Autonomic Query Optimization in Oracle9i

Kaustav Ghoshal
IBM India Software Labs

Abstract 3
A Query Optimizer 4
Query Optimization in Oracle 9i 4
SQL Transformation 5
Materialized View Rewrite 5
Index Selection for Materialized Views 6
Materialized View Invalidation 7
Loading and Refreshing of Materialized Views 7
Query Rewrite 8
Query Rewrite Integrity Modes 9
Correctness of Result 9
Summary Advisor 9
Providing a Workload 10
Cost Model and Statistics 10
Optimizer Statistics 11
Object-level Statistics 11
System Statistics 11
User-defined Statistics 12
Statistics Management 12
Parallel Sampling 12
Monitoring 13
Automatic Histogram Determination 13
Dynamic Sampling 13
Correlation 13
Transient Data 14
Dynamic Runtime Optimizations 14
Dynamic Degree of Parallelism 15
Dynamic Memory Allocation 15
A Self Learning Optimizer 16
Deferred Feedback-Based Learning 17
Immediate Feedback-Based Learning 18
Research issues in Autonomic Query Optimization 19
Stability and Convergence 19
Detecting and Exploiting Correlation 20
Reoptimization 21
Learning More Information 21

Abstract

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

Structured Query Language (SQL) has emerged as an industry standard for querying relational database management systems, largely because a user need only specify what data are wanted, not the details of how to access those data. A query optimizer uses a mathematical model of query execution to determine automatically the best way to access and process any given SQL query. This model is heavily dependent upon the optimizer's estimates for the number of rows that will result at each step of the query execution plan (QEP), especially for complex queries involving many predicates and/or operations. These estimates rely upon statistics on the database and modeling assumptions that may or may not be true for a given database.

This paper, discusses a model of an autonomic query optimizer that automatically self-validates itself without requiring any user interaction to repair incorrect statistics or cardinality estimates. By monitoring queries as they execute, the autonomic optimizer compares the optimizer's estimates with actual cardinalities at each step in a QEP, and computes adjustments to its estimates that may be used during future optimizations of similar queries. Moreover, the detection of estimation errors can also trigger reoptimization of a query in mid-execution. The autonomic refinement of the optimizer's model can result in a reduction of query execution time by orders of magnitude at negligible additional run-time cost. Also discussed in this paper are various methods and configurations that one would like to do, and which will serve as a prerequisite for the learning optimizer.

A Query Optimizer

Query optimization is of great importance for the performance of a relational database, especially for the execution of complex SQL statements. The task of a query optimizer is to determine the best strategy for performing each query. The query optimizer chooses, for example, whether or not to use indexes for a given query, and which join techniques to use when joining multiple tables.

These decisions have a tremendous effect on SQL performance, and query optimization is a key technology for every application mainly in the case of data warehouses and analysis systems. The process of query optimization is entirely transparent to the application and the end-user. Because applications may generate very complex SQL, query optimizers must be extremely sophisticated and robust to ensure good performance. For example, query optimizers transform SQL statements, so that these complex statements can be transformed into equivalent, but better performing, SQL statements.

Query optimizers are typically 'cost-based' although Oracle has provision for both cost based and rule based optimization. In a cost-based optimization strategy, multiple execution plans are generated for a given query, and then an estimated cost is computed for each plan. The query optimizer chooses the plan with the lowest estimated cost.

Query Optimization in Oracle 9i

Oracle's query optimizer consists of four major components:

SQL transformations: Oracle transforms SQL statements using a variety of sophisticated techniques during query optimization. The purpose of this phase of query optimization is to transform the original SQL statement into a semantically equivalent SQL statement that can be processed more efficiently.

Execution plan selection: For each SQL statements, the optimizer chooses an execution plan (which can be viewed using Oracle's EXPLAIN PLAN facility or via Oracle's "v$sql_plan" views). The execution plan describes all of the steps when the SQL is processed, such as the order in which tables are accessed, how the tables are joined together and whether tables are accessed via indexes. The optimizer takes in account all possible execution plans for each SQL statement, and chooses the best one out of these.

Cost model and statistics: Oracle's optimizer relies upon cost estimates for the individual operations that make up the execution of a SQL statement. In order for the optimizer to choose the best execution plans, the optimizer needs the best possible cost estimates. The cost estimates are based upon in-depth knowledge about the I/O, CPU, and memory resources required by each query operation, statistical information about the database objects (tables, indexes, and materialized views), and performance information

regarding the hardware server platform. The process for gathering these statistics and performance information needs to be both highly efficient and highly automated.

Dynamic runtime optimization: Not every aspect of SQL execution can be optimally planned ahead of time. Oracle thus makes dynamic adjustments to its query-processing strategies based on the current database workload. The goal of dynamic optimizations is to achieve optimal performance even when each query may not be able to obtain the ideal amount of CPU or memory resources.

