Oracle Optimizer: What's new in Oracle 12c?

Maria Colgan
Oracle
Redwood Shores, CA

Keywords: Database, Performance, SQL Tuning

Introduction

The optimizer is one of the most fascinating components of the Oracle Database, since it is essential to the processing of every SQL statement. The optimizer determines the most efficient execution plan for each SQL statement based on the structure of the given query, the available statistical information about the underlying objects, and all the relevant optimizer and execution features.

With each new release the optimizer evolves to take advantage of new functionality and the new statistical information to generate better execution plans. Oracle Database 12c makes this evolution go a step further with the introduction of a new adaptive approach to query optimizations.

Adaptive Query Optimization

By far the biggest change to the optimizer in Oracle Database 12*c* is Adaptive Query Optimization. Adaptive Query Optimization is a set of capabilities that enable the optimizer to make run-time adjustments to execution plans and discover additional information that can lead to better statistics. This new approach is extremely helpful when existing statistics are not sufficient to generate an optimal plan. There are two distinct aspects in Adaptive Query Optimization, adaptive plans, which focuses on improving the initial execution of a query and adaptive statistics, which provide additional information to improve subsequent executions.

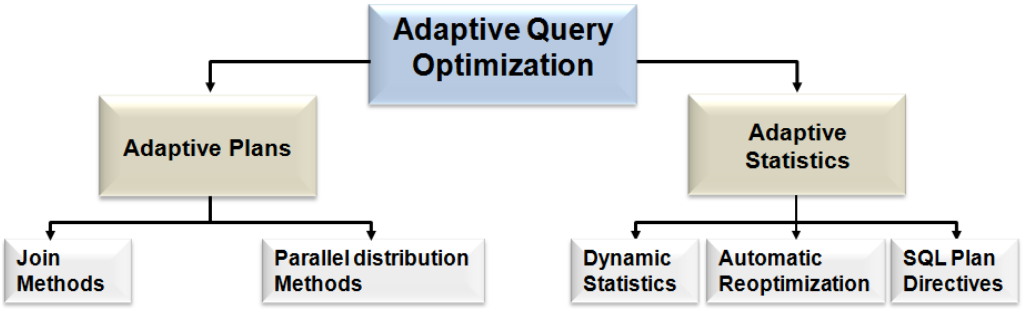


Figure 1. The components that make up the new Adaptive Query Optimization functionality

Adaptive Plans

Adaptive plans enable the optimizer to defer the final plan decision for a statement, until execution time. The optimizer instruments its chosen plan (the default plan), with statistics collectors so that at runtime, it can detect if its cardinality estimates, differ greatly from the actual number of rows seen by the operations in the plan. If there is a significant difference, then the plan or a portion of it can be automatically adapted to avoid suboptimal performance on the first execution of a SQL statements.

Adaptive Join Methods

The optimizer is able to adapt join methods on the fly by predetermining multiple subplans for portions of the plan. For example, in Figure 2 the optimizer's initial plan choice (default plan) for joining the order_items and product_info tables is a nested loops join via an index access on the product_info table. An alternative subplan, has also been determined that allows the optimizer to switch the join type to a hash join. In the alternative plan the product_info table will be accessed via a full table scan.

During execution, the statistics collector gathers information about the execution and buffers a portion of rows coming into the subplan. In this example, the statistics collector is monitoring and buffering rows coming from the full table scan of order_items. Based on the information seen in the statistics collector, the optimizer will make the final decision about which subplan to use. In this case, the hash join is chosen as the final plan since the number of rows coming from the order_items table is larger than the optimizer initially estimated. After the optimizer chooses the final plan, the statistics collector stops collecting statistics and buffering rows, and just passes the rows through instead. On subsequent executions of the child cursor, the optimizer disables buffering, and chooses the same final plan. Currently the optimizer can switch from a nested loops join to a hash join and vice versa. However, if the initial join method chosen is a sort merge join no adaptation will take place.

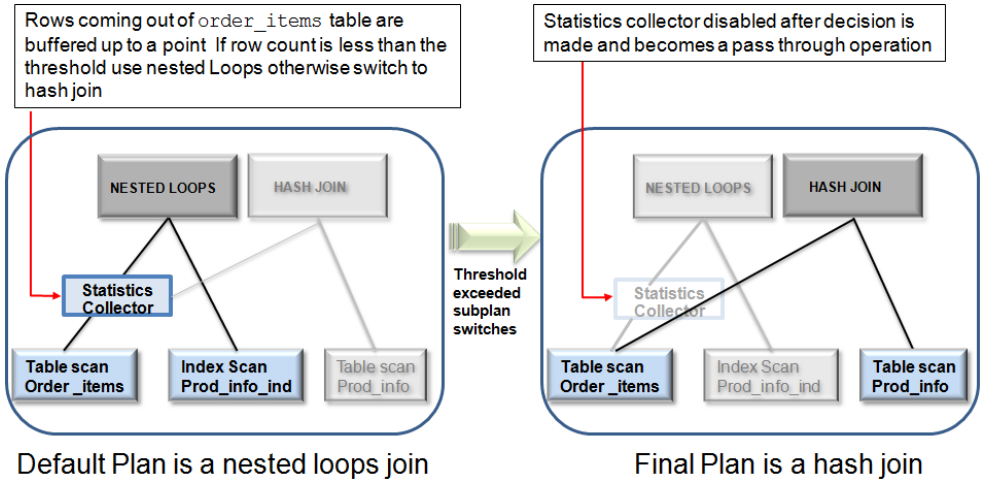


Figure 2. Adaptive execution plan for the join between the Order_items and Prod_info tables

By default, the explain plan command will show only the initial or default plan chosen by the optimizer. Whereas the DBMS_XPLAN.DISPLAY_CURSOR function displays only the final plan used by the query.

Adaptive Parallel Distribution Methods

When a SQL statement is executed in parallel certain operations, such as sorts, aggregations, and joins require data to be redistributed among the parallel server processes executing the statement. The distribution method chosen by the optimizer depends on the operation, the number of parallel server processes involved, and the number of rows expected. If the optimizer inaccurately estimates the number of rows, then the distribution method chosen could be suboptimal and could result in some parallel server processes being underutilized.

With the new adaptive distribution method, HYBRID HASH the optimizer can defer its distribution method decision until execution, when it will have more information on the number of rows involved. A statistics collector is inserted before the operation and if the actual number of rows buffered is less

than the threshold the distribution method will switch from HASH to BROADCAST. If however the number of rows buffered reaches the threshold then the distribution method will be HASH. The threshold is defined as 2 X degree of parallelism.

Adaptive Statistics

The quality of the execution plans determined by the optimizer depends on the quality of the statistics available. However, some query predicates become too complex to rely on base table statistics alone and the optimizer can now augment these statistics with adaptive statistics.

Dynamic statistics

During the compilation of a SQL statement, the optimizer decides if the available statistics are sufficient to generate a good execution plan or if it should consider using dynamic sampling. Dynamic sampling is used to compensate for missing or insufficient statistics that would otherwise lead to a very bad plan. For the case where one or more of the tables in the query does not have statistics, dynamic sampling is used by the optimizer to gather basic statistics on these tables before optimizing the statement. The statistics gathered in this case are not as high a quality (due to sampling) or as complete as the statistics gathered using the DBMS_STATS package.

In Oracle Database 12*c* dynamic sampling has been enhanced to become dynamic statistics. Dynamic statistics allow the optimizer to augment existing statistics to get more accurate cardinality estimates for not only single table accesses but also joins and group-by predicates. A new level, 11 has also been introduced for the initialization parameter OPTIMIZER_DYNAMIC_SAMPLING. Level 11 enables the optimizer to automatically decide to use dynamic statistics for any SQL statement, even if all basic table statistics exist. The optimizer bases its decision, to use dynamic statistics, on the complexity of the predicates used, the existing base statistics, and the total execution time expected for the SQL statement. For example, dynamic statistics will kick in for situations where the Optimizer previously would have used a guess. For example, queries with LIKE predicates and wildcards.

Given these new criteria it's likely that when set to level 11, dynamic sampling will kick-in more often than it did before. This will extend the parse time of a statement. In order to minimize the performance impact, the results of the dynamic sampling queries will be persisted in the cache, as dynamic statistics, allowing other SQL statements to share these statistics. SQL plan directives, which are discussed in more detail below, also take advantage of this level of dynamic sampling.

Automatic Reoptimization

In contrast to adaptive plans, automatic reoptimization changes a plan on subsequent executions after the initial execution. At the end of the first execution of a SQL statement, the optimizer uses the information gathered during the initial execution to determine whether automatic reoptimization is worthwhile. If the execution information differs significantly from the optimizer's original estimates, then the optimizer looks for a replacement plan on the next execution. The optimizer will use the information gathered during the previous execution to help determine an alternative plan. The optimizer can reoptimize a query several times, each time learning more and further improving the plan. Oracle Database 12*c* supports multiple forms of reoptimization.

Statistics Feedback

Statistics feedback (formally known as cardinality feedback) is one form of reoptimization that automatically improves plans for repeated queries that have cardinality misestimates. During the first execution of a SQL statement, the optimizer generates an execution plan and decides if it should

enable statistics feedback monitoring for the cursor. Statistics feedback is enabled in the following cases: tables with no statistics, multiple conjunctive or disjunctive filter predicates on a table, and predicates containing complex operators for which the optimizer cannot accurately compute cardinality estimates.

At the end of the execution, the optimizer compares its original cardinality estimates to the actual cardinalities observed during execution and, if estimates differ significantly from actual cardinalities, it stores the correct estimates for subsequent use. It will also create a SQL plan directive so other SQL statements can benefit from the information learnt during this initial execution. If the query executes again, then the optimizer uses the corrected cardinality estimates instead of its original estimates to determine the execution plan. If the initial estimates are found to be accurate no additional steps are taken. After the first execution, the optimizer disables monitoring for statistics feedback.
Figure 3 shows an example of a SQL statement that benefits from statistics feedback. On the first execution of this two-table join, the optimizer underestimates the cardinality by 8X due to multiple, correlated, single-column predicates on the customers table.

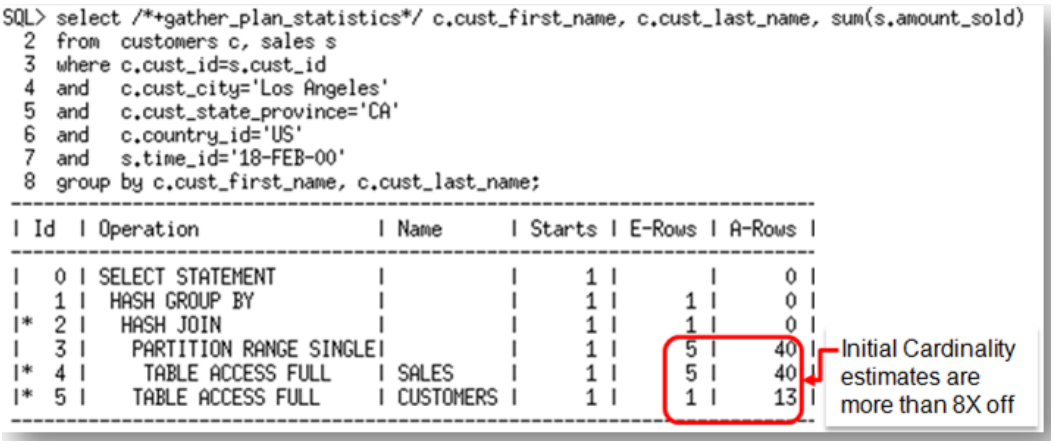


Figure 3. Initial execution of a SQL statement that benefits from Automatic Reoptimization statistics feedback

