19° ORACLE Database

# The Optimizer In Oracle Database 19c



### 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 19c 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.

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 27 for details.

### **DISCLAIMER**

This document in any form, software or printed matter, contains proprietary information that is the exclusive property of Oracle. Your access to and use of this confidential material is subject to the terms and conditions of your Oracle software license and service agreement, which has been executed and with which you agree to comply. This document and information contained herein may not be disclosed, copied, reproduced or distributed to anyone outside Oracle without prior written consent of Oracle. This document is not part of your license agreement nor can it be incorporated into any contractual agreement with Oracle or its subsidiaries or affiliates.

This document is for informational purposes only and is intended solely to assist you in planning for the implementation and upgrade of the product features described. 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 in this document remains at the sole discretion of Oracle.

Due to the nature of the product architecture, it may not be possible to safely include all features described in this document without risking significant destabilization of the code.

# TABLE OF CONTENTS

Introduction	2
Automatic Indexing	4
Configuring and Monitoring Automatic Indexing	5
Adaptive Query Optimization	8
Adaptive Plans	8
Optimizer Statistics	19
High-frequency Automatic Statistics Gathering	19
Histograms	19
Online Statistics Gathering	19
Real-time Statistics	20
Incremental Statistics	20
SQL Quarantine	22
Optimizer Statistics Advisor	24
SQL Plan Management	25
Enhancements to DBMS Explain Plan	25
Hint Usage Reporting	25
SQL Plan Comparison	26
INITIALIZATION PARAMETER CHANGES	28
Oracle Database 18c made the following initialization parameters available:	28
References	29

# **AUTOMATIC INDEXING**

This Oracle Database 19c new feature is available on certain Oracle Database platforms. Check the Oracle Database Licensing Guide for more information.

### INDEXING CHALLENGES

Indexes are useful for most types of workload and particularly critical for high performance in online transaction processing (OLTP) and operational data store (ODS) environments. However, identifying the best indexes is not straightforward and can require a considerable amount of manual labor. This is largely because index management is a *continuous* process. Applications are rarely static; they change in response to changing business requirements. For this reason, there will be changes in data volumes, of course, but also less obvious changes in data value distributions and the way data is organized, processed and accessed.

Traditionally, DBAs have been responsible to monitoring performance and deciding when and where to add, change or remove indexes in a tactical and often ad-hoc manner. This ad-doc approach to index maintenance is prone to error because it is almost impossible to quantify the effect any change – both positive *and* negative. This may lead to a database that has many more indexes than necessary, where indexes have been gradually added over time and there is a reluctance to remove any of them for fear of negative consequences. This will lead to an increase the system resources required to maintain indexes when data is modified and processed. In addition, over-indexed environments often suffer from less stable SQL execution plans as the sheer number of indexes make the optimizer's choice of index access path more and more finely balanced.

### **AUTOMATIC INDEXING**

Automatic indexing addresses these issues. It is not a simple advisor, but instead it is an expert system that *implements* indexes based on what a performance engineer skilled in index tuning would do. The Oracle Database analyzes the application workload and identifies the queries that will benefit from additional indexes. In other words, it identifies candidate indexes and validates them before implementation, and the entire process is fully automatic.

Here is a summary of the workflow:

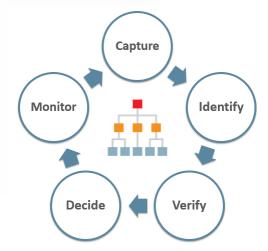


Figure 1 Automatic indexing workflow

The steps can be summarized as follows:

### Capture

 Periodically capture SQL statements from the application workload in a SQL tuning set. This tuning set called the automatic SQL tuning set (ASTS).

### Identify

- Identify candidate indexes that may benefit the application workload.
- Creates unusable and invisible index candidates (this is a data dictionary metadata change only)

### Verify

- The optimizer verifies which index candidates will be used by the captures SQL statements
- · Materialize the successful candidates and measure their effect on the performance of the captured SQL statements
- The indexes remain invisible to the application workload and all verification is done outside the application workflow

### Decide

- Indexes that are found to offer a significant performance improvement are made visible to the application workload.
- Indexes that are found to provide insufficient performance benefits remain invisible to the application workload.
- Automatic indexing offers a mixed solution where necessary. If there is an overall benefit in using a particular auto-index, then it will be
  marked visible. If individual queries suffer from a performance regression, SQL plan baselines are used to prevent them from using
  regressed plans.

### **Monitor**

 SQL performance and index usage is continuously monitored. Indexes that have not been used for a configurable period of time will be dropped.

# **Configuring and Monitoring Automatic Indexing**

Automatic indexing requires little to no manual intervention, but a package called DBMS\_AUTO\_INDEX package is provided for changing a small number of defaults. The feature can be enabled as follows:

```
SQL> exec dbms_auto_index.configure('AUTO_INDEX_MODE', 'IMPLEMENT')
```

And disabled:

```
SQL> exec dbms_auto_index.configure('AUTO_INDEX_MODE', 'OFF')
```

The number of days unused auto-indexes are retained is controlled as follows:

```
SQL> exec dbms_auto_index.configure('AUTO_INDEX_RETENTION_FOR_AUTO', '373')
```

It is possible to specify which schemas are subject to auto indexing:

```
SQL> -- Exclude SH and HR

SQL> exec dbms_auto_index.configure('AUTO_INDEX_SCHEMA', 'SH', FALSE)

SQL> exec dbms_auto_index.configure('AUTO_INDEX_SCHEMA', 'HR', FALSE)

SQL> -- Remove HR from exclusion list

SQL> exec dbms_auto_index.configure('AUTO_INDEX_SCHEMA', 'HR', NULL)

SQL> -- Remove all schemas from exclusion list

SQL> exec dbms_auto_index.configure('AUTO_INDEX_SCHEMA', NULL, TRUE)

SQL> -- Include SH for auto indexing but exclude HR

SQL> exec dbms_auto_index.configure('AUTO_INDEX_SCHEMA', 'SH', FALSE)

SQL> exec dbms_auto_index.configure('AUTO_INDEX_SCHEMA', 'HR', TRUE)
```

All parameter settings (and schemas that have been included and excluded) can be seen as follows:

```
SQL> select * from dba_auto_index_config;
```

A report on auto index activity can be generated. For example:

```
SQL> set linesize 300 trims on pagesize 1000 long 100000
SQL> column report format a120

SQL> SELECT
          dbms_auto_index.report_activity(sysdate-30,null,'text','all','all') report
FROM dual;
```

Here is an extract from an example report:

```
REPORT
GENERAL INFORMATION
                               : 29-DEC-2018 05:09:37
: 28-JAN-2019 05:09:37
 Activity start
 Activity end
                          : 743
 Executions completed
 Executions interrupted : 0
Executions with fatal error : 0
SUMMARY (AUTO INDEXES)
 Index candidates
                                                   : 16
: 2 (2 / 0)
 Indexes created (visible / invisible)
                                                     268.44 MB (268.44 MB / 0 B)
 Space used (visible / invisible)
 Indexes dropped
 SQL statements verified SQL statements improved (improvement factor)
                                                     3 (493.3x)
 SQL plan baselines created
 Overall improvement factor
SUMMARY (MANUAL INDEXES)
 Unused indexes
                    : 0 B
 Space used
 Unusable indexes : 0
INDEX DETAILS

    The following indexes were created:

*: invisible
| Owner | Table | Index
                                          | Key | Type | Properties |
                   SYS_AI_Orn9u2kmxxbs7 | F1
 AUTOI | FACT1 | SYS_AI_151bnmf3xsxaw | F2,F3 | B-TREE | NONE
VERIFICATION DETAILS
1. The performance of the following statements improved:
 Parsing Schema Name : AUTOI
SQL ID
SQL Text
                       : 34xymh9usuxzz
                       : select /* TESTQ3 */ sum(f4) from fact1 where f1 = 10
 Improvement Factor : 2268.9x
PLANS SECTION
- Original
 Plan Hash Value : 4087791341
```

Id   Operation	Name	Rows	Bytes	Cost	Time
0   SELECT STATEMENT   1   SORT AGGREGATE   2   TABLE ACCESS FULL	FACT1	1 98	8   784	51280 51280	00:00:03

# - With Auto Indexes

Plan Hash Value : 911518680

Id   Operation		Name	I I	Rows	Bytes	(	Cost	Time
0   SELECT STATEMENT   1   SORT AGGREGATE   2   TABLE ACCESS BY INDEX ROWID BATCHED   * 3   INDEX RANGE SCAN		FACT1 SYS_AI_0rn9u2kmxxbs7	     	1   1   98   98	8 8 784	     	102   102   3	00:00:01   00:00:01   00:00:01

Predicate Information (identified by operation id):

\* 3 - access("F1"=10)

# 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.

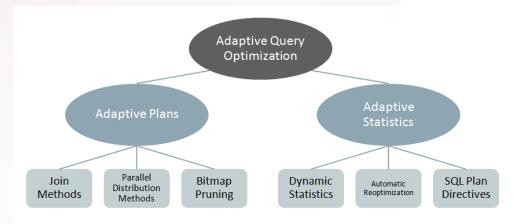


Figure 2: The components that make up the new Adaptive Query Optimization functionality

### **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.

From Oracle Database 12.2 onwards, adaptive plans are enabled and disabled using the database parameter optimizer adaptive plans. Use of the default value of TRUE (enabled) is recommended.

