ORACLE **12**^{c}
DATABASE

Optimizer with Oracle Database 12c Release 2

ORACLE®

Table of Contents

Disclaimer

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Introduction

The Oracle 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.

This paper introduces all of the new optimizer and statistics related features in Oracle Database 12c Release 2[1] and provides simple, reproducible examples to make it easier to get acquainted with them, especially when migrating from previous versions. It also outlines how existing functionality has been enhanced to improve both performance and manageability.

Some Oracle Optimizer features have been broken out of this paper and covered in their own. Specifically, they are:

» Optimizer Statistics and Optimizer Statistics Advisor
» SQL Plan Management
» Approximate Query Processing

To get a complete picture of the Oracle Optimizer, it is recommended that you read this paper in conjunction with the relevant papers listed in the *References* section. See page 28 for details.

---

1 Oracle Database 12c Release 2 (12.2), the latest generation of the world's most popular database, is now available in the Oracle Cloud

Adaptive Query Optimization

By far the biggest change to the optimizer in Oracle Database 12c 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 execution of a query and **adaptive statistics**, which uses additional information to improve query execution plans.
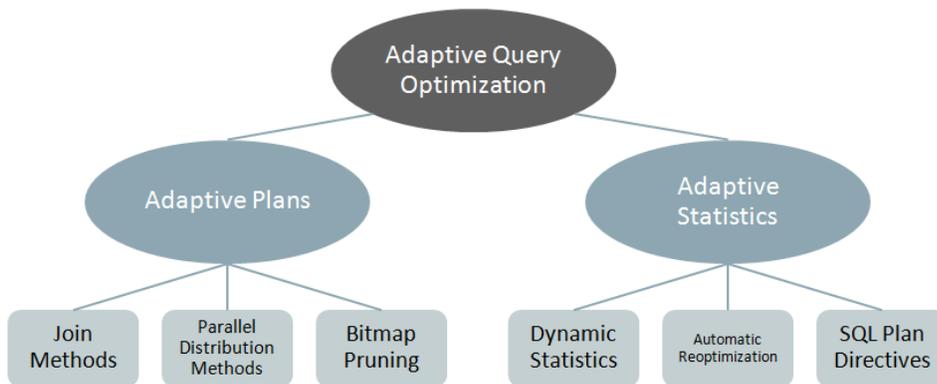

Figure 1: The components that make up the new Adaptive Query Optimization functionality

The adaptive features enabled by default in Oracle Database 12c Release 2 differs from Oracle Database 12c Release 1. See the "Initialization Parameter" section below for details.

Adaptive Plans

An *adaptive plan* will be chosen by the Optimizer if certain conditions are met; for example, when a query includes joins and complex predicates that make it difficult to estimate cardinality accurately. Adaptive plans enable the optimizer to defer the 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 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 will be automatically adapted to avoid suboptimal performance.

Adaptive Join Methods

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

During the initial execution, the *statistics collector* gathers information about the execution and buffers a portion of rows coming into the sub-plan. The Optimizer determines what statistics are to be collected, and how the plan should be resolved for different values of the statistics. It computes an "inflection point" which is the value of the statistic where the two plan choices are equally good. For instance, if the nested loops join is optimal when the scan of an *orders* table produces fewer than 10 rows, and the hash join is optimal when the scan of *orders* produces more than 10 rows, then the inflection point for these two plans is 10. The optimizer computes this value, and configures a buffering statistics collector to buffer and count up to 10 rows. If at least 10 rows are produced by the scan, then the join method is resolved to hash join; otherwise it is resolved to nested loops join. In Figure 2, the statistics collector is monitoring and buffering rows coming from the full table scan of *orders*. Based on the information seen in the statistics collector, the optimizer will make the decision about which sub-plan to use. In this case, the hash join is chosen since the number of rows coming from the orders table is larger than the optimizer initially estimated.
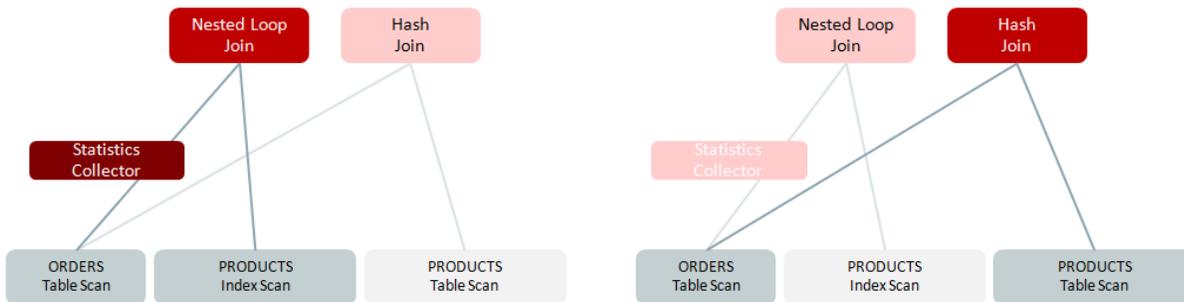
Figure 2: Adaptive execution plan for the join between ORDERS and PRODUCTS. Default plan on the left, chosen plan on the right.

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.

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 the plan actually used by the query.

Figure 3: Explain plan and DBMS_XPLAN.DISPLAY_CURSOR plan output for the scenario represented in Figure 2.

To see all of the operations in an adaptive plan, including the positions of the statistics collectors, the additional format parameter 'adaptive' must be specified in the DBMS_XPLAN functions. In this mode, an additional notation "-" appears in the *Id* column of the plan, indicating the operations in the plan that were not used (inactive).

```
SELECT *
FROM table(DBMS_XPLAN.DISPLAY_CURSOR(FORMAT=>'ADAPTIVE'));

-------------------------------------------------------
| Id  | Operation                    | Name     |
-------------------------------------------------------
|    0 | SELECT STATEMENT            |          |
|    1 |  SORT AGGREGATE             |          |
|  * 2 |   HASH JOIN                 |          |
|-   3 |    NESTED LOOPS             |          |
|-   4 |     NESTED LOOPS            |          |
|-   5 |      STATISTICS COLLECTOR   |          |
|  * 6 |       TABLE ACCESS FULL     | ORDERS   |
|- * 7 |       INDEX RANGE SCAN      | PROD_IDX |
|-   8 |      TABLE ACCESS BY INDEX ROWID| PRODUCTS |
|  * 9 |    TABLE ACCESS FULL        | PRODUCTS |
-------------------------------------------------------

Note
-----
   - this is an adaptive plan (rows marked '-' are inactive)
```

Figure 4: Complete adaptive plan displayed using 'ADAPTIVE' format parameter in DBMS_XPLAN.DISPLAY_CURSOR

SQL Monitor visualizes all operations if "Full" is selected in the "Plan" drop down box. The inactive parts of the plan are grayed out (see Figure 5). If the "Plan Note" icon is clicked, a pop-up box will be displayed confirming that the plan is an adaptive plan.
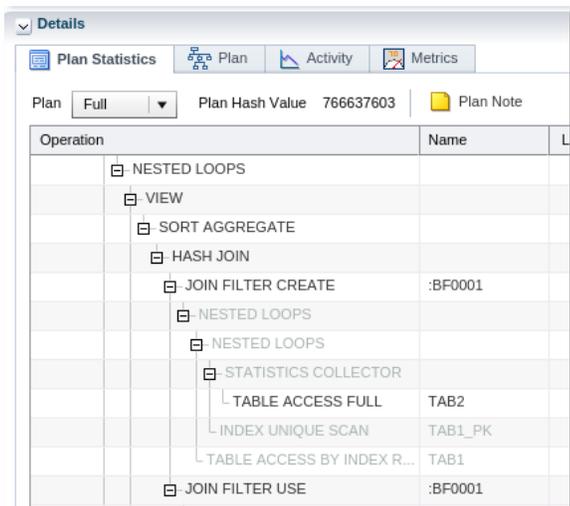

Figure 5: SQL Monitor showing and adaptive plan

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.

Figure 6 shows an example of a SQL Monitor execution plan for a join between EMP and DEPT that is executed in parallel. One set of parallel server processes (producers or pink icons) scan the two tables and send the rows to another set of parallel server processes (consumers or blue icons) that actually do the join. The optimizer has decided to use the HYBRID HASH distribution method. The first table accessed in this join is the DEPT table. The rows coming out of the DEPT table are buffered in the statistics collector, on line 6 of the plan, until the threshold is exceeded or the final row is fetched. At that point the optimizer will make its decision on a distribution method.

Figure 6: SQL Monitor execution plan for hash join between EMP & DEPT that uses adaptive distribution method

To understand which distribution method was chosen at runtime, the easiest way to find this information is to look at the OTHER column in SQL Monitor. This column shows a binocular icon in the lines with PX SEND HYBRID HASH row source. When you click the icon, you can see the distribution method used at runtime.

Figure 7: Hybrid hash distribution method

For the adaptive distribution methods there are three possible values reported in this dialog box: 6 = BROADCAST, 5 = ROUND-ROBIN, and 16 = HASH distribution.

Adaptive Bitmap Index Pruning

When the optimizer generates a star transformation plan, it must choose the right combination of bitmap indexes to reduce the relevant set of ROWIDs as efficiently as possible. If there are many indexes, some of them might not reduce the ROWID set very substantially but will nevertheless introduce significant processing cost during query execution. Adaptive plans are therefore used to prune out indexes that are not significantly filtering down the number of matched rows.

DBMS_XPLAN.DISPLAY_CURSOR will reveal adaptive bitmap pruning in a SQL execution plan with the *adaptive* keyword in a similar manner to the example shown in Figure 3. For example, consider the following SQL execution plan showing the bitmap index CAR_MODEL_IDX being pruned:

```
-----------------------------------------------------------
| Id  | Operation                      | Name          |
-----------------------------------------------------------
|   0 | SELECT STATEMENT               |               |
|   1 |  SORT GROUP BY NOSORT          |               |
|   2 |   NESTED LOOPS                 |               |
|   3 |    BITMAP CONVERSION TO ROWIDS |               |
|   4 |     BITMAP AND                 |               |
|   5 |      BITMAP MERGE              |               |
|   6 |       BITMAP KEY ITERATION     |               |
|   7 |        TABLE ACCESS FULL        | COLORS        |
|   8 |        BITMAP INDEX RANGE SCAN  | CAR_COLOR_IDX |
|-  9 |      STATISTICS COLLECTOR      |               |
|- 10 |       BITMAP MERGE             |               |
|- 11 |        BITMAP KEY ITERATION    |               |
|- 12 |         TABLE ACCESS FULL       | MODELS        |
|- 13 |         BITMAP INDEX RANGE SCAN | CAR_MODEL_IDX |
|  14 |      STATISTICS COLLECTOR      |               |
|  15 |       BITMAP MERGE             |               |
|  16 |        BITMAP KEY ITERATION    |               |
|  17 |         TABLE ACCESS FULL       | MAKES         |
|  18 |         BITMAP INDEX RANGE SCAN | CAR_MAKE_IDX  |
|  19 |    TABLE ACCESS BY USER ROWID  | CARS          |
-----------------------------------------------------------

Note
-----
   - this is an adaptive plan (rows marked '-' are inactive)
```

Figure 8: Example of adaptive bitmap index pruning.

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.

Beginning with Oracle Database 12c Release 1, 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. Also, from Oracle Database 12c Release 1, a new level 11 has 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, such as queries with LIKE predicates and wildcards.

The default dynamic sampling level is 2, so it's likely that when set to level 11, dynamic sampling will kick-in much 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 dynamic sampling queries are held in the database Server Result Cache[2] in Oracle Database 12c Release 1 and in the SQL plan directive repository from Oracle Database 12c Release 2 onwards. This allows multiple SQL statements to share a common set of statistics gathered by dynamic sampling. SQL plan directives, which are discussed in more detail below, also take advantage of this level of dynamic sampling.

Automatic Re-optimization

During the first execution of a SQL statement, an execution plan is generated as usual. During optimization, certain types of estimates that are known to be of low quality (for example, estimates for tables which lack statistics or tables with complex predicates) are noted, and monitoring is enabled for the cursor that is produced. If feedback monitoring is enabled for a cursor by the system, cardinality estimates in the plan are compared to the actual cardinalities seen during execution. If estimates are found to differ significantly from the actual cardinalities, 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 re-optimize a query several times, each time learning more and further improving the plan. Oracle Database 12*c* supports multiple forms of re-optimization.

---

2 For more information on the Server Result Cache, see Oracle documentation: *Database Performance Tuning Guide, Tuning the Result Cache*

Statistics Feedback

Statistics feedback (formally known as cardinality feedback) is one form of re-optimization 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.

Figure 9 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.

```
SQL> select /*+gather_plan_statistics*/ c.cust_first_name, c.cust_last_name, sum(s.amount_sold)
  2  from   customers c, sales s
  3  where c.cust_id=s.cust_id
  4  and    c.cust_city='Los Angeles'
  5  and    c.cust_state_province='CA'
  6  and    c.country_id='US'
  7  and    s.time_id='18-FEB-00'
  8  group by c.cust_first_name, c.cust_last_name;
-----------------------------------------------------------------------------
| Id  | Operation                  | Name      | Starts | E-Rows | A-Rows |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT           |           |      1 |        |      0 |
|   1 |  HASH GROUP BY             |           |      1 |      1 |      0 |
|*  2 |   HASH JOIN                |           |      1 |      1 |      0 |
|   3 |    PARTITION RANGE SINGLE  |           |      1 |      5 |     40 |     Initial Cardinality
|*  4 |     TABLE ACCESS FULL      | SALES     |      1 |      5 |     40 |     estimates are
|*  5 |     TABLE ACCESS FULL      | CUSTOMERS |      1 |      1 |     13 |     more than 8X off
-----------------------------------------------------------------------------
```

Figure 9: Initial execution of a SQL statement that benefits from automatic re-optimization statistics feedback

Where estimates vary greatly from the actual number of rows returned, the 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.

```
SQL> select sql_id, child_number, sql_text, is_reoptimizable
  2  from   v$sql
  3  where  sql_text like 'select /*+gather_plan_statistics*/ c.cust_first_name,%';

SQL_ID          CHILD_NUMBER SQL_TEXT                         IS_REOPTIMIZABLE
--------------- ------------ -------------------------------- ----------------
302pwu6ww1ys4              0 select /*+gather_plan_statisti Y
                             cs*/ c.cust_first_name, c.cust
                             _last_name, sum(s.amount_sold)
                              from   customers c, sales s wh
                             ere c.cust_id=s.cust_id and
                             c.cust_city='Los Angeles' and
                               c.cust_state_province='CA' a
                             nd   c.country_id='US' and    s
                             .time_id='18-FEB-00' group by
                             c.cust_first_name, c.cust_last
                             _name
```

Figure 10: 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.

```
SQL_ID  302pwu6ww1ys4  child number 1
-----------------------------------------
select /*+gather_plan_statistics*/ c.cust_first_name, c.cust_last_name,
sum(s.amount_sold) from  customers c, sales s where c.cust_id=s.cust_id
and    c.cust_city='Los Angeles' and    c.cust_state_province='CA' and
c.country_id='US' and    s.time_id='18-FEB-00' group by
c.cust_first_name, c.cust_last_name

Plan hash value: 3650331407

-------------------------------------------------------------
| Id  | Operation                | Name      | Starts | E-Rows | A-Rows |
-------------------------------------------------------------
|   0 | SELECT STATEMENT         |           |     1  |        |      1 |
|   1 |  HASH GROUP BY           |           |     1  |      1 |      1 |
|*  2 |   HASH JOIN              |           |     1  |      8 |      8 |
|*  3 |    TABLE ACCESS FULL     | CUSTOMERS |     1  |     13 |     13 |
|   4 |    PARTITION RANGE SINGLE|           |     1  |     40 |     40 |
|*  5 |     TABLE ACCESS FULL    | SALES     |     1  |     40 |     40 |
-------------------------------------------------------------
```
Cardinality estimates based on execution statistics are now accurate

```
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
-----
   - statistics feedback used for this statement
```

Figure 11: 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> select sql_id, child_number, sql_text, is_reoptimizable
  2  from   v$sql
  3  where  sql_text like 'select /*+gather_plan_statistics*/ c.cust_first_name,%';

SQL_ID         CHILD_NUMBER SQL_TEXT                        IS_REOPTIMIZABLE
-------------- ------------ ------------------------------- ----------------
302pwu6ww1ys4             0 select /*+gather_plan_statisti  Y
                            cs*/ c.cust_first_name, c.cust
                            _last_name, sum(s.amount_sold)
                             from   customers c, sales s wh
                            ere c.cust_id=s.cust_id and
                            c.cust_city='Los Angeles' and
                             c.cust_state_province='CA' a
                            nd    c.country_id='US' and    s
                            .time_id='18-FEB-00' group by
                            c.cust_first_name, c.cust_last
                            _name

302pwu6ww1ys4             1 select /*+gather_plan_statisti  N
                            cs*/ c.cust_first_name, c.cust
                            _last_name, sum(s.amount_sold)
                             from   customers c, sales s wh
                            ere c.cust_id=s.cust_id and
                            c.cust_city='Los Angeles' and
                             c.cust_state_province='CA' a
                            nd    c.country_id='US' and    s
                            .time_id='18-FEB-00' group by
                            c.cust_first_name, c.cust_last
                            _name
```

Figure 12: New plan generated using execution statistics from initial execution

Performance Feedback

Another form of re-optimization is Performance Feedback, which helps to improve the degree of parallelism chosen for repeated SQL statements when Automatic Degree of Parallelism (AutoDOP)[3] is enabled in adaptive mode (see `OPTIMIZER_ADAPTIVE_STATISTICS` on page 25).

When AutoDOP is enabled in adaptive mode, during the first execution of a SQL statement, the optimizer determines if the statement should execute in parallel and if so what parallel degree should be used. The parallel degree is chosen based on the estimated performance of the statement. Additional performance monitoring is also enabled for the initial execution of any SQL statement the optimizer decides to execute in parallel.

At the end of the initial execution, the parallel degree chosen by the optimizer is compared to the parallel degree computed base on the actual performance statistics (e.g. CPU-time) gathered during the initial execution of the statement. If the two values vary significantly then the statement is marked for re-optimization and the initial execution performance statistics are stored as feedback to help compute a more appropriate degree of parallelism for subsequent executions.

If performance feedback is used for a SQL statement then it is reported in the note section under the plan as shown in Figure 13.

```
| Id | Operation            | Name | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|  0 | SELECT STATEMENT     |      |       |       |       | 357G(100)|           |
|  1 |  MERGE JOIN          |      | 2500T |  66P  |       | 357G(100)|999:59:59 |
|  2 |   SORT JOIN          |      | 100M  | 1621M | 5370M | 3601K  (9)| 00:02:21 |
|  3 |    TABLE ACCESS FULL| EMP  | 100M  | 1621M |       | 71577 (60)| 00:00:03 |
|* 4 |   SORT JOIN          |      | 100M  | 1239M | 4596M | 3121K  (9)| 00:02:02 |
|  5 |    TABLE ACCESS FULL| DEPT | 100M  | 1239M |       | 54434 (47)| 00:00:03 |
-------------------------------------------------------------------------------------

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

   4 - access("EMP"."DEPTNO"="DEPT"."DEPTNO")
       filter("EMP"."DEPTNO"="DEPT"."DEPTNO")

Note
-----
   - automatic DOP: Computed Degree of Parallelism is 1 because of parallel threshold
   - performance feedback used for this statement
```

Figure 13: Execution plan for a SQL statement that was found to run better serial by performance feedback

---

3 For more information on Auto DOP please refer to the whitepaper "Parallel Execution with Oracle Database 12c Fundamentals". See Reference 5.

SQL plan directives

SQL plan directives are automatically created based on information learnt via automatic re-optimization. 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 14).

```
---------------------------------------------------------------
| 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 14: 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 (altered or deleted) using the package DBMS_SPD but 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 15).

```
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 15: Monitoring SQL plan directives automatically created based on information learnt via re-optimization

There a two types of SQL plan directive rows: DYNAMIC_SAMPLING and DYNAMIC_SAMPLING_RESULT. The *dynamic sampling* type 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.

The *dynamic sampling result* type is present from Oracle Database 12c Release 2 onwards and signifies where results from dynamic sampling queries are stored in the SQL directive repository (instead of the Server Result Cache as used by Oracle Database 12c Release 1).

```
SELECT  TO_CHAR(d.directive_id) dir_id, o.object_name,
        o.subobject_name col_name, o.object_type, d.type, d.state, d.reason
FROM    dba_sql_plan_directives d, dba_sql_plan_dir_objects o
WHERE   d.directive_id=o.directive_id
AND     o.owner = 'SPD'
ORDER BY 1,2,3,4,5;


DIR_ID                OBJECT_NAME   COL_NAME       OBJECT_TYPE   TYPE                     STATE    REASON
--------------------- ------------- -------------- ------------- ------------------------ -------- -------------------------------------
1679456716794556153S  SALES                        TABLE         DYNAMIC_SAMPLING_RESULT  USABLE   VERIFY CARDINALITY ESTIMATE
1732576763047032330S  SALES         ASSET_ID       COLUMN        DYNAMIC_SAMPLING         USABLE   SINGLE TABLE CARDINALITY MISESTIMATE
1732576763047032330S  SALES         PAID           COLUMN        DYNAMIC_SAMPLING         USABLE   SINGLE TABLE CARDINALITY MISESTIMATE
1732576763047032330S  SALES                        TABLE         DYNAMIC_SAMPLING         USABLE   SINGLE TABLE CARDINALITY MISESTIMATE
1785294270231606416S  SALES         STATE          COLUMN        DYNAMIC_SAMPLING         USABLE   SINGLE TABLE CARDINALITY MISESTIMATE
1785294270231606416S  SALES                        TABLE         DYNAMIC_SAMPLING         USABLE   SINGLE TABLE CARDINALITY MISESTIMATE
```

Figure 16: Dynamic sampling results stored in the SQL plan directive repository in Oracle Database 12c Release 2 onwards.

SQL plan directives can be used by Oracle to determine if extended statistics[4], 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, the database can automatically create that column group the next time statistics are gathered on the appropriate table. This step is "always on" in Oracle Database 12c Release 1, but from Oracle Database 12c Release 2, it is controlled by the DBMS_STATS preference AUTO_STAT_EXTENSIONS. Note that the default is OFF, so to enable automatic column group creation the following step is required:

```
EXEC DBMS_STATS.SET_GLOBAL_PREFS('AUTO_STAT_EXTENSIONS', 'ON')
```

Extended statistics will 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.

---

4 More information on extended statistics can be found in the paper "Understanding Optimizer Statistics with Oracle Database 12c". See Reference 1.

Optimizer Statistics

Optimizer statistics are a collection of data that describe the database and the objects in it. The optimizer uses these statistics to choose the best execution plan for each SQL statement. Being able to gather the appropriate statistics in a timely manner is critical to maintaining acceptable performance on any Oracle system. With each new release, Oracle strives to provide the necessary statistics automatically.

A summary is presented here, but full details can be found in Reference 1, *Understanding Optimizer Statistics with Oracle Database 12c Release 2.*

New types of histograms

Histograms tell the optimizer about the distribution of data within a column. By default, the optimizer assumes a uniform distribution of rows across the distinct values in a column and will calculate the cardinality for a query with an equality predicate by dividing the total number of rows in the table by the number of distinct values in the column used in the equality predicate. The presence of a histogram changes the formula used by the optimizer to determine the cardinality estimate, and allows it to generate a more accurate estimate. From Oracle 12c Release 1 onwards there are two additional types of histogram, namely, top-frequency and hybrid. They allow the optimizer to derive improved cardinality estimates for more intractable data skews. Prior to Oracle Database 12c Release 1, there were two types of histograms, frequency and height balanced.

Online Statistics Gathering

When an index is created, Oracle automatically gathers optimizer statistics as part of the index creation by piggybacking the statistics gather on the full data scan and sort necessary for the index creation (this has been available since Oracle Database 9i). From Oracle Database 12c Release 1 onwards, the same technique is now applied for direct path operations such as, create table as select (CTAS) and insert as select (IAS) operations into empty tables. Piggybacking the statistics gather as part of the data loading operation, means no additional full data scan is required to have statistics available immediately after the data is loaded. The additional time spent on gathering statistics is small compared to a separate statistics collection process, and it guarantees to have accurate statistics readily available from the get-go.

Incremental Statistics

Gathering statistics on partitioned tables consists of gathering statistics at both the table level (global statistics) and at the (sub)partition level. If data in (sub)partitions is changed in any way, or if partitions are added or removed, then the global-level statistics must be updated to reflect the changes so that there is correspondence between the partition-level and global-level statistics. For large partitioned tables, it can be very costly to scan the whole table to reconstruct accurate global-level statistics. For this reason, incremental statistics were introduced in Oracle Database 11g to address this issue, whereby *synopses* were created for each partition in the table. These data structures can be used to derive global-level statistics – including non-aggregatable statistics such as column cardinality - without scanning the entire table.

Incremental Statistics and Staleness

In Oracle Database 11g, if incremental statistics were enabled on a table and a single row changed in one of the partitions, then statistics for that partition were considered stale and had to be re-gathered before they could be used to generate global level statistics.

 In Oracle Database 12c a new preference called `INCREMENTAL_STALENESS` allows you to control when partition statistics will be considered stale and not good enough to generate global level statistics. By default,

INCREMENTAL_STALENESS is set to NULL, which means partition level statistics are considered stale as soon as a single row changes (same behavior as in Oracle Database 11g).

Alternatively, it can be set to USE_STALE_PERCENT or USE_LOCKED_STATS. USE_STALE_PERCENT means the partition level statistics will be used as long as the percentage of rows changed (in the respective partition or subpartition) is less than the value of the preference STALE_PERCENTAGE (10% by default). USE_LOCKED_STATS means if statistics on a partition are locked, they will be used to generate global level statistics regardless of how many rows have changed in that partition since statistics were last gathered.

Incremental Statistics and Partition Exchange Loads

One of the benefits of partitioning is the ability to load data quickly and easily, with minimal impact on the business users, by using the exchange partition command. The exchange partition command allows the data in a non-partitioned table to be swapped into a specified partition in the partitioned table. The command does not physically move data; instead it updates the data dictionary to exchange a pointer from the partition to the table and vice versa.

In previous releases, it was not possible to generate the necessary statistics on the non-partitioned table to support incremental statistics during the partition exchange operation. Instead statistics had to be gathered on the partition after the exchange had taken place, in order to ensure the global statistics could be maintained incrementally.

In Oracle Database 12c, the necessary statistics (synopsis) can be created on the non-partitioned table prior to the exchange so that, statistics exchanged during a partition exchange load can automatically be used to maintain incrementally global statistics.

More Compact Synopses

The performance for statistics gathering with incremental statistics can come with the price of high disk storage of synopses (they are stored in the SYSAUX tablespace). More storage is required for synopses for tables with a high number of partitions and a large number of columns, particularly where the number of distinct values (NDV) is high. Besides consuming storage space, the performance overhead of maintaining very large synopses can become significant. Oracle Database 12c Release 2 introduces a new algorithm for gathering and storing NDV information, which results in much smaller synopses while maintaining a similar level of accuracy to the previous algorithm.

Concurrent Statistics

In Oracle Database 11g, concurrent statistics gathering was introduced. When the global statistics gathering preference CONCURRENT is set, Oracle employs the Oracle Job Scheduler and Advanced Queuing components to create and manage one statistics gathering job per object (tables and / or partitions) concurrently.

In Oracle Database 12c, concurrent statistics gathering has been enhanced to make better use of each scheduler job. If a table, partition, or sub-partition is very small or empty, the database may automatically batch the object with other small objects into a single job to reduce the overhead of job maintenance.

Automatic Column Group Detection

Extended statistics were introduced in Oracle Database 11g. They help the optimizer improve the accuracy of cardinality estimates for SQL statements that contain predicates involving a function wrapped column (e.g. UPPER(LastName)) or multiple columns from the same table that are used in filter predicates, join conditions, or group-by keys. Although extended statistics are extremely useful it can be difficult to know which extended statistics should be created if you are not familiar with an application or data set.

Auto column group detection, automatically determines which column groups are required for a table based on a given workload. The detection and creation of column groups is a simple three-step procedure[5].

New Reporting Subprograms in DBMS_STATS package

Knowing when and how to gather statistics in a timely manner is critical to maintain acceptable performance on any system. Determining what statistics gathering operations are currently executing in an environment and how changes to the statistics methodology will impact the system can be difficult and time consuming.

In Oracle Database 12c, new reporting subprograms have been added to the `DBMS_STATS` package to make it easier to monitor what statistics gathering activities are currently going on and what impact changes to the parameter settings of these operations will have. The DBMS_STATS subprograms are REPORT_STATS_OPERATIONS, REPORT_SINGLE_STATS_OPERATION and REPORT_GATHER_*_STATS.

Figure 17 shows an example output from the REPORT_STATS_OPERATIONS function. The report shows detailed information about what statistics gathering operations have occurred, during a specified time window. It gives details on when each operation occurred, its status, and the number of objects covered and it can be displayed in either text or HTML format.

```
set long 1000000

select dbms_stats.report_stats_operations(
        since=>systimestamp-1,
        until=>systimestamp,
        detail_level=>'TYPICAL',
        format=>'TEXT') as report
from dual;
```

```
REPORT
--------------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------------
| Operation Id | Operation          | Target   | Start Time        | End Time          | Status    | Total Tasks | Successful Tasks | Failed Tasks | Active Tasks |
--------------------------------------------------------------------------------------------------------------------------------
| 2680         | gather_table_stats | STATS.T3 | 11-AUG-16         | 11-AUG-16         | COMPLETED | 1           | 1                | 0            | 0            |
|              |                    |          | 04.05.45.107179 AM| 04.05.45.162335 AM|           |             |                  |              |              |
|              |                    |          | -07:00            | -07:00            |           |             |                  |              |              |
--------------------------------------------------------------------------------------------------------------------------------
| 2679         | gather_table_stats | STATS.T2 | 11-AUG-16         | 11-AUG-16         | COMPLETED | 1           | 1                | 0            | 0            |
|              |                    |          | 04.05.45.016893 AM| 04.05.45.099127 AM|           |             |                  |              |              |
|              |                    |          | -07:00            | -07:00            |           |             |                  |              |              |
--------------------------------------------------------------------------------------------------------------------------------
| 2678         | gather_table_stats | STATS.T2 | 11-AUG-16         | 11-AUG-16         | COMPLETED | 1           | 1                | 0            | 0            |
|              |                    |          | 04.05.44.930627 AM| 04.05.45.013153 AM|           |             |                  |              |              |
|              |                    |          | -07:00            | -07:00            |           |             |                  |              |              |
--------------------------------------------------------------------------------------------------------------------------------
```

Figure 17: Reporting stats operations.

---

5 For information on creating column groups, see the white paper *Understanding Optimizer Statistics With Oracle Database 12c.*

Optimizer Statistics Advisor

It is well known that inferior statistics cause query performance problems. It is relatively easy to identify stale, out-of-date statistics and missing statistics, but poor quality statistics can be harder to identify: such as inconsistencies between tables and indexes, primary-key/foreign-key relationships and so on.

Inconsistencies in statistics are usually a result of not following recommended approaches, but it is not always easy to strictly adhere to these for a number of reasons. For example, Oracle continuously enhances statistics gathering features but enhancements can be overlooked post-upgrade (a good example is the recommendation to use AUTO_SAMPLE_SIZE rather than fixed percentages). DBAs may use legacy scripts to gather statistics manually so that there is a reluctance to change "proven" procedures. Sometimes statistics gathering can be overlooked and statistics might not be maintained during batch processing and there may be a perceived lack of time in batch windows. There are many "inherited" systems too, where nobody understands the scripts that are used to maintain statistics.

To address these issues, Oracle Database 12.2 includes a new feature called the Optimizer Statistics Advisor. The goal of the advisor is to analyze how statistics are gathered, validate the quality of statistics already gathered and check the status of auto stats gathering (for example, checking for successful completion). To achieve this, it examines the data dictionary with respect to a set of *rules*. Where exceptions to the rules are found, *findings* may be generated and these, in turn, may lead to specific *recommendations*. The advisor will generate a report that lists findings (with the associated "broken" rule), and then list specific *recommendations* to remedy the situation. Finally, the recommendations can be implemented using a set of *actions.* Actions can be output in the form of a SQL script or they can be implemented automatically.

Full details can be found in Reference 2, *Best Practices for Gathering Optimizer Statistics with Oracle Database 12c.*

New and Enhanced Optimization Techniques

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. In Oracle Database 12*c*, several new query optimizations were introduced.

Oracle Database12c Release 1 Onwards

Partial Join Evaluation

Partial join evaluation is an optimization technique that is performed during join order generation. The goal of this technique is to avoid generating duplicate rows that would otherwise be removed by a distinct operator later in the plan. By replacing the distinct operator with an inner join or a semi-join earlier in the plan, the number of rows produced by this step will be reduced. This should improve the overall performance of the plan, as subsequent steps will only have to operate on a reduced set of rows. This optimization can be applied to the following types of query block: `MAX()`, `MIN()`, `SUM (DISTINCT)`, `AVG (DISTINCT)`, `COUNT (DISTINCT)`, `DISTINCT,` branches of the `UNION`, `MINUS`, `INTERSECT` operators, `[NOT] EXISTS` sub queries, etc.

Consider the following `DISTINCT` query:

```
SQL> Select distinct order_id
  2  From    orders o, customers c
  3  Where   o.customer_id = c.customer_id
  4  And     order_id < 2400;
```

In Oracle Database 11g, the join between `ORDERS` and `CUSTOMERS` is a hash join that must be fully evaluated before a unique sort is done to eliminate any duplicate rows.

```
------------------------------------------------------------
| Id | Operation             | Name      | Rows | Cost (%CPU)|
------------------------------------------------------------
|  0 | SELECT STATEMENT      |           |      |  8 (100)|
|  1 |  SORT UNIQUE          |           |  46  |  7  (29)|
|  2 |   HASH JOIN           |           |  46  |  6  (17)|
|  3 |    TABLE ACCESS FULL  | ORDERS    |  46  |  2   (0)|
|  4 |    TABLE ACCESS FULL  | CUSTOMERS |  319 |  3   (0)|
------------------------------------------------------------
```

Figure 18: Oracle Database 11g plan requires complete join between ORDERS & CUSTOMERS with duplicates removed via a unique sort

With partial join evaluation, the join between ORDERS and CUSTOMERS is converted to a semi-join, which means as soon as one match is found for a CUSTOMER_ID in the CUSTOMERS table the query moves on to the next CUSTOMER_ID. By converting the hash join to a semi-join, the number of rows flowing into the HASH UNIQUE is greatly reduced because the duplicates for the same join key have already been eliminated. The plan for the transformed SQL is shown in Figure 19.

```
------------------------------------------------------------
| Id | Operation             | Name      | Rows | Cost (%CPU)|
------------------------------------------------------------
|  0 | SELECT STATEMENT      |           |      |  7 (100)|
|  1 |  HASH UNIQUE          |           |  46  |  7  (29)|
|  2 |   HASH JOIN SEMI      |           |  46  |  6  (17)|
|  3 |    TABLE ACCESS FULL  | ORDERS    |  46  |  2   (0)|
|  4 |    TABLE ACCESS FULL  | CUSTOMERS |  319 |  3   (0)|
------------------------------------------------------------
```

Figure 19: Oracle database 12c plan shows semi join between ORDERS & CUSTOMERS resulting in no duplicates being generated

Null Accepting Semi-joins

It is not uncommon for application developers to add an `IS NULL` predicate to a SQL statement that contains an `EXISTS` sub-query. The additional `IS NULL` predicate is added because the semi join resulting from the `EXISTS` sub-query removes rows that have null values, just like an inner join would. Consider the following query:

```
SQL> Select p.prod_id, s.quantity_sold, s.cust_id
  2  From    products p,sales s
  3  Where   p.prod_list_price > 11
  4  And     p.prod_id = s.prod_id
  5  And     (S.cust_id is NULL
  6            OR Exists ( Select 1
  7                        From    customers c
  8                        Where   c.cust_id = s.cust_id
  9                        And     c.country_id = 'US'));
```

The assumption here is that the column s.cust_id may have null values and we want to return those rows. Prior to Oracle Database12c the `EXISTS` sub-query cannot be un-nested because it appears in an `OR` predicate (disjunction) with the `IS NULL` predicate. Not being able to un-nest the sub-query results in a suboptimal plan where the sub-query is applied as a filter after the join between the `SALES` and `PRODUCTS` table.

```
---------------------------------------------------------------
| Id | Operation            | Name      | Rows  | Cost (%CPU)|
---------------------------------------------------------------
|  0 | SELECT STATEMENT     |           |       | 1536 (100)|
|  1 |  FILTER              |           |       |           |
|  2 |   HASH JOIN          |           | 917K  | 1536  (22)|
|  3 |    TABLE ACCESS FULL | PRODUCTS  |  756  |    9   (0)|
|  4 |    TABLE ACCESS FULL | SALES     | 918K  | 1449  (18)|
|  5 |   TABLE ACCESS FULL  | CUSTOMERS |    1  |  423   (7)|
---------------------------------------------------------------
```

Figure 20: Oracle database 11g plan shows the EXISTS sub-query being applied as a filter after the join.

In Oracle Database 12c a new type of semi-join has been introduced, called a null-accepting semi-join. This new join extends the semi-join algorithm to check for null values in join column of the table on the left hand side of the join. In this case that check would be done on s.cust_id. If the column does contain a null value, then the corresponding row from the `SALES` table is returned, else the semi-join is performed to determine if the row satisfies the join condition. The null-accepting semi-join plan is shown in figure 32 below.

```
-----------------------------------------------------------------
| Id | Operation                | Name      | Rows  | Cost (%CPU)|
-----------------------------------------------------------------
|  0 | SELECT STATEMENT         |           |       | 1816 (100)|
|  1 |  HASH JOIN               |           | 4311  | 1816  (28)|
|  2 |   TABLE ACCESS FULL      | PRODUCTS  |  757  |   12   (9)|
|  3 |   HASH JOIN RIGHT SEMI NA |          | 4316  | 1803  (29)|
|  4 |    TABLE ACCESS FULL     | CUSTOMERS |   33  |    6  (17)|
|  5 |    TABLE ACCESS FULL     | SALES     | 918K  | 1665  (23)|
-----------------------------------------------------------------
```

Figure 21: Oracle database 12c plan shows the EXISTS subquery has been unnested and a null-accepting semi-join is used between customers and sales.

Scalar Sub-query Un-nesting

A scalar sub-query is a sub-query that appears in the `SELECT` clause of a SQL statement. Scalar sub-queries are not un-nested, so a correlated scalar sub-query (one that references a column outside the sub-query) needs to be evaluated for each row produced by the outer query. Consider the following query:

```
SQL> Select c.cust_id, c.cust_last_name, c.cust_city,
  2          (Select avg(s.quantity_sold)
  3           From    sales s
  4           Where  s.cust_id = c.cust_id) avg_quan
  5  From    Customers c
  6  Where   c.cust_credit_limit > 50000;
```

In Oracle Database 11g, for each row in the `CUSTOMERS` table where the `CUST_CREDIT_LIMIT` is greater than 50,000 the scalar sub-query on the `SALES` table must be executed. The `SALES` table is large and scanning it multiple times is very resource intensive.