Oracle additionally has a legacy optimizer, the rule-based optimizer. This optimizer exists solely for backwards compatibility, and will be taken off from the next release.

SQL Transformation

There are many possible ways to express a complex query using SQL. Oracle mainly implements SQL transformation in two main categories

Heuristic query transformation: These transformations are applied to incoming SQL transformations whenever possible. These transformations always provide equivalent or better query performance, so that Oracle asserts that applying these conditions would not degrade the performance of the query.

Cost based query transformation: Oracle mainly uses a cost-based approach for several classes of query optimization. Under this approach, the transformed query is compared with the original query and then it's the task of the optimizer to determine the path that would lead to the least possible execution time. Since this is the default approach that Oracle uses, more emphasis is given to this part. Some aspects of the cost based approach can also be automated using some precomputed results.

Materialized View Rewrite

The concept of reusability comes in the case of one or more tables being accessed very frequently and the same data or a subset of this data being used in all the cases. In such a situation it is not wise to seek the desired data from the table, but rather it makes sense to maintain some sort of caching mechanism that the next query can utilize to make the data fetching much faster.

Precomputing and storing commonly used data in the form of materialized views can really aid in the making query processing fast. Oracle can transform the SQL queries so that one or more tables that are referenced in the query can be replaced by a reference to a materialized view. If the materialized view is smaller than the original table or has a better access path, the transformed SQL statement could be executed much faster than the original table.

The materialized view can be thought of as a special kind of a view which physically exists inside the database, it can contain joins and or aggregate and exists to improve query execution time by precalculating expensive joins and aggregation operations prior to execution. The optimizer should be able to learn about the frequency with which some parts of a particular table or a join are being accessed, and thus retain this materialized view for further use.

A materialized view definition can include aggregation, such as SUM, MIN, MAX, AVG, COUNT (*), COUNT(x), COUNT (DISTINCT), VARIANCE or STDDEV, one or more tables can be joined together and a GROUP BY. It may be indexed and partitioned and basic DDL operations such as CREATE, ALTER, DROP may be applied.

Since a materialized view is an object in the database then in any ways, a materialized view behaves like an index because:

- The purpose of the materialized view is to increase query execution performance
- The existence of a materialized view is transparent to the SQL application

This is a SQL statement to create a materialized view

```
CREATE MATERIALIZED VIEW COSTS
PCTFREE 0
STORAGE (initial 8k next 8k pctincrease 0)
BUILD IMMEDIATE
REFRESH FAST ON DEMAND
ENABLE QUERY REWRITE
AS
SELECT time_id, prod_name,
        SUM (unit_cost) AS sum_units,
        COUNT (unit_cost) AS count_units,
        COUNT (*) AS cnt
FROM costs c, products p
WHERE c.prod_id = p.prod_id
GROUP BY time_id, prod_name;
```

This is a typical implementation of a materialized view using the normal parameters

Index Selection for Materialized Views

Depending on the number of rows in the materialized view and whether it will be incrementally refreshed, it may be necessary to create indexes on the materialized views. Therefore, consideration should be given to creating a unique, local index which contains all of the materialized view keys. Other indexes could include a single-column bitmap indexes on each materialized view key column. This consideration should be learnt with time and the optimizer should be intelligent enough to determine the nature of the data.

Materialized View Invalidation

Materialized views are constantly being monitored to ensure that the data they contain is fresh. The purpose of invalidating a materialized view is to ensure that invalid data is not returned. A materialized view will be marked as stale whenever an object on which it is based is changed.

The state of the materialized view can he determined by querying the table USER_MVIEWS. If there is any doubt about the state of a materialized view, one can issue the command ALTER MATERIALIZED COMPILE to ensure that the latest status is shown.

Loading and Refreshing of Materialized Views

The main problem that is associated with the usage of summary tables is the initial loading and the subsequent updating of the summary. These issues are now addressed because summary management provides mechanisms to

- Fully refresh the data
- Perform a fast refresh, that is add/merge only the changes
- Automatically update a materialized view whenever the changes are made

Oracle 9i provides the following refresh methods:

- Complete
- fast(only the changes are applied)
- force, so a fast if possible, otherwise perform a complete refresh

These operations may be performed by:

- On demand refreshing by:
    - Specific materialized views(DBMS_MVIEW.REFRESH)
    - Those materialized views dependent on a table(DBMS_MVIEW.REFRESH_DEPENDENT)
    - All materialized views(DBMS_MVIEW.REFRESH_ALL_MVIEWS)
- On commit, whenever the tables on which the materialized view is defined are changed.

In the case of a data warehouse where the data is accessed quite frequently it makes sense to refresh the data in the materialized views, whereas in a scenario where the data is rarely accessed the on commit refresh mode is preferred.

Query Rewrite

One of the major benefits of using summary management which the end user will really benefit from is the query rewrite capability. It is a query optimizations technique that transforms a user query written in the terms of tables and views, to execute faster by fetching data from materialized views. The Oracle 9i server automatically rewrites any appropriate SQL application to use the materialized views.

The compositions of query does not have to exactly match the definition of the materialized view because this would require that the DBA knew in advance what queries would be executed against the data. This of course, especially with respect to data warehouses where one of the main benefits to an organization is to suddenly execute a new query. Therefore, query rewrite will still occur even is only part is the query can be satisfied by the materialized view.

Query rewrite occurs when the following parameters are set

ALTER SESSION SET QUERY_REWRITE_ENABLED =TRUE or
ALTER SYSTEM SET QUERY_REWRITE_ENABLED =TRUE

Or when the materialized view is defined, it is eligible for query rewrite, by including the clause ENABLE QUERY REWRITE.

One of the methods that help in the automation of the queries is the summary joinback method of query rewriting

Some times a query may contain reference to a column which is not stored in a summary table but can the obtained by joining back the materialized view to the appropriate dimension table. For example, consider the query

```
SELECT c.cust_last_name,
        sum (s.quantity_sold) AS quantity,
        sales s, customers c, products p
FROM sales s, customers c, products p
WHERE c.cust_id =s.cust_id
AND s.prod_id = p.prod_id
GROUP BY c.cust_last_name, p.prod_id
```

This query references the column c.cust_last_name which is not in the materialized view all_cust_salses_mv, but cust_last_name is functionally dependent on c.cust_id because of the hierarchical relationship between them .This means that this query can the rewritten on terms of all_cust_sales_mv, which is joined back to the customers table no order to obtain c.cust_last_name column.

Query Rewrite Integrity Modes

Summary management will endeavor to identify inconsistent materialized views and mark them accordingly, but to overcome these problems, three integrity levels are available, which are selected by the parameter QUERY_REWRITE_INTEGRITY

- STALE_TOLERATED
    - In this mode, a materialized view will always be used even if it is stale.

- TRUSTED
    - In this mode, the optimizer trusts that the data in the materialized views is fresh and that the relationships declared in dimensions and RELY constraints are correct. In this mode, the optimizer will also use prebuilt materialize views or materialized views based on views, and it will also used relationships that are not enforced as well as those that are enforced.
- ENFORCED(by default)
    - In this mode, the optimizers will only use materialized views that it knows contain fresh data and it will only use those relationships that are based on ENABLED_VALIDATED primary/unique/foreign key constraints.

Correctness of Result

Whenever a SQL query uses a materialized view rather than the actual source of the data, there are instances when the results returned may be different.

- A materialized view can be out of synchronization with the detail data. This generally happens because the refresh procedure is pending and STALE_TOLERATED integrity mode has been selected
- Join columns may violate referential integrity. In this case, some child side rows are not rolled up into exactly one parent side row. To avoid this situation, the system enforced integrity whose overheads are negligible and benefits are significant, is used.

It is possible to create a rolling materialized view, which is when the materialized view contains information about rows that no longer exist in the detail data. For example, the materialized view may contain 18 months worth of data, but the detail tables only contain the last 6 months. Therefore, if a query were ever to go against the base table rather than the materialized view than different results would show up.

Summary Advisor

When the decision is first made to use the materialized views an initial set has to be defined. To help resolve this problem, Summary Management contains a component

called the summary advisor which can either be invoked by calling a procedure or from the Oracle Enterprise Manager and it can provide the following information

- Recommended materialized views based on a collected or hypothetical workload
- Estimate the size of the materialized view
- Report actual utilization of the materialized views based on collected workload.
- Define filters to use against a workload
- Load and validate the workload
- Purge filters, workloads, and results

Before using the summary advisor, the procedure DBMS_STATS should be run to gather cardinality information on the tables and materialized views in the database. This information is used as a part of the prediction process.

Providing a Workload

Although the summary advisor can recommend materialized views without a workload, it performs best when it has a workload, which in Oracle9i, can the provided in the form of:

- User-defined (DBMS_OLAP.LOAD_WORKLOAD_USER)
- Current contents of SQL cache (DBMS_OLAP.LOAD_WORKLOAD_CACHE)
- Current contents of SQL trace (DBMS_OLAP.LOAD_WORKLOAD_TRACE)

A user-defined workload involves storing the queries in a table in the database. This will then be read by the summary advisor and taken as its workload. Alternatively, the current queries in the SQL cache can be made into a workload and used as input to the summary advisor.

If the Oracle Trace is available, an event set called the Summary Workload is provided. When enabled, it collects workload statistics comprising of the name of each materialized view and the ideal materialized view that could have been used in that particular case.

Although only one workload can be used at a time as input to the recommendation procedure RECOMMEND_MVIEW_STRATEGY, multiple workloads may be stored in the database and the compared to see which one generates the best recommendation.

