19c ORACLE Database

SQL Plan Management in Oracle Database 19c

ORACLE WHITE PAPER / MARCH 13, 2019

ORACLE

INTRODUCTION

The performance of any database application heavily relies on consistent query execution. While the Oracle Optimizer is perfectly suited to evaluate the best possible plan without any user intervention, a SQL statement's execution plan can change unexpectedly for a variety of reasons including re-gathering optimizer statistics, changes to the optimizer parameters or schema/metadata definitions. The lack of a guarantee that a changed plan will always be better leads some customers to freeze their execution plans (using stored outlines) or lock their optimizer statistics. However, doing so prevents them from ever taking advantage of new optimizer functionality (like new access paths), that would result in improved plans. An ideal solution to this conundrum would preserve the current execution plans amidst environment changes, yet allow changes only for better plans.

SQL Plan Management (SPM) provides such a framework and allows for complete controlled plan evolution. With SPM the optimizer automatically manages execution plans and ensures that only known or verified plans are used. When a new plan is found for a SQL statement, it will not be used until it has been verified to perform better than the current plan.

This paper provides an in-depth explanation of how SPM works and why it should be a critical part of every DBAs toolkit. The paper is divided into three sections. The first section describes the fundamental aspects of SPM and how they work together to provide plan stability and controlled plan evolution. It then discusses how SPM interacts with other Oracle Database features that influence the optimizer plan selection. The final section provides a more step-by-step guide on how to use SPM to ensure consistent database and application performance during some of the more daunting tasks of a DBA, including upgrades.

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

SQL Plan Management 4

Interaction with Other Performance Features 15

SQL Plan Management Use-Cases 18

Conclusion 23

References 23

SQL PLAN MANAGEMENT

Introduction

SQL plan management (SPM) ensures that runtime performance will not degrade due to execution plan changes. To guarantee this, only accepted execution plans are used; any plan evolution that does occur is tracked and evaluated at a later point in time, and only accepted if the new plan shows a noticeable improvement in runtime. SQL Plan Management has three main components:

Plan Capture:

- Creation of *SQL plan baselines* that store accepted execution plans for all relevant SQL statements. SQL plan baselines are stored in the *SQL management base* in the SYSAUX tablespace.

Plan Selection:

- Ensures only accepted execution plans are used for statements with a SQL plan baseline and records any new execution plans found for a statement as unaccepted plans in the SQL plan baseline.

Plan Evolution:

- Evaluate all unaccepted execution plans for a given statement, with only plans that show a performance improvement becoming accepted plans in the SQL plan baseline.

SQL Management Base

The SQL management base (SMB) is a logical repository in the data dictionary, physically located in the SYSAUX tablespace. In the context of SPM, it stores the following structures:

SQL Plan History

- The SQL plan history is the set of SQL execution plans generated for SQL statements over time.
- The history contains both SQL plan baselines and unaccepted plans.

SQL Plan Baselines

- A SQL plan baseline is an accepted plan that the optimizer is allowed to use for a SQL statement. In the typical use case, the database accepts a plan into the plan baseline only after verifying that the plan performs well.

SQL Statement Log

- A series of query signatures used to identify queries that have been executed more than once during automatic plan capture (see below).

Plan Capture

For SPM to become active, the SQL management base must be seeded with a set of acceptable execution plans, which will become the SQL plan baseline for the corresponding SQL statements. There are two different ways to populate the SQL management base: automatically or manually.

AUTOMATIC PLAN CAPTURE

Automatic plan capture is enabled by setting the init.ora parameter `OPTIMIZER_CAPTURE_SQL_PLAN_BASELINES` to `TRUE` *(*default `FALSE`*)*. When enabled, a SQL plan baseline will be automatically created for any *repeatable* SQL statement provided it doesn't already have one. Repeatable statements are SQL statements that are executed more than once during the capture period. To identify repeatable SQL statements, the optimizer logs the SQL signature (a unique SQL identifier generated from the normalized SQL text) of each SQL statement executed the first time it is compiled. The SQL signatures are stored in the SQL statement log in the SQL management base.

If the SQL statement is executed again, the presence of its signature in the statement log will signify it to be a repeatable statement. A SQL plan baseline is created for the repeatable statements, which includes all of the information needed by the optimizer to reproduce the current cost-based execution plan for the statement, such as the SQL text, outline, bind variable values, and compilation environment. This initial plan will be automatically marked as accepted. If some time in the future a new plan is found for this SQL statement, the execution plan will be added to the SQL plan baseline but will be marked unaccepted.

Automatic plan capture is not enabled by default as it would result in a SQL plan baseline being created for every repeatable SQL statement executed on the system, including all monitoring and recursive SQL statements. On an extremely busy system this could potentially flood the SYSAUX tablespace with unnecessary SQL plan baselines. Note also that the first plan captured for each statement is automatically accepted, even if it isn't the most performant plan. Automatic plan capture should therefore only be enabled when the default plans generated for critical SQL statements are performing as expected.

The Oracle Database includes a way to limit which SQL statements are captured using filters. For example, Figure 1 shows how to target a specific database schema and view the configuration settings in `DBA_SQL_MANAGEMENT_CONFIG`:

```
exec dbms_spm.configure('AUTO_CAPTURE_PARSING_SCHEMA_NAME','SCOTT')

select parameter_name,parameter_value
from   dba_sql_management_config;

PARAMETER_NAME                           PARAMETER_VALUE
---------------------------------------- ----------------
SPACE_BUDGET_PERCENT                     10
PLAN_RETENTION_WEEKS                     53
AUTO_CAPTURE_PARSING_SCHEMA_NAME         SCOTT
AUTO_CAPTURE_MODULE
AUTO_CAPTURE_ACTION
AUTO_CAPTURE_SQL_TEXT
```

Figure 1: Setting and viewing the SPM configuration information

MANUAL PLAN CAPTURE

Manually loading plans into SPM is the most common method to populate SQL plan baselines and is especially useful when a database is being upgraded from a previous version, or when a new application is being deployed. Manual loading can be done in conjunction with or instead of automatic plan capture, and can be done for a single statement or all of the SQL statements in an application. Execution plans that are manually loaded are automatically accepted to create new SQL plan baselines, or added to existing SQL plan baselines as accepted plans. Plans can be manually loaded from four different sources, using either the functions in the `DBMS_SPM` package or through Oracle Enterprise Manager (EM):

- From a SQL Tuning Set
- From the cursor cache
- From the AWR repository (from Oracle Database 12c Release 2)
- Unpacked from a staging table
- From existing stored outlines

From a SQL Tuning Set

A SQL tuning set (STS) is a database object that includes one or more SQL statements, their execution statistics and execution context (which can include the execution plan). You can also export SQL tuning sets from a database and import them into another. One or more plans can be loaded into SPM from an STS using the PL/SQL procedure `DBMS_SPM.LOAD_PLANS_FROM_SQLSET`. The plans manually loaded from the STS will be automatically accepted.

From the Cursor Cache

Plans can be loaded directly from the cursor cache into SQL plan baselines. By applying a filter on the SQL statement text, module name, SQL_ID or parsing schema, a SQL statement or set of SQL statements can be identified and their plans captured using the PL/SQL procedure `DBMS_SPM.LOAD_PLANS_FROM_CURSOR_CACHE`.

From the AWR Repository

SQL plan baselines can be captured directly from the AWR repository using the PL/SQL procedure `DBMS_SPM.LOAD_PLANS_FROM_AWR`. Plans can be loaded between a specified begin and end AWR snapshot and, in common with the other manual approaches, filters can be used to limit which SQL statements are selected to be stored as SQL plan baselines.

Copying SQL Plan Baselines Using a Staging Table

Just as it is possible to transfer optimizer statistics from one database system to another, it is possible to transfer SQL plan baselines via a staging table. SQL plan baselines can be packed into a staging table, on the source system, using the PL/SQL procedure `DBMS_SPM.PACK_STGTAB_BASELINE`. The staging table can then be exported from one system and imported into another, using a database utility like Data Pump. Once the staging table is imported, the SQL plan baselines can be unpacked from the staging table using the PL/SQL procedure `DBMS_SPM.UNPACK_STGTAB_BASELINE`. Once unpacked, the SQL plan baselines will be active and will be used the next time the corresponding SQL statements are executed.

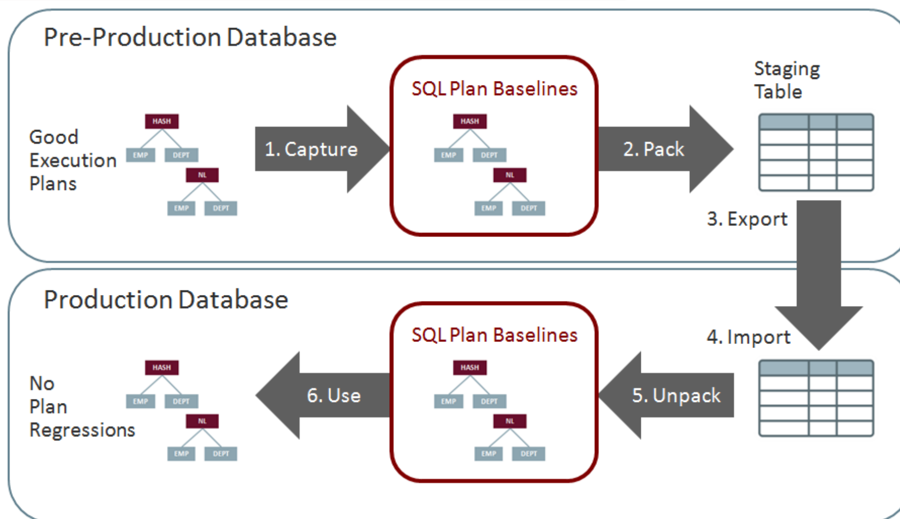

Figure 2: Copying SQL plan baselines from one database to another.

From Existing Stored Outlines

In earlier releases of Oracle Database, stored outlines were the only mechanism available to preserve an execution plan. With stored outlines, only one plan could be used for a given SQL statement, and no plan evolution was possible. Stored outlines were deprecated in Oracle Database 11g, and it is strongly recommended that any existing stored outlines be migrated to SPM.

Stored outlines can be migrated to SQL plan baselines using the PL/SQL procedure `DBMS_SPM.MIGRATE_STORED_OUTLINE`. You can specify stored outlines to be migrated based on their name, category, or associated SQL text, or you can simply migrate all stored outlines in the system.

CAPTURING ADDITIONAL PLANS FOR EXISTING BASELINES

Regardless of which method you use to initially create a SQL plan baseline, any subsequent new plan found for that SQL statement will be added to the plan baseline as an unaccepted plan. This behavior is not dependent on the initialization parameter `OPTIMIZER_CAPTURE_SQL_PLAN_ BASELINES` and will occur even if this parameter is set to `FALSE` (the default). These newly added plans will not be used until the plan has been verified to perform better than the best existing accepted plan in the SQL plan baseline.

Plan Selection

Each time a SQL statement is compiled, the optimizer first uses the traditional cost-based search method to build a best-cost plan. If the initialization parameter `OPTIMIZER_USE_SQL_PLAN_BASELINES` is set to `TRUE` (the default value), then before the cost based plan is executed the optimizer will check to see if a SQL plan baseline exists for this statement. SQL statements are matched to SQL plan baselines using the signature of the SQL statement. A signature is a unique SQL identifier generated from the normalized SQL text (uncased and with whitespaces removed). This is the same technique used by SQL profiles and SQL patches. This comparison is done as an in-memory operation, thus introducing no measurable overhead to any application.

If any accepted SQL plan baselines exists for the SQL statement, the generated cost-based plan is compared to plans in the SQL plan baseline (a match is detected using plan hash values). If a match is found, and the SQL plan baseline is in an accepted state, the optimizer proceeds with this plan. Otherwise, if no match is found, the newly generated plan is added to the SQL plan baseline as an unaccepted plan. It will have to be verified before it can be accepted. Instead of executing the newly generated plan, the optimizer will cost each of the accepted plans for the SQL statement and pick the one with the lowest cost (note that a SQL plan baseline can have more than one accepted plan for a given statement). However, if a change in the system (such as a dropped index) causes all of the accepted plans to become non-reproducible, the optimizer will use the newly generated cost-based plan and will store this plan in the SQL plan history as an unaccepted plan.

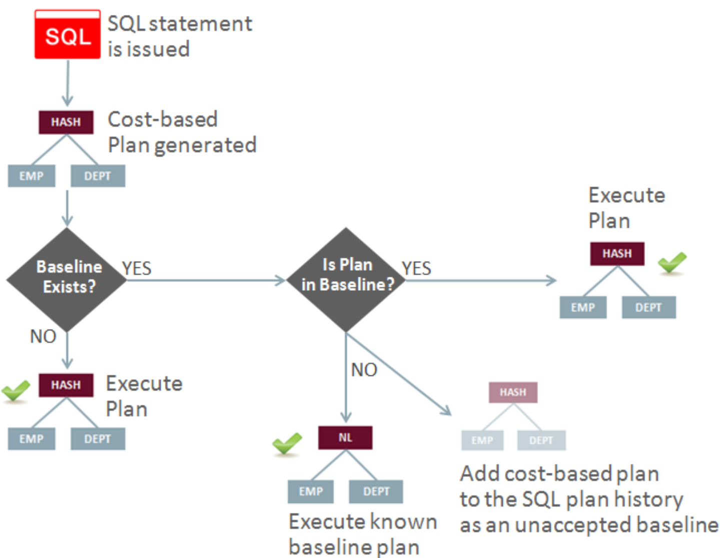

Figure 3: Flowchart of how plan selection works.

It is also possible to influence the optimizer's choice when it is selecting from the plans in a SQL plan baseline. One or more plans in the SQL plan baseline can be marked as fixed. Fixed plans indicate to the optimizer that they are preferred. The optimizer will select the fixed plan with the lowest cost unless none of the fixed plans are reproducible. In that case, the optimizer will cost the remaining (accepted and non-fixed) plans in the SQL plan baselines, and select the one with the lowest cost.

Note that costing an existing plan is not nearly as expensive as a full cost-based optimization. The optimizer is not looking at all possible alternatives, but at specific ones indicated by the plan, like a given access path.

You should also be aware that no new plans are added to a SQL plan baseline that contains a fixed plan, even if a new cost-based plan is found at parse.

Plan Evolution

When the optimizer finds a new plan for a SQL statement, the plan is added to the SQL plan baseline as an unaccepted plan that needs to be verified before it can become an accepted plan. Verification is the process of comparing the execution performances of the non-accepted plan and the best accepted plan (plan with the lowest cost). The execution is performed using the conditions (e.g., bind values, parameters, etc.) in effect at the time the unaccepted plan was added to the SQL plan baseline. If the unaccepted plan's performance is better, it will be automatically accepted otherwise it will remain unaccepted but its `LAST_VERIFIED` attribute will be updated with the current timestamp. Automatic plan evolution (see below) will consider the plan again if at least 30 days has passed since it was last verified (and as long as the SQL is still being executed). You can prevent this by disabling the plan with *dbms_spm.alter_sql_plan_baseline*, but it is recommended that you leave it enabled and continue to allow automatic evolution to prioritize which execution plans to verify. After all, if changes are made to a system then an alternative plan may prove to be beneficial.

The performance criteria used by the evolve process are the same as those used by the SQL Tune Advisor and SQL Performance Analyzer. An aggregate performance statistic is calculated for each test execution based the elapse time, CPU time and buffer gets. The performance statistics are then compared and if the new plan shows a performance improvement of 1.5 over the existing accepted plan, then it will be accepted.

The results of the evolve process are recorded in the data dictionary and can be viewed any time using the `DBMS_SPM.REPORT_EVOLVE_TASK` function.

```
SUMMARY SECTION
-------------------------------------------------------------------------------
  Number of plans processed  : 1
  Number of findings         : 1
  Number of recommendations  : 1
  Number of errors           : 0
-------------------------------------------------------------------------------

DETAILS SECTION
-------------------------------------------------------------------------------
 Object ID          : 2
 Test Plan Name     : SQL_PLAN_b4awtfj8v79r8ae9b4305
 Base Plan Name     : SQL_PLAN_b4awtfj8v79r842949306
 SQL Handle         : SQL_b22b997451b3a6e8
 Parsing Schema     : SH
 Test Plan Creator  : SH
 SQL Text           : select /* q1_group_by */ prod_name, sum(quantity_sold)
                    from products p, sales s where p.prod_id = s.prod_id and
                    p.prod_category ='Girls' group by prod_name

Execution Statistics:
-----------------------------
                    Base Plan                     Test Plan
                    ----------------------------  ----------------------------
 Elapsed Time (s):  .000137                       .000117
 CPU Time (s):      .000089                       .000089
 Buffer Gets:       7                             5
 Optimizer Cost:    16                            10
 Disk Reads:        0                             0
 Direct Writes:     0                             0
 Rows Processed:    5                             5
 Executions:        10                            10


FINDINGS SECTION
-------------------------------------------------------------------------------

Findings (1):
-----------------------------
 1. The plan was verified in 0.05000 seconds. It passed the benefit criterion
    because its verified performance was 1.50892 times better than that of the
    baseline plan.

Recommendation:
-----------------------------
 Consider accepting the plan. Execute
 dbms_spm.accept_sql_plan_baseline(task_name => 'TASK_92', object_id => 2,
 task_owner => 'SPM');
```

Figure 4: Example evolve report

An evolve report describes which plans were tested with the actions performed, and a side-by-side comparison of the performance criteria of each execution. It also has a findings section that clearly explains what happened during the verification process and whether or not the new plan should be accepted. Finally, there is a recommendations section that gives detailed instructions on what to do next, including the necessary syntax. In the evolve report shown in Figure 4, the unaccepted *test plan* performed more than 1.5 times better than the current accepted plan and the recommendation is therefore to accept this plan so that it will be stored as a new SQL plan baseline.

As with plan capture, plan evolution can be done automatically or manually.

AUTOMATIC PLAN EVOLUTION

From Oracle Database 12c Release 1 onward, automatic plan evolution is done by the SPM Evolve Advisor. The SPM Evolve Advisor is an AutoTask (`SYS_AUTO_SPM_EVOLVE_TASK`), which operates during the nightly maintenance window and automatically runs the evolve process for unaccepted plans in SPM. The AutoTask ranks all unaccepted plans in SPM (newly found plans ranking highest) and then runs the evolve process for as many plans as possible before the maintenance window ends.

All of the unaccepted plans that perform better than the existing accepted plan in their SQL plan baseline are automatically accepted. However, any unaccepted plans that fail to meet the performance criteria remain unaccepted and their `LAST_VERIFIED` attribute will be updated with the current timestamp. The AutoTask will not attempt to evolve an unaccepted plan again for at least another 30 days and only then if the SQL statement is active (`LAST_EXECUTED` attribute has been updated). The results of the nightly evolve task can be viewed using the `DBMS_SPM.REPORT_AUTO_EVOLVE_TASK` function.

AUTOMATIC SQL PLAN MANAGEMENT IN ORACLE DATABASE 19C

The SQL plan management evolve advisor is fully automated in Oracle Database 19c. Here is a summary of the new workflow:

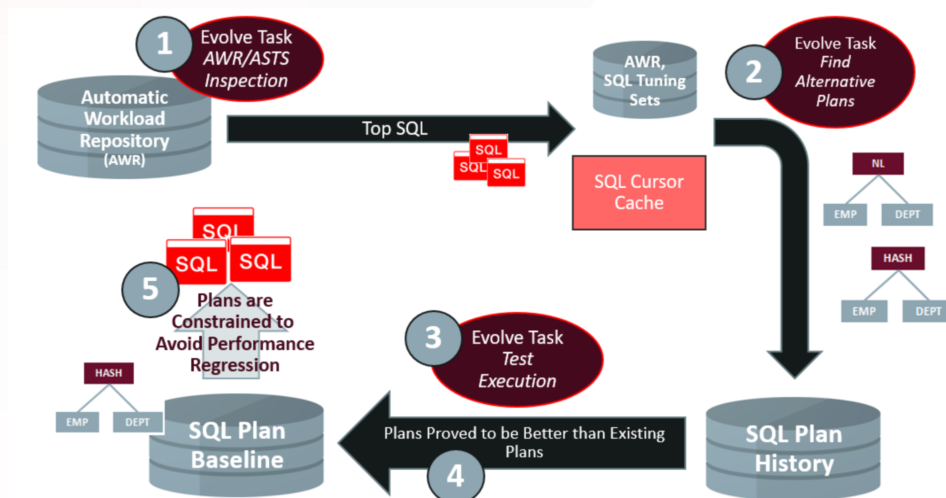

Figure 5 – Automatic SQL plan management

1. The SPM evolve task uses the automatic workload repository (AWR) and, when available, the automatic SQL tuning set (ASTS) used by automatic indexing to determine which SQL statements have a significant impact on system performance.
2. The database cursor cache and various SQL plan repositories are inspected to identify alternative SQL execution plans for statements identified in AWR. All plans found are captured and saved in the SQL plan history.
3. The SQL evolve advisor test executes the captured plans and determines which are best.
4. Plans that are found to perform better than existing plans are added to the SQL plan baseline.
5. Plans are constrained by SQL plan baselines to avoid query regression.

In this way, the Oracle Database will (without manual intervention) determine which SQL execution plans have the best performance and reduce the risk that SