Optimizer Plan Change Management: Improved Stability and Performance in Oracle 11g

Mohamed Ziauddin, Dinesh Das, Hong Su, Yali Zhu, Khaled Yagoub
Oracle Corporation
500 Oracle Parkway
Redwood Shores, CA 94065, USA
{mohamed.ziauddin, dinesh.das, hong.su, yali.zhu, khaled.yagoub}@oracle.com

ABSTRACT

Execution plans for SQL statements have a significant impact on the overall performance of database systems. New optimizer statistics, configuration parameter changes, software upgrades and hardware resource utilization are among a multitude of factors that may cause the query optimizer to generate new plans. While most of these plan changes are beneficial or benign, a few rogue plans can potentially wreak havoc on system performance or availability, affecting critical and time-sensitive business application needs. The normally desirable ability of a query optimizer to adapt to system changes may sometimes cause it to pick a sub-optimal plan compromising the stability of the system.

In this paper, we present the new SQL Plan Management feature in Oracle 11g. It provides a comprehensive solution for managing plan changes to provide stable and optimal performance for a set of SQL statements. Two of its most important goals are preventing sub-optimal plans from being executed while allowing new plans to be used if they are verifiably better than previous plans. This feature is tightly integrated with Oracle's query optimizer.

SQL Plan Management is available to users via both command-line and graphical interfaces. We describe the feature and then, using an industrial-strength application suite, present experimental results that show that SQL Plan Management provides stable and optimal performance for SQL statements with no performance regressions.

1. INTRODUCTION

The performance of SQL statements depends heavily on the optimality of execution plans generated by the query optimizer. This means that a query optimizer has the unenviable task of generating efficient execution plans for SQL statements of varied characteristics: simple vs. complex, lightweight vs. resource intensive, recursive vs. non-recursive. The query optimizers of most commercial DBMSs, such as Oracle, IBM DB2, and Microsoft SQL Server, do a very good job of producing efficient execution plans for the majority of SQL statements. In cases where they fail, SQL plan tuning techniques are employed to overcome the optimizer limitations [5].

Furthermore, query optimizers are also very good at adapting to the database and system changes and producing execution plans according to the current state. However, the plan adaptability comes at a price. If the query optimizer generates a different execution plan for a SQL statement due to a change, there is no guarantee that the new plan will execute better than the old one.

When a plan change occurs, it results in one of three possible outcomes: 1) the SQL statement continues to run with similar performance, 2) the SQL statement runs faster than before – a performance improvement, or 3) the SQL statement runs slower than before – a performance regression. The last outcome poses a serious problem, especially if the SQL statement has a stringent performance requirement.

An execution plan change can occur for a number of reasons. Some of the obvious ones are: 1) a database upgrade, and consequently a new version of the query optimizer, 2) change in the statistics used by the optimizer, 3) presence of host language variables in the SQL statement such that different amount of data are processed with different values, 4) change in the set of indexes and materialized views affecting the availability of access paths, and 5) change in the system configuration parameters affecting the efficiency (i.e., cost) of operations such as hash join, sort, and aggregation. Execution plan change is a consequence of the plan adaptability.

To address the performance risk associated with plan adaptability, a technique called plan stability is offered by most commercial DBMS vendors. Plan stability, as the name suggests, tries to stabilize the SQL performance by greatly reducing or completely disabling the ability of the query optimizer to adapt to database system changes for a set of SQL statements. The plan stability technique works by issuing directives to the query optimizer to generate a specific plan. The optimizer directives can dictate how to generate parts of a plan or a full plan.

Various commercial database vendors have offered different solutions to address the problem of performance regressions arising from execution plan changes. Oracle has offered a plan stability feature called Stored Outlines since Oracle 8i [14]. A Stored Outline contains directives for the optimizer to generate a fixed execution plan. Oracle also offers optimizer hints that can be embedded in the SQL statement text, which direct the

PVLDB '08, August 23-28, 2008, Auckland, New Zealand
Copyright 2008 VLDB Endowment, ACM 978-1-60558-306-8/08/08

optimizer to fix all or parts of an execution plan. IBM DB2 allows database users to create optimization profiles, which contain hints for the optimizer to generate specific query execution plans [4]. SQL Server offers a feature called plan forcing based on plan guides created by database users [15]. A plan guide contains optimizer hints guiding the optimizer in its plan generation process. Note that Oracle optimizer hints, DB2 optimization profiles, and SQL Server plan guides are also used in the manual tuning of poorly performing SQL statements.

But plan stability also comes at a price. It prevents potential performance gains that would have been possible without its use. For example, a fixed execution plan may no longer be optimal due to data growth. A much more serious issue occurs when some optimizer directives stop working due to system changes, and as a consequence the original plan is no longer reproducible. Consider, for example, an optimizer directive that references an index that is later dropped so that the only access path now available is a full table scan. If this directive was specified along with another directive to use nested-loops join, then the query optimizer acts upon the second directive while ignoring the first one since it is no longer applicable. This will produce a plan using nested-loops join with full table scan as the inner access path, resulting in severe performance degradation unless the scanned table is very small. This is an unfortunate outcome because it defeats the very objective of plan stability.

Plan adaptability and plan stability have their pros and cons, and the two objectives often conflict. This means that the use of one comes at the expense of the other. This is the main gist of the problem faced by database users in managing the performance of SQL statements in their applications. This problem is becoming more challenging with the increasing complexity of industrial strength business applications such as Oracle E-Business Suite, Siebel Business Analytics, and SAP Business Warehouse. The success of these applications depends to a large extent on ensuring both optimal and stable performance of their SQL workload.

In Oracle 11g Release 1, we have introduced a comprehensive solution called *SQL Plan Management*, which allows database users to maintain stable yet optimal performance for a set of SQL statements. SQL Plan Management incorporates the positive attributes of plan adaptability and plan stability, while simultaneously avoiding their shortcomings. It has two main objectives: 1) prevent performance regressions in the face of database system changes, and 2) offer performance improvements by gracefully adapting to the database system changes.

Section 2 describes SQL Plan Management concepts and architecture in detail. In section 3, we present experimental results to show how SQL Plan Management can prevent performance regressions caused by plan changes, while still allowing for performance improvements. We describe related work in section 4, and conclude in section 5.

2. SQL PLAN MANAGEMENT

SQL Plan Management (SPM) enables database users to easily manage the execution plans and performance of various SQL statements. A managed SQL statement is one for which SPM has been enabled. SPM can be configured to work automatically or it can be manually controlled either wholly or partially.

SPM helps prevent performance regressions by enabling the detection of plan changes for managed SQL statements. For this purpose, a *plan history* consisting of different execution plans generated for each managed SQL statement is maintained on disk. The part of the Oracle database dictionary that stores the plan history and other SPM related information is called the *SQL Management Base* (SMB). An enhanced version of the Oracle optimizer, called *SPM aware optimizer*, accesses, uses, and manages the information stored in the SMB. The plan history enables the SPM aware optimizer to determine if the best-cost plan it has produced using the cost-based method is a brand new plan or not. A brand new plan represents a plan change that has potential to cause performance regression. For this reason, the SPM aware optimizer does not choose a brand new best-cost plan. Instead, it chooses from a set of *accepted* plans. An accepted plan is one that has been either verified to not cause performance regression or designated to have good performance. A set of accepted plans is called a *SQL plan baseline*, which represents a subset of the plan history.

A brand new plan is added to the plan history as a non-accepted plan. Later, an SPM utility verifies its performance, and keeps it as a non-accepted plan if it will cause a performance regression, or changes it to an accepted plan if it will provide a performance improvement. The plan performance verification process ensures both plan stability and plan adaptability.

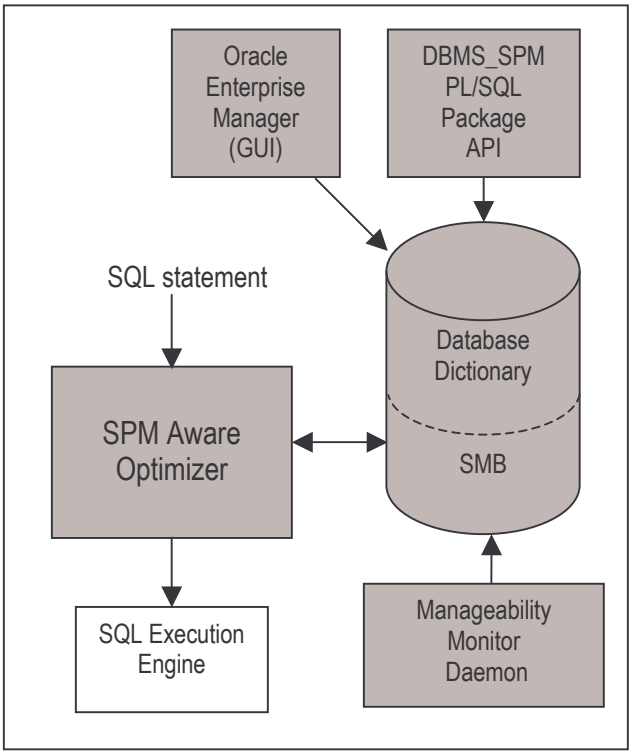

Figure 1: SPM Functional Architecture

The SPM architecture is shown in Figure 1. It consists of several functional components, a disk store, a background daemon, and user interfaces. These are listed below.

- SPM Aware Optimizer – Enhanced Oracle optimizer that uses and manages SPM information.
- SQL Management Base – a subset of the Oracle database dictionary, where plan history, SQL plan baselines, and other information of managed SQL statements is stored.
- Manageability Monitor Daemon – a background shadow process that periodically purges unused information from the SMB.
- SPM User Interfaces – a new DBMS_SPM PL/SQL (Oracle's procedural language) package, and Oracle Enterprise Manager screens (GUI).

In Figure 1, a shaded component represents an enhanced version of an existing component or a newly introduced component. For example, DBMS_SPM is a new PL/SQL package created for the SPM user interface, while SPM aware optimizer is an enhanced version of Oracle's query optimizer.

Not shown in Figure 1 but employed by the shaded components are the three main functional methods of SPM:

- SQL Plan Baseline Capture – methods of capturing and storing relevant information about execution plans of managed SQL statements on disk.
- SQL Plan Baseline Selection – a method used by the SPM aware optimizer to detect plan changes using stored plan history and to select appropriate plans for managed SQL statements.
- SQL Plan Baseline Evolution – methods of evolving SQL plan baselines by accepting plans after they are verified not to cause plan regressions.

2.1 SPM Aware Optimizer

The SPM aware optimizer is central to the SPM functional architecture. The main objectives of the SPM aware optimizer are to: 1) identify repeatable SQL statements, 2) use and maintain plan history and SQL plan baselines of managed SQL statements, 3) detect if a best-cost plan is a brand new plan (i.e., it is not found in the plan history), 4) detect if a best-cost plan represents a potential performance risk (i.e., it is in the plan history but not in the SQL plan baseline), and 5) when necessary, reproduce accepted plans and select the least costly one.

When enabled, the SPM aware optimizer determines if a SQL statement is *repeatable*. A SQL statement is repeatable if it is compiled or executed twice. Statement repeatability is used as a pre-condition for the automatic capture of SQL plan baselines, which is described in the next section. The processing logic of the SPM aware optimizer is detailed in the next three sections.

2.2 SQL Plan Baseline Capture

The SQL plan baselines for various SQL statements can be captured manually, or automatically, or both. A database user can load a set of execution plans that are known to have good performance as SQL plan baselines using DBMS_SPM package procedures. Plans for a set of SQL statements can be first captured into a SQL tuning set [5][19] before they are loaded, or plans can be directly loaded from the cursor cache (i.e., cache of recently executed plans). All manually loaded plans are assumed to be accepted plans, so they are added to the existing SQL plan baselines, or used to create new SQL plan baselines.

Automatic SQL plan baseline capture can be enabled for individual user sessions or for the entire system by setting the configuration parameter *optimizer_capture_sql_plan_baselines* to TRUE. This enables three activities by the SPM aware optimizer: 1) automatic recognition of repeatable SQL statements, 2) automatic creation of plan history and SQL plan baseline for recognized repeatable SQL statements, and 3) addition of new plans, as they are found, to the plan history of managed SQL statements. The first best-cost plan found for a newly recognized repeatable SQL statement is considered to be an accepted plan, resulting in the automatic creation of a SQL plan baseline. New best-cost plans found subsequently are considered non-accepted plans and added to the plan history.

