How Cost Based Optimizer Works?

Jože Senegačnik, Oracle ACE Director
DbProf d.o.o.
Ljubljana, Slovenia

Keywords
Cost Based Optimization, Statistical Optimizer, Selectivity, Cardinality

Introduction

The first versions of Oracle database until version 7 were using so called „Rule Based Optimizer“ or RBO which was preparing the execution plan according to predefined rules. These rules were ranked according to the quality of the access path ranging from the fastest (ROWID access, index access) to the slowest or most time consuming like full table scan.
One of the biggest problems of the RBO was the rules which were strictly followed but at the same time the amount of data to be processed was not taken into the consideration. Unfortunately the size of the databases started to grow and tables became greater and greater, however the RBO had no clue about the number of rows which should be returned. The rules were perfect for retrieving a small number of rows but when there were many rows to be retrieved then the rules failed and the data access path became suboptimal.
For this reason Oracle introduced the first version of statistical optimizer or how it was officially called “Cost Based Optimizer” (CBO) in later releases of version 7 database. Although Oracle encouraged users to use the new CBO, it was hardly usable so most of the users stayed with old RBO.

The CBO was slightly improved in version 8.0 and significantly in version 8i (8.1.x) when it became somehow mature product with still a lot of teething problems. Oracle version 9i made a big progress by introducing the system statistics. Until then Oracle had no chance to take into account the speed of CPU or disk access during the optimization phase. With the new system statistics Oracle was able to distinct between different systems and to prepare the execution plan by knowing the system characteristics.
The CBO was further improved in version 10g which was armed with automatic statistics gathering. In 10g Oracle introduced new query transformations which were now based on the cost calculation, what means that they are not applied if the cost of execution of the transformed statement is higher than the cost of untransformed query.
Version 11g brought in some new statistics gathering features and the extended statistics, which is actually used to correct the problem of column dependency. With extended statistics one can virtually create histograms on combinations of columns which are together used in the “where clause” of the SQL statement but are somehow dependent and therefore break the CBO assumption of column independency.
New data access methods were introduced in latest versions of the database which opened the place for better optimization.

Basics of cost based optimization

The cost based optimization is not following predefined set of ranked rules but rather uses the comparison of overall cost of execution of the statement as the main selection criteria for the winning plan. The execution plan with the lowest cost is the winning plan which is subsequently used by the runtime engine (a.k.a. row source generator).
But what is the cost? Actually the cost is a representation of the time which is expected to be required for the execution. It is an internal Oracle measure which, what we will see later, combines not only the cost of physical access measured in number of I/O operations but also the cost of CPU. If a deduction of cost was possible in early versions of Oracle database, it is really a hard task to accomplish in the

latest versions. Therefore one should consider the cost only as the internal measure and should never consider comparing cost among different statements.

The goal of the **estimator** is to estimate the overall cost of a given plan. If statistics are available, then the estimator uses them to compute the measures, otherwise it uses default values. The statistics improve the degree of accuracy of the measures. The computed measures are: *selectivity, cardinality and cost*. So the cost calculation is based on two measures: **selectivity** and **cardinality**.

Selectivity
Selectivity is the first and the most important measure. It represents a fraction of rows from a row set where row set can be a base table or a result of a 'join' or a 'group by' operator. The estimator uses statistics to determine the selectivity of a certain predicate. The selectivity is thus tied to a query predicate, such as 'id = 12445', or a combination of predicates, such as 'id = 12445' and 'status = 'A''. The purpose of query predicates is to limit the scope of the query to a certain number of rows that we are interested in. Therefore, the selectivity of a predicate indicates how many rows from a row set will pass the predicate test. Selectivity lies in a value range from 0.0 to 1.0 where a selectivity of 0.0 means that no rows will be selected from a row set and a selectivity of 1.0 means that all rows will be selected. Selectivity is the reciprocal of the number of distinct values that a certain column has (1/NDV where NDV means number of distinct values). The query selects rows that all contain one out of *n* distinct values. The estimator assumes a *uniform distribution* of data. This means that each value out of *n* distinct values will return the same number of rows. But there are situations when this is not true and as a result we can face large variations in the number of duplicates. For such cases we have to create a histogram. The purpose of a histogram is to help the estimator to generate good selectivity estimates for columns with non-uniform data distribution. The estimator will always use a histogram when available (under certain circumstances).
When there are no statistics available, the estimator uses an internal default value for selectivity. The predicate type governs which kind of internal defaults will be used. The equality predicate is expected to return fewer rows then a range predicate.