Cost Model and Statistics

A cost-based optimizer works by estimating the cost of various alternative execution plans and choosing the plan with the best (that is, lowest) cost estimate. Thus, the cost model is a crucial component of the optimizer, since the accuracy of the cost model directly impacts the optimizer's ability to recognize and choose the best execution plans.

The ‘cost’ of an execution plan is based upon careful modeling of each component of the execution plan. The cost model incorporates detailed information about all of Oracle’s access methods and database structures, so that it can generate an accurate cost for each type of operation. Additionally, the cost model relies on ‘optimizer statistics’, which describe the objects in the database and the performance characteristics of the hardware platform. These statistics are described in brief below. In order for the cost-model to work effectively, the cost model must have accurate statistics.

Oracle has many features to help simplify and automate statistics-gathering. The cost model is a very sophisticated component of the query optimizer. Not only does the cost-model understand each access method provided by the database, but it must also consider the performance effects of caching, I/O optimizations, parallelism, and other performance features. Moreover, there is no single definition for costs. For some applications, the goal of query optimization is to minimize the time to return the first row or first set of N rows, while for other applications; the goal is to return the entire result set in the least amount of time. Oracle’s cost-model supports both of these goals by computing different types of costs based upon the preference of the DBA.

Optimizer Statistics

When optimizing a query, the optimizer relies on a cost model to estimate the cost of the operations involved in the execution plan (joins, index scans, table scans, etc.). This cost model relies on information about the properties of the database objects involved in the SQL query as well as the underlying hardware platform. In Oracle, this information, the optimizer statistics, comes in two flavors: object-level statistics and system statistics.

Object-level Statistics

Object-level statistics describe the objects in the database. These statistics track values such as the number of blocks and the number of rows in each table, and the number of levels in each b-tree index. There are also statistics describing the columns in each table. Column statistics are especially important because they are used to estimate the number of rows that will match the conditions in the WHERE-clauses of each query. For every column, Oracle’s column statistics have the minimum and maximum values, and the number of distinct values. Additionally, Oracle supports histograms to better optimize queries on columns which contain skewed data distributions. Oracle supports both height-balanced histograms and frequency histograms, and automatically chooses the appropriate type of histogram depending on the exact properties of the column.

System Statistics

System statistics describe the performance characteristics of the hardware platform. The optimizer’s cost model distinguishes between the CPU costs and I/O costs. However, the speed of the CPU varies greatly between different systems and moreover the ratio

between CPU and I/O performance also varies greatly. Hence, rather than relying upon a fixed formula for combining CPU and I/O costs, Oracle provides a facility for gathering information about the characteristics of an individual system during a typical workload in order to determine the best way to combine these costs for each system. The information collected includes CPU-speed and the performance of the various types of I/O (the optimizer distinguishes between single-block, multi-block, and direct-disk I/Os when gathering I/O statistics). By tailoring the system statistics for each hardware environment, Oracle's cost model can be very accurate on any configuration from any combination of hardware vendors.

User-defined Statistics

Oracle also supports user-defined cost functions for user-defined functions and domain indexes. Customers who are extending Oracle's capabilities with their own functions and indexes can fully integrate their own access methods into Oracle's cost model. Oracle's cost model is modular, so that these user-defined statistics are considered within the same cost model and search space as Oracle's own built-in indexes and functions.

Statistics Management

The properties of the database tend to change over time as the data changes, either due to transactional activity or due to new data being loaded into a data warehouse. In order for the object-level optimizer statistics to stay accurate, those statistics need to be updated when the underlying data has changed. The problem of gathering the statistics poses several challenges for the DBA.

Statistics gathering can be very resource intensive for large databases. Determining which tables need updated statistics can be difficult. Many of the tables may not have changed very much and recalculating the statistics for those would be a waste of resources. However, in a database with thousands of tables, it is difficult for the DBA to manually track the level of changes to each table and which tables require new statistics. Determining which columns need histograms can be difficult. Some columns may need histograms, others not. Creating histograms for columns that do not need them is a waste of time and space. However, not creating histograms for columns that need them could lead to bad optimizer decisions. Oracle's statistics-gathering routines address each of these challenges.

Parallel Sampling

The basic feature that allows efficient statistics gathering is sampling. Rather than scanning (and sorting) an entire table to gather statistics, good statistics can often be gathered by examining a small sample of rows. The speed-up due to sampling can be dramatic since sampling not only the amount of time to scan a table, but also

subsequently reduces the amount of time to process the data (since there is less data to sorted and analyzed). Further speed-up for gathering statistics on very large databases can be achieved by using sampling in conjunction with parallelism. Oracle's statistics gathering routines automatically determines the appropriate sampling percentage as well as the appropriate degree of parallelism, based upon the data-characteristics of the underlying table.