```
-----------------------------------------------------------
| Id | Operation           | Name     | Rows | Cost (%CPU)|
-----------------------------------------------------------
|  0 | SELECT STATEMENT    |          |      |  426 (100)|
|  1 |  SORT AGGREGATE     |          |   1  |           |
|  2 |   TABLE ACCESS FULL | SALES    | 144  | 1414  (16)|
|  3 | TABLE ACCESS FULL   | CUSTOMERS|   5  |  426   (8)|
-----------------------------------------------------------
```

Figure 22: Oracle database 11g plan shows the scalar subquery has to be evaluated for every row returned from customers table.

Un-nesting the scalar sub-query and converting it into a join would remove the necessity of having to evaluate it for every row in the outer query. In Oracle Database 12c scalar sub-queries can be un-nested and in this example the scalar sub-query on the `SALES` table is converted into a group-by view. The group-by view is guaranteed to return a single row, just as the sub-query was. An outer join is also added to the query to ensure a row from the `CUSTOMERS` table will be returned even if the result of the view is NULL. The transformed query will be as follows,

```
SQL> Select   c.cust_id, c.cust_last_name, c.cust_city, V.avg_quan
   2 From     Customers c,
   3          (Select avg(s.quantity_sold) avg_quan, s.cust_id
   4           From    sales s
   5           Group by s.cust_id) V
   6 Where    c.cust_credit_limit > 50000
   7 And      c.cust_id = V.cust_id (+);
```

```
-----------------------------------------------------------
| Id | Operation           | Name     | Rows  | Cost (%CPU)|
-----------------------------------------------------------
|  0 | SELECT STATEMENT    |          |       | 1805 (100)|
|  1 | HASH GROUP BY       |          |   1   | 1805  (29)|
|  2 | HASH JOIN OUTER     |          |  65   | 1803  (29)|
|  3 |   TABLE ACCESS FULL | CUSTOMERS|   1   |    6  (17)|
|  4 |   TABLE ACCESS FULL | SALES    | 918K  | 1665  (23)|
-----------------------------------------------------------
```

Figure 23: Oracle database 12c plan shows the scalar sub-query has been un-nested with the use of an outer join and group view.

Multi-Table Left Outer Join

Prior to Oracle Database12*c*, having multiple tables on the left of an outer join was illegal and resulted in an ORA-01417 error.

```
SQL> select c.channel_desc, s.amount_sold
  2 from channels c, sales s, costs ct
  3 where c.channel_id = ct.channel_id
  4 and    c.channel_id = s.channel_id(+)
  5 and    ct.prod_id   = s.prod_id (+);
and    c.channel_id = s.channel_id(+)
                 *
ERROR at line 4:
ORA-01417: a table may be outer joined to at most one other table
```

Figure 24: Multi-table left outer join not supported in Oracle Database 11g.

The only way to execute such a query was to translate it into ANSI syntax. However, the implementation of such ANSI syntax results in a *lateral view*[6] being used. Oracle is unable to merge a lateral view, so the optimizer's plan choices are limited in terms of join order and join method, which may result in a sub-optimal plan.

---

6 A lateral view is a view that references columns from a table that is not inside the view.

```
SQL> select c.channel_desc, s.amount_sold
  2  from    channels c inner join sales_returns sr
  3          on (c.channel_id = sr.channel_id)
  4    left outer join sales s
  5          on (c.channel_id = s.channel_id
  6          and sr.prod_id   = s.prod_id)
  7  where  c.channel_id = '9';
```

Lateral view that
cannot be merged

```
----------------------------------------------------------------------------------
| Id | Operation            | Name          | Rows  | Bytes |TempSpc| Cost (%CPU)|
----------------------------------------------------------------------------------
|  0 | SELECT STATEMENT     |               |       |       |       | 47893 (100)|
|  1 |  MERGE JOIN OUTER    |               | 366M  |  13G  |       | 47893  (67)|
|  2 |   SORT JOIN          |               | 114K  | 3028K | 9032K |  3637  (10)|
|  3 |    VIEW              |               | 114K  | 3028K |       |  1436  (17)|
|  4 |     NESTED LOOPS     |               | 114K  | 2131K |       |  1436  (17)|
|* 5 |      TABLE ACCESS FULL| CHANNELS     |    1  |   12  |       |     2   (0)|
|* 6 |      TABLE ACCESS FULL| SALES_RETURNS| 229K  | 1570K |       |  1434  (17)|
|* 7 |   SORT JOIN          |               | 918K  |  10M  |  42M  | 13796   (9)|
|  8 |    TABLE ACCESS FULL | SALES         | 918K  |  10M  |       |  1434  (17)|
----------------------------------------------------------------------------------
```

Figure 25: ANSI syntax results in a plan with a lateral view, which cannot be merged, thus limiting the join order

In Oracle Database 12*c*, multi-table left outer join specified in Oracle syntax (+) are now supported. It is also possible to merge multi-table views on the left hand side of an outer-join. The ability to merge the views enables more join orders and join methods to be considered, resulting in a more optimal plan being selected.

```
SQL> select c.channel_desc, s.amount_sold
  2  from    channels c, sales s, sales_returns sr
  3  where  c.channel_id = '9'
  4  and    c.channel_id = sr.channel_id
  5  and    c.channel_id = s.channel_id(+)
  6  and    sr.prod_id   = s.prod_id(+);
```

```
--------------------------------------------------------------------------------
| Id | Operation             | Name          | Rows  | Bytes  | Cost (%CPU)|
--------------------------------------------------------------------------------
|  0 | SELECT STATEMENT      |               |       |        |  3380 (100)|
|* 1 |  HASH JOIN OUTER      |               | 36469 | 1139K  |  3380  (24)|
|* 2 |   HASH JOIN           |               |  1037 | 20740  |  1688  (24)|
|* 3 |    TABLE ACCESS FULL  | CHANNELS      |     1 |    13  |     2   (0)|
|* 4 |    TABLE ACCESS FULL  | SALES_RETURNS |  2074 | 14518  |  1685  (24)|
|* 5 |   TABLE ACCESS FULL   | SALES         |  2074 | 24888  |  1685  (24)|
--------------------------------------------------------------------------------
```

Figure 26: New multi-table left outer join support allows view merging and results in a more optimal plan

Group-by and Aggregation Elimination

Many applications contain queries that have a form where a group-by query block has a single table that is a group-by view. Under certain conditions, the group-by clauses and aggregate functions of the two query blocks can be eliminated. The resulting query is simpler and contains fewer group-by clauses and aggregate functions.

Grouping and aggregation are expensive operations, and their elimination may lead to a more optimal execution plan. Further, this type of elimination triggers view merging, which, in turn, may result in other optimizations to be applied.

Consider the following example, the outer query is transformed so that is included a single group-by operation rather than two:

```
SELECT v.column1, v.column3, SUM(v.sm)
FROM (SELECT t1.column1, t1.column2, t1.column3, SUM(t1.item_count) AS sm
      FROM    t1, t2
      WHERE   t1.column4 > 3
      AND     t1.id = t2.id
      AND     t2.column5 > 10
      GROUP BY t1.column1, t1.column2, t1.column3) V
GROUP BY v.column1, v.column3;
```

Group-by and Aggregation Elimination →

```
SELECT v.column1, v.column3, SUM(v.sm)
FROM    t1, t2
WHERE   t1.column4 > 3
AND     t1.id = t2.id
AND     t2.column5 > 10
GROUP BY v.column1, v.column3;
```

Figure 27: An example of group-by and aggregation elimination

The corresponding SQL execution plans will be as follows:

```
---------------------------------------------
| Id  | Operation                   | Name  |
---------------------------------------------
|   0 | SELECT STATEMENT            |       |
|   1 |  HASH GROUP BY              |       |
|   2 |   VIEW                      |       |
|   3 |    HASH GROUP BY            |       |
|   4 |     MERGE JOIN              |       |
|*  5 |      TABLE ACCESS BY INDEX ROWID| T1   |
|   6 |       INDEX FULL SCAN       | T1_I  |
|*  7 |      SORT JOIN              |       |
|*  8 |       TABLE ACCESS FULL     | T2    |
---------------------------------------------
```

→