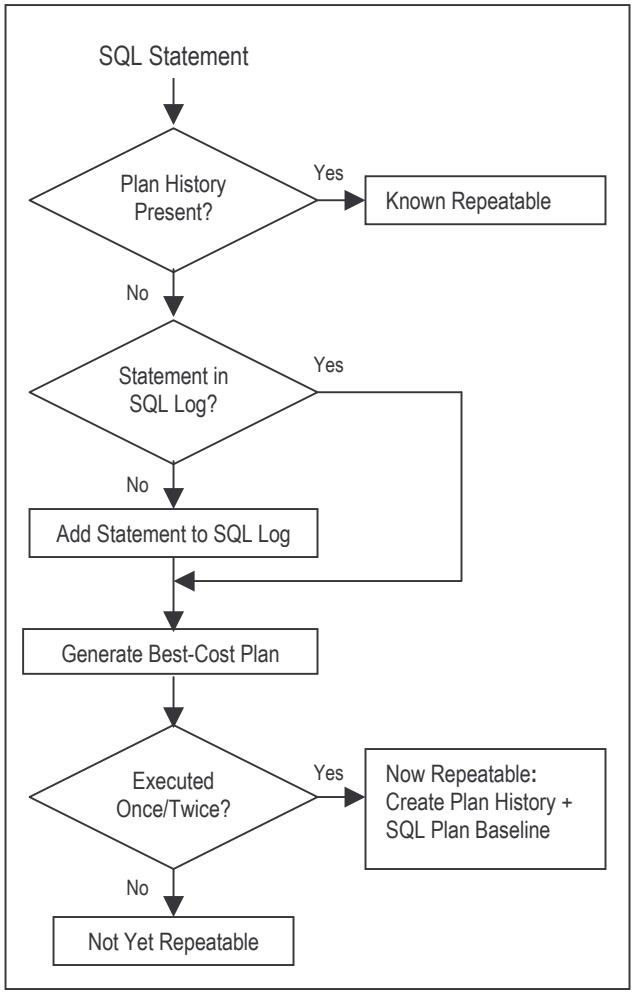

Figure 2: Automatic Capture Process

In order to recognize a repeatable SQL statement, the SPM aware optimizer stores the statement into SQL log. If the same SQL statement is compiled again, its presence in SQL log signifies it to be a repeatable statement. A statement compiled once and

executed twice is also recognized as repeatable. The steps involved in recognizing a SQL statement to be repeatable and the automatic capture of SQL plan baseline for it, are depicted in the flowchart shown in Figure 2. The last conditional in this flowchart is a check for one or two executions. A SQL plan baseline is created upon first execution of SQL statement if it was previously seen (i.e. it was found in SQL log); otherwise it is created on second execution.

The plan history of a SQL statement and, by inference, its SQL plan baseline, only contain unique plans. Thus, for example, a SQL statement may be executed several times with different sets of host language variable values or using different configuration parameter values, but there may be only a few distinct plans produced and stored in the plan history.

Note that the database user can disable the automatic capture of SQL plan baselines after the capture has been completed for a SQL workload of interest. This means that no additions will be made to the set of managed SQL statements. However, SPM will continue to manage existing SQL statements, and the SPM aware optimizer will add newly found best-cost plans to the plan history of these statements.

2.3 SQL Plan Baseline Selection

The SPM aware optimizer uses a conservative plan selection method called SQL plan baseline selection. The main objective of this method is to avoid potential performance regressions for managed SQL statements.

When a managed SQL statement is compiled, the SPM aware optimizer builds a best-cost plan using the normal cost-based method. It checks whether the best-cost plan is the same as one of the plans in the plan history or the SQL plan baseline. If no match is found, then the best-cost plan represents a brand new plan and the SPM aware optimizer adds it to the plan history as a non-accepted plan. However, if a match is found in the plan history but not in the SQL plan baseline, then it is a non-accepted plan. In either scenario, the SPM aware optimizer tries to reproduce and cost each of the accepted plans in the SQL plan baseline and selects the one with the least cost. If, however, none of the accepted plans in the SQL plan baseline is fully reproduced, the SPM aware optimizer selects the best-cost plan. This is a better choice than selecting a partially reproduced accepted plan because the latter is neither fully cost-based nor fully verified and so has a greater potential of causing unpredictable performance. The flowchart shown in Figure 3 illustrates the plan selection logic.

Note that the SPM aware optimizer does not use a SQL plan baseline at the outset. It builds a best-cost plan first because this can be a brand new plan, so it is important, at the very first opportunity, to generate and add this plan and its execution context (e.g., host language variable values) to the plan history. This approach helps in the evolution of SQL plan baselines which is described in the next section.

Also note that the overhead in reproducing accepted plans from a SQL plan baseline is not significant because each accepted plan is stored with a full set of optimizer directives. This eliminates all but one plan alternatives that are normally considered by the query optimizer.

The SQL plan baseline selection method can be turned off for individual user sessions or for the entire system by setting the configuration parameter *optimizer_use_sql_plan_baselines* to FALSE. Alternatively, it can be turned off for specific managed SQL statements by disabling their SQL plan baselines.

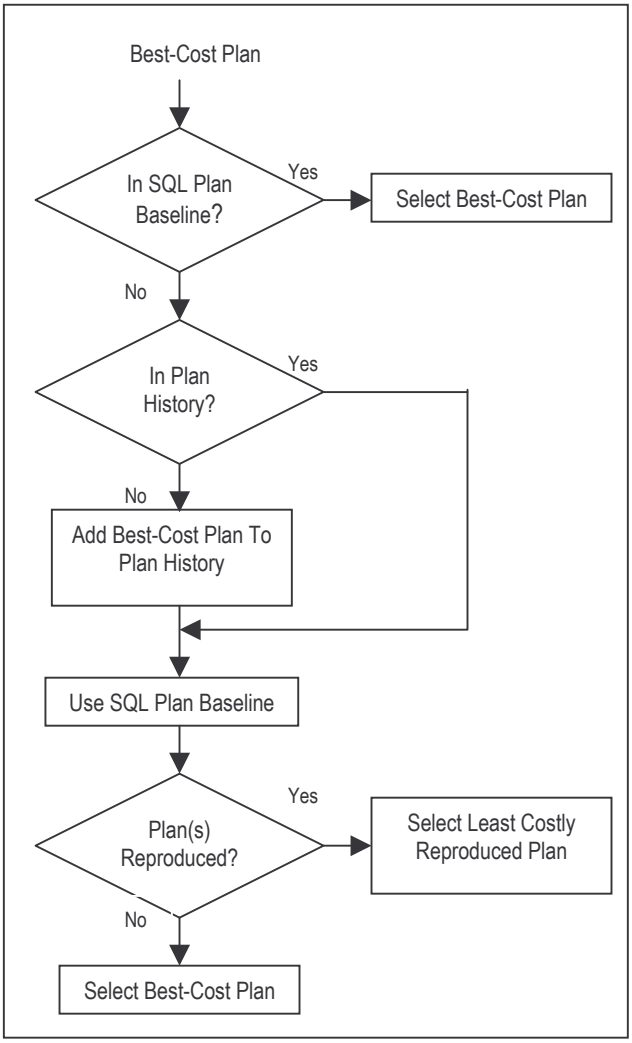

Figure 3: Plan Selection Process

2.4 SQL Plan Baseline Evolution

A SQL plan baseline is evolved when an accepted plan is added to it. This process is called SQL plan baseline evolution and its main objective is to improve the performance of managed SQL statements by accruing positive impacts of the database system changes.

The SQL plan baseline of a managed SQL statement normally starts with a single accepted plan. However, some SQL statements perform well when executed with different execution plans under different conditions. For example, a SQL statement containing host language variables tends to have several good plans. So it is sensible to evolve its SQL plan baseline from one accepted plan to several. It is also sensible to evolve a SQL plan baseline following a significant change in the database system. For example, the creation of a materialized view can dramatically improve the performance of a SQL statement when a new plan

using it as an access path is found. Also, after a database upgrade, the new optimizer may find new plans based on enhanced optimization techniques, which may result in improved performance.

There are many ways of evolving a SQL plan baseline. Different methods of SQL plan baseline evolution are listed below.

1) One or more plans can be loaded from either the cursor cache or a SQL tuning set into an existing SQL plan baseline.
2) A managed SQL statement can be tuned using the SQL tuning advisor [5]. When the SQL tuning advisor finds an execution plan with better performance, it makes a recommendation to accept a SQL profile. Accepting the SQL profile adds the tuned plan to the SQL plan baseline as an accepted plan.
3) The database user can run an SPM utility (i.e., a DBMS_SPM package procedure) to evolve one or more SQL plan baselines. This utility runs a new plan side by side with an accepted plan chosen by the SPM aware optimizer from the corresponding SQL plan baseline. The two plans are run using the same execution context (e.g., host language variable values) in which the new plan was found. The new plan is accepted when its performance is better than that of the accepted plan by an internally defined threshold. A successful verification results in the evolution of the SQL plan baseline, which now includes the new plan. Note that the existing accepted plan is not removed since it may be better under a different execution context. The processing logic of this method is illustrated in Figure 4.

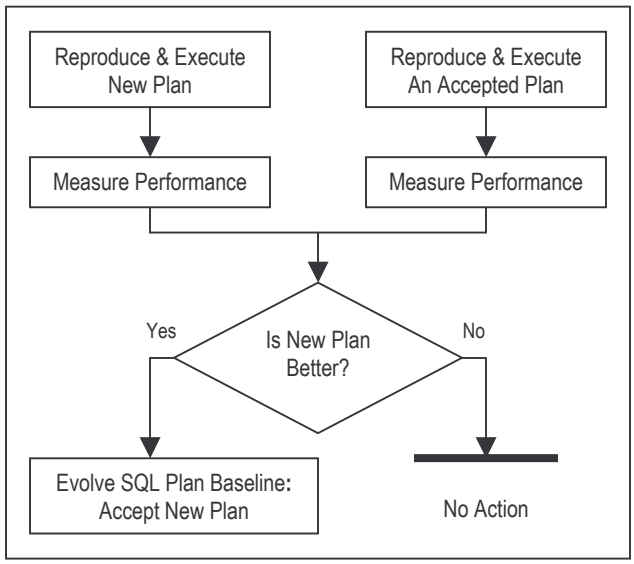

Figure 4: Plan Performance Verification Process

A setup for automatic SQL plan baseline evolution can be easily accomplished by running the SPM utility as a job which is periodically scheduled by the Oracle Scheduler.

2.5 SQL Management Base

SQL Management Base (SMB) is a part of the Oracle database dictionary where SPM information such as SQL log, plan history, and SQL plan baselines are stored. Since the SMB is a disk store for the management of SQL statements, other SQL management objects such as SQL profiles [5] are also stored here. The SMB resides in the SYSAUX tablespace separate from the SYSTEM tablespace. SYSAUX is a predefined system auxiliary tablespace.

Figure 5 shows the layout of SPM information stored in the SMB. As can be seen, SQL plan baselines are a subset of the plan histories, and plan histories are maintained for a subset of SQL statements (i.e., repeatable ones). The SQL log contains all SQL statements (repeatable as well as ad hoc) seen by the SPM aware optimizer.

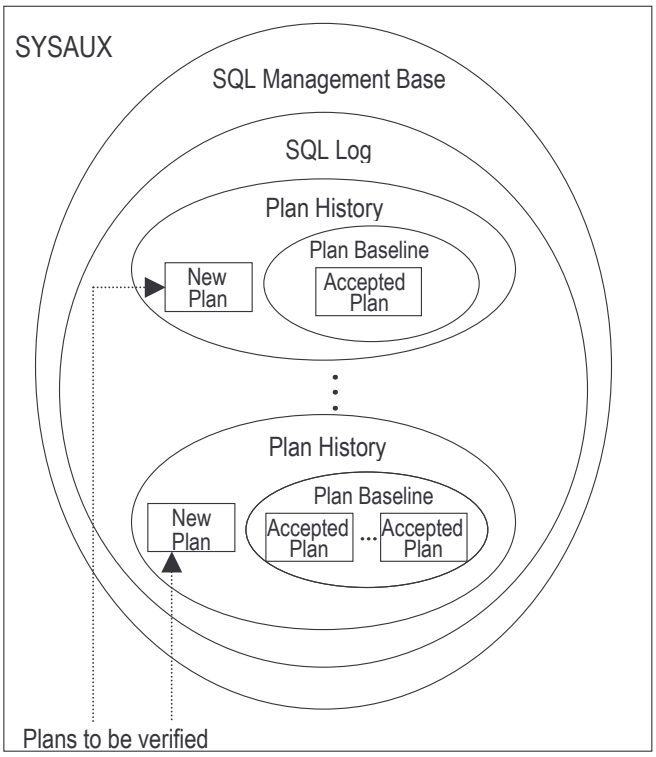

Figure 5: SPM Information Stored in SMB

Proactive management of SPM information is done through an automatically scheduled purging task. For example, information pertaining to a plan left unused for a period exceeding 53 weeks is purged from the SMB. The database user can configure the SMB and change the unused plan retention period from a default value of 53 weeks to a value between 5 weeks and 523 weeks.