Monitoring

Another key feature for simplifying statistics management is monitoring. Oracle keeps track of how many changes (inserts, updates, and deletes) have been made to a table since the last time statistics were collected. Those tables that have changed sufficiently to merit new optimizer statistics are marked automatically by the monitoring process. When the DBA gathers statistics, Oracle will only gather statistics on those tables which have been significantly modified.

Automatic Histogram Determination

Oracle's statistics-gathering routines also implicitly determine which columns require histograms. Oracle takes this decision by examining two characteristics: the data-distribution of each column, and the frequency with which the column appears in the WHERE-clause of SQL statements. For columns which are both highly-skewed and commonly appear in WHERE-clauses, Oracle will create a histogram. These features together virtually automate the process of gathering optimizer statistics. With a single command, the DBA can gather statistics on an entire schema, and Oracle will implicitly determine which tables require new statistics, which columns require histograms, and the sampling level and degree of parallelism appropriate for each table.

Dynamic Sampling

Unfortunately, even accurate statistics are not always sufficient for optimal query execution. The optimizer statistics are by definition only an approximate description of the actual database objects. In some cases, these static statistics are incomplete. Oracle addresses those cases by supplementing the static object-level statistics with additional statistics that are gathered dynamically during query optimization. There are primarily two scenarios in which the static optimizer statistics are inadequate:

Correlation

Often, queries have complex WHERE-clauses in which there are two or more conditions on a single table. Here is a very simple example:

```
SELECT * FROM EMP
```

```
WHERE JOB_TITLE = 'VICE PRESIDENT'
AND SAL < 40000
```

The optimization approach is to assume that these two columns are independent. That is, if 5% of the employees are ‘Vice President’ and 40% of the employees have a salary less than 40,000, then the simple approach is to assume that .05 * .40 = .02 of employees match both criteria of this query. This simple approach is incorrect in this case. Job title and salary are correlated, since employees with a job title of ‘Vice President’ are much more likely to have higher salaries. Although the simple approach indicates that this query should return 2% of the rows, this query may in actuality return zero rows.

The static optimizer statistics, which store information about each column separately, do not provide any indication to the optimizer of which columns may be correlated. Moreover, trying to store statistics to capture correlation information is a daunting task: the number of potential column combinations is exponentially large.

Transient Data

Some applications will generate some intermediate result set that is temporarily stored in a table. The result set is used as a basis for further operations and then is deleted or simply rolled back. It can be very difficult to capture accurate statistics for the temporary table where the intermediate result is stored since the data only exists for a short time and might not even be committed. There is no opportunity for a DBA to gather static statistics on these transient objects.

Oracle's dynamic sampling feature addresses these problems. While a query is being optimized, the optimizer may notice that a set of columns may be correlated or that a table is missing statistics. In those cases, the optimizer will sample a small set of rows from the appropriate table(s) and gather the appropriate statistics on the-fly. In the case of correlation, all of the relevant conditions in the WHERE clause are applied to those rows simultaneously to directly measure the impact of correlation. This dynamic sampling technique is also very effective for transient data; since the sampling occurs in the same transaction as the query, the optimizer can see the user's transient data even if that data is uncommitted.

Dynamic Runtime Optimizations

The workload on every database fluctuates, sometimes greatly, from hour to hour, from daytime workloads to evening workloads, from weekday workloads to weekend workloads, and from normal workloads to end-of-quarter and end-of year workloads. No set of static optimizer statistics and fixed optimizer models can cover all of the dynamic aspects of these ever-changing systems. Dynamic adjustments to the execution strategies are mandatory for achieving good performance. For this reason, Oracle’s query optimization extends beyond just access path selection.

Oracle has a very robust set of capabilities which allow for adjustments to the execution strategies for each query based not only on the SQL statement and the database objects, but also the current state of the entire system at the point in time when the query is executing. The key consideration for dynamic optimization is the appropriate management of the hardware resources, such as CPU and memory. The hallmark of dynamic optimization is the dynamic adjustments of execution strategies for each query so that the hardware resources are utilized to maximize the throughput of all queries. While most other aspects of query optimization focus on optimizing only a single SQL statement, dynamic optimization focuses on optimizing each SQL statement in the context of all of the other SQL statements that are currently executing.

Dynamic Degree of Parallelism

Parallelism is a great way to improve the response time of a query on a multiprocessor hardware. However, the parallel execution of the query will likely use slightly more resources in total than serial execution. Hence, on a heavily loaded system with resource contention, parallelizing a query or using too high a degree of parallelism can be counterproductive.

On other hand, on a lightly loaded system, queries should have a high degree parallelism to leverage the available resources. Therefore, relying on a fixed degree of parallelism is a bad idea since the workload on the system varies over time. Oracle automatically adjusts the degree of parallelism for query, throttling it back as the workload increases in order to avoid resource contention. As the workload decreases, the degree of parallelism is again increased.

Dynamic Memory Allocation

Some operations (primarily, sorts and hash joins) are faster if they have access to more memory. These operations typically process the data multiple times; the more data that can fit into memory, the less data will need to be stored on disk in temporary table spaces between each pass. In the best case, sorts and hash joins can occur entirely in memory so that temporary disk space is not used at all. Unfortunately, there is only a finite amount of physical memory available on the system and it has to be shared by all the operations that are executing concurrently. If memory is over allocated, then swapping will occur and performance will deteriorate dramatically. On the other hand, if there is memory available that could be used to speed up the execution of a sort or a hash join, it should be allocated or the performance of the operation will be suboptimal. The challenge is to provide the optimal amount of memory for each query: enough memory to process the query efficiently, but not too much memory so that other queries can receive their share of memory as well.

Assigning a fixed amount of memory to each SQL statement is not an effective solution. As the database workload increases, more and more memory will be required to handle

the increasing number of queries. Eventually, the physical memory on the system would be exhausted, and the performance of the system would degrade dramatically. A slightly cleverer, but nevertheless inefficient, approach is to give each query an equal-sized portion of memory, so that if there are 100 concurrent queries, then each query gets 1% of the available memory and if there are 1000 concurrent queries, then each query gets .1% of the available memory.

This solution is insufficient, because each query operates on different-sized data sets, and each query may have a different number of sort and hash-join operations. If each query was given a fixed amount of memory, then some queries would have far too much memory while other queries would have insufficient memory. Oracle's dynamic memory management resolves these issues. The DBA specifies the total amount of system memory available to Oracle for SQL operations. Oracle manages the memory so that each query receives an appropriate amount of memory within the boundary that the total amount of memory for all queries does not exceed the DBA-specified limit.

For each query, the optimizer generates a profile with the memory requirements for each operation in the execution plan. These profiles not only contain the ideal memory levels (that is, the amount of memory needed to complete an operation entirely in memory) but also the memory requirements for multiple disk passes. At runtime, Oracle examines the amount of available memory on the system and the query's profile, and allocates memory to the query to provide optimal performance in light of the current system workload. Even while queries are running,

Oracle will continue to dynamically adjust the memory for each query. If a given query is using less memory than anticipated, that memory will be re-assigned to other queries. If a given query can be accelerated with the addition of more memory, then that query will be given additional memory when it is available. The decision about how each individual operation is affected by altering the memory allocations is based upon its memory profile. When dynamically adjusting memory allocations, Oracle will pick the individual operations that are best suited for the change with respect to the impact on overall performance. This unique feature greatly improves both the performance and manageability of the database, and relieves the DBA from managing memory allocations – a problem that is impossible to manually resolve to perfection.

A Self Learning Optimizer

This section describes the architecture of the proposed autonomic optimizer that observes actual query execution and uses actual cardinalities to autonomically validate and refine the estimates from its model and to reoptimize the current query, without requiring user intervention. In the following sections two essential functions of the autonomic optimizer are discussed: deferred learning for future queries and progressive optimization of the query currently under execution.

Deferred Feedback-Based Learning

Deferred learning exploits empirical results from actual executions of queries to validate the optimizer's model incrementally, deduce what part of the optimizer's model is in error, and compute adjustments to the optimizer's model for future query optimizations. Deferred learning with proposed model works under the assumption that future queries will be similar to previous queries, that is, they will share one or several predicates. The proposed model currently corrects the statistics for tables (which may be out of date) and estimates the selectivity of individual predicates in this way.

The feedback loop is comprised of four steps, as seen in Figure 1: monitoring, analysis, feedback, and feedback exploitation. At query compilation time, the monitoring component saves the cardinality estimates derived by the optimizer for the best (i.e., least-cost) plan, and during query execution saves the actual cardinalities observed for that plan. The analysis component uses the information thus learned to identify modeling errors and compute corrective adjustments. This analysis is a stand-alone process that may be run separately from the database server and even on another system. The feedback component modifies the catalog statistics of the database according to the learned information. The exploitation component closes the feedback loop by using the learned information in the system catalog to provide adjustments to the query optimizer's cardinality estimates.

Figure 1 Deferred learning

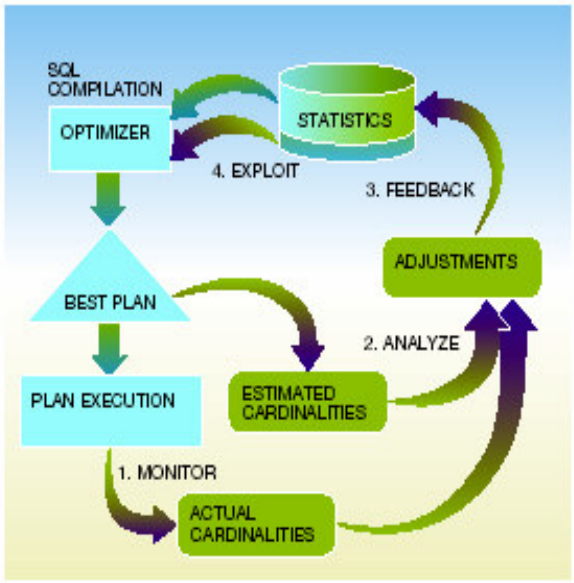

Figure 2 A query plan that can be reoptimized dynamically

The four components can operate independently, but form a consecutive sequence that constitutes a continuous learning mechanism by incrementally capturing plans, monitoring their execution, analyzing the monitor output, and computing adjustments to be used for future query compilations. This mechanism enables deferred learning, since only future queries will benefit from the feedback.