# 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 3, 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 3, 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.

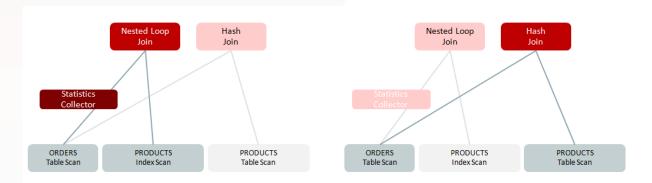


Figure 3: Adaptive execution plan for the join between ORDERS and PRODUCTS. Default plan on the left, chosen plan on the right.

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.

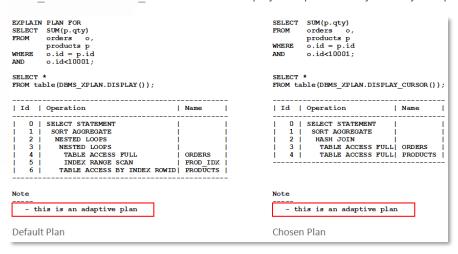


Figure 4: Explain plan and DBMS\_XPLAN.DISPLAY\_CURSOR plan output for the scenario represented in Figure 3.

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).

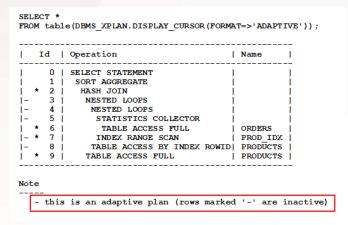


Figure 5: Complete adaptive plan displayed using 'ADAPTIVE' format parameter in DBMS\_XPLAN.DISPLAY\_CURSOR

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 6). If the "Plan Note" icon is clicked, a pop-up box will be displayed confirming that the plan is an adaptive plan.

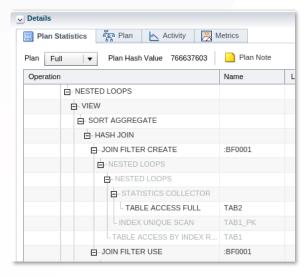


Figure 6: SQL Monitor showing and adaptive plan

### 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 7 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.

Operation	on	Name	Line	Other	Estimated F
∯ ⊟-S	ELECT STATEMENT		0		
§ b	PX COORDINATOR		1		
åå E	PX SEND QC (RANDOM)	:TQ10002	2		
දි <sub>කි</sub>	HASH JOIN BUFFERED		3	<b>€</b>	
<u>25</u>	PX RECEIVE		4		
836	PX SEND HYBRID HASH	:TQ10000	5	æ	
236	- STATISTICS COLLECTOR		6		
226	PX BLOCK ITERATOR		7		
226	TABLE ACCESS FULL	DEPT	8		
88	PX RECEIVE		9		
236 1865	- PX SEND HYBRID HASH	:TQ10001	10	â	
<u>256</u>	PX BLOCK ITERATOR		11		
236	TABLE ACCESS FULL	EMP	12		

Figure 7: 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 8: 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 4. For example, consider the following SQL execution plan showing the bitmap index CAR\_MODEL\_IDX being pruned:

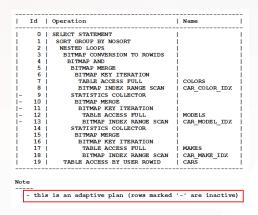


Figure 9: Example of adaptive bitmap index pruning.

### 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.

The use of adaptive statistics can extend SQL statement parse times, which can become an overhead in systems that have high hard parse rates. In addition, SQL execution plans are likely to be more dynamic and less consistent. This feature is disabled by default from Oracle Database 12c Release 2 onwards. Systems that have low hard parse rates and complex queries (particularly ad-hoc queries) may benefit from this feature by setting the database initialization parameter optimizer\_adaptive\_statistics to TRUE.

### 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 was 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 <code>OPTIMIZER\_DYNAMIC\_SAMPLING</code>. 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 and will increase hard parse times.

### **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 19*c* supports multiple forms of re-optimization.

### 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. After the first execution, the optimizer disables monitoring for statistics feedback.

Figure 10 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.