For the plan purging method to work properly, the last-executed timestamp is updated in the SMB when a plan executes, provided its on-disk value is at least one week old. The staleness criterion is applied to prevent frequent disk updates and to avoid possible locking or latching contention.

The total space occupied by the SMB is measured regularly and checked against a defined limit based on the percent size of the SYSAUX tablespace. For example, when the total space occupied

by the SMB exceeds 10 percent of the SYSAUX tablespace size, a warning is generated and written to the alert log. The database user can configure the SMB and change the SMB space budget from a default value of 10 percent of the SYSAUX size to a value between 1 and 50 percent.

2.6 Manageability Monitor Daemon

Manageability Monitor daemon is a background process periodically scheduled by the Manageability Monitor (MMON). This process runs nightly and performs the purging of unused SPM information from the SMB based on a defined plan retention period. It also manages the space for SQL log by purging very old batches of SQL statements from it. The daemon process also measures the total space occupied by the SMB and generates database alerts when the occupied space exceeds a defined space budget limit.

2.7 SPM User Interfaces

A database user with appropriate database privilege can access and manipulate SPM information using either the DBMS_SPM PL/SQL package command line interface, or the Oracle Enterprise Manager graphical user interface. Either interface allows the database user similar access and manipulation of SPM information.

The SPM user interfaces allow the database user to perform the following activities:

1) View plan history and SQL plan baselines of managed SQL statements using the DBA_SQL_PLAN_BASELINES dictionary view.
2) Perform explain of plans in the plan history using DISPLAY_SQL_PLAN_BASELINE.
3) Configure the SMB using the CONFIGURE procedure.
4) Load execution plans from cursor cache or SQL tuning set into the SMB as SQL plan baselines using LOAD_PLANS_FROM_CURSOR_CACHE and LOAD_PLANS_FROM_SQLSET.
5) Modify selected attributes of plans in the plan history using ALTER_SQL_PLAN_BASELINE. For example, the user can disable all plans in a SQL plan baseline and thus disable the use of SPM for the corresponding SQL statement.
6) Drop plans from the plan history using DROP_SQL_PLAN_BASELINE.
7) Evolve SQL plan baselines of one or more managed SQL statements using EVOLVE_SQL_PLAN_BASELINE.
8) Pack plans in the plan history into a staging table using PACK_STGTAB_BASELINE and then, optionally, export the staging table to another system.
9) Unpack plans into the plan history from an imported staging table using UNPACK_STGTAB_BASELINE.

The graphical user interface consists of several Oracle Enterprise Manager screens that show SPM information as well as contain action knobs representing various SPM operations such as SMB configuration, SQL plan baseline evolution, and explain of plans in the plan history.

3. EXPERIMENTAL RESULTS

We evaluated the SPM feature using an Oracle E-Business Suite (EBS) workload. EBS is Oracle's end-to-end Enterprise Resource Planning application software that is comprised of several components (like Financials, Human Resources, etc.) and modules (like General Ledger, Payroll, etc.). The application works with the Oracle database as the underlying data repository. The functionality in EBS is used for both OLTP and decision support applications; for example, a simple order entry and a complex reporting query are both supported within the EBS.

EBS uses SQL statements to interact with the database. Both DML and queries are used for this purpose. Many of these SQL statements are very complex referencing several tables, views, subqueries, functions, operators, expressions, host language variables and aggregates. As a result, the optimizer sometimes generates a sub-optimal plan for some of these statements. Manual or automatic SQL tuning [5] can sometimes remedy this, but customers are often limited in the solutions they can adopt for such pre-packaged application suites.

For our experiment, we executed 232500 queries from the EBS. The database was 60GB in size with almost 25000 tables and 45000 indexes. The largest table contained more than 10 million rows and the smallest tables had fewer than 10 rows. There were also more than 40000 PL/SQL packages. We used queries because they are generally more complex than DML statements. (It is important to note, however, that SPM also supports DML and complex DDL statements.) Some of these queries were lightweight while others were very resource-intensive.

Our goal was to evaluate whether SPM was able to effectively prevent performance regressions due to sub-optimal plans while also improving performance by using new accepted plans. Plan changes normally happen because of changes to a system. Two of the changes that customers are most concerned about are database upgrades and optimizer statistics gathering. The former is an infrequent activity while the latter is often very frequent (daily or weekly). Both changes have the potential to severely impact system performance if plans regress. We wanted to test whether SPM provided plan stability (avoiding sub-optimal plans) and plan adaptability (using verified better plans).

We devised our experiment to measure the success of SPM in the face of both a database upgrade and statistics gathering. We executed the query workload seven times with a specific action or system change (described below) between each run. The experiments were performed using Oracle Database 10g Release 2 and Oracle Database 11g Release 1; we call them 10g and 11g, respectively, in the remainder of this section. We used the CPU time for each query execution as the performance metric. This is because we found that the CPU time was the most stable and reliable indicator of performance; other metrics like elapsed time and page reads varied widely between multiple executions of the same execution plan. Also, CPU time is a function of parse time, execution time, page reads and other indicators of performance, so it is a convenient metric to use. Thus, for our experiments, the lower the total CPU time for a query or a workload, the better the performance.

Table 1 shows the seven experimental runs and the action or change prior to each run. Each experiment was a full run of the 232500 queries in the workload, with each run executing all the

queries in the same order. The first experiment, Exp1, was performed using 10g while all subsequent runs were on 11g. Before Exp1 was run, we captured the plans for all queries in a SQL Tuning Set. We used the Exp1 run on 10g as the basis for measuring the improvement or regression for each of the other six runs.

Exp2 was executed after upgrading the database from 10g to 11g; optimizer statistics were not re-gathered nor were any other changes made to the system.

Table 1: Query Workload Runs

Prior Action	Experiment	Database	SPM used
Save 10g plans in SQL Tuning Set	Exp1	10g	No
Upgrade database from 10g to 11g	Exp2	11g	No
Load 10g plans as SQL plan baselines	Exp3	11g	Yes
Verify and evolve new plans from Exp3	Exp4	11g	Yes
Gather 11g optimizer statistics	Exp5	11g	Yes
Verify and evolve new plans from Exp5	Exp6	11g	Yes
Disable SPM	Exp7	11g	No

After Exp2, we loaded all the plans from the SQL Tuning Set (captured prior to Exp1) as SQL plan baselines for the queries. In Exp3, the SPM aware optimizer used the same plan for each query as in Exp1 while at the same time it captured new best-cost plans in the plan history to be later verified.