Immediate Feedback-Based Learning

The monitored cardinalities need not be used for subsequent queries alone. If the actual cardinalities are significantly different from the estimated cardinalities, the chosen query plan could be highly suboptimal.

Generally, response time and memory are optimized if each row is processed completely and returned to the user in a pipelined plan. But occasionally, the rows of an intermediate result must be fully materialized, either as a sorted or unsorted temporary table (TEMPORARY), which is called a materialization point. TEMPORARY afford a natural opportunity to count the number of rows and possibly to reoptimize the plan before any rows are returned to the user, thereby avoiding returning duplicate rows that are caused by restarting the query. However, two important issues arise:

- Since reoptimization involves a cost, when is it worthwhile?
- How can reoptimization be done efficiently?

The first question is answered in this subsection. For the second question, the easiest solution is to simply rerun the query "from scratch" under a new plan. However, this would waste all the work that has been done up to the materialization point, which was saved in the TEMPORARY. In most cases, it is preferable for the reoptimized plan to avoid having to redo that work by instead accessing that TEMPORARY in the reoptimized plan.

For example, Figure 2 shows a query plan for a simple two-table join that groups/aggregates the Orders table by Product_Id before the join. The sort that may be needed to accomplish this aggregation must materialize its entire input before proceeding and thus constitutes a TEMPORARY. Since most aggregations can be performed incrementally as the rows are sorted, the TEMPORARY will, at its conclusion, contain the GROUP BY result. The optimal join algorithm (nested loop join, hash join, or merged join) for subsequently joining Orders and Products tables depends crucially on the size of this GROUP BY result. The query optimizer could choose a suboptimal join algorithm if it under- or over-estimates the size of this result.

However, during query execution, the optimizer can monitor the size of the GROUP BY result, and reoptimize in case of severe estimation errors, for example, by changing the join algorithm if needed. Such reoptimization becomes more complex for more elaborate query plans with multiple materialization points. TEMPORARYs can be encapsulated as tables and the remaining portion of the query plan can be converted after the

TEMPORARYs into an SQL query, which can then be resubmitted to the query optimizer. Unfortunately, this approach has two problems. First, it may not be optimal to reuse a TEMPORARY as it is. In cases where the size of the TEMPORARY is much larger than expected, the optimal plan might be to reuse only a part of the TEMPORARY, or even ignore the TEMPORARY completely, in favor of a totally new plan that directly uses the base tables. Moreover, the remaining portion of the plan beyond the TEMPORARY may not always be expressible as an SQL statement, especially if it contains update operations, which are fed from sub queries.

A better alternative is not to encapsulate the TEMPORARYs, but instead to define them as materialized views and expose their definition to the query optimizer. The optimizer can then rely on standard view-matching techniques to identify TEMPORARYs that are worthwhile to reuse. The cost of reoptimization using additional materialized views is almost identical to the cost of optimizing the original query, since the optimizer only has to investigate one alternative intermediate table access method per materialized view. Moreover, once TEMPORARYs are defined as materialized views, there is no reason to limit their use to the current query only. All subsequent queries can potentially exploit materialized TEMPORARYs, just as they exploit user-defined materialized views. Of course, this approach could lead to an avalanche of such views, so that the query engine would have to periodically delete rarely used ones.

Research issues in Autonomic Query Optimization

This initial prototype of the Oracle autonomic optimizer has uncovered a number of challenging research problems that require solutions for any practical application of the optimizer in a product. Discussed below are these problems and possible approaches to their solution.

Stability and Convergence

A cardinality model refined by feedback has to take incomplete information into account. While some cardinalities may be deduced from query feedback—constituting hard facts—others are derived from statistics and modeling assumptions—forming uncertain knowledge. The learning rate of the system is largely dependent on the workload and the accuracy of statistics and assumptions. Assuming independence of predicates, when in fact the data are correlated, usually results in underestimation of the cardinalities of the intermediate results, which are used by the optimizer when determining the cost of a QEP. This underestimation will cause the optimizer to prefer a plan based on uncertain knowledge over one based on hard facts. The underestimation of cardinalities can result in a complete exploration of the search space; the system will converge only after trying out and learning about all QEPs that contain underestimation. Overestimation, however, may result in a local minimum (i.e., a suboptimal QEP); the optimizer will prefer other QEPs over a QEP with overestimates. Hence overestimates are unlikely to ever be

discovered or corrected. To reach a reasonable form of stability, the autonomic optimizer should initially use an exploratory mode, for example, before going into production. This mode will initially involve more risks by choosing promising QEPs based on uncertain knowledge, thus validating the model and gathering hard facts about data distribution and workload.