Here are some cases of predicate selectivity (a.k.a. Filter Factor (FF)) calculations:

• c1 = value	density (= 1/c1.num_distinct*)
• c1 like value	density (= 1/c1.num_distinct*)
• c1 > value	(Hi -value) / (Hi -Lo)
• c1 < value	(value -Lo) / (Hi -Lo)
• c1 >= value	(Hi -value) / (Hi -Lo) + c1.num_distinct
• c1 <= value	(value -Lo) / (Hi -Lo) + c1.num_distinct
• c1 between	(upper -lower) / (Hi -Lo) + 2*c1.num_distinct
	or the selectivity of c1 <= upper if that is smaller

When we combine several predicates in the "where clause" certain rules apply which could be seen on Figure 1.

One of very important assumptions of the CBO is the uniform data distribution. In real life we have very few cases when we have uniformly distributed data. For cases when we have to deal with skew or non-uniform distribution Oracle uses data distribution histograms.
Histograms are stored in the dictionary and computed during the statistics gathering task (either by DBMS_STATS package or deprecated ANALYZE). In latest versions of Oracle database several new options were introduced for gathering histograms. The "AUTO" option works fine for majority of cases and in now days there are only special cases when one has to change the settings to be different

than AUTO. It is beyond the scope of this paper to discuss all the details in gathering and using histograms.

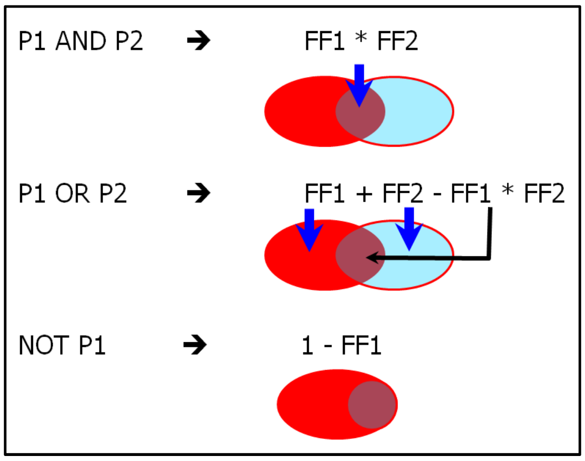

Figure 1: Combining predicates

When histograms are available they are used in calculations of selectivity (or as we can name it also the filtering factor - FF): The optimizer uses the density in its selectivity calculation, not number of distinct values. The density is calculated differently for columns with histograms, not simply as 1/number of distinct values.

Not only related to the histograms but in general the cause for a poor access plan is more often the incorrect estimate of the cardinality of a row source than the incorrect estimate of an index access cost. And histograms play a great role in determining the right selectivity especially in cases when the distribution is skew.

Oracle uses two types of histograms: Frequency and Height balanced histograms. In **Frequency histograms** there is one bucket for each distinct value, storing exact cardinalities. They are limited only to certain number of distinct values (actually 254) otherwise the space used for storing the histogram would be too big.

When there are more distinct values Oracle uses so called **height balanced histogram** where column values are divided up so that each bucket contains the same number of values

- Maximum values are recorded as endpoints.
- The special zero bucket records the minimum value.
- Column values occurring more than once as endpoint are popular values.
- For non-popular values the density statistic is used to estimate selectivity for equality predicates.