After Exp3, we verified all the new plans captured by the SPM aware optimizer. Some of the new plans were accepted as a result and became part of the SQL plan baselines. In other words, some queries now had two accepted plans in their SQL plan baselines. Exp4 was run at this point.

We then gathered optimizer statistics for all the objects using Oracle's recommended statistics gathering options. Statistics gathering is a frequent customer activity that sometimes causes the optimizer to generate sub-optimal plans. In our experiments, even though no DML activity was present, the 11g statistics were different and more accurate than those in 10g because of various enhancements in the statistics gathering algorithms in 11g [3]. We ran Exp5 after gathering statistics. For some queries, this caused the SPM aware optimizer to generate new plans and add them to the plan history.

Since new plans were added by Exp5, we verified all of them and some were accepted and added to the SQL plan baselines. As a result, some queries now had three accepted plans. We then ran Exp6.

Exp3 through Exp6 were run using the SPM aware optimizer. It was reasonable to ask what the workload performance would have been had we not used SPM at all. For this reason, we performed Exp7. It was run with SPM disabled but with 11g optimizer statistics as mentioned above.

Table 2 shows the number of new best-cost plans that were automatically added to the plan history by the SPM aware optimizer in Exp3 and Exp5 and the number of plans that were subsequently accepted.

Table 2: Number of best-cost plans added by SPM aware optimizer and the number subsequently accepted

Experiment	Number of plans added to plan history	Number of those plans accepted
Exp3	29648	811
Exp5	38280	2594

Table 3 shows the number of queries with one, two or three plans in the plan history and the number of queries with one, two or three plans in their SQL plan baseline. Thus, 7138 queries had three plans in their plan history but only 132 of those queries had three accepted plans. Tables 2 and 3 show that a majority of the best-cost plans found by the optimizer failed to meet the internal performance threshold required for SQL plan baseline evolution.

Table 3: Number of queries with one, two or three plans in their plan history and SQL plan baseline

Number of plans	Number of queries with those plans in plan history	Number of queries with those plans in SQL plan baseline
1	171707	229106
2	53655	3262
3	7138	132

Exp1 and Exp2 were straightforward workload executions in 10g and 11g with the same optimizer statistics. We expected that Exp2 would perform better than Exp1 because of enhancements in both the optimizer and SQL execution engine in 11g. As we will see below, there were some sub-optimal plans in Exp2. We wanted to see whether the subsequent experiments using the SPM aware optimizer prevented these plans from being used, thus boosting performance.

Figure 6 on the next page shows the cumulative CPU time for the entire query workload for each of the seven experiments. Exp1, on 10g, took 4873s. Exp2, on 11g, took 4508s. Thus, without any changes in the system apart from the database upgrade, there was an improvement of 7.5%. There are several things to note here. First, some of the improvement was undoubtedly due to an improved optimizer in 11g. Second, even for plans that did not

change between the two versions, the execution performance was often much better in 11g due to a better SQL execution engine.

The cumulative CPU time for Exp3 was 4552s. Recall that Exp3 was run using SQL plan baselines from 10g. The performance of Exp3 was worse than Exp2 because it prevented the use of any new unverified 11g plans, many of which would be better plans. This becomes clear when we look at the performance of Exp4.

Exp4 was run after SQL plan baseline evolution meaning that each query was executed using the better of the 10g and 11g plans. The performance of Exp4 (4490s) was better than that of Exp2 (4508s) because some of the cost-based plans in Exp2 were sub-optimal. Exp4 prevented these sub-optimal plans from being executed because they failed the performance requirement during the SQL plan baseline evolution process.

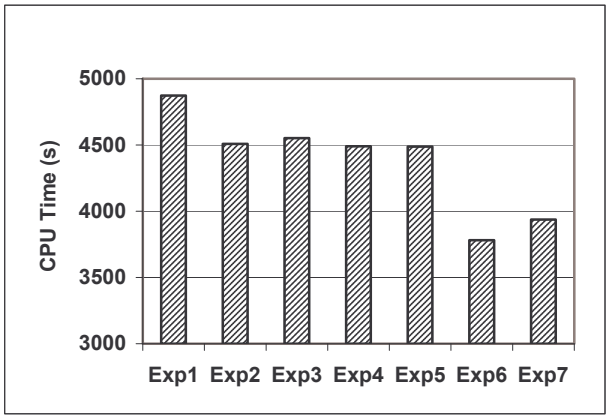

Figure 6. Cumulative CPU Time of EBS Workload

The performance of Exp5 was similar to Exp4 since only accepted plans were used in each run. However, after SQL plan baseline evolution, several new plans were accepted and, indeed, Exp6 showed that using these better plans resulted in a substantial performance gain of 22%.

Using default best-cost plans without SPM, Exp7 had a performance of 3937s. The performance of Exp6 was better than Exp7 by almost 4% because Exp6 executed only the verified optimal plans, whereas there were some sub-optimal plans in Exp7.

Figure 6 clearly shows a significant performance improvement of the entire EBS workload by using SPM. However, there was a subset of queries for which the SPM aware optimizer was able to find new best-cost plans. Therefore, this set of queries has the potential of adversely impacting the system performance. Thus, it is meaningful to look at the effect of SPM on their performance.

Figure 7 shows the cumulative CPU time for each experiment but limited to those queries where the SPM aware optimizer found new plans in Exp3 and Exp5. There were 60793 such queries with a total of 67931 new plans. Compared to 10g (i.e., Exp1), we can see that using SPM in Exp6 gave us a 44% improvement. Without SPM (Exp7), we realized an improvement of 38% over 10g. The improvement was less than in Exp6 because of the adverse effects of a few sub-optimal plans in Exp7. In other words, SPM prevented sub-optimal plans from being executed in Exp6. Thus, while there are significant enhancements in the optimizer and SQL execution engines in 11g to improve performance compared to 10g, using SPM provided an even greater improvement.