A second operational mode will be biased toward QEPs that are based on experience. This mode favors QEPs based on hard facts over slightly cheaper QEPs based on uncertain knowledge. The transition between the modes would be gradual. To overcome the local optimum caused by overestimation, it is necessary to explore uncertain knowledge used for presumably suboptimal, but promising QEPs, for example, by synchronous or asynchronous sampling.

Detecting and Exploiting Correlation

In practical applications, data are often highly correlated. In a country database, for instance, the selectivity of the conjunction (*country* _ "India" and *city* _ "Pune") is not correctly derived by multiplying the individual selectivity of *country* _ "India" and *city* _ "Pune", because the columns *country* and *city* are correlated —only India has a city called Pune. Since correlation constitutes a violation of the independence assumption, selectivity estimates for predicates involving correlation can be off by orders of magnitude in state-of-the-art query optimizers. There is an opportunity to detect and correct such errors in the approach followed.

Correlations pose many challenges. First, there are many types of correlation, ranging from functional dependencies between columns, especially referential integrity, to more subtle and complex cases, such as an application-specific constraint that an item is supplied by at most 20 suppliers. Second, correlations may involve more than two columns, and hence more than two predicates in a query, with subsets of those columns having varying degrees of correlation. Moreover, a single query can only provide evidence that two or more columns are correlated for specific values. For complex queries involving several predicates, isolating which subsets of predicates are correlated and the degree of correlation can be extremely difficult.

Another difficult research problem is to generalize correlations from specific values to relationships between columns: How many different values from executing multiple queries having predicates on the same columns are required to safely conclude that those columns are, in general, correlated, and to what degree? Instead of waiting for that many queries to execute, correlation detection could instead identify promising combinations of columns —even from different tables—on which the statistics utility would then collect multidimensional histograms. In addition, the observed information can be used to pinpoint errors in the cardinality model, populate the database statistics, or to adjust the erroneous estimates by creating an additional layer of statistics.

Reoptimization

As discussed in the subsection, "Immediate feedback-based learning," immediate learning can change the plan for a query at run time, when the actual cardinalities are significantly different from the estimated cardinalities. But the new plan could itself be quite expensive, if it cannot make use of prior TEMPORARYs efficiently. The optimizer will find this out during reoptimization, but the cost of reoptimization could itself be significant. Therefore it is crucial to determine, *without reoptimizing*, when it will be worthwhile to reoptimize.

However the question is not how inaccurate the optimizer's estimate is; it is whether the plan is suboptimal under the new cardinalities and whether the cost difference is enough to pay for the reoptimization. One heuristic looks at the nature of the plan operators and decides whether a change in the input cardinality for an operator is likely to make the operator suboptimal. Alternatively, the optimizer can be enhanced to pick not only the optimal plan, but also the range of selectivity for each predicate within which the plan is optimal. This prediction of the sensitivity of any plan to any one -parameter is extremely hard, because of nonlinearities in the cost model. The number of reoptimization attempts for a single query has also to be limited because the convergence problem of the subsection, "Stability and convergence" is even more serious here.

Learning More Information

Learning and adapting to a dynamic environment is not restricted to cardinalities and selectivity. Using a feedback loop, many costs and parameters currently estimated by the optimizer can be made self-validating. For example, the dominant aspect of cost, the number of physical I/Os, is currently estimated probabilistically from estimated hit ratios, assuming each application gets an equal share of the buffer pool. The optimizer could validate these estimates by observing actual I/Os, actual hit ratios, and/or actual times to access tables for a given plan.

Another example is the maximum amount of memory allocated to perform a particular sort in a plan. If the DBMS detected by query feedback that a sort operation could not be performed in main memory, it could adjust the sort heap size to avoid external sorting for future sort operations. Feedback is not limited to services and resources consumed by the DBMS, but also extend to the applications that the DBMS serves.

For example, the DBMS could measure how many of the rows in a query's result are actually consumed by each application and optimize each query's performance for just that portion of the result, for example, by effectively appending the OPTIMIZE FOR _n_ ROWS clause of SQL to that query. Similarly, feedback from executions could be used to automatically set many configuration parameters for shared resources that are currently set manually. Physical parameters such as the network rate, disk access time, and disk transfer rate are used to weight the contribution of these resources to plan costs, and are usually considered to be constant after an initial set-up. However, setting these

parameters using measured values is more autonomic and more accurately captures the effective rate. In the same way, the allocation of memory among different buffer pools, the total sort heap, and so on, can be tuned automatically according to hit ratios that were recently observed.