Cardinality

Cardinality represents **the number of rows in a row source** (table, result of previous operations). Analyzing the table captures the base cardinality. If table statistics are not available, then the estimator uses the number of extents occupied by the table and the default row length (100 bytes) to estimate the base cardinality. Computed cardinality (sometimes referred also as 'Effective' cardinality) is the number of rows that are selected from a base table when predicates are applied. The computed cardinality is computed as the product of the table cardinality (base cardinality) and combined selectivity of all predicates specified in where clause. Each predicate is acting as a successive filter on

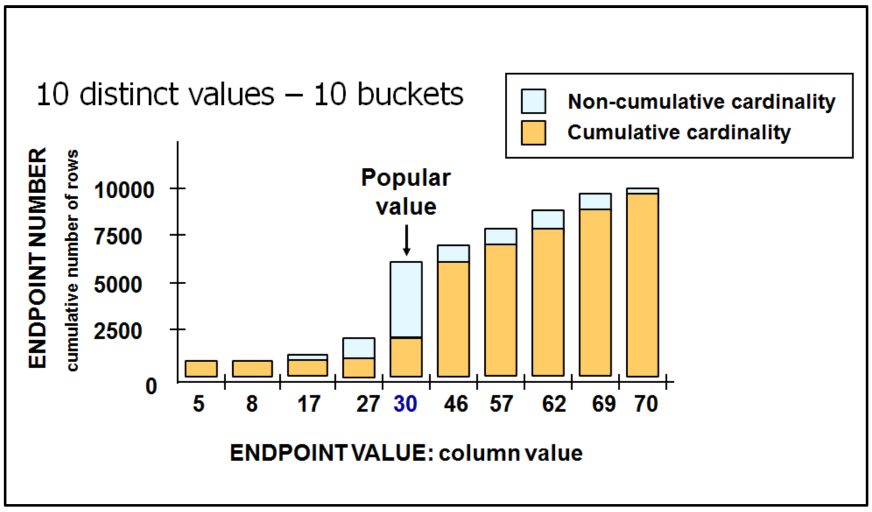

Figure 2: Frequency Histogram

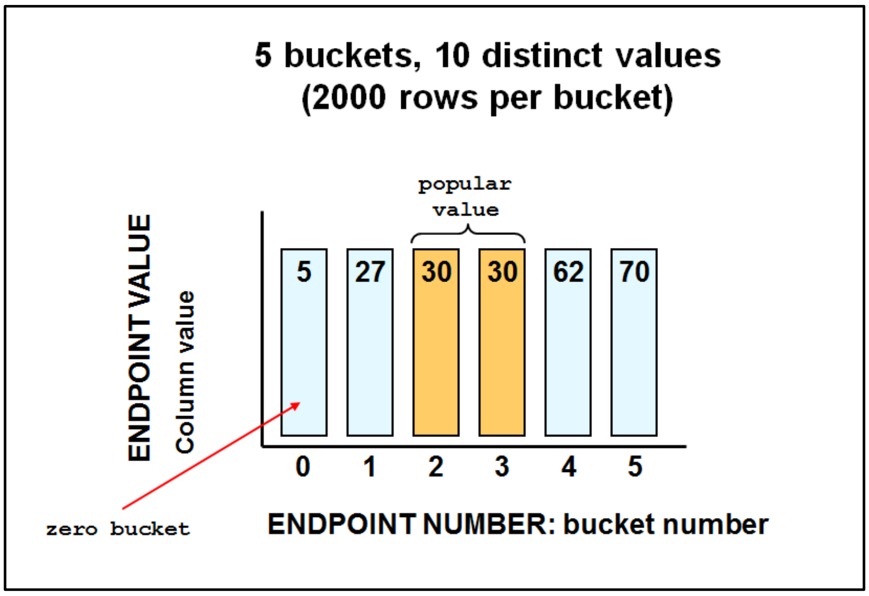

Figure 3: Height based histogram

the rows of the base table. The computed cardinality equals its base cardinality when there is no predicate specified. The computed cardinality for each row source (table or result of previous operations) is visible in the explain plan. The computed cardinality thus determines how many rows will be selected and is used as a measure in the subsequent calculations of cost.

Cost

The cost used by the CBO represents an estimate of the number of disk I/Os and the amount of CPU and memory used in performing an operation. We mentioned before that we can understand this alos as some kin of execution time estimation. The cost represents units of work or resource used. The CBO uses disk I/O, CPU usage, and memory usage (in join operations, sorting,…) as units of work. The operation can be a full table scan, accessing rows by using an index, join operation, sorting a row source, a group by operation and many others. So the cost of a query plan is the number of work units that were estimated as necessary when the statement is executed.