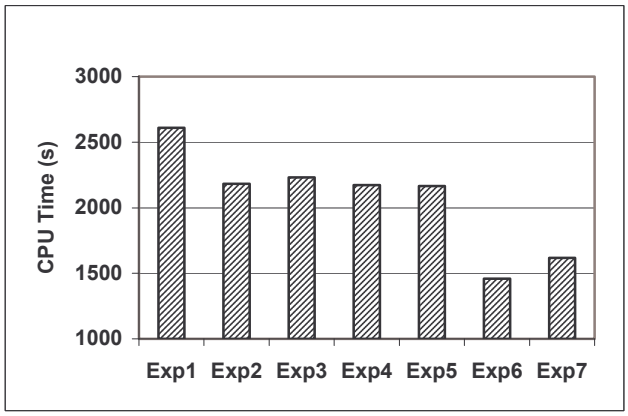

Figure 7. Cumulative CPU Time of 60793 Queries with New Plans in EBS Workload

A main objective of SPM is to prevent performance regressions of individual SQL statements. Figure 8 shows the number of improved and regressed plans for each experiment compared to Exp1. Exp2 was the 11g run with 10g statistics and we can see that there were a few plan regressions. Likewise, Exp7 was the 11g run with 11g statistics; it, too, had a few regressions from Exp1. All the other experiments used SPM and thus, had no regressions. The number of improved plans increased steadily in each experiment as each new plan that was found by the SPM aware optimizer was verified and accepted.

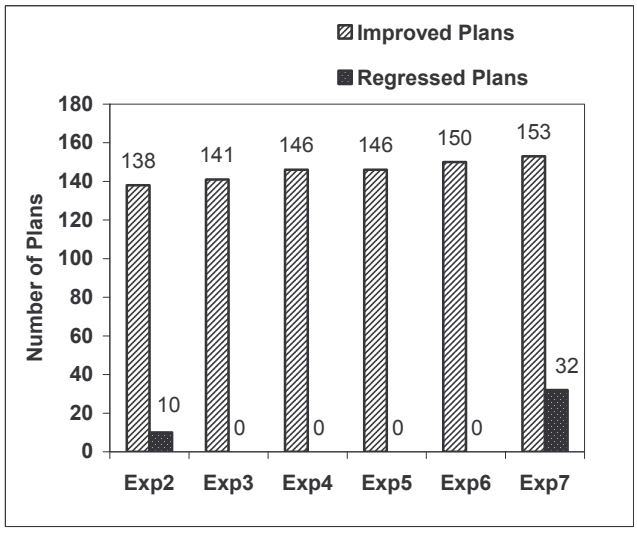

Figure 8. Number of Improved Plans and Regressed Plans in 11g Compared to 10g

Note that Exp7 (with SPM disabled) had slightly more improved plans than Exp6 (using SPM). This is because some best-cost plans failed to meet the internal performance threshold for SQL plan baseline evolution and thus were not used in Exp6. The most noteworthy observation in Figure 8 is that there were 32 sub-optimal plans in Exp7 while Exp6 had none; SPM prevented all of these plans from being used.

Our experimental results using an industrial-strength application and DBMS provide clear validation that SPM boosts performance by preventing any sub-optimal plans from being executed while simultaneously ensuring that the most optimal plans are recognized as accepted plans. In other words, SPM accomplishes its dual objectives of plan adaptability and plan stability.

4. RELATED WORK

Several commercial databases provide plan stability features. For example, SQL Server provides a series of hints [13] that can override any execution plan the query optimizer might select for a query. The hints provided include join hints, table hints and query hints. A join hint specifies that query optimizer enforce a join strategy between two tables. A table hint, for example, specifies that the query optimizer use a table scan, one or more indexes, or a locking method with the specified table. While join hints and table hints affect only part of the query (e.g., the specified joins and tables), query hints in contrast apply to the whole query. For example, a query hint can specify that nested-loops must be used for all joins in the query.

SQL Server also provides a new feature called *Plan Guides* [15]. This feature injects query hints into SQL statements in batches, stored procedures, and so forth. However, it does not require any modification to the query itself. This is very useful when the query for which the plan has to be influenced or forced originates in a non-modifiable application. When this feature is enabled, every SQL query statement or batch is first compared against the optimizer's cached plan store to check for a match. If one exists, the cached query plan is used to execute the query. If not, the query or batch is checked against the set of existing plan guides in the current database for a match. If an active plan guide exists for the statement and its context, the original matching statement is substituted with the one from the plan guide. After this is done, the query plan is compiled and cached and the query executed.

DB2 Version 9 allows users to provide *optimization profiles* to guide the optimizer to generate a desired execution plan [4][16]. An optimization profile is stored in the XML format. It specifies the target query and the guidelines for the optimizer (e.g., use a certain index for table access). All running SQL statements look for matches in active optimization profiles. A successful text match of a query specified in the profile triggers the guidelines associated with the query.

All of the above approaches constrain the whole plan or a part thereof. As noted in Section 1, the advantage of these approaches is that there are no unexpected plan changes that could possibly lead to performance regressions. The disadvantage of fixing plans is a potentially lost opportunity for better plans.

Another approach to addressing plan regression is adaptive query optimization. During plan execution, the optimizer learns which parts of the plan are potentially sub-optimal. It then uses the feedback either to re-optimize the currently running SQL statement or in the optimization of other SQL statements.

Adaptive query optimization has been actively studied in academia [6][7][8][9][18]. For example, Kabra and Dewitt [9] propose a new statistic collector operator which is placed at critical points in a query plan. This operator collects the real-time cardinality and size of the input tuples. If a significant discrepancy is found between the observed and estimated cardinality or input sizes, the plan execution is stopped and the sub-optimal portion of the plan is re-optimized.

Adaptive query optimization techniques have also been explored in commercial databases. Oracle Rdb [1][2], a commercial database system on VMS platforms, chooses access methods at run time. It runs multiple access methods competitively and then picks the best one. LEO [10][11][12][17], DB2's learning optimizer, is designed to repair incorrect statistics and cardinality estimates in a query execution plan. By monitoring previously executed queries, LEO compares the statistics and the cardinality estimates with the actual values. If significant discrepancies are found, LEO feeds the actual values back into the optimizer for re-optimization. LEO also materializes the partial results obtained in the execution of the current plan so they can be reused in the re-optimized plan.

A possible drawback of adaptive query optimization is that the re-optimization is done in an iterative manner. For example, the optimizer may choose a different access method in the first re-optimization cycle and then choose a different join order in the second re-optimization cycle. It may take several cycles (and possibly a long time) to converge to an optimal plan.

Adaptive query processing addresses the plan regression in a *reactive* manner. In other words, it tries to correct a plan when the plan is found to have degraded performance. In contrast, SPM addresses the plan regression in a *proactive* manner. It directs the optimizer to choose from a set of plans, which are known to have good performance. Plan regression is prevented in the first place.