After the initial execution the optimizer compares its original cardinality estimates to the actual number of rows returned by the operations in the plans. The estimates vary greatly from the actual number of rows return, so this cursor is marked IS_REOPTIMIZIBLE and will not be used again. The IS_REOPTIMIZIBLE attribute indicates that this SQL statement should be hard parsed on the next execution so the optimizer can use the execution statistics recorded on the initial execution to determine a better execution plan.

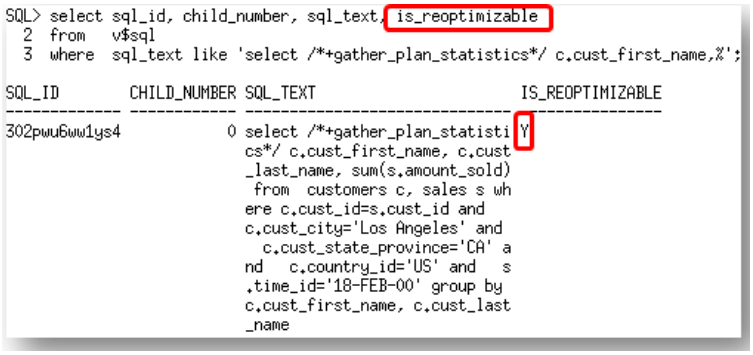


Figure 4. Cursor marked IS_REOPTIMIZIBLE after initial execution statistics vary greatly from original cardinality estimates

A SQL plan directive is also created, to ensure that the next time any SQL statement that uses similar predicates on the customers table is executed, the optimizer will be aware of the correlation among these columns.

On the second execution the optimizer uses the statistics from the initial execution to determine a new plan that has a different join order. The use of statistics feedback in the generation of execution plan is indicated in the note section under the execution plan.