```
---------------------------------------------
| Id  | Operation                   | Name  |
---------------------------------------------
|   0 | SELECT STATEMENT            |       |
|   1 |  HASH GROUP BY              |       |
|   2 |   MERGE JOIN                |       |
|*  3 |    TABLE ACCESS BY INDEX ROWID| T1   |
|   4 |     INDEX FULL SCAN         | T1_I  |
|*  5 |    SORT JOIN                |       |
|*  6 |     TABLE ACCESS FULL       | T2    |
---------------------------------------------
```

Figure 28: SQL execution plans with and without the transformation

Oracle Database12c Release 2 Onwards

Cost-Based OR Expansion Transformation

New in Oracle Database 12.2 is the *cost-based* OR expansion transformation. This transformation is a significant enhancement to the pre-12.2 "OR expansion", which has been available since Oracle Database 9i.

An OR expansion transformation can be used to optimize queries that contain OR clauses (technically known as *disjunctions)*. The basic idea of OR expansion is to transform a query containing disjunctions into the form of a UNION ALL query of two or more branches. This is done by splitting the disjunction into its components and associating each component with a branch of a UNION ALL query. For example:

```
SELECT * FROM prods T1, shops T2
WHERE    (T1.vendid1 = 10 OR T2.vendid2 = 20)
AND      T1.delivery_batch = T2.delivery_batch;
```

OR Transformation →

```
SELECT * FROM prods T1, shops T2
WHERE    T1.vendid1 = 10
AND      T1.delivery_batch = T2.delivery_batch
UNION ALL
SELECT * FROM prods T1, shops T2
WHERE    T2.vendid2 = 20
AND      T1.delivery_batch = T2.delivery_batch
AND      LNNVL(T1.vendid1=10);
```

Note: LNNVL function avoids duplicates

Figure 29: An example of a cost-based OR expansion transformation

OR expansions can enable more efficient access paths (index accesses, partition pruning) and sometimes open up alternative join methods. Prior to Oracle Database 12.2, the transformation is indicated in a SQL execution plan using the CONCATENATION operation, which is semantically equivalent to the UNION-ALL operator. From Oracle

Database 12.2, the UNION-ALL operation will be shown instead, reflecting the underlying changes that have been made to improve the transformation. In particular, there are more opportunities for other transformations to be applied on top of UNION ALL branches (because the algorithm for costing alternative access methods has been improved). Each of the UNION-ALL branches can be executed in parallel (this was not the case with the CONCATENATION operator), so expect to see performance improvements for decision support applications making use of parallel execution.

Consider the following SQL execution plans, comparing Oracle Database 12.1 and 12.2:

Oracle Database 12.1

```
-------------------------------------------------------
| Id  | Operation                  | Name            |
-------------------------------------------------------
|   0 | SELECT STATEMENT           |                 |
|   1 |  SORT AGGREGATE            |                 |
|   2 |   CONCATENATION            |                 |
|*  3 |    HASH JOIN               |                 |
|   4 |     TABLE ACCESS FULL      | T_4K            |
|   5 |     MERGE JOIN CARTESIAN   |                 |
|*  6 |      INDEX FAST FULL SCAN  | T_4K_CONCAT4    |
|   7 |      BUFFER SORT           |                 |
|   8 |       INDEX FAST FULL SCAN | T_10K_HUNDRED   |
|*  9 |    HASH JOIN               |                 |
|  10 |     NESTED LOOPS           |                 |
|* 11 |      TABLE ACCESS FULL     | T_4K            |
|* 12 |      INDEX RANGE SCAN      | T_10K_HUNDRED   |
|* 13 |     INDEX FAST FULL SCAN   | T_4K_CONCAT4    |
-------------------------------------------------------
```

Oracle Database 12.2

```
----------------------------------------------------------------
| Id  | Operation                 | Name               |
----------------------------------------------------------------
|   0 | SELECT STATEMENT          |                    |
|   1 |  SORT AGGREGATE           |                    |
|*  2 |   HASH JOIN               |                    |
|   3 |    VIEW                   | VW_JF_SET$9CE2290B |
|   4 |     UNION-ALL             |                    |
|*  5 |      HASH JOIN            |                    |
|*  7 |       INDEX FAST FULL SCAN| T_4K_CONCAT4       |
|*  9 |       TABLE ACCESS FULL   | T_4K               |
|* 10 |      HASH JOIN            |                    |
|* 11 |       TABLE ACCESS FULL   | T_4K               |
|  12 |       INDEX FAST FULL SCAN| T_4K_CONCAT3       |
|  13 |    INDEX FAST FULL SCAN   | T_10K_HUNDRED      |
----------------------------------------------------------------
```

Figure 30: Comparing Oracle Database 12.1 and 12.2

Note how CONCATENATION has been replaced by UNION-ALL, and in this case an additional transformation has become available to eliminate an additional scan of the T_10K_HUNDRED table.

Sub-query Elimination

Many applications have queries that contain a single-table sub-query in their WHERE clause. Under the following conditions, such a query can be optimized by eliminating the sub-query:

» The sub-query contains a single table.

» The table is also present in the outer query.

» The column involved in the connecting/correlating predicate is the same.

The transformation will eliminate table access paths from the SQL execution plan. For example:

```
SELECT ta.a
FROM   t1 ta
WHERE  ta.b IN
    (SELECT tb.b FROM t1 tb);
```

```
-----------------------------------
| Id | Operation         | Name |
-----------------------------------
|  0 | SELECT STATEMENT  |      |
|* 1 | HASH JOIN SEMI    |      |
|  2 |  TABLE ACCESS FULL| T1   |
|  3 |  TABLE ACCESS FULL| T1   |
-----------------------------------
```

Sub-query Elimination →

```
SELECT ta.a
FROM   t1 ta
WHERE  ta.b IS NOT NULL;
```

```
-----------------------------------
| Id | Operation         | Name |
-----------------------------------
|  0 | SELECT STATEMENT  |      |
|* 1 |  TABLE ACCESS FULL| T1   |
-----------------------------------
```

Figure 31: An example of sub-query elimination

Enhanced Join Elimination

If the result of a query with and without a join is the same, then the join can be eliminated. This is the principle behind this transformation, which relies on primary/unique key and foreign key constraints. Prior to Oracle Database 12.2, the transformation could only be applied to single-column key constraints. From Oracle Database 12.2, the transformation is supported in more case, in particular it is now possible to use the transformation with multi-column key constraints and deferrable constraints under certain conditions.

The transformation is applied iteratively, so that join elimination can trigger further join elimination.

Figure X, illustrates the affect that the transformation will have. In this example, access to the DEPATMENTS table is eliminated:

```
SELECT e.ename, d.deptno, d.org_id
FROM   employees e,
       departments d
WHERE  e.deptno = d.deptno
AND    e.org_id = d.org_id;
```

Join Elimination →