SPM is seamlessly integrated with two important features in Oracle. The first is Automatic SQL Tuning [5]. Given a problematic query and a time limit, the SQL tuning advisor performs a comprehensive analysis to determine how to generate a better plan for the given query within the specified time limit. It analyzes three aspects that are most likely to affect the plan optimality. First, it analyzes whether the statistics that are needed by the query are missing or stale. Second, it verifies whether the optimizer's cardinality estimates of intermediate results are correct. Third, it uses the past execution history of a SQL statement to determine the correct optimizer settings. Users can implement recommendations made by the SQL tuning advisor to apply potential fixes for a query, such as re-gathering optimizer statistics. Users can also accept SQL profiles recommended by the SQL tuning advisor. When SQL profiles are created, the SQL tuning advisor evolves the SQL plan baselines of managed SQL statements by adding tuned plans to them.

SPM is also integrated with SQL Performance Analyzer (SPA) [19]. SPA enables a database user to analyze the impact of planned system changes, such as upgrades, configuration parameter changes, schema changes, or new optimizer statistics. This analysis helps the user to tune the system before the changes are implemented in a production database. SPA takes a SQL workload, executes each SQL statement before and after the

planned change, compares the results of the two executions, and reports the impact of the change on the SQL workload and on each SQL statement. For those SQL statements whose performance would degrade after the planned change, SPA recommends the creation of SQL plan baselines.

5. CONCLUSION

Poor system performance caused by sub-optimal query execution plans is a well-known problem and one that continues to vex database users. Remedial measures, like adding query directives, are often needed to fix such problems, but these techniques may be error-prone and have the possibility of introducing unknown risks in a production application. Such measures also require a high level of expertise and can be costly, requiring an unscheduled outage to apply patches.

In this paper, we have described the SQL Plan Management feature introduced in Oracle 11g. It provides a novel, unique, and comprehensive solution to managing execution plan changes. Database users can identify their critical SQL statements in advance and use SPM to manage plan changes for them. This ensures that plan changes for these statements will occur only for provably better plans, thus eliminating the risk posed by sub-optimal plans.

We used a large real-world commercial application suite to validate that SQL Plan Management provides stable and optimal performance for a set of SQL statements. Our experiments showed that not only were there zero performance regressions for the workload, there were significant performance improvements due to new and better plans.

We have also described how SQL Plan Management can be used manually or automatically via command-line or graphical user interfaces. Once enabled, SQL Plan Management provides a complete solution to managing optimizer plan changes with minimal user intervention. Thus, it is a very cost-effective means of ensuring a predictable system performance for mission-critical applications.

6. ACKNOWLEDGMENTS

Our thanks to testing team members Holly Casaletto, Vinay Ponnam, Romi Rai, Arvind Shukla, and Anand Viswanathan for conducting the performance experiments and helping us measure the quality and efficacy of SPM. We would like to also thank Pete Belknap, Benoit Dageville, Karl Dias, Hakan Jakobsson, Cetin Ozbutun, and Mohamed Zait for their contribution in developing SPM ideas and practical strategies.

7. REFERENCES

[1] Antoshenkov, G. Dynamic optimization of index scan restricted by Booleans, *ICDE*, 1996, pages 430-440.

[2] Antoshenkov, G. and Ziauddin, M. Processing and optimization in Oracle Rdb, *VLDB Journal*, 1996, volume 5, number 4, pages 229-237.

[3] Chakkappen, S., Cruanes, T., Dageville, B., Jiang, L., Shaft, U., Su, H. and Zait, M. Efficient and Scalable Statistics Gathering for Large Databases in Oracle 11g, *SIGMOD*, 2008, pages 1053-1063.

[4] Chen, K. Influence query optimization with optimization profiles and statistical views in DB2 9, 2006, http://www.ibm.com/developerworks/db2/library/techarticle/dm-0612chen/index.html

[5] Dageville, B., Das, D., Dias, K., Yagoub, K., Zaït, M. and Ziauddin, M. Automatic SQL tuning in Oracle 10g, *VLDB*, 2004, pages 1098-1109.

[6] Graefe, G. and Cole, R. Optimization of dynamic query evaluation plans, *SIGMOD*, 1994, pages 150-160.

[7] Ioannidis, Y., Ng, R. T., Shim, K. and Sellis, T. Parametric query optimization. *VLDB*, 1992, pages 103 – 114.

[8] Ives, Z. Efficient query processing for data integration, Ph.D. thesis, University of Washington, 2002.

[9] Kabra, N. and Dewitt, D. Efficient mid-query reoptimization of sub-optimal query execution plans, *SIGMOD*, 1998, pages 106 - 117.

[10] Markl, V., Lohman, G. M. and Raman, V. LEO: An autonomic query optimizer for DB2, *IBM Systems Journal*, volume 42, number 1, 2003, pages 98 –106.

[11] Markl, V. and Lohman, G. M. Learning table access cardinalities with LEO, *SIGMOD*, 2002, pages 613.

[12] Markl, V., Raman, V., Simmen, D. E., Lohman, G. M. and Pirahesh, H. Robust query processing through progressive optimization, *SIGMOD*, 2004, pages 659-670.

[13] Microsoft, Hints (Transact-SQL), SQL Server 2005 Books Online, 2007, http://msdn2.microsoft.com/en-us/library/ms187713.aspx

[14] Oracle, Using Plan Stability, Oracle Performance Tuning Guide, Oracle Database, 11g Release 1 (11.1) Documentation, Chapter 18, 2007.

[15] Patel, B. A., Forcing query plans, *Microsoft TechNet*, 2005, http://www.microsoft.com/technet/prodtechnol/sql/2005/frcqupln.mspx

[16] Qiao, L., Soetarman, B., Fuh, G., Pannu, A., Cui, B., Beavin T. and Kyu, W. A framework for enforcing application policies in database systems, *SIGMOD*, 2007, pages 981 – 992

[17] Stillger, M., Lohman, G., Markl, V. and Kandil, M. LEO – DB2's learning optimizer, *VLDB*, 2001, pages 19 –28

[18] Urhan, T., Franklin, M.J. and Amsaleg, L. Cost-based query scrambling for initial delays, *SIGMOD*, 1998, pages 130 - 141.

[19] Yagoub, K., Belknap, P., Dageville, B., Dias, K., Joshi, S. and Yu, H. Oracle's SQL performance analyzer. *IEEE Data Engineering Bulletin*, 2008, volume 31, number 1.