Closing The Query Processing Loop in Oracle 11g*

Allison W. Lee, Mohamed Zait
Oracle USA
400 Oracle Parkway
Redwood Shores,
CA 94065, USA
E-mail: {allison.waingold,
mohamed.zait}@oracle.com

ABSTRACT

The role of a query optimizer in a database system is to find the best execution plan for a given SQL statement based on statistics about the objects related to the tables referenced in the statement. These statistics include the tables themselves, their indexes, and other derived objects. Statistics include the number of rows, space utilization on disk, distribution of column values, etc. Optimization also relies on system statistics, such as the I/O bandwidth of the storage sub-system. All of this information is fed into a cost model. The cost model is used to compute the cost whenever the query optimizer needs to make a decision for an access path, join method, join order, or query transformation. The optimizer picks the alternative that yields the lowest cost.

The quality of the final execution plan depends primarily on the quality of the information fed into the cost model as well as the cost model itself. In this paper, we discuss two of the problems that affect the quality of execution plans generated by the query optimizer: the cardinality of intermediate results and host variable values. We will give details of the solutions we introduced in Oracle 11g. Our approach establishes a bridge from the SQL execution engine to the SQL compiler. The bridge brings valuable information to help the query optimizer assess the impact of its decisions and make better decisions for future executions of the SQL statement. We illustrate the merits of our solutions based on experiments using the Oracle E-Business Suite workload.

1. INTRODUCTION

Database query processing refers to the process inside a database management system (DBMS) that is responsible for compiling and executing SQL statements. The compilation process (performed by the SQL Compiler) takes as input a SQL statement text (with optional host variables) and produces an execution plan. The execution process (performed by the Execution Engine) takes as input the execution plan and returns the result of the execution. An execution plan contains the detailed steps necessary to execute the SQL statement. These steps are expressed as a set of database operators that consumes and produces rows. The processing order and implementation of the operators are decided by the query optimizer, using a combination of query transformations and physical optimization techniques.

The execution plan generated for the SQL statement is just one of the many alternative execution plans considered by the query optimizer. The query optimizer selects the execution plan with the lowest cost. Cost is a proxy for performance; the lower the cost, the better the performance (e.g. response time) of the query is expected to be. The cost model used by the query optimizer accounts for the IO, CPU, and network utilization during query execution. The cost model relies on object statistics (e.g. number of rows, number of blocks, and distribution of column values) and system statistics (e.g. IO bandwidth of the storage subsystem).

Figure 1 illustrates the lifecycle of a SQL statement inside the SQL compiler and the execution engine. A SQL statement goes through the Parser, Semantic Analysis (SA), and Type-Check (TC) first before reaching the optimizer. The Oracle optimizer performs a combination of logical and physical optimization techniques [1]. The Query Transformer (QT) is responsible for selecting the best combination of transformations (e.g. subquery unnesting and view merging) while the Plan Generator (PG) is responsible for selecting access paths, join methods, and join orders. The QT calls the PG for every candidate set of transformations and retains the one that yields the lowest cost. The PG calls the Cost Estimator (CE) for every alternative access path, join method, and join order and keeps the one that has the lowest cost. The Code Generator (CG) stores the optimizer decisions into a structure called a *cursor*. All cursors are stored in a shared memory area of the database server called the Cursor Cache (CC). The goal of caching cursors in the cursor cache is to avoid compiling the same SQL statement every time it is executed, i.e. subsequent executions of the same statement will use the cached cursor instead of going through the SQL compiler. The Dictionary contains the database metadata (definitions of tables, indexes, views, constraints, etc) as well as object and system statistics. When processing a SQL statement, the SQL compiler components look in the dictionary for information about the objects referenced in the statement, e.g. the optimizer reads the statistics about a column referenced in the WHERE clause. At

Patents have been filed for the techniques described in this paper.

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

run-time the cursor corresponding to a SQL statement is identified based on several criteria, among which is the SQL text, the compilation environment, and authentication rules. If a matching cursor is found then it is used to execute the statement, otherwise the SQL compiler builds a new one. Several cursors may exist for the same SQL text, e.g. if the same SQL text is submitted by two users that have different authentication rules. All the factors that affect the execution plan, such as whether a certain optimization is enabled by the user running the SQL statement, are used in the algorithm used to match a cursor from the CC.

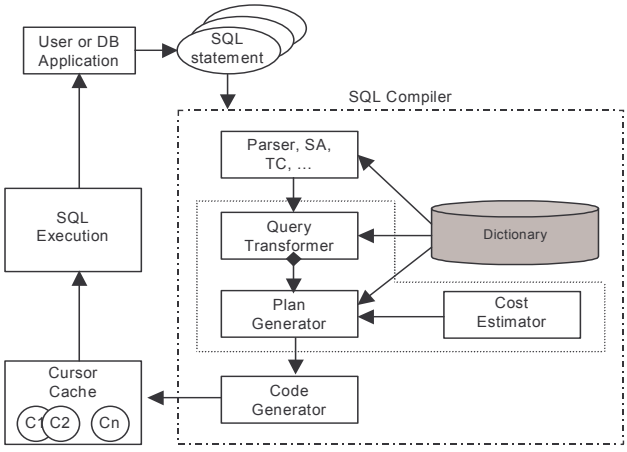

Figure 1: Architecture of the Oracle SQL Engine

The quality of the final execution plan produced by the query optimizer depends on the quality of the information used by the cost model and the cost model itself. In the remainder of this section we discuss two of the problems that affect the quality of executions plans: cardinality (number of rows) of intermediate results and host variables.

1.1 Cardinality Estimation

The cardinality of tables involved in a SQL statement directly affects the cost computed by the cost estimator. All database systems provide a special purpose utility to gather statistics on database objects. The utility can be called directly by a database administrator (DBA) or is automatically called from an automatic process maintained by the database system [3]. Since the data is constantly changing, the DBA or the automatic statistics maintenance process periodically refreshes the statistics whenever they are determined to be stale, e.g. when update statements touch more than 10% of the rows in a table. In addition to using the statistics on base tables, the optimizer derives statistics for intermediate results. An intermediate result is the output of a database operator such as a scan, join, union, etc. For several reasons, statistics on intermediate results are often inaccurate, leading to poor choices by the optimizer. The consequences are execution plans that perform very badly and negatively affect the customer's business processes. For example, consider the SQL statement in Q1. It retrieves the product names and translated US English product names for bulk purchases of an item with a specified price.

Q1

```
SELECT d.translated_name,
       p.product_name
FROM order_items oi, products p,
product_descriptions pd
WHERE oi.unit_price = 15
  AND oi.quantity > 1
  AND p.product_id = oi.product_id
  AND pd.product_id = p.product_id
  AND pd.language_id = 'US'
```

The corresponding execution plan is shown in Figure 2 (a). Note that `products` is a view that joins `product_information` (`PI`) and `product_descriptions` (`PD`). The optimizer selected nested-loops to join `order_items` (`OE`) and `product_information` (`PI`) tables. Prior to making this decision the optimizer has to estimate the number of rows generated by the table scan node (`TS`) on table `order_items` (`OE`) taking into consideration the filters [`oi.unit_price = 15 AND oi.quantity > 1`]. The estimated number of rows comes much lower than the actual number, causing the nested-loops join cost to be under-estimated. Figure 2 (b) shows the alternative plan that the optimizer would produce if the actual number of rows were known during optimization; a hash-based join would have been selected instead of the nested-loops join. Boxes are used to represent base tables (`PD` for `product_descriptions`, etc.) and circles are used to represent database operators (`TS` for Table Scan, `IS` for index scan, `HJ` for hash-join, `NL` for nested-loops join).

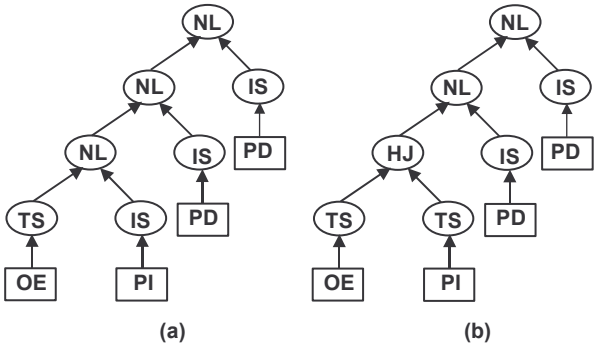

Figure 2: Execution Plans for Q1

This problem has been the subject of several research papers. Database system vendors also introduced technologies to tackle this problem in their most recent software releases [3]. In Oracle 10g, the concept of SQL profiling was introduced as part of the Automatic SQL Tuning feature. This and other such enhancements will be explained and compared in Section 4.

1.2 Host Variables

The cursor cache is most useful when SQL statements contain *host variables*. They are referred to as *bind variables* in Oracle DBMS, while other DBMS use the term *host variables*. We will use the term bind variables in the rest of this paper. These bind variables are declared and initialized in a program written using a host programming language (e.g. PL/SQL, C, C++) then

embedded inside SQL statements submitted from that program. The cursor corresponding to a SQL statement is re-used when different values of the bind variables are specified. For example, consider the statement in Q2. It retrieves the earliest start time (START_TIME), latest end time (END_TIME), and the count of all employees for a given job title (JOB_ID). The cursor built for this statement will be re-used for all executions even when different values are used for bind variable :JOB_V.

Q2

```
SELECT count(distinct e.employee_id),
       min(j.start_date),
       max(j.end_date)
FROM employees e, job_history jh
WHERE e.job_id = :job_v
  AND e.employee_id = jh.employee_id
```

When optimizing SQL statements containing bind variables, the Oracle query optimizer *peeks* at the bind variable values in order to compute cost as well as cardinality of derived tables. Bind variable peeking helps the optimizer compute cost and cardinality more accurately, especially in the presence of data skew or correlation. However, in releases prior to Oracle 11g, the plan generated based on the bind variable values submitted with the first execution is re-used for subsequent executions of the same SQL statement.

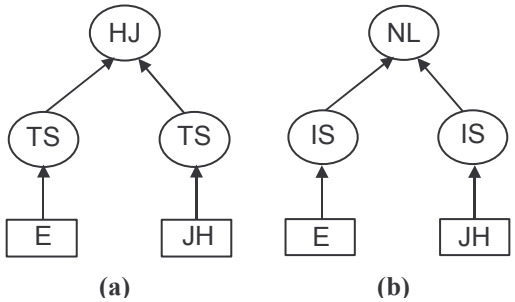

Figure 3: Executions Plans for Q2

The problem with this approach is that a plan that is optimal when the bind variable value is 'Clerk' may be sub-optimal when the bind variable value is 'VP'. The selectivity[1] of the predicate e.job_id = :job_v in the above example when the bind variable value is 'Clerk' is very different from the selectivity when the bind value is 'VP', e.g. *0.1* versus *0.001*. Assuming an index exists on column job_id, the optimizer would have selected a full table scan when optimizing with the bind value 'Clerk' and an index range scan when optimizing with the value 'VP'. The join method would also have been different, e.g. nested-loops join would be more appropriate than hash-join if the variable value is 'VP' because fewer rows would qualify from the employees table. Figure 3 shows the two possible execution plans depending on the variable value. The optimizer will generate the execution plan in Figure 3 (a) if it uses the bind variable value 'Clerk', while it will generate the execution plan in Figure 3 (b) if it uses value 'VP'.

To workaround this problem database developers have to decide between two choices where there is no clear winner:

1. Getting an execution plan that performs well for all variable values, but cursors are not shared leading to high compilation overhead and shared memory usage.
2. Avoid the compilation overhead and reduce shared memory utilization, but at the expense of getting a plan that works for a limited set of the variable values.

To this end some database vendors provide knobs for database developers to explicitly state to the system the desired behavior (to share or not to share cursors, to peek or not to peek bind variable values) or the user has to decide whether to write application code using bind variables or literals in the first place.

In Oracle 11g, we enhanced the SQL Engine architecture to let the optimizer know how accurate the decisions made during compilation turned out to be during execution. We will illustrate changes we made to the SQL Engine, and demonstrate how these changes improve the performance of SQL statements from the Oracle E-Business suite. Figure 4 shows the new design of the SQL Engine. The main difference with the older design is the addition of two new steps:

1. Update the corresponding cursor with information collected during or at the end of a statement execution.
2. Use the information reported from execution during query optimization.

These two steps essentially establish a feedback loop from the SQL execution component to the SQL compiler component. The details of these and other changes to the SQL Engine are given in the following section.

The rest of this paper is organized as follows. In Section 2, we present enhancements made to the SQL Engine in Oracle 11g. Section 3 contains the result of our performance experiments. Related work is summarized in Section 4. Finally, we conclude the paper in Section 5.

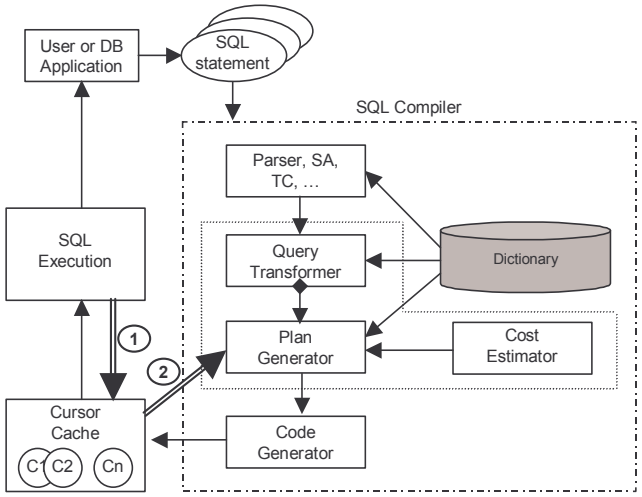

Figure 4: Architecture of the SQL Engine in Oracle 11g

[1] The selectivity of a predicate P on a table R is defined as the probability for P to evaluate to true for any row of R.

