g introduces many advisors, each responsible for analyzing and optimizing the performance of its respective sub-components. ADDM and the SQL Tuning Advisor are presented later in this paper; other advisors include: *Segment Advisor* that analyzes space wastage by objects due to internal and external fragmentation; *Memory Advisors* that continuously monitor the database instance and auto-tune the memory utilization between the various memory pools for shared memory and process private memory [5]; *Undo Advisor* that provides optimal sizing of the Undo space.

2.4 Resolve Phase

The various advisors, after having performed their analysis, provide as output a set of recommendations that can be implemented or applied to the database. Each recommendation is accompanied by a benefit, in DbTime units, which the workload would experience should the recommendation be applied. The recommendations may be automatically applied by the database (e.g., the memory resizing by the memory advisors) or it may be initiated manually. This constitutes the Resolve phase.

Applying recommendations to the system closes an iteration of that particular tuning loop. The influence of the recommendations on the workload will then be observed in future performance measurements. Further tuning loops may be initiated until the desired level of performance is attained.

3 ADDM

The Automatic Database Diagnostic Monitor (ADDM) in Oracle 10g automates the entire process of diagnosing performance issues and suggests relevant tuning recommendations with the primary objective of maximizing the total database throughput. This advisor is executed out-of-the-box once every hour, each time an AWR snapshot is produced. Results of these analyses are kept by default for a month making it very easy for the DBA to address past performance issues.

Automatic performance diagnosis is very challenging because modern database systems have complicated interactions between their sub-components and have the ability to work with a variety of applications. This results in a very large list of potential performance issues such an automatic analysis could identify. Also, as new database technologies and applications are introduced, and older ones are made obsolete, it is pivotal that automatic diagnostic and tuning solutions can easily be adapted to accommodate such changes.

ADDM was designed with the following objectives:

- Should posses a holistic view of the database and understand the interactions between various database components.
- Should be capable of distinguishing symptoms from the actual root-cause of performance bottlenecks.
- Should provide mechanisms to diagnose performance issues on their first occurrence.
- Should easily keep up with changing technologies.

ADDM uses DbTime to identify database components that require investigation and also to quantify performance bottlenecks. Identifying the component consuming the most database time is equivalent to finding the single database component that when tuned will provide the greatest benefit. In other words, it is looking for ways to process a given set of user requests in the least amount of database time.

3.1 DBTime-graph and ADDM Methodology

The first step in automatic performance tuning is to correctly identify the root causes of performance problems, Only then is it possible to explore effective tuning recommendations to solve or alleviate the issue. ADDM looks at the database time spent in two independent dimensions: the first dimension looks at the database time spent in various phases of processing user requests, and includes categories like 'connecting to the database', 'optimizing SQL statements', 'executing SQL statements'; the second dimenstion looks at the database time spent using or waiting for various database resources used in processing user requests, and includes both hardware resources like CPU and I/O devices, and software resources like database locks and application locks.

ADDM looks at the database time spent in each category under both these dimensions and drills down into the categories that had consumed significant database time. This two dimensional correlation gives ADDM a very good judgment in zooming in to the more significant performance issues. The drill down process can be represented using a directed-acyclic-graph as shown in Figure 2, which we call the *DBTime-graph*.

It should be noted that this DBTime-graph is not a decision tree for a rule-based diagnosis system, where a set of rules is organized in the form of a decision tree that is traversed either to find the goal given a particular set of data or to find the data given a particular goal [1]. The DBTime-graph has various properties that differentiates itself from rule-based decision trees: (a) each node in this graph looks at the amount of database time consumed