The aim of the **plan generator** is to try out different possible plans and the plan with the lowest cost is the winner. While generating possible plans the plan generator permutes access paths (full tables scan, index access, fast full index scan, index skip scan, etc.), join orders of tables and join methods (nested loops, hash join, sort-merge join). Oracle constantly improves the possible access paths and adds new ones from version to version. One should bear this in mind when using optimizer hints. Optimizer hints, when used (accepted) by the CBO, are in fact directives that instruct the CBO to use a certain execution plan. When new features become available, hints prevent the CBO from preparing a better plan because it has to obey hints that dictate a certain execution plan. Thereby using hints is the last resort in the SQL statement tuning process.
A lot of developers and DBAs use cost as the most important measure in the SQL statement tuning process. The misunderstanding of the cost concept leads them to erroneous conclusion that lower cost means faster execution and vice versa. When they accidentally use the right hint and run the statement but the response time is much better, they ask themselves how this is possible because the cost is higher but execution is faster. Another very common misconception is to compare the cost and the performance of two completely different SQL statements and make some conclusions from that. As we saw from the above description of how cost is calculated, it can only be treated as the **CBO's internal measure** that is used in the process of selecting the optimal plan. "Cost" is the result of the "price" of the access method and the estimated cardinality of the row source. When we recall that the cardinality of a row source (i.e. table, result of previous operations) is calculated from the base cardinality of the row source and the estimated selectivity, we suddenly discover the origin for sub-optimal execution plans. Thus both factors that are used in a cardinality computation can contribute to the plan becoming sub-optimal. Incorrectly estimated selectivity and an inaccurate base cardinality of the table have same effect. How can one see the possible danger that is hiding in the execution plan? We mentioned before that the estimated cardinality as well as cost is reported for each step in the execution plan. People usually do not pay too much attention to the cardinality but rather (and that's the major problem) to the estimated cost. The theoretical execution plan, produced by the *explain plan* command, also contains the estimated cardinality of the final result.

Query Transformations

For every SQL statement the CBO first tries to perform so called "query transformation". The goal of query transformations is to enhance the query performance. The transformation generates semantically equivalent form of statement, which produces the same results, but significantly differs in performance. The transformation rely on algebraic properties that are not always expressible in SQL, e.g, anti-join and semi-join.

The CBO supports different approaches. The "Automatic approach" always performs transformations for which it is known that they always produce a faster plan – they are still part of the latest versions of the CBO. In versions before 10g the CBO used so called »**Heuristic-based transformations**«. For heuristic-based transformation there was an assumption that such transformations produce faster execution plan most of the time. Unfortunately the real life and many problems with suboptimal execution plans prooved that this is not true. So since 10gR1 the CBO uses so called "**Cost-based Transformations**". Actually the transformation does not always produce a faster execution plan. The CBO teherfore costs non transformed query and transformed query and picks the cheapest execution plan.
Important fact to be remembered here is that the query transformation may span more than one query block – for instance a query can be transformed across the main select statement and also its sub-query.
It is beyond the scope of this paper to discuss all possible transformations. The CBO when writing the trace file (triggered by event 10053) writes some details about performed transformations and names them:

– JPPD - join predicate push-down
– OJPPD - old-style (non-cost-based) JPPD
– FPD - filter push-down
– PM - predicate move-around
– CVM - complex view merging
– SPJ - select-project-join
– SJC - set join conversion
– SU - subquery unnesting
– OBYE - order by elimination
– OST - old style star transformation
– ST - new (cbqt) star transformation
– CNT - count(col) to count(*) transformation
– JE - Join Elimination
– JF - join factorization
– SLP - select list pruning
– DP - distinct placement