2. FEEDBACK-BASED OPTIMIZATIONS

Oracle 11g contain two new features that address the problems of cardinality estimation and generation of plans for queries containing bind variables. Both of these features use the paradigm summarized in Figure 4: for certain queries, monitoring information is collected at execution time and fed back to the cursor cache (arrow 1 in the figure), so that on subsequent executions, this new information is provided to the plan generator (arrow 2), to improve the plan that is generated.

2.1 Cardinality Feedback

Cardinality feedback aims to address cardinality misestimates by the Oracle cost-based optimizer due to complex predicates, and inaccurate or missing statistics. During the first execution of a SQL statement, an execution plan is generated as usual. During optimization, certain types of estimates that are known to be low quality (for instance, estimates for tables which lack statistics or tables with complex predicates) are noted, and monitoring is enabled for the cursor that is produced. If cardinality feedback monitoring is enabled for a cursor, then at the end of execution, the estimates for single table cardinalities are compared to the actual cardinalities. If an estimate is found to be significantly different from the actual value, then the correct value is stored for later use. On subsequent executions, the query is re-optimized, and the correct cardinality is substituted for the usual estimate. If the estimate is found to be relatively close to the original estimate after the first execution, then the monitoring is disabled for future executions.

Recall query Q1 from above, which retrieves product names and translated US English product names for bulk purchases of an item with a specified price.

Q1

```
SELECT d.translated_name,
       p.product_name
FROM order_items oi, products p,
product_descriptions pd
WHERE oi.unit_price = 15
  AND oi.quantity > 1
  AND p.product_id = oi.product_id
  AND pd.product_id = p.product_id
  AND pd.language_id = 'US'
```

The combination of equality and inequality filter predicates makes it difficult to estimate the single table cardinality of `order_items`. The cardinality is incorrectly estimated and the plan in Figure 2 (a) is chosen. Because this combination of predicates is known to be difficult to estimate cardinality for, cardinality feedback monitoring is enabled. At the end of the first execution, the actual single table cardinality is found to be significantly different from the estimate. So the new cardinality is stored in the cursor and the cursor is marked to be re-optimized.

On the second execution of the query, the cursor cache finds a matching cursor that is marked for re-optimization, so it does not share the cursor. Instead, the query is re-optimized, and the stored cardinality replaces the usual estimate. This leads to a different plan, shown in Figure 2 (b).

Cardinality feedback is enabled in a variety of cases where the quality of cardinality estimates is known to be suspicious. These includes tables with no statistics, combining selectivity of multiple conjunctive or disjunctive filter predicates, and predicates containing complex operators that the optimizer cannot accurately compute selectivity estimates for. In some cases, there are other optimizations available for these complex predicates; for instance, dynamic sampling or multi-column statistics [3] allow the optimizer to more accurately estimate selectivity of conjunctive predicates. In cases where these optimizations apply, cardinality feedback is not enabled. However, if, for instance, multi-column statistics are not present for the relevant combination of columns, the optimizer can fall back on cardinality feedback.

Cardinality feedback is useful for queries where the data volume being processed is stable over time. For a query on volatile tables, the first execution statistics are not necessarily reliable. This feature is not meant to evolve plans over time as the data in the table changes; it is meant to address queries where the plan is not correct to begin with. That is why the monitoring for cardinality feedback is disabled after the first execution. Disabling monitoring also minimizes the impact of cardinality feedback on queries where it is not necessary – for queries where the estimates are correct, this one time overhead is not dissimilar to the one time overhead of optimizing a query.

The performance impact of cardinality feedback on query optimization is minimal. Supplying cardinality estimates based on feedback is achieved using existing infrastructure in the Oracle optimizer. The infrastructure introduced for Oracle Automatic SQL Tuning [5] allows cardinality estimates to be supplied to the optimizer using optimizer hints. If a cardinality estimate is supplied, the optimizer is saved the time to generate the estimate, so if anything optimizer performance improves for this case. For all other cases, there is some minor overhead for hint processing. While we do not have experimental data to support it, we do not believe that this has a significant impact on optimizer performance.

Oracle currently supports cardinality feedback for single table cardinalities only. Feedback of join cardinalities presents some additional challenges. The intermediate join cardinalities from execution only include the join prefixes in the plan that is actually used. If the optimizer were to use these as corrected estimates during compilation, it would compare costs of plans for some join orders that use higher quality estimates derived from execution feedback, and some that use the usual optimizer estimates. This apples to oranges comparison is likely to cause strange behavior. Oracle Automatic SQL Tuning [5] provides an offline alternative for improving plans where intermediate cardinality estimates are inaccurate. Real-time feedback of join cardinalities warrants consideration for future work.

2.2 Bind-Aware Cursor Sharing

In Oracle 11g, we introduced a new feature called *bind-aware cursor sharing*, to better optimize SQL statements containing bind variables. The new feature helps the optimizer generate better performing execution plans. Oracle uses feedback techniques to determine when to use this feature. We first summarize the feature, and then discuss the need for and use of feedback techniques.

2.2.1 Bind-Aware Cursor Sharing

We discussed earlier, in Section 1.2, the approach of bind variable peeking to allow the optimizer to estimate cardinalities and costs

of a query containing bind variables. This approach allows the optimizer to choose an optimal plan for the first value of a bind variable encountered, but the plan may not be optimal for subsequent values. For this reason, there has always been a trade-off when using bind variables in a query – bind variables result in less shared memory consumption and fewer costly query compilations, but for certain queries, this may result in unpredictable run-time performance (depending on what bind variable values are used).

In Oracle 11g, *bind aware cursor sharing* (BACS) was introduced to solve this problem. Prior to this extension, SQL developers had to choose between always sharing the same plan (by using bind variables) and never sharing the same plan (using no bind variables). With this new feature, the database automatically decides whether an existing plan is likely to be optimal for a new bind variable value; in this case, it shares the existing plan, otherwise it re-optimizes using the current bind variable value.