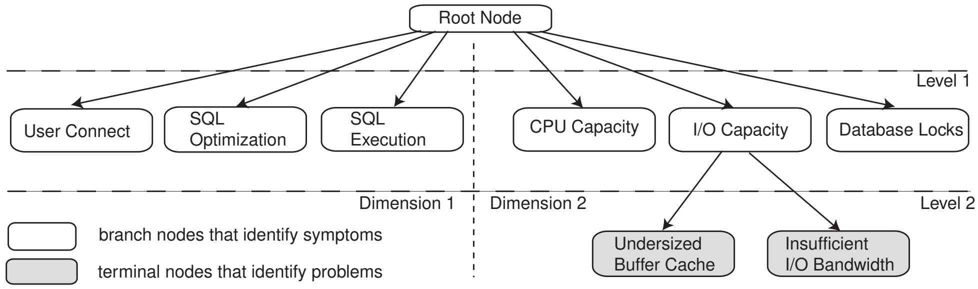

Figure 2: A Sample DBTime-Graph.

by a particular database component or resource; (b) all nodes in this graph are gauged with the same measure - DbTime; (c) all the children of a particular node are unconditionally explored whenever the database time spent in that node is significant; and (d) database time attributed to a particular node should be completely contained in the database time attributed to each of its parents. Any node that complies with all these properties can be added to the DBTime-graph making it easy to evolve with changing technologies, unlike the decision tree of a rule-based diagnosis system [1].

ADDM explores this DBTime-graph starting at the root-node and visiting all the children of a node if the database time consumed is significant. Branch nodes in this graph identify the performance impact of what is usually a symptom of a bottleneck, whereas the terminal nodes identify particular root-causes that can explain all the symptoms that were significant along the path in which the terminal node was reached. For example, in Figure 2, the branch node ”I/O Capacity” would measure database time spent in all I/O requests. Whenever significant database time was spent in I/O requests all the children of the ”I/O Capacity” node would be explored, which are the two terminal nodes in this example. The ”Undersized Buffer Cache” node would look for a particular root-cause, which is to see if the data-block buffer cache was undersized causing excessive number of I/O requests. The ”Insufficient I/O Bandwidth” node would look for hardware issues that could slow down all I/O requests.

Once a terminal node identifies a root-cause, it measures its impact in DbTime units. It then explores ways that can solve or alleviate the problem and comes up with actionable tuning recommendations based on the various workload measurements gathered. The nodes also estimate the maximum possible database time that could be saved by the suggested tuning recommendations, which need not necessarily be equal to the database time attributed to the root-cause.

It is interesting to note that ADDM doesn't traverse the entire DBTime-graph, rather it prunes the uninteresting sub-graphs. This is possible only because a node's database time is contained in the database time attributed to its parents. Consequently the cost of an ADDM analysis depends only on the number of actual performance problems that were affecting the database, and not on the actual load on the database or the number of issues that ADDM could potentially diagnose.

3.2 Workload Measurements

ADDM analysis can only be done if the appropriate data is available. Our first and most important requirement is that we collect all the data ADDM needs for each node in the DBTime-graph. ADDM needs data for the following operations: quantifying the impact in DbTime for the database components and operations; finding recommendations for alleviating root-cause problems and estimating the potential benefit in DbTime units. Our second requirement is the ”minimal intrusion principle”; it states that the act of collecting measurements for performance diagnostics should not cause a significant degradation in performance. All the data collection is done as part of the AWR snapshot mechanism described earlier. The various types of measurements include:

Database Time Measurements: The first priority in an ADDM analysis is to establish the main components that consume significant database time. This measurement is a cumulative non-decreasing function of time whose value over any time period can be got by a difference of the respective values from the start and end points. Direct measurements can only be done on database operations that usually take significant time to finish. The decision about which operations should be measured must be based on the cost of measurement (i.e. start and end a timer) and the expected length and quantity of such operations. For example, measuring the total time spent in I/O operations is reasonable while measuring the time spent in critical sections is not. Our solution to capture short duration operations is to use sampling, both frequency-based as well as time-based sampling.