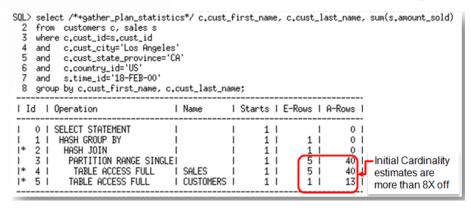


Figure 10: Initial execution of a SQL statement that benefits from automatic re-optimization statistics feedback

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.

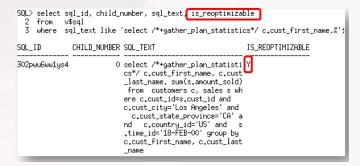


Figure 11: Cursor marked IS\_REOPTIMIZIBLE after initial execution statistics vary greatly from original cardinality estimates

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.

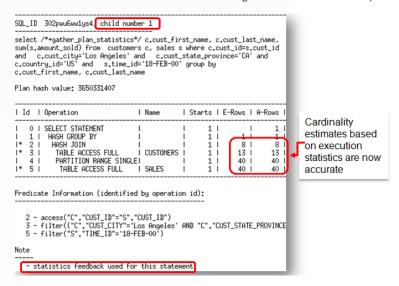


Figure 12: New plan generated using execution statistics from initial execution

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

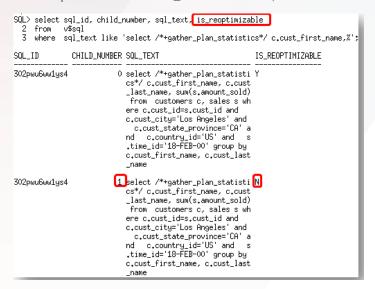


Figure 13: New plan generated using execution statistics from initial execution

### 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 (Auto DOP)<sup>1</sup> is enabled with parallel\_degree\_policy = ADAPTIVE.

When Auto DOP 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 14.

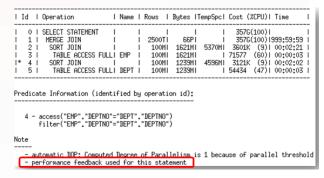


Figure 14: Execution plan for a SQL statement that was found to run better serial by performance feedback

### 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 15).

l Id	Operation	l Name	1.3	Starts	I E-Rows	I A-Rows I
i* 3	I SELECT STATEMENT I HASH GROUP BY I HASH JOIN I TABLE ACCESS FULL I PARTITION RANGE SINGLE I TABLE ACCESS FULL		  -  s    -	1 1 1 1 1	i 13 i 13	i 8 i I 13 I I 40 I
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",""ITME ID"=118-FED-FED-						
Note  - dunamic statistics used; dunamic sampling (level=2)  - 2 Sql Plan Directives used for this statement						

Figure 15: 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 16).

```
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
       from dba_sql_plan_directive_id) dir_id, o.owner, o.object_name
from dba_sql_plan_directives d, dba_sql_plan_dir_objects o
where d.DIRECTIVE_IID=o.DIRECTIVE_ID
and o.owner in ('SH')
order by 1,2,3,4,5;
                                                                          COUNTRY_ID
16334867421200019996
                                                  CUSTOMERS
                                                                                                              COLUMN DYNAMIC SAMPLING NEW
                                                                                                                                                                                  SINGLE TABLE CARDINALITY MISESTIMATE
                                                                          CUST_CITY COLUMN DYNAMIC_SAMPLING NEW
CUST_STATE_PROVINCE COLUMN DYNAMIC_SAMPLING NEW
                                                                                                                                                                                  SINGLE TABLE CARDINALITY MISESTIMATE
SINGLE TABLE CARDINALITY MISESTIMATE
16334867421200019996
                                        SH
                                                  CUSTOMERS
                                                                                                             TABLE DYNAMIC_SAMPLING NEW
COLUMN DYNAMIC_SAMPLING NEW
COLUMN DYNAMIC_SAMPLING NEW
TABLE DYNAMIC_SAMPLING NEW
                                                                                                                                                                                  SINGLE TABLE CARDINALITY MISESTIMATE
SINGLE TABLE CARDINALITY MISESTIMATE
SINGLE TABLE CARDINALITY MISESTIMATE
 16334867421200019996
17286749297683543730
                                                                          CUST_ID
TIME_ID
 17286749297683543730
                                                  SALES
SALES
   286749297683543730
                                                                                                                                                                                  SINGLE TABLE CARDINALITY MISESTIMATE
```