Bind-aware cursors have extra sharing criteria related to a *bind profile* that is generated for the query. The bind profile stores information about the selectivity estimates of the predicates containing bind variables. If a different value for the bind variable would cause the optimizer to generate a different selectivity estimate, then that bind's value should be considered during cursor matching. To do this, the cursor's bind profile stores selectivity ranges for each bind-containing predicate. When deciding whether to share a cursor, the bind profile for the current bind values is computed, and the two profiles are compared. If the current selectivity estimates fall within the acceptable range for the stored bind profile, then the cursor is shared; otherwise, Oracle recompiles the query with the current bind values. If it turns out that the same execution plan is chosen by the optimizer, then the optimizer merges the bind profiles for the two cursors, and frees one cursor. This prevents contention in the cursor cache due to many equivalent cursors appearing in the cache. Each bind profile starts with a relatively small selectivity range based on the selectivity for the bind value that the cursor was originally compiled for, and over time, the applicable selectivity ranges grow as new bind variable values are encountered.

Recall query Q2 from above.

Q2

```
SELECT count(distinct e.employee_id),
       min(j.start_date),
       max(j.end_date)
FROM employees e, job_history jh
WHERE e.job_id = :job_v
  AND e.employee_id = jh.employee_id
```

The `employees` table is skewed on the `job_id` column – certain jobs, like `'Clerk'` or `'Sales Rep'` are common, while other jobs, like `'VP'`, are much less common. The optimal plan for a very common job is a full table scan following by a hash join, whereas for the less common jobs, it is a nested loop join with index range scan. Consider executing this query three times, with three jobs: `'Sales Rep'`, `'Clerk'`, and `'VP'`. Suppose that 10% of the records in the `employees` table are Sales Reps, 12% are Clerks, and 0.05% are Vice Presidents. Using bind-aware cursor sharing, on the first execution, the optimizer would choose the hash join plan, and the bind profile would contain a selectivity range of approximately 10%. On the second execution, the optimizer computes the selectivity for `'Clerk'`, and finds that it is quite close to that of the existing cursor. Thus, that cursor would be shared. On the third execution, the optimizer computes the selectivity for `'VP'` and finds that it is not at all close to the existing cursor's selectivity. So the query is optimized with the current bind value, and a new plan is generated. That cursor's bind variable profile contains a selectivity range around 0.05%.

Bind-aware cursor sharing requires that additional computation take place at cursor matching time, to determine if a cursor is a good fit for the current bind values. In particular, the optimizer must compute the selectivity estimate of each of the relevant predicates for the current bind values. In order to reduce the slowdown to cursor matching, we have restricted the types of predicates supported to be those whose selectivity can be computed quickly. Efficient cursor matching was a hard requirement for this feature, so reducing the scope of the feature or slightly increasing the shared memory utilization is more acceptable than slowing down cursor matching. Efficiently computing selectivity at cursor matching time requires additional information be stored in the BACS context, including some of the column statistics for the columns appearing in these predicates, and a compile-time representation of the predicate. The types of predicates supported include equality and range predicates of the form:

```
<col> <op> <bind>
```

For instance, these predicates are supported:

```
name = :bnd
salary > :bnd
```

Whereas this predicate is not:

```
substr(name, 1, 5) = :bnd
```

For equality predicates, the object statistics must include a histogram for the column appearing in the predicate; otherwise, the selectivity estimate of the predicate will not change depending on the bind value.

2.2.2 Feedback for Bind-Aware Cursor Sharing

Bind-aware cursor sharing introduces overhead in terms of both time and shared memory utilization. There is additional overhead to check if a bind-aware cursor matches a new *bind set* (a set of bind variable values), and if it does not, incurring an additional compilation consumes additional resources, and leads to more cursors vying for space in the cursor cache. The additional compilation slows down the current execution of the query, but it also consumes more machine resources in general. The additional cursors that are created can cause contention in the cursor cache, which leads to other cursors (for different queries) being aged out. This in turn requires that those other queries will incur the cost of compilation on subsequent executions. So the problem of cursor cache contention can snowball to create system-wide problems. Furthermore, in order to decide if an existing plan is appropriate for a newly-encountered set of bind variable values, additional context information is stored in the cursor, so the cursor size is also slightly increased, resulting in greater shared memory utilization. For a fixed size cursor cache, this also means that fewer cursors can fit into the cache, again leading to possible cache contention.

All of these additional forms of overhead may be worthwhile if the result is improved plans for commonly executed queries. Our solution is to use feedback from execution to target specific queries for bind-aware cursor sharing. This allows us to avoid the added overhead for queries where it is unlikely to improve performance. We call the use of feedback to determine the mode of cursor sharing *adaptive cursor sharing* (ACS).

If a query is eligible for BACS, the cursor is initially in a monitored state where it does not use BACS, but its execution characteristics are monitored to determine if it is likely to be useful. The cursor is monitored using Oracle's *rowsource profiling*, a feature introduced in Oracle 11g, which tracks certain execution statistics for all queries at the rowsource level. One of these statistics, the number of rows processed, is used in the BACS determination. The numbers of rows produced by all rowsources are combined to generate a value that represents the amount of data processed in that execution of the query. If there is a lot of variation in the amount of data processed by a query as the bind values change, then it is likely that choosing different execution plans for different bind values may be beneficial. If after several executions, such variation is noted, then the cursor is switched to bind-aware cursor sharing for future executions. This monitoring takes place at the end of execution during a sample of all of the executions. The sample rate decreases over time, since after many executions if Oracle has not seen variation in the amount of data processed, it is likely it never will. This decreases the overhead introduced by the monitoring. The data points representing amount of data processed are stored in a histogram, to reduce the memory consumption. The computation determining if there is variation in the amount of data processed is actually based on variation across the histogram buckets. For example, consider the simple single-table query in query Q3.

Q3

```
SELECT count(e.employee_id)
FROM employees e
WHERE e.job_id = :job_v
```

The amount of data processed by the query depends on the `job_v` variable. Consider two scenarios: (A) over a number of executions, several of the executions query on common jobs, and several query on uncommon jobs, and (B) all of the executions query on common jobs. Figure 5 shows the histograms representing the amount of data processed across executions for the two scenarios.

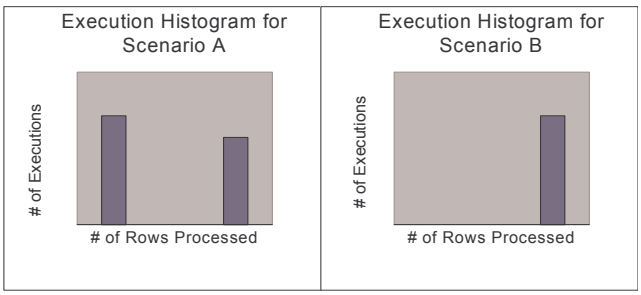

Figure 5: Execution Statistic Histograms

In scenario A, BACS would be enabled; the histogram shows that there is variation in the number of rows processed across different executions. In scenario B, no such variation is reflected in the histogram, so BACS would not be enabled.

To summarize the entire process end-to-end, consider again query Q2 from above, and the three executions previously described. Using ACS, after the first execution, the statistics about the relatively small number of rows processed are recorded. After the second execution, the statistics are again recorded, and checked to see if BACS is warranted. Since the number of rows processed across the two executions is similar, BACS is not enabled. After the third execution, the statistics are recorded, and Oracle finds that there is substantial variation in the amount of data processed across the three runs. Hence, BACS is enabled. Future executions of the query will use BACS. However, BACS requires additional context information be stored in the cursor, which has not been stored up until this point, so the optimizer must in essence start from scratch on the next execution. The next execution will trigger re-optimizing the query, and the relevant context information is stored.

2.3 An Abstract Feedback Process

These two features show commonalities in the process by which feedback is used to improve query optimization. This common process provides a template for future feedback-based optimizations.

Figure 6 summarizes the state transitions that a cursor can go through using these feedback-based optimizations, and the interactions with the query compiler. The compiler initially generates a cursor that is in a monitoring state, and populates some compiler information (e.g. optimizer cardinality estimates) into the context structure. The cursor continues to be in the monitoring state for some period of time, and may transition to a no monitoring state (i.e. Oracle pre-11g behavior) or to a feedback-enhanced state. During the monitoring phase, the context is further populated with execution information. In the feedback-enhanced state, there is additional interaction with the query compiler on subsequent executions. The context may be further populated with improved compilation information during this phase.

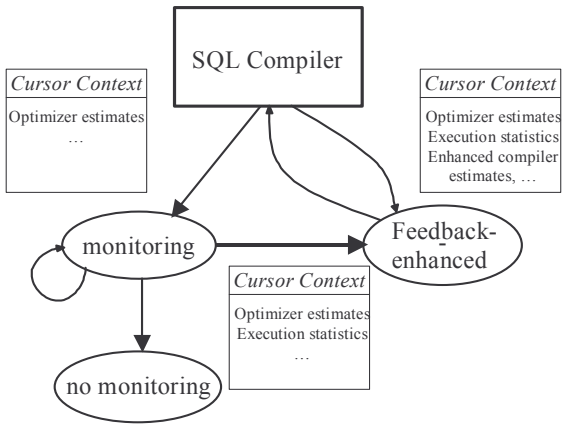

Figure 6: Cursor State Transitions with Feedback

3. PERFORMANCE STUDIES

We conducted experiments to measure the impact of cardinality feedback and adaptive cursor sharing. The experiments were conducted on an Oracle E-Business Suite sample database and workload. We first give a brief overview of the database and workload, followed by a discussion of the experiments.

3.1 Oracle E-Business Suite

The Oracle E-Business Suite (EBS) is a suite of business applications that includes Human Resources, Financial, CRM, Supply Chain and other applications. The EBS schema consists of approximately 25,000 tables and 45,000 indexes. Most of the schema is highly normalized. For our experiments, we used a sample database that is used for testing purposes. This sample database is 60 GB in size, with tables ranging from over a million rows to fewer than ten rows. The workload consists of approximately 332,000 queries from these various applications. The queries vary in complexity from simple one-table queries to queries containing joins of up to 140 tables. The most complex queries are significantly more complex than queries we have seen from other OLTP applications, containing complex and deeply nested subqueries. We believe the workload to be representative of the different types of OLTP queries that are encountered in the real world. The workload lacks long-running analytics queries submitted by batch job, but since these two features are targeted at shorter-running frequently executed queries, we believe this to be a sufficient workload for our experiments.

3.2 Cardinality Feedback

We conducted an experiment using an Oracle E-Business Suite (EBS) demo database, and a workload consisting of approximately 40,000 queries from EBS. These queries were selected from the larger test workload; we discarded non-SELECT statements, and queries containing bind variables. Each query was executed twice, and information about the execution plans used for each execution was gathered. Our goal was to determine what portion of these queries is impacted by cardinality feedback (monitored and recompiled), and what impact on performance this has. Ideally we would see a new plan and a performance improvement for each query that is recompiled. Executing each query exactly twice is not realistic. In reality, some queries may never be re-executed, and some may be executed many more times. We generally assume that queries are compiled once and run many times while the cursor is cached. For such queries, any improvement from cardinality feedback will be magnified by repeated execution, and the overhead of the additional compilation will be amortized over all subsequent runs.

Figure 7 summarizes the frequency of monitoring and recompilation. A little over one-third of the queries were monitored. Other queries were not monitored, because the optimizer did not believe selectivity estimates were uncertain enough to warrant runtime checking of the optimizer's estimates. Of the monitored queries, the vast majority did not trigger a recompilation. In total, 10% of the queries were compiled a second time, and 40% of those generated a new plan with the accurate cardinality recorded from execution.

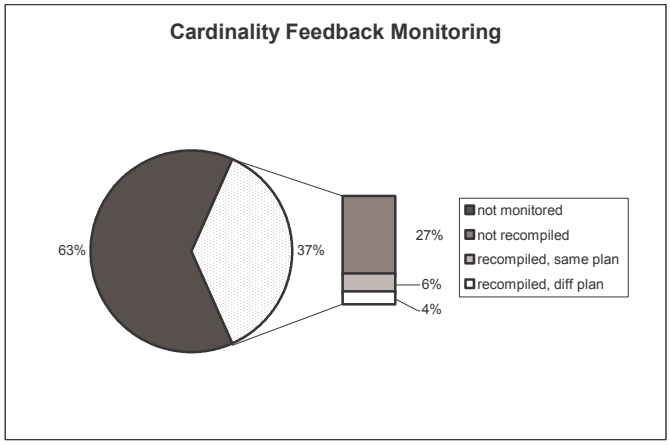

Figure 7: Cardinality Feedback Frequencies

Figure 8 summarizes the performance impact of cardinality feedback on the queries with new plans. We used buffer gets as a measure of query performance; a buffer get represents a logical I/O operation. While a buffer get will not necessarily result in disk access, it represents a potential disk access, depending on the state of the buffer cache. We measured the performance improvement or degradation using the following formula:

improvement/degradation = (before BG – after BG)/before BG

where *before BG* is the number of buffer gets with the feature disabled and the *after BG* is the number of buffer gets with the feature enabled.

Of the 1317 queries with different plans on the second execution, 924 improved, 214 degraded, and 179 had the same performance. The amount of improvement in the improved queries (68%) was greater than the degradation among the degraded queries (29%), so overall, there was a 58% improvement. The degraded queries are an interesting example of how better estimates can sometimes lead to a worse plan, due to weaknesses in the cost model or bad estimates elsewhere. Query degradation with better estimates may seem surprising, but there are many explanations for this. For instance, misestimates in cardinalities throughout a plan might complement each other – an overestimate in one table's cardinality may counteract an underestimate in a joined table's cardinality. This can result in improved estimates leading to a worse plan. But overall, one would expect improved estimates to lead to better plans, as is the case with this workload.

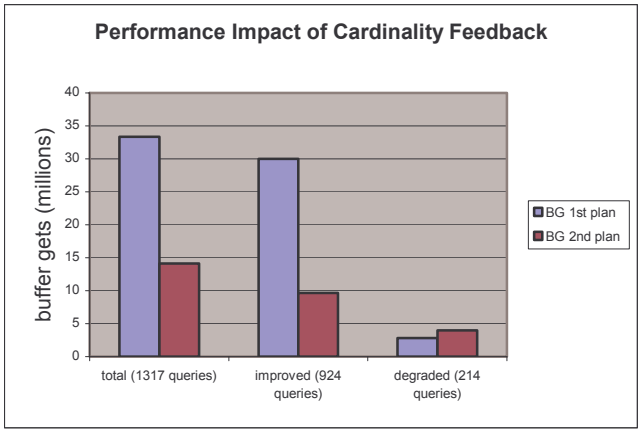

Figure 8: Impact of Cardinality Feedback

Figure 9 shows the distribution of the improvement across the queries. Most of the queries show an improvement between 0 and 100%, and the improvements are nearly evenly distributed over this range. The concentration around zero includes the 179 queries that had no change.

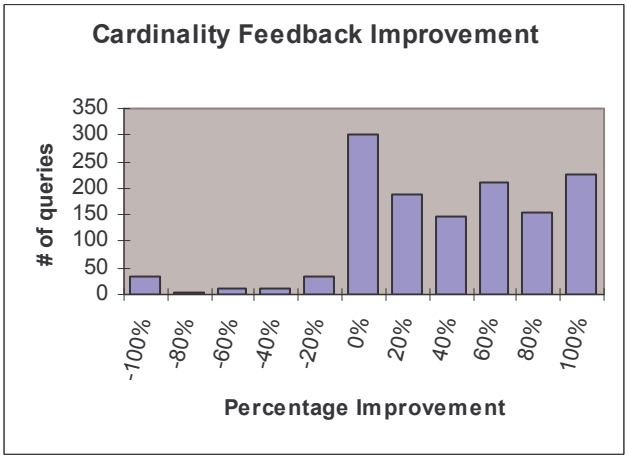

Figure 9: Distribution of Cardinality Feedback Improvement

3.3 Bind Variables

We conducted an experiment for adaptive cursor sharing using the same Oracle EBS sample database and workload. Bind values from a realistic run of the system were not available to us, so we used a tool to generate bind values. The bind value generation tool essentially reverse engineers a query to return combinations of bind values which when supplied to the query will return non-empty results. The distribution of the bind sets mirrors the distribution of the data – values appearing in more rows of the table are more likely to be generated in one of the sample bind sets. Due to the simplicity of the tool, bind values could not be generated for all queries. For instance, bind values appearing in complex predicates could not be generated, e.g.:

```
substr(:bind, 1, 5) = col
```

Because of these restrictions, our experiment may be skewed towards more simplistic queries for which bind value generation succeeded.

We started with a set of approximately 400 queries and their generated bind values. These queries were selected because they had been found to generate multiple plans in previous experiments where binds were replaced with literals (effectively disabling cursor sharing entirely). The queries had between 1 and 6 binds variables in them, with an average of 2.4. The number of bind sets generated ranged from 2 to 316 per query, with an average of 55. We ran each query with each bind set, and collected information on what execution plan was used for each execution. With adaptive cursor sharing enabled, 58 of the queries (approximately 14%) showed more than one plan was used. We then re-ran this set of 58 queries in two other configurations, for a total of three runs. The three configurations can be summarized as follows:

- ACS: Use adaptive cursor sharing to determine the mode of cursor sharing
- ALWAYS_SHARE: Always share the cursor, regardless of bind values. This is equivalent to the Oracle pre-11g default behavior.
- NEVER_SHARE: Disable cursor sharing entirely, and always create a new cursor for each bind set. This is equivalent to replacing bind variables with literals for each bind set.

Our goal was to determine how many distinct plans each technique generates (as a measure of consumption of cursor cache), and the overall performance of the queries. We would expect NEVER_SHARE to have the best performance, but with the most cursor cache consumption and greater compilation overhead; and ALWAYS_SHARE to have the worst performance with the least cursor cache consumption (one plan per query). ACS represents a compromise between these two concerns. At best, ACS could approach the performance characteristics of NEVER_SHARE, with minimal impact on cursor cache contention.

We found that eight queries performed worse with NEVER_SHARE than with ALWAYS_SHARE. This is an example of a worse plan being generated even when better information is input to the optimizer. This behavior makes it difficult to compare the performance of ACS to NEVER_SHARE, so we removed these eight queries from further analysis. This left us with 50 queries.

Figure 10 summarizes the performance results of this study. As expected, ACS results in performance that is somewhere between NEVER_SHARE and ALWAYS_SHARE. ACS showed an 18% improvement compared with ALWAYS_SHARE. NEVER_SHARE showed a further 14% improvement. Our choice of workload may limit the improvement available when we do not share plans. Oracle EBS has been heavily tuned to avoid a combination of histograms and bind variables in queries where they can result in significant plan instability. Hence, the total improvement between ALWAYS_SHARE and NEVER_SHARE is not huge. But the goal of adaptive cursor sharing is merely to close the gap between these two configurations. In this experiment, we are more than halfway to closing the gap. We would expect the performance improvement to be higher if we lifted the restrictions on which histograms are created in Oracle EBS (e.g. by using the Oracle automatic histogram feature), or used binds in more cases where literals are used today.

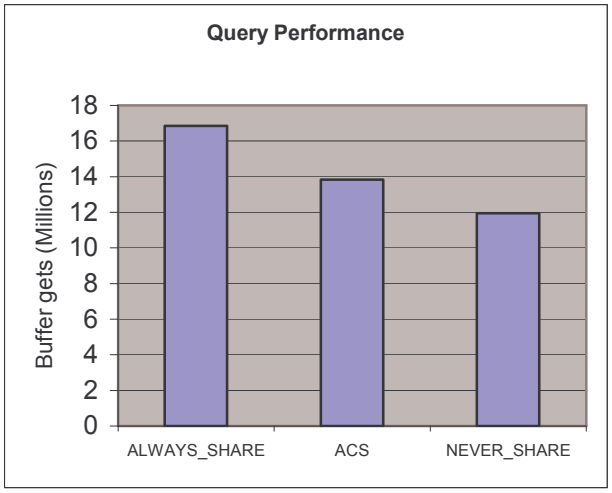

Figure 10: Cursor Sharing Impact on Query Performance

Figure 11 summarizes the number of distinct plans generated across the entire set of 50 queries in the three experiments. With NEVER_SHARE, we generate more than twice as many (149) plans as with ALWAYS_SHARE (65). Again ACS falls into the middle ground, with 119 distinct plans. Note that even with ALWAYS_SHARE there is slightly more than 1 cursor per query. This is because there are other cursor sharing criteria that must be met in order to share a cursor, independent of the bind variables.

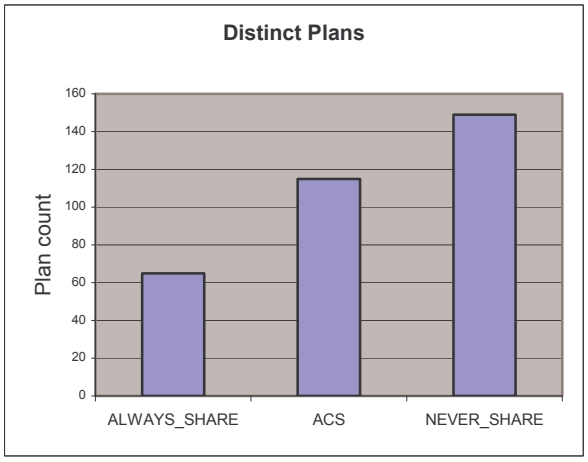

Figure 11: Cursor Sharing Impact on Number of Plans

This experiment demonstrates that adaptive cursor sharing offers improved execution performance for queries over classic cursor matching, without the cursor cache contention and compilation overhead caused by not using bind variables.

3.3.1 Feedback in Adaptive Cursor Sharing

It is clear from the above experiment that adaptive cursor sharing offers benefits for some queries containing bind variables. We conducted a second experiment to evaluate the necessity of using feedback to determine when to enable bind-aware cursor sharing. The impetus for using feedback to enable BACS is to control the overhead in terms of number of compilations (which adds latency to some query executions as well as consuming more machine resources overall) and number of distinct plans generated (which results in cursor cache contention).

We conducted an experiment on the original 428 queries that were known to produce different plans for different bind values. We ran each query with each bind set under two different configurations. In the first, adaptive cursor sharing was enabled as usual. In the second, the feedback aspect was disabled, so that every query eligible for bind-aware cursor sharing would use it; in other words, the monitoring phase was skipped, and bind-aware cursor sharing was enabled from the first execution. We collected information on the total number of plans generated for each query (which represents the number of compilations) and the total number of child cursors (distinct plans) for that query resident in the cursor cache after executing all of the bind value sets. Our goal was to determine if using feedback from execution really does reduce the two types of overhead mentioned above.

Figure 12 summarizes the results of this experiment. Without feedback, the number of compilations (6193) is nearly five times as high as with feedback (1256). Without feedback, the number of resident child cursors (1006) is nearly twice as high as with feedback (506). This experiment demonstrates the need for runtime information to determine when it is appropriate to enable BACS. Using feedback helps to achieve our goals of reducing overhead both in terms of additional compilations and cursor cache consumption.

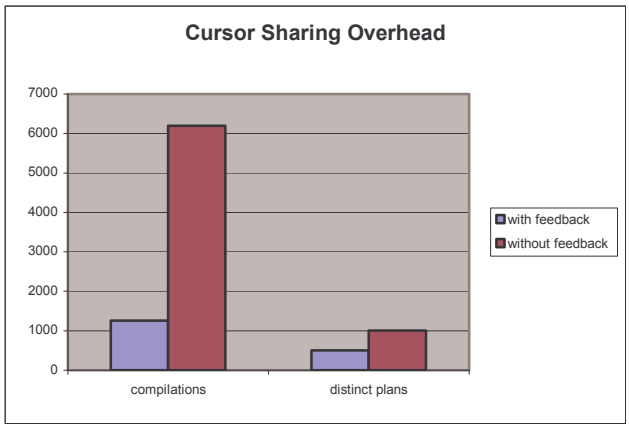

Figure 12: Feedback Impact on Cursor Sharing Overhead

4. RELATED WORK

The work described in this paper falls in the general area of feedback systems, where the output of a system is used to control its behavior. The system adapts when fed new information. There are several applications of this concept, and in this paper we are concerned with its application to improve the performance of a database system in processing SQL statements. In general, a feedback system is implemented around two main components: a decision component (DC) and a monitoring component (MC). DC makes decisions based on parameters updated by MC.

An example of a feedback system is the Automatic Memory Manager introduced in Oracle 9i [6]. In that paper the memory manager decides the maximum amount of memory a database operator (such as sort or hash) can consume during its execution. The amount of memory is dynamically updated by a monitoring component based on the number of active operators and their consumption patterns. The operators adapt to directives from the memory manager during their execution.

Another example is the work described in [3], [8], [9], and [11] to deal with the use of inaccurate or incomplete information that leads the optimizer to generate sub-optimal execution plans. While these papers proposed solutions to address a problem that is similar to one of the two problems we raised in our work, there are still differences that are worth highlighting.

First, despite the fact that the approach proposed in [3] automates the tuning of a SQL statement, it still relies on the user to activate the tuning process using the most expensive statements identified by the system. The user does not play a role in our approach. Furthermore, the feedback happens at the end of every execution of the statement, and is continuously updated with new information.

Second, unlike the work described in [8] and [9], our approach does not affect the current execution of the SQL statement; the plan does not change during execution. One of our major design goals is to have a solution that works out of the box because it is part of the production version of the Oracle DBMS. This requires that the current execution of the statement not be affected and the overhead from monitoring be extremely low. The information gathered at the end of execution will benefit future executions. Furthermore, we screen the statements during query optimization to pick the candidates for monitoring. This helps limit the monitoring overhead to the ones that we think may benefit from feedback. These statements represent 37% of the SQL workload used in the experiments described in Section 3.2. Furthermore, we stop monitoring the statement if feedback does not help the optimizer to find a better plan.