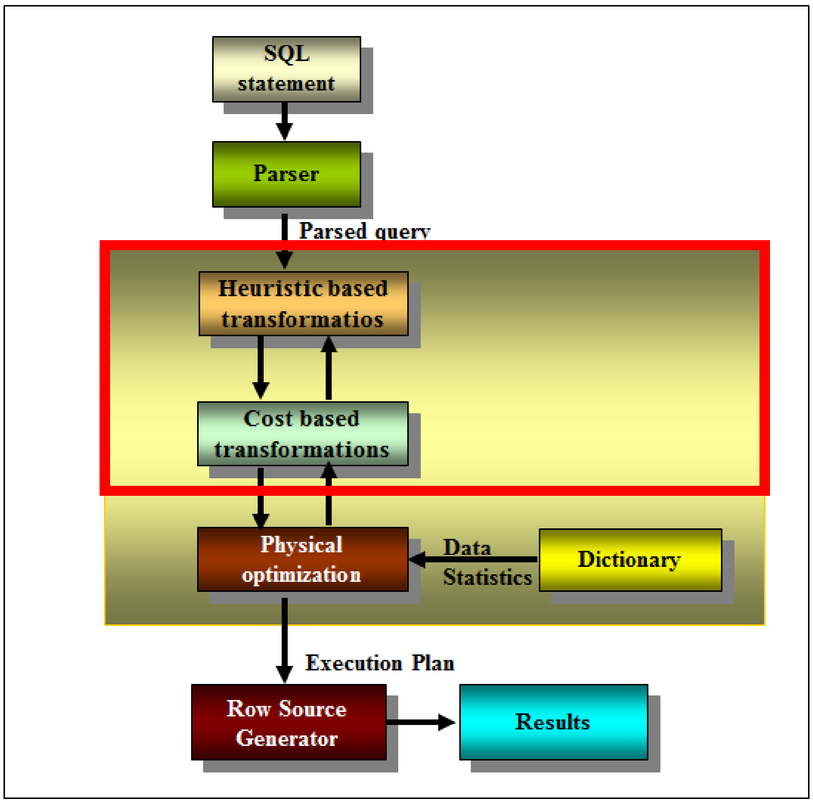

Figure 4: Query Transformations

Another fact to remember here is that every transformation can be source for the next transformation. The main goal of different transformations is to open the possibility to use another access path to data – for instance after the transformation an index access is possible while in the untransformed statement it was not possible.
Therefore each new transformation is opening possibility for another transformation and by this solution the search space for the optimal execution plan is getting bigger and bigger. As all these

transformations are "cost-based" the CBO calculates the cost of execution for each transformation and finally the winning plan is the plan with the lowest cost.

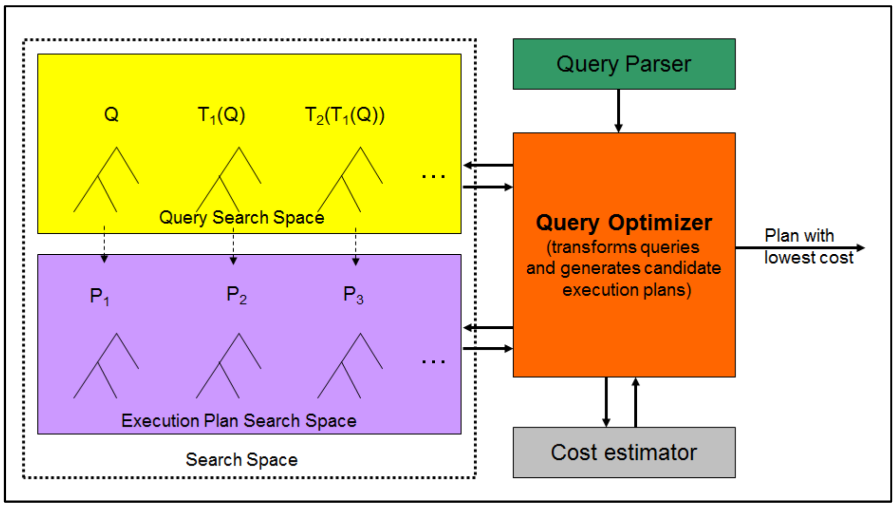

Figure 5: Logical and Physical Optimization

The query transformation is a logical optimization and takes place in the first step. The so called "**physical optimization**" takes place in subsequent step when the CBO calculates the cost of different access paths (full table scan, index access, …), join method (nested loop join, hash join, sort-merge join) and join order permutation (table T1 and T2, result is joined with T3, ….).