Figure 16: 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
         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;
                                     COL_NAME
DIR_ID
                     OBJECT_NAME
                                                                OBJECT_TYPE
                                                                                          TVDE
                                                                                                                  STATE
                                                                                                                              REASON
16794567167945561532 SALES
17325767630470323308 SALES
                                                                                          DYNAMIC_SAMPLING_RESULT USABLE DYNAMIC_SAMPLING USABLE
                                                                                                                              VERIFY CARDINALITY ESTIMATE
SINGLE TABLE CARDINALITY MISESTIMATE
                                      ASSET_ID
                                                                COLUMN
17325767630470323308 SALES
                                     PAID
                                                                COLUMN
                                                                                          DYNAMIC SAMPLING
                                                                                                                  USABLE
                                                                                                                              SINGLE TABLE CARDINALITY MISESTIMATE
17325767630470323308 SALES
                                                                TABLE
                                                                                          DYNAMIC SAMPLING
                                                                                                                  USABLE
                                                                                                                              SINGLE TABLE CARDINALITY MISESTIMATE
17852942702316064160 SALES
                                     STATE
                                                                COLUMN
                                                                                          DYNAMIC SAMPLING
                                                                                                                  USARLE
                                                                                                                              STNGLE TABLE CARDINALITY MISESTIMATE
17852942702316064160 SALES
```

Figure 17: Dynamic sampling results stored in the SQL plan directive repository in Oracle Database 12c Release 2 onwards.

From Oracle Database 12c Release 2 onwards, SQL plan directives are created but they are not used at parse time to modify SQL execution plans. This prioritizes plan stability over maximizing query execute performance. The optimizer's use of SQL plan directives is controlled by the database initialization parameter optimizer\_adaptive\_statistics, which has the default value of FALSE. This setting is recommended for most systems.

In addition to finding optimal SQL execution plans, SQL plan directives can be used by the database to determine if extended statistics<sup>2</sup>, 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. The recommended (and default) value is OFF. This prioritizes plan stability over maximizing query performance. Automatic column group creation can be enabled using the following step:

```
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.

<sup>2</sup> More information on extended statistics can be found in the paper "Understanding Optimizer Statistics with Oracle Database 19c". 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* 19c.

### **High-frequency Automatic Statistics Gathering**

This Oracle Database 19c new feature is available on certain Oracle Database platforms. Check the Oracle Database Licensing Guide for more information.

Stale statistics can mislead the optimizer and in some cases can result in a sub-optimal plan. To address this, automatic optimizer statistics collection was introduced in Oracle 11g. The automated maintenance task infrastructure schedules statistics gathering to occur in *maintenance windows*. By default, one window is scheduled for each day of the week and statistics collection runs in all predefined maintenance windows.

For systems with rapidly changing datasets, data can drastically change in a short period time, rapidly rendering statistics stale and making daily statistics gathering insufficient.

New to Oracle Database 19c, high-frequency automatic statistics collection is introduced as a *complement* to the existing auto task. It is decoupled from the maintenance window. By configuring the DBMS\_STATS preferences, the DBA can customize the frequency and maximum run time according to their requirement. Note that the existing automatic statistics collection running in the predefined maintenance window will not be affected, and high-frequency automatic statistics task will not start during the maintenance window.

### **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.

### **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). The same technique is 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.

### **Real-time Statistics**

This Oracle Database 19c new feature is available on certain Oracle Database platforms. Check the Oracle Database Licensing Guide for more

Online statistics gathering for create table as select (CTAS) and direct path insert was first introduced in Oracle Database 12c Release 1. This allows statistics to be accurate and available immediately, but prior to Oracle Database 19c, this capability was not applicable to conventional database manipulation language (DML) operations which dominate the way data is manipulated in (for example) traditional OLTP systems.

Real-time statistics extends the online statistic gathering techniques to conventional insert, update and merge DML operations. In order to minimize the performance overhead of generating these statistics, only the most essential optimizer statistics are gathered during DML operations. The collection of the remaining stats (such as the number of distinct values) is deferred to the automatic statistics gathering job, high-frequency stats gathering and/or the manual invocation of the DBMS\_STATS API. See Reference 2, Best Practices for Gathering Optimizer Statistics with Oracle Database 19c, for more information.

### **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

A DBMS\_STATS 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 (the 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.

The Oracle Database allows you to create the necessary statistics (synopses) on load tables before they are exchanged into the partitioned table. This means that global partition statistics can be maintained incrementally the moment that the load table is exchanged with the relevant table partition.

# 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 introduced 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

When the global statistics gathering preference CONCURRENT is set, Oracle employs the Oracle Job Scheduler to create and manage one statistics gathering job per object (tables and / or partitions) concurrently.