Third, the approach proposed in [11] computes adjustments based on the deviation between the estimated and actual cardinalities for filter predicates on the tables in a SQL statement, and stores them in the dictionary tables. The next time a SQL statement is compiled the optimizer checks whether adjustments exist and if so uses them. While the idea of correcting cardinality estimation is similar to ours, the two approaches differ in a few ways:

1. LEO monitors every SQL statement, while the Oracle optimizer monitors a subset of SQL statements, those where cardinality estimates are known to be low quality. This helps limit the overhead of monitoring and make the solution more practical in a production environment.
2. If the recompilation of a SQL statement based on feedback information did not cause a different plan then the Oracle optimizer will disable monitoring for it. LEO will keep monitoring the SQL statement.
3. LEO stores the feedback information (cardinality adjustments) in dictionary tables and readjusts them when new statistics are collected for the related tables. We believe that this approach can lead to incorrect adjustments over time because the readjustment process cannot predict how the data distribution has shifted in the new statistics. The Oracle optimizer starts the learning from the beginning every time new statistics are collected. The feedback information is kept in memory allowing a fast access to it.
4. While LEO has both an online and offline option to analyze the feedback information, it seems that it is harder to design the online option correctly as per the paper. So we assume that the offline option was used. The Oracle optimizer performs the feedback online at the end of the statement execution with minimal overhead.

The second aspect of our work relates to the selection of execution plans for SQL statements containing bind variables. Based on our research of the literature on this topic, we believe that our work is the first to propose a feedback-based approach to this problem.

In [4] it is assumed that the value of bind variables is unknown during query optimization. The solution proposed in that paper consists of delaying the decisions that depend on the value of the bind variables to query start time. For every such decision the plan would contain multiple choices rooted to a special operator, choose-plan. The fundamental difference with our approach is that the impact on the plan is limited to the plan fragment under the choose-plan operator while our approach allows the whole query plan to change. The second difference is that our approach relies on feedback from execution to decide when re-optimization is needed. Last, the approach proposed in [4] might work well for an optimizer that deals only with localized decisions such as access path and join method. However, it is not practical for a modern query optimizer, where an early decision can impact the whole plan. For example, in Oracle the value of a bind variable can affect a simple decision such as an access path, as well as whether to apply a transformation such as join predicate pushdown [1].

All the approaches available in commercial systems rely on the SQL developer to decide whether the plan built for such SQL statements be shared or not when executed using different bind variable values.

IBM DB2 Version 9 [7] offers several options to the SQL developer using the REOPT command:

1. Always – re-generates the plan for the SQL statement each time the statement is executed. This is equivalent to disabling cursor sharing.
2. None or Once – generates the plan for the SQL statement once, using the bind variables values specified during the first execution.
3. Auto – re-generates the plan for the SQL statement every time the bind variable values change and an execution plan has not been generated for the specified bind variable values.

The SQL developer is clearly in charge of telling the system what to do and the system does not contain any form of feedback based on previous optimizations or executions of the same statement.

Microsoft SQL Server 2005 [10] allows a SQL developer to write SQL statements with bind variables (e.g. SQL statement is embedded in Transact-SQL program). It also provides a process

to convert literal-based statements into bind variable-based statements, where literals are essentially substituted by bind variables. The purpose of this process is to offer the same benefits from using bind variables (reduce compilation overhead and the memory used to store cursors in the cache) without requiring the SQL developer to rewrite existing applications. This process is referred to as parameterization. There are two variants of this process and the user is expected to pick which one to use: auto and forced parameterization. Under the auto variant, the system re-uses the same plan for different variable values when it is certain this does not lead to a sub-optimal plan. This variant is only used for very simple statements, i.e. involving a single table, seriously limiting its application. The forced variant can be used for more complex statements. However, the execution plan generated for the statement is shared regardless of the bind variable values. The documentation warns that because of forced parameterization the query optimizer might choose sub-optimal execution plans for SQL statements and that it should not be used for environments that rely heavily on "indexed views and indexes on computed columns." It is further recommends that it "should only be used by experienced database administrators after determining that doing this does not adversely affect performance."

5. CONCLUSION

In this paper, we discussed two of the problems that affect the quality of execution plans generated by query optimizers: bind variables and inaccurate cardinality estimation for intermediate results. The solutions we proposed have been implemented in the Oracle DBMS and shipped in a production release, Oracle 11g. Our approach relies on monitoring the execution of SQL statements to feed back a summary of the execution to the SQL compiler, more specifically the query optimizer. The feedback allows the query optimizer to adapt its decisions with the goal of improving the performance of future executions of the SQL statements. Two of the main design goals of our approach are (1) zero input from SQL developers and zero maintenance for the DBA, (2) extremely low overhead from monitoring the SQL statements that is limited to only the statements that may benefit from the feedback. We demonstrated, using a real SQL workload, how our approach can be used to improve the performance of SQL statements. The SQL workload comes from the Oracle E-Business Suite, used by tens of thousands of customers worldwide. The framework we developed can be leveraged to support feedback of other information that will further improve the database system performance. We plan to add such support in the future as well as improving the framework itself, e.g. by making the feedback information persistent (disk-based).

6. ACKNOWLEDGMENTS

We would like to thank Vadim Tropashko for his efforts in the validation of the initial ideas discussed in this paper.

7. REFERENCES

[1] Ahmed, R., Lee, A., Witkowski, A., Das, D., Su, H., Zait, M., and Cruanes, T. "Cost-based Query Transformation in Oracle". *In Proc. of the 32nd International Conference on Very Large Databases, 2006*. Pages 1026-1036.

[2] Avnur, R., Hellerstein, J. M. "Eddies: Continuously Adaptive Query Processing". *In Proc. of the ACM SIGMOD International Conference on Management of Data, 2000.* Pages 261–272.

[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". *In Proc. of the ACM SIGMOD, 2008*.

[4] Cole, R., Graefe, G. "Optimization of Dynamic Query Execution Plans". *In Proc. of the ACM SIGMOD, 1994*. Pages 150-160

[5] Dageville, B., Das, D., Dias, K., Yagoub, K., Zait, M., and Ziauddin, M. "Automatic SQL Tuning in Oracle 10g". *In Proc. of the 30th International Conference on Very Large Databases, 2004*. Pages 1098 - 1109.

[6] Dageville, B., and Zait, M. "Automatic Memory Management in Oracle 9i". *In Proc. of the 28th International Conference on Very Large Databases, 2002*. Pages 962 - 973.

[7] DB2 Version 9.1 - http://www-1.ibm.com/support/docview.wss?rs=71&uid=swg27009474

[8] Markl, V., Raman, V., Simmen, D., Lohman, G., Pirahesh, H., and Cilimdzic, M. "Robust Query Processing Through Progressive Optimization". *In Proc. of the ACM SIGMOD, 2004*. Pages 659 - 670.

[9] Navin, K., and DeWitt, D. J. "Efficient mid-query re-optimization of sub-optimal query execution plans", *In Proc. of the ACM SIGMOD, 1998*. Pages 106 – 117.

[10] SQL Server 2005 Books Online – http://msdn2.microsoft.com/en-us/library/ms130214.aspx.

[11] Stillger, M., Lohman, G., Markl, V., Kandil, M. "LEO – DB2's Learning Optimizer". *In Proc. of the 27th International Conference on Very Large Databases, 2001*. Pages 19-28.