CBO, objects and functions developed in PL/SQL

Since Oracle introduced the possibility to define different objects in the database it also introduced so called "Extensible optimizer". The "normal" optimizer deals with the predefined Oracle tables (heap organized, index organized, clusters). But when one introduces an object into the database he also has to define the selectivity and cost of execution of its methods. The goal of the "extensible optimizer" is to introduce the development framework which one has to use in order to develop functions which are called by the CBO each time it is dealing with an object. The introduction of the objects also requires special type of statistics gathering – for instance how one can gather statistics on object type "CUSTOMER". The framework for the extensible optimizer is defined in catodci.sql (In $ORACLE_HOME/rdbms/admin directory). The ODCI stands for "Oracle Data Cartridge Interface". The whole interface is presented in the below Figure 6.

The DBMS_STATS or ANALYZE calls ODCIStatsColect and ODCIStatsDelete functions which store and delete the statistics. The statistics can be stored in a separate tables which are later on used in the ODCIStatsSelectivity function to estimate the selectivity. The whole interface has to be implemented as a statistical type. The implementation details of such type are beyond the scope of such paper.

For simple cases, especially for functions which are developed in PL/SQL language by the users and are used in "where clause" there is a simple method how one can define a default selectivity and cost of the execution. For such cases Oracle introduced the SQL command "ASSOCIATE STATISTICS WITH…".

Let us look at a simple case. We have a function named f2 which returns some value and in the "where clause" of the SQL statement we have a construct like this: "`where ... and f2(…)=10`". Let us assume that we perform several SQL statements inside the f2 function. Unfortunately the CBO does not know what kind of operations we perform inside the f2 function and therefore has no clue about the selectivity of the function (how many rows will pass this part of the filtering condition) and also the cost of execution. Therefore the CBO uses some default value for the selectivity and cost which is in most of the cases extremely wrong. By providing a more meaningful value with "ASSOCIATE STATISTICS WITH" command for selectivity and cost one can significantly improve the selectivity and cost calculation and therefore increasing the likelihood to get near optimal execution plan.

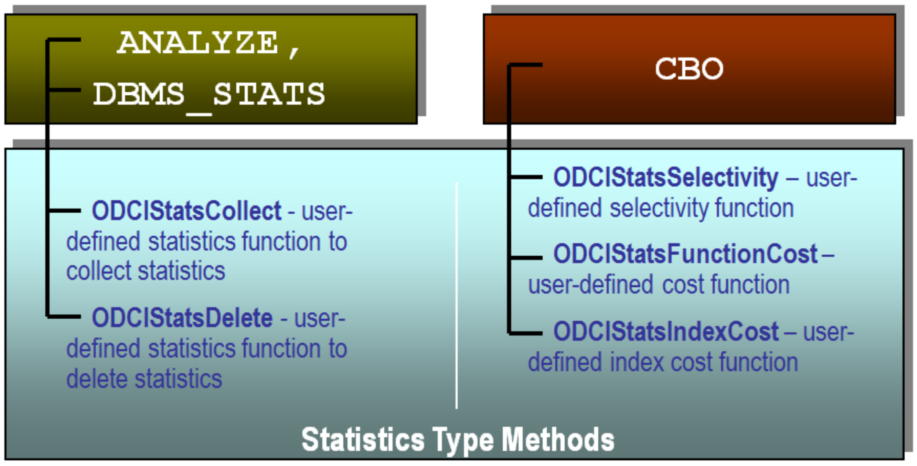
ANALYZE,
DBMS_STATS
CBO
ODCIStatsCollect - user-defined statistics function to collect statistics
ODCIStatsDelete - user-defined statistics function to delete statistics
ODCIStatsSelectivity – user-defined selectivity function
ODCIStatsFunctionCost – user-defined cost function
ODCIStatsIndexCost – user-defined index cost function
Statistics Type Methods

Figure 6: The Extensible Optimizer Function Calls

Corrective Mechanisms

Every automated system has to have some sort of corrective mechanism for situations when things go wrong. The CBO is not an exception. Since it was introduced in version 7 the CBO got some interesting correction mechanisms which were developed together with the development of the CBO.

Optimizer Hints were one of the first corrective mechanisms developed and are still there. In every new version new hints are introduced because they are used in other mechanism used for execution plan stability named "SQL Plan Baselines".

The optimizer hints as their name suggests should be hints for the optimizer how to perform the optimization, however they are actually commands. If they are used by the CBO they always influence the optimization path and majority of other facts is overlooked. For instance if someone hints the CBO to use the index access path instead of full table scan the CBO will produce an execution plan which will always enforce index access path although in certain circumstances a full table scan would be much more appropriate method.

In the earliest releases there was an impression that it is enough to use only one hint to achieve certain optimization. However when a new version was introduced such statement might get again suboptimal plan because several hints should be used.

"**Stored outlines**" were introduced in version 8i which were actually remembering the execution plan for a certain SQL statement in the way that a bunch of optimizer hints were stored in the dictionary and used on subsequent parses in order to generate the same execution plan. Unfortunately the mechanism was not bullet proof because it was not taking in the account the range of values of bind variables which could be passed to the optimizer in runtime.

To correct this problem Oracle introduced so called "**Adaptive cursor sharing**" mechanism together with the "**SQL Plan Baselines**". With these two mechanisms which are actually based on the result of monitoring the execution of the statement in runtime Oracle introduced a new feedback loop in the optimization phase of the statement. Actually the CBO monitors the cardinalities of the execution plan steps and compares them with the estimations which were made during the preparation of the original plan. When the run time cardinalities are way off from the estimates a new execution plan is prepared which is stored as a new child in the V$SQL performance view. The cardinalities may fluctuate due to changed parameters of the same SQL statement passed through bind variables. When a SQL statement is sensitive to changed bind values it is marked as bind sensitive and if new plan has to be introduced it is marked as bind aware. So in the situation when we create a SQL Plan Baseline for a certain cursor there might be several execution plans stored under it which are used according to the ranges of bind variables. So now Oracle looks at the values of bind variables not only at the optimization phase during hard parse but also monitors them later on if the cursor is bind aware. With such feedback mechanism Oracle was finally solving the problem of the execution plan instability. Unfortunately due to some bugs the mechanism is still not 100% bullet proof and might fail from time to time.

"**Automatic Cardinality Feedback Tuning**" is the feature of 11gR2. By using the result of monitoring the cardinalities achieved in the actual execution of SQL statement and comparing them with estimates the CBO might silently decide to change the execution plan on subsequent execution of the statement by reparsing the statement and producing a better plan. For the subsequent parse the CBO scales down or up as necessary the cardinality estimates according to the values seen at the first execution and corrected cardinality estimates may produce another, potentially better plan. Also this feature is still experiencing some teething problems and was significantly improved in the last release of 11gR2. One can detect if the automatic cardinality feedback was used by looking at the output of the DBMS_XPLAN package which under notes section of the execution plan writes if this feature was used.

Conclusion

The CBO is a very complex piece of code which takes into the account relatively small number of facts (statistics about our data) in order to prepare optimal execution plan for our queries. Most of the time (more than 99,99% of the time) the CBO is able to prepare a near optimal execution plan. However, there are situations when the data distribution is non-uniform or the developer don't provide enough data (like user defined statistics gathering or PL/SQL function costing) or there are some other reasons when the CBO is not able to prepare near optimal plan. For such situations Oracle has introduced the corrective mechanisms which in most cases will resolve the problem. If everything fails one can still use optimizer hints to force the CBO to use certain access paths or join methods.

Kontaktadresse:
Jože Snegačnik
DbProf d.o.o.
Smrjene 153
SI-1291 Škofljica
Slovenia

Telefon:	+386 41 72 44 61
E-Mail	joze.senegacnik@dbprof.com
Internet:	www.dbprof.com