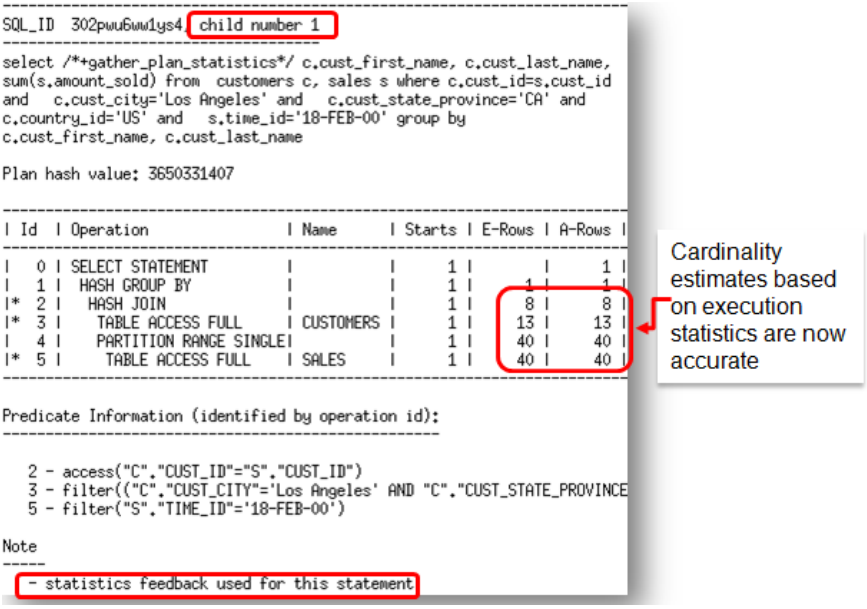


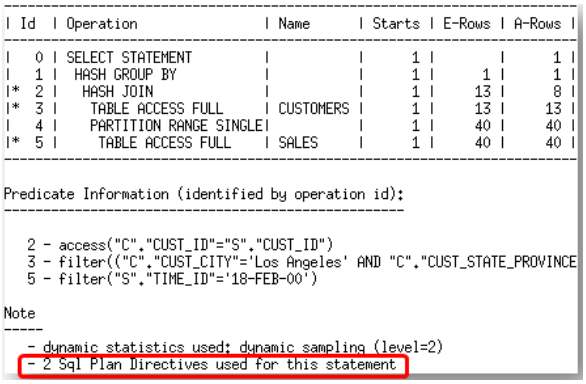
Figure 5. New plan generated using execution statistics from initial execution

The new plan is not marked IS_REOPTIMIZIBLE, so it will be used for all subsequent executions of this SQL statement.

SQL plan directives

SQL plan directives are automatically created based on information learnt via Automatic Reoptimization. A SQL plan directive is additional information that the optimizer uses to generate a more optimal execution plan. For example, when joining two tables that have a data skew in their join columns, a SQL plan directive can direct the optimizer to use dynamic statistics to obtain a more accurate join cardinality estimate.

SQL plan directives are created on query expressions rather than at a statement or object level to ensure they can be applied to multiple SQL statements. It is also possible to have multiple SQL plan directives used for a SQL statement. The number of SQL plan directives used for a SQL statement is shown in the note section under the execution plan (Figure 6).

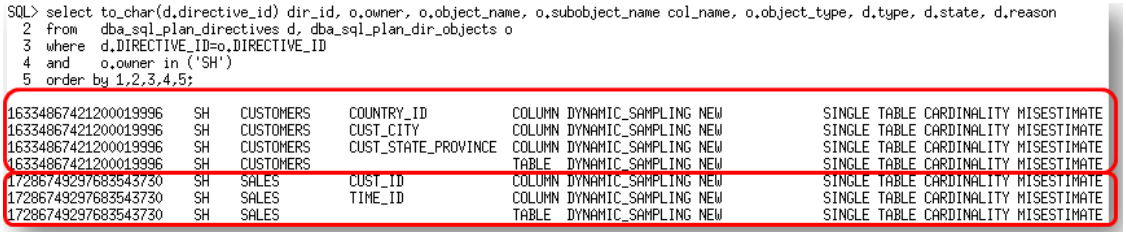


```
-------------------------------------------------------------------------------
| Id  | Operation                | Name      | Starts | E-Rows | A-Rows |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT         |           |      1 |        |      1 |
|   1 |  HASH GROUP BY           |           |      1 |      1 |      1 |
|*  2 |   HASH JOIN              |           |      1 |     13 |      8 |
|*  3 |    TABLE ACCESS FULL     | CUSTOMERS |      1 |     13 |     13 |
|   4 |    PARTITION RANGE SINGLE|           |      1 |     40 |     40 |
|*  5 |     TABLE ACCESS FULL    | SALES     |      1 |     40 |     40 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("C"."CUST_ID"="S"."CUST_ID")
   3 - filter(("C"."CUST_CITY"='Los Angeles' AND "C"."CUST_STATE_PROVINCE
   5 - filter("S"."TIME_ID"='18-FEB-00')

Note
-----
   - dynamic statistics used: dynamic sampling (level=2)
   - 2 Sql Plan Directives used for this statement
```

Figure 6. The number of SQL plan directives used for a statement is shown in the note section under the plan

The database automatically maintains SQL plan directives, and stores them in the SYSAUX tablespace. Any SQL plan directive that is not used after 53 weeks will be automatically purged. SQL plan directives can also be manually managed using the package DBMS_SPD. However, it is not possible to manually create a SQL plan directive. SQL plan directives can be monitored using the views DBA_SQL_PLAN_DIRECTIVES and DBA_SQL_PLAN_DIR_OBJECTS (See Figure 7).

As mentioned above, a SQL plan directive was automatically created for the SQL statement shown in Figure 3 after it was discovered that the optimizer's cardinality estimates vary greatly from the actual number of rows return by the operations in the plan. In fact, two SQL plan directives were automatically created. One to correct the cardinality misestimate on the customers table caused by correlation between the multiple single-column predicates and one to correct the cardinality misestimate on the sales table.



```
SQL> select to_char(d.directive_id) dir_id, o.owner, o.object_name, o.subobject_name col_name, o.object_type, d.type, d.state, d.reason
  2  from   dba_sql_plan_directives d, dba_sql_plan_dir_objects o
  3  where  d.DIRECTIVE_ID=o.DIRECTIVE_ID
  4  and    o.owner in ('SH')
  5  order by 1,2,3,4,5;

16334867421200019996    SH    CUSTOMERS     COUNTRY_ID           COLUMN DYNAMIC_SAMPLING NEW            SINGLE TABLE CARDINALITY MISESTIMATE
16334867421200019996    SH    CUSTOMERS     CUST_CITY            COLUMN DYNAMIC_SAMPLING NEW            SINGLE TABLE CARDINALITY MISESTIMATE
16334867421200019996    SH    CUSTOMERS     CUST_STATE_PROVINCE  COLUMN DYNAMIC_SAMPLING NEW            SINGLE TABLE CARDINALITY MISESTIMATE
16334867421200019996    SH    CUSTOMERS                          TABLE  DYNAMIC_SAMPLING NEW            SINGLE TABLE CARDINALITY MISESTIMATE
17286749297683543730    SH    SALES         CUST_ID              COLUMN DYNAMIC_SAMPLING NEW            SINGLE TABLE CARDINALITY MISESTIMATE
17286749297683543730    SH    SALES         TIME_ID              COLUMN DYNAMIC_SAMPLING NEW            SINGLE TABLE CARDINALITY MISESTIMATE
17286749297683543730    SH    SALES                              TABLE  DYNAMIC_SAMPLING NEW            SINGLE TABLE CARDINALITY MISESTIMATE
```

Figure 7. Monitoring SQL plan directives automatically created based on information learnt via reoptimization

Currently there is only one type of SQL plan directive, 'DYNAMIC_SAMPLING'. This tells the optimizer that when it sees this particular query expression (for example, filter predicates on country_id, cust_city, and cust_state_province being used together) it should use dynamic sampling to address the cardinality misestimate.

SQL plan directives are also used by Oracle to determine if extended statistics[1], specifically column groups, are missing and would resolve the cardinality misestimates. After a SQL directive is used the optimizer decides if the cardinality misestimate could be resolved with a column group. If so, it will automatically create that column group the next time statistics are gathered on the appropriate table. The extended statistics will then be used in place of the SQL plan directive when possible (equality predicates, group bys etc.). If the SQL plan directive is no longer necessary it will be automatically purged after 53 weeks.

[1] More information on extended statistics can be found in the paper Understanding Optimizer Statistics

Take for example, the SQL plan directive 16334867421200019996 created on the customers table in the previous example. This SQL plan directive was created because of the correlation between the multiple single-column predicates used on the customers table. A column group on CUST_CITY, CUST_STATE_PROVINCE, and COUNTRY_ID columns would address the cardinality misestimate. The next time statistics are gathered on the customers table the column group is automatically created.

```
SQL> BEGIN
  2    dbms_stats.gather_table_Stats('SH','CUSTOMERS');
  3  END;
  4  /

PL/SQL procedure successfully completed.

SQL>
SQL>
SQL> Select table_name, extension_name, extension
  2  From   dba_stat_extensions
  3  Where  owner='SH'
  4  And    table_name='CUSTOMERS';

TABLE_NAME   EXTENSION_NAME                    EXTENSION
------------ ----------------------------------------
CUSTOMERS    SYS_STSMZ$C3AIHLPBROI#SKA58H_N    ("CUST_CITY","CUST_STATE_PROVINCE","COUNTRY_ID")
```

Figure 8. Column group automatically created based on SQL plan directive

The next time this SQL statement is executed the column group statistics will be used instead of the SQL plan directive. The state column in `DBA_SQL_PLAN_DIRECTIVES` indicates where in this cycle a SQL plan directive is currently.

Once the single table cardinality estimates have been address additional SQL plan directives may be created for the same statement to address other problems in the plan like join or group by cardinality misestimates.

```
SQL> select to_char(d.directive_id) dir_id, o.owner, o.object_name, o.subobject_name col_name, o.object_type, d.type, d.state, d.reason
  2  from   dba_sql_plan_directives d, dba_sql_plan_dir_objects o
  3  where  d.DIRECTIVE_ID=o.DIRECTIVE_ID
  4  and    o.owner in ('SH')
  5  order by 1,2,3,4,5;

DIR_ID               OWNER OBJECT_NAME   COL_NAME            OBJECT TYPE            STATE          REASON
-------------------- ----- ------------- ------------------- ------ --------------- -------------- ------------------------------------
1622550251047685531 1 SH    CUSTOMERS                         TABLE  DYNAMIC_SAMPLING PERMANENT      JOIN CARDINALITY MISESTIMATE
1622550251047685531 1 SH    SALES                             TABLE  DYNAMIC_SAMPLING PERMANENT      JOIN CARDINALITY MISESTIMATE
1633486742120001999 6 SH    CUSTOMERS     COUNTRY_ID          COLUMN DYNAMIC_SAMPLING HAS_STATS      SINGLE TABLE CARDINALITY MISESTIMATE
1633486742120001999 6 SH    CUSTOMERS     CUST_CITY           COLUMN DYNAMIC_SAMPLING HAS_STATS      SINGLE TABLE CARDINALITY MISESTIMATE
1633486742120001999 6 SH    CUSTOMERS     CUST_STATE_PROVINCE COLUMN DYNAMIC_SAMPLING HAS_STATS      SINGLE TABLE CARDINALITY MISESTIMATE
1633486742120001999 6 SH    CUSTOMERS                         TABLE  DYNAMIC_SAMPLING HAS_STATS      SINGLE TABLE CARDINALITY MISESTIMATE
1728674929768354373 0 SH    SALES         CUST_ID             COLUMN DYNAMIC_SAMPLING NEW            SINGLE TABLE CARDINALITY MISESTIMATE
1728674929768354373 0 SH    SALES         TIME_ID             COLUMN DYNAMIC_SAMPLING NEW            SINGLE TABLE CARDINALITY MISESTIMATE
1728674929768354373 0 SH    SALES                             TABLE  DYNAMIC_SAMPLING NEW            SINGLE TABLE CARDINALITY MISESTIMATE
```

Figure 9. Multiple SQL plan directives created over time for the SQL statement seen if figure 3

Conclusion

The optimizer is considered one of the most fascinating components of the Oracle Database because of its complexity. Its purpose is to determine the most efficient execution plan for each SQL statement. It makes these decisions based on the structure of the query, the available statistical information it has about the data, and all the relevant optimizer and execution features.

In Oracle Database 12*c* the optimizer takes a giant leap forward with the introduction of a new adaptive approach to query optimizations and the enhancements made to the statistical information available to it. The new adaptive approach to query optimization enables the optimizer to make run-time adjustments to execution plans and to discover additional information that can lead to better statistics. Leveraging this information in conjunction with the existing statistics should make the optimizer more aware of the environment and allow it to select an optimal execution plan every time.

Contact address:

Maria Colgan
Oracle
Redwood Shores, CA

Email Maria.Colgan@oracle.com
Internet: http:blogs.oracle.com/optimizer