Active Session History: We use regular time-based sampling to capture the activity in a system since it is not practical to collect a complete system trace of operations. This enables ADDM to narrow down root-causes of problems and give effective recommendations. We call the collection of sampled data the "Active Session History" (ASH). Each sample contains information about what the database server is doing on behalf of each connected user (a.k.a. "session") at the time of sampling. We only collect data for sessions that are actively using the database during the sample time. If a specific operation consumes significant database time during the analysis period, there is a high probability that this operation will appear in a significant number of samples in ASH. This enables ADDM to diagnose such operations even if we do not measure them directly.

System Configuration Data: We collect system configuration data related to database settings. Since database settings do not change very often we maintain a full log of changes. This data can be crucial to giving recommendations for fixing specific problems. Examples of such data are size of memory components (like buffer cache), number of CPUs used by the system, special query optimizer settings.

Simulation Data: Sometimes, estimating the impact of a specific area of the database requires a simulation of various possible alternatives. For example to find that the buffer cache is the root-cause of an I/O issue we must determine that we spent time reading data blocks that were in the buffer cache at some point in time and were replaced. In other words, we need to determine how many read I/O operations could have been saved given an infinite buffer cache. Our solution is to simulate and quantify the effect of various cache sizes.

4 SQL Tuning Advisor

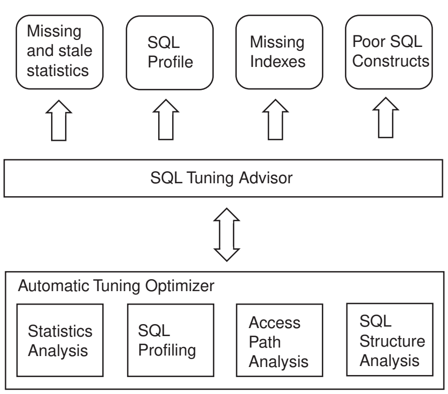

Figure 3: Automatic SQL Tuning Architecture.

The inherently complex activity of SQL Tuning requires 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. Oracle10g address this problem by introducing a new advisor, the "Automatic SQL Tuning Advisor", implemented as a core enhancement of the Oracle query optimizer. It 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. Figure 3 shows the architecture of the Automatic SQL Tuning component. We term the special extension of the query optimizer as the Automatic Tuning Optimizer.

The advantage of using the Oracle query optimizer as the basis for Automatic SQL Tuning is multifold: tuning is done by the same component that is responsible for selecting the execution plan; future enhancements to the query optimizer are automatically considered; customized optimizer settings can be used based on the execution history of the SQL statement. The SQL Tuning Advisor acts as the front-end, accepting one or more SQL statements and passing it to the Automatic Tuning Optimizer along with other input parameters, such as a time limit. It then displays the results in the form of tuning recommendations, each with a rationale and an estimate of the benefit in DBTime units

4.1 SQL Profiling

The query optimizer relies on data and system statistics to function properly and by employing probabilistic models on these base statistics the query optimizer derives various data size estimates. Some of the main reasons for a sub-optimal plan include: missing or stale base statistics, wrong estimation of intermediate result sizes, and inappropriate optimization parameter settings.

To overcome these limitatione we introduce SQL profiling, a new concept that denotes the capability within the optimizer to obtain auxiliary information specific to a SQL statement based on 1) statistics analysis, 2) estimates analysis, and 3)parameters settings. A SQL Profile object is then built from this auxiliary information.

Once the user, acting on the recommendation generated, accepts a SQL Profile, it is stored in Oracle's data dictionary. When this SQL statement (same text with potentially different host variables and/or literal values) is subsequently presented to the system the optimizer will retrieve the SQL Profile from the dictionary and use it along with other statistics to build a well-tuned execution plan. The use of a SQL Profile remains completely transparent to the user, and more importantly its creation and use don't require changes to the application source code. The following is done as part of profiling:

Statistics Analysis: The goal here is to verify whether statistics are missing or stale. The Automatic Tuning Optimizer checks each of the statistics required during plan generation. It uses sampling to check the accuracy of the stored statistic. Iterative sampling with increasing sample size is used to meet this objective to obtain greater accuracy if needed. If a statistic is found to be stale, auxiliary information is generated to compensate for staleness. If it is missing, auxiliary information is generated to supply the missing statistic.