### 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<sup>3</sup>.

### **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.

Reporting subprograms in DBMS\_STATS package 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 18 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.

### 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   04.05.45.107179 AM   -07:00	11-AUG-16   04.05.45.162335 AM   -07:00	COMPLETED	1	1	0	0   
2679   	gather_table_stats   	STATS.T2	11-AUG-16   04.05.45.016893 AM   -07:00	11-AUG-16   04.05.45.099127 AM   -07:00	COMPLETED	1	1	0	0   
2678   	gather_table_stats   	STATS.T2	11-AUG-16   04.05.44.930627 AM   -07:00	11-AUG-16   04.05.45.013153 AM   -07:00	COMPLETED	1	1	0	0

Figure 18: Reporting stats operations.

3 For information on creating column groups, see the white paper *Understanding Optimizer Statistics With Oracle Database 19c.* 

# **SQL QUARANTINE**

This Oracle Database 19c new feature is available on certain Oracle Database platforms. Check the Oracle Database Licensing Guide for more information.

### **RUNAWAY SQL STATEMENTS**

Runaway SQL statements can be a threat to database service levels by consuming significant system resources if they remain unchecked. This is a particular problem if multiple runaway queries are running concurrently. The Oracle Database Resource Manager (DBRM) has always offered the ability to terminate a SQL statement that exceed certain resource utilization thresholds (such as CPU and IO) as well as the ability to terminate queries that exceed maximum runtime thresholds. However, this does not prevent repeated execution of runaway queries. A problem query may be run over and over again, each time consuming significant resources before being terminated each time.

# For example:

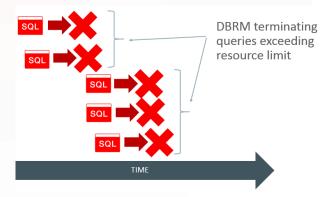


Figure 19 - Runaway SQL statements

For the sake of example, we will assume that the machine CPU utilization looks something like this in response to the runaway queries being executed. In this case, when three queries are running concurrently, the CPU is nearly at 100%:

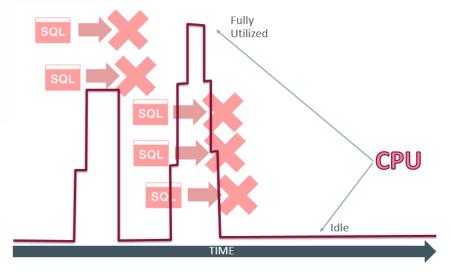


Figure 20 - CPU consumed by runaway SQL

### **SQL QUARANTINE**

The new Oracle Database 19c feature SQL Quarantine can be used to eliminate the overhead of runaway queries. When DBRM detects a SQL statement is exceeding a resource or run-time limit, the SQL execution plan used by the statement is quarantined. If the SQL statement is executed again and it is using the same SQL execution plan then it will be terminated immediately. This can significantly reduce the amount of system resource that would otherwise be wasted. In the following example, there is high utilization when the initial queries are executed, but once they are quarantined they no longer consume system resources because they are terminated prior to execution:

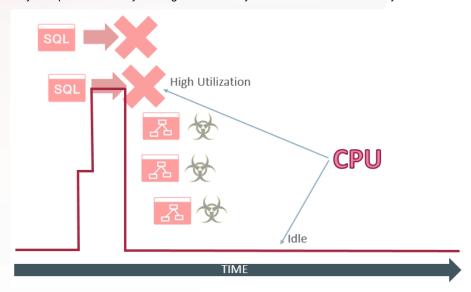


Figure 21 - CPU saved by SQL quarantine

DBRM must be configured to apply a limit. In this case, queries executing for more than 5 seconds will be terminated:

A DBRM exception is raised as follows:

```
SQL> select /* Time consuming query */ * from ...;

ERROR at line 1:

ORA-00040: active time limit exceeded - call aborted
Elapsed: 00:00:07.59
```

Quarantine information is held in memory and flushed to disk periodically. Once the quarantine configuration is persisted in the data dictionary, the following error condition will be raised:

```
SQL> select /* Time consuming query */ * from ...;

ERROR at line 1:
ORA-56955: quarantined plan used
Elapsed: 00:00:00.00
```

# 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 postupgrade (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.

From Oracle Database 12 Release 2, a feature called the Optimizer Statistics Advisor is available. 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 19c.

### 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 were made to SQL plan management in Oracle Database 12c:

- Automatic plan evolution
- Filtered auto capture
- Capture plans from AWR Repository

Oracle Database 18c adds:

Support for a subset of SQL plan management functionality in Oracle Database Standard Edition

Oracle Database 19c adds:

 Prevention of SQL regression with a new default 'AUTO' evolution mode that locates and tests alternative SQL execution plans from multiple sources to find the most optimal.