```
SELECT e.ename, e.deptno, e.org_id
FROM   employees e
WHERE  e.deptno is not null
AND    e.org_id is not null;
```

Figure 32: An example of join elimination

Approximate Query Processing

Oracle Database 12c added the new and optimized SQL function, APPROX_COUNT_DISTINCT() to provide approximate count distinct aggregation. Processing of large volumes of data is significantly faster than the exact aggregation, especially for data sets with a large number of distinct values, with negligible deviation from the exact result.

The need to count distinct values is a common operation in today's data analysis. Optimizing the processing time and resource consumption by orders of magnitude while providing almost exact results speeds up any existing processing and enables new levels of analytical insight.

Oracle Database 12.2 extends this functionality to include:

» Approximate versions for percentile and median (APPROXIMATE_PERCENTILE and APPROXIMATE_MEDIAN).

» Materialized view support and query rewrite.

» The ability to use the approximate SQL functions with zero code changes by setting a session-level or system-level database parameter.

Full details can be found in Reference 3, *Analytical SQL in Database 12c Release 2*.

SQL Plan Management

SQL Plan Management (SPM) is an exceptionally important feature to consider for critical applications that require guaranteed SQL execution plan stability. SPM is furthermore fundamental for any database upgrade to evolve execution plans from one optimizer version to another in a controlled manner, managing execution plans and ensuring that only known or verified plans are used.

A number of enhancements have been made to SQL plan management in Oracle Database 12c:

- Automatic Plan Evolution

- Enhanced Auto Capture

- Capture from AWR Repository

These features are covered in detail in Reference 4, *SQL Plan Management with Oracle Database 12c Release 2*.

Initialization Parameters

There are several new initialization parameters that govern the optimizer and its new features in Oracle Database 12c. Below are the details on the new parameters.

OPTIMIZER_ADAPTIVE_FEATURES (Introduced in Oracle Database 12c Release 1, obsolete in Oracle Database 12c Release 2)

This parameter is obsolete in Oracle Database 12c Release 2 and has been superseded by `OPTIMIZER_ADAPTIVE_PLANS` and `OPTIMIZER_ADAPTIVE_STATISTICS`, covered below, to provide customers a more fine grained control mechanism for the optimizer's adaptive features.

In Oracle Database 12c Release 1, the use of adaptive query optimization functionality (including adaptive joins and the creation and use of SQL plan directives) is controlled by the `OPTIMIZER_ADAPTIVE_FEATURES` parameter.

If `OPTIMIZER_ADAPTIVE_FEATURES` is set to `TRUE`, then all of the adaptive query optimization features will be used if `OPTIMIZER_FEATURES_ENABLE` is set to 12.1.0.1 or above.

If `OPTIMIZER_ADAPTIVE_FEATURES` is set to `FALSE` then none of the adaptive query optimization features will be used.

OPTIMIZER_ADAPTIVE_PLANS (New in Oracle Database 12c Release 2, superseding OPTIMIZER_ADAPTIVE_FEATURES)

The use of the adaptive plan functionality is controlled by the `OPTIMIZER_ADAPTIVE_PLANS` parameter. The default value for this parameter is `TRUE`. The features controlled by this parameter are:

» Adaptive joins
» Bitmap pruning
» Parallel distribution method

If `OPTIMIZER_ADAPTIVE_PLANS` is set to `TRUE`, then the adaptive plan features will be used if `OPTIMIZER_FEATURES_ENABLE` is set to 12.1.0.1 or above.

If `OPTIMIZER_ADAPTIVE_PLANS` is set to `FALSE`, then the adaptive plan features will not be used.

OPTIMIZER_ADAPTIVE_STATISTICS (New in Oracle Database 12c Release 2, superseding OPTIMIZER_ADAPTIVE_FEATURES)

The use of the adaptive statistics functionality is controlled by the `OPTIMIZER_ADAPTIVE_STATISTICS` parameter. The default value for this parameter is `FALSE`. The features controlled by this parameter are:

» The use of SQL Plan Directives (SPDs) for query optimization
» Statistics feedback for joins
» Adaptive dynamic sampling for parallel queries
» Performance feedback

If `OPTIMIZER_ADAPTIVE_STATISTICS` is set to `TRUE`, then the adaptive statistics features will be used if `OPTIMIZER_FEATURES_ENABLE` is set to 12.1.0.1 or above.

If `OPTIMIZER_ADAPTIVE_STATISTICS` is set to `FALSE`, then the adaptive statistics features will not be used. SQL plan directives will continue to be created by the optimizer, but they will not be used to refine SQL execution plans with dynamic sampling.

Setting `OPTIMIZER_ADAPTIVE_STATISTICS` to false preserves statistics feedback functionality that was introduced in Oracle Database 11g (where this feature was called cardinality feedback).

OPTIMIZER_ADAPTIVE_REPORTING_ONLY

This parameter is available beginning with Oracle Database 12c Release 1. In order to get a better understand of how many SQL statements will be affect by the new adaptive plans, it is possible to enable adaptive plan in a reporting mode only by setting `OPTIMIZER_ADAPTIVE_REPORTING_ONLY` to `TRUE` (default is `FALSE`). In this mode, information needed to enable adaptive join methods is gathered, but no action is taken to change the plan. This means the default plan will always be used but information is collected on how the plan would have adapted in non-reporting mode. When this parameter is set to TRUE, the decisions taken by the optimizer are revealed using the REPORT formatting parameter in DBMS_XPLAN.DISPLAY_CURSOR. Figure 33 shows an example of how the report can be viewed and, for brevity, the default NESTED LOOPS plan has been edited out:

```
alter session set optimizer_adaptive_reporting_only=true;

select  count(*)
from    sales a,country b
where   a.id = b.id
and     a.id<10001;

SELECT *
FROM table(DBMS_XPLAN.DISPLAY_CURSOR(FORMAT=>'LAST +REPORT'));

...default NESTED LOOPS plan displayed here...

Adaptive plan:
-------------
This cursor has an adaptive plan, but adaptive plans are enabled for
reporting mode only.  The plan that would be executed if adaptive plans
were enabled is displayed below.

Plan hash value: 2609362579

---------------------------------------------------------------------
| Id | Operation          | Name  | Rows | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------
|  0 | SELECT STATEMENT   |       |      |       | 18 (100)|          |
|  1 |  SORT AGGREGATE    |       |    1 |    10 |         |          |
|* 2 |   HASH JOIN        |       |   10 |   100 | 18  (12)| 00:00:01 |
|* 3 |    TABLE ACCESS FULL| SALES |   10 |    50 |  8  (25)| 00:00:01 |
|* 4 |    INDEX RANGE SCAN | I1    |    1 |     5 |  1   (0)| 00:00:01 |
---------------------------------------------------------------------

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

   2 - access("A"."ID"="B"."ID")
   3 - filter("A"."ID"<10001)
   4 - access("B"."ID"<10001)

Note
-----
   - this is an adaptive plan
```

Figure 33: Displaying an adaptive plan report

It is usually more useful to inspect how many SQL execution plans would be affected if adaptive plans were enabled. For example, after setting this parameter to TRUE you can inspect cursors in the cursor cache as follows:

```
set serveroutput on
declare
  cursor si is
    select sql_id,
           child_number
    from  v$sql
    where is_resolved_adaptive_plan = 'Y'
    and   parsing_schema_name not in ('SYS','SYSTEM');
begin
  for r in si
  loop
    for p in (
        select *
        from   table (dbms_xplan.display_cursor(
                      sql_id=>r.sql_id,
                      cursor_child_no=>r.child_number,
                      format=>'+report'))
    )
    loop
       dbms_output.put_line( p.plan_table_output );
    end loop;
  end loop;
end;
/
```

Figure 34: Viewing adaptive plan reports for SQL statements in the cursor cache

OPTIMIZER_DYNAMIC_SAMPLING

Although the parameter `OPTIMIZER_DYNAMIC_SAMPLING` is not new, it does have a new level, 11 that controls the creation of dynamic statistics. When set to level 11 the optimizer will automatically determine which statements would benefit from dynamic statistics, even if all of the objects have statistics.

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.

As always, we hope that by outlining in detail the changes made to the optimizer and statistics in this release, the mystery that surrounds them has been removed and this knowledge will help make the upgrade process smoother for you, as being forewarned is being forearmed!

References

At the time of writing, the Oracle Database 12c Release 2 versions of the following white papers have not been published. They will be available soon, but in the meantime they cover Oracle Database 12c Release 1.

1. Understanding Optimizer Statistics with Oracle Database 12c Release 2
   http://www.oracle.com/technetwork/database/bi-datawarehousing/twp-statistics-concepts-12c-1963871.pdf

2. Best Practices for Gathering Optimizer Statistics with Oracle Database 12c Release 2
   http://www.oracle.com/technetwork/database/bi-datawarehousing/twp-bp-for-stats-gather-12c-1967354.pdf

3. Analytical SQL in Database 12c Release 2
   http://www.oracle.com/technetwork/database/bi-datawarehousing/wp-sqlnaturallanguageanalysis-2431343.pdf

4. SQL Plan Management with Oracle Database 12c Release 2
    http://www.oracle.com/technetwork/database/bi-datawarehousing/twp-sql-plan-mgmt-12c-1963237.pdf

5. Parallel Execution with Oracle Database 12c Fundamentals
   http://www.oracle.com/technetwork/database/bi-datawarehousing/twp-parallel-execution-fundamentals-133639.pdf

Hardware and Software, Engineered to Work Together

Oracle is committed to developing practices and products that help protect the environment