Estimates Analysis: One of the main features of a cost-based query optimizer is its ability to derive the size of intermediate results. Errors in estimates result in sub-optimal plans and can be caused by a combination of factors like uniform distribution assumption, column correlation and an inadequate statistical model for complex predicates. During SQL profiling, various standard estimates are validated by running parts of the query on a sample of the input dataset. When errors are found, compensatory information is added to the SQL Profile.

Parameter Settings Analysis: Here the past execution history of a SQL statement is used to determine the best optimizer settings. For example, the history may show that the output of a SQL statement is often partially consumed, consequently a setting to produce the first n rows is generated, where n is derived from this execution history.

4.2 Access Path Analysis

Creating suitable indexes is a well-known tuning technique that can significantly improve the performance of SQL statements. 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. The recommendation is given only if the performance can be improved by a large factor.

4.3 SQL Structure Analysis

Often a SQL statement can be high-load simply due to the way it is written. This usually happens when there are different, but not semantically equivalent ways to write a statement to produce same result. It is important to understand that the optimizer, as part of regular plan generation process, already does semantically equivalent transformations. 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 constraints may not exist in the database but instead are enforced by the application. The Automatic Tuning Optimizer performs a cost-based what-if analysis to identify missed query rewrite opportunities and issues recommendations.

5 Conclusions

In this paper we describe the Oracle's Self Tuning Architecture and how it enables a comprehensive automatic tuning solution. We then described two automatic tuning solutions: ADDM and SQL Tuning Advisor.

ADDM seeks to improve the overall throughput of the database via a comprehensive top-down performance analysis of the system. By using database time in conjunction with the two-dimensional DBTime-graph ADDM is able to quickly isolate the root causes of performance bottlenecks and provide very specific actionable recommendations, obtained by using fine-grained sampling data. Please refer to [7] for more details.

The SQL Tuning Advisor is based on the Automatic Tuning Optimizer, an extension of the Oracle query optimizer. We have described the multipronged approach to SQL Tuning, and the unique concept of SQL Profiling that results in a SQL Profile object associated with the SQL statement and used subsequently during plan generation. For more information please refer to [6].

Acknowledgements

The authors would like to thank all the members of the Server Manageability team at Oracle for their valuable contribution in making Oracle the first self-managed database and helping us to write this paper. In addition, we would like to specially thank Uri Shaft for helping us with LaTeX.

References

[1] D. G. Benoit. Automatic Diagnosis of Performance Problems in Database Management Systems, PhD Thesis, Queen's University, Canada, 2003.

[2] K. P. Brown, M. Mehta, M.J. Carey, M. Livny: Towards Automated Performance Tuning for Complex Workloads. VLDB 1994.

[3] S. Chaudhuri and G. Weikum: Rethinking Database System Architecture: Towards a Self-tuning RISC-style Database System. VLDB 2000.

[4] S. Chaudhuri, B. Dageville, G. M. Lohman: Self-Managing Technology in Database Management Systems. VLDB 2004.

[5] B. Dageville, M. Zait: SQL Memory Management in Oracle9i. VLDB 2002, 962-973.

[6] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zait, M. Ziauddin: Automatic SQL Tuning in Oracle10g. VLDB 2004.

[7] K. Dias, M.Ramacher, U. Shaft, V.Venkataramani, G.Wood: Automatic Performance Diagnosis and Tuning in Oracle. Proceedings of the 2005 CIDR Conf.

[8] Gartner Group: Total Cost of Ownership: The Impact of System Management Tools, 1996.

[9] Hurwitz Group: Achieving Faster Time-to-Benefit and Reduced TCO with Oracle Certified Configurations, March 2002.

[10] G. Weikum, A. Mönkeberg, C. Hasse, P. Zabback: Selftuning Database Technology and Information Services: From Wishful Thinking to Viable Engineering, VLDB 2002.