These features are covered in detail in Reference 4, SQL Plan Management with Oracle Database 19c.

# ENHANCEMENTS TO DBMS EXPLAIN PLAN

# **Hint Usage Reporting**

Hints used in SQL statements can be difficult to 'debug'. For example, it is not always easy to understand why a particular hint is not being used. There is no error reporting for hints, so if they are incorrectly specified in any way, they will be ignored silently.

Hint usage reporting addresses this. It provides detailed information like invalid hints, conflicting hints, ignored hints (e.g. because of the query block location specified) and hints that successfully affected the final plan. Here is an example:

```
EXPLAIN PLAN FOR
SELECT /*+ INDEX(t1) FULL(@sel$2 t1) */ COUNT(*) FROM jobs t1
WHERE t1.job_id IN (SELECT /*+ FULL(t1) */ job_id FROM employees t1);
SELECT * FROM TABLE(DBMS_XPLAN.display(format => 'typical'));
                                                        | Bytes | Cost (%CPU)| Time
l Id
      | Operation
                                  | Name
                                                Rows
    0
         SELECT STATEMENT
                                                       1
                                                                             (34)
                                                                                    00:00:01
          SORT AGGREGATE
                                                       1
                                                               17
     2
           NESTED LOOPS
                                                                          3
2
                                                      19
                                                              323
                                                                             (34)
                                                                                    00:00:01
     3
                                                              963
                                                                               (0)
                                                     107
                                                                                    00:00:01
             SORT UNIQUE
     4
              TABLE ACCESS FULL
                                    EMPLOYEES
                                                     107
                                                              963
                                                                               (0)
                                                                                    00:00:01
 *
     5
             INDEX UNIQUE SCAN | JOB_ID_PK
                                                                8
                                                                                    00:00:01
Hint Report (identified by operation id / Query Block Name / Object Alias): Total hints for statement: 3 (U - Unused (1))
         SEL\$5DA710D3 / "T1"@"SEL\$2" U - FULL(t1) / hint overridden by another in parent query block
                FULL (@sel$2 t1)
         SEL$5DA710D3 / "T1"@"SEL$1"
                INDEX(t1)
```

New format specifiers have been created to support hint reporting:

- HINT\_REPORT
- HINT\_REPORT\_USED
- HINT\_REPORT\_UNUSED

For example:

```
SELECT * FROM TABLE(DBMS_XPLAN.display(format => 'basic +hint_report'));
```

# **SQL Plan Comparison**

The plan comparison feature (hence forth referred to compare plans tool) takes as input a reference plan and an arbitrary list of test plans. It compares the reference plan with each of the test plans in the list and produces a difference report. Such a tool is useful in a variety of use cases:

- New plans: When a plan change occurs, users want to compare it with an old plan (for instance stored in AWR).
- SQL Performance Analyzer (SPA), SQL Tune: How does a plan generated based on a new SQL profile or plan generated through SPA differs from the original plan?
- SQL Plan Management: When an accepted baseline plan is not reproduced by SPM, how does the original plan that is stored in the baseline differs from the generated plan? What are the differences between the different plan baselines captured for the same query?
- Plan generated from a different SQL statement or based on the effect of hints/parameters: How does adding a particular hint or changing the value of a parameter affects the plan?

Plan comparison goes beyond a simple line-by-line comparison because SQL execution plans are *execution trees*. A more complex analysis is required and logical differences must be identified.

Consider two queries:

```
select count(*)
         products p, sales s
p.prod_id = s.prod_id
from
where
and
         p.prod_min_price > 200;
select count(*)
         products p, sales s
p.prod_id = s.prod_id
from
where
         s.quantity_sold = \overline{43};
They can be compared as follows:
VARÍABLE v_rep CLOB
BEGIN
   :v_rep := DBMS_XPLAN.COMPARE_PLANS(
     reference_plan => cursor_cache_object('0hxmvnfkasg6q', NULL),
compare_plan_list => plan_object_list(cursor_cache_object('10dqxjph6bwum', NULL)),
                              => 'TEXT',
=> 'TYPICAL',
     type
     level
                              => 'ALL');
     section
END;
```

# A report can be generated as follows:

```
SET PAGESIZE 50000
SET LONG 100000
SET LINESIZE 210
COLUMN report FORMAT a200
SELECT :v_rep REPORT FROM DUAL;
REPORT
COMPARE PLANS REPORT
 Total number of plans : 2
Number of finding
 Number of findings
COMPARISON DETAILS
                        : 1 (Reference Plan)
 Plan Number
 Plan Found
                        : Yes
 Plan Source
                        : Cursor Cache
 SQL ID
Child Number
                        : Ohxmvnfkasg6q
                        : 0
                        : 19.0.0.0
: "SH"
 Plan Database Version
 Parsing Schema
 SQL Text
                        : select count(*) from products p, sales s where
                        p.prod_id = s.prod_id and s.quantity_sold = 43
Plan
 Plan Hash Value : 3519235612
                               | Name | Rows | Bytes | Cost | Time
| Id | Operation
       SELECT STATEMENT
    0
                                                           469
                                             1
    1
          SORT AGGREGATE
           PARTITION RANGE ALL
                                                           469
                                                                 00:00:01
  *
            TABLE ACCESS FULL | SALES |
                                                          469 | 00:00:01 |
Predicate Information (identified by operation id):
* 3 - filter("S"."QUANTITY_SOLD"=43)
 Plan Number
 Plan Found
 Plan Source
                          Cursor Cache
 SQL ID
                          10dqxjph6bwum
 Child Number
                        : 19.0.0.0
 Plan Database Version
                          "SH"
                        : "SH"
: select count(*) from products p, sales s where
 Parsing Schema
 SQL Text
Plan
 Plan Hash Value : 3037679890
| Id | Operation
                                                             | Rows | Bytes | Cost | Time
    0
        SELECT STATEMENT
    1
          SORT AGGREGATE
  *
                                                               781685 İ
                                                                        10161905
                                                                                      34
                                                                                           00:00:01
           HASH JOIN
            TABLE ACCESS FULL
                                             PRODUCTS
                                                                                           00:00:01
                                                                   61
                                                                             549
```

4   PARTITION RANGE ALL   5   BITMAP CONVERSION TO ROWIDS   6   BITMAP INDEX FAST FULL SCAN   SALES_PROD_BIX	918843   3675372   29   00:00:01     918843   3675372   29   00:00:01
Predicate Information (identified by operation id):	
* 2 - access("P"."PROD_ID"="S"."PROD_ID") * 3 - filter("P"."PROD_MIN_PRICE">200)	
Notes	
- This is an adaptive plan	
Comparison Results (1):	
1. Query block SEL\$1: Transformation JOIN REMOVED FROM QUE only in the reference plan (result query block: SEL\$A43	RY BLOCK occurred D1678).

# INITIALIZATION PARAMETER CHANGES

# Oracle Database 18c made the following initialization parameters available:

optimizer\_ignore\_hints

Setting to TRUE will all the optimizer to ignore hints (other than hints intended to control parallel execution or direct path load).

The default value is FALSE.

optimizer\_ignore\_parallel\_hints

Setting to TRUE will allow the optimizer to ignore hints that control parallel operations (e.g. PARALLEL and NOPARALLEL).

The default value is FALSE.

# **REFERENCES**

- 1. Understanding Optimizer Statistics with Oracle Database 19c
- 2. Best Practices for Gathering Optimizer Statistics with Oracle Database 19c
- 3. SQL Plan Management with Oracle Database 19c

# **ORACLE CORPORATION**

# **Worldwide Headquarters**

500 Oracle Parkway, Redwood Shores, CA 94065 USA

# **Worldwide Inquiries**

+ 1.650.506.7000 + 1.800.ORACLE1

FAX + 1.650.506.7200

oracle.com

### CONNECT WITH US

Call +1.800.ORACLE1 or visit oracle.com. Outside North America, find your local office at oracle.com/contact.



blogs.oracle.com/oracle





# Integrated Cloud Applications & Platform Services

Copyright © 2019, Oracle and/or its affiliates. All rights reserved. This document is provided for information purposes only, and the contents hereof are subject to change without notice. This document is not warranted to be error-free, nor subject to any other warranties or conditions, whether expressed orally or implied in law, including implied warranties and conditions of merchantability or fitness for a particular purpose. We specifically disclaim any liability with respect to this document, and no contractual obligations are formed either directly or indirectly by this document. This document may not be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without our prior written permission. This device has not been authorized as required by the rules of the Federal Communications Commission. This device is not, and may not be, offered for sale or lease, or sold or leased, until authorization is obtained. (THIS FCC DISLAIMER MAY NOT BE REQUIRED. SEE DISCLAIMER SECTION ON PAGE 2 FOR INSTRUCTIONS.)

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group. 0319

Author: Nigel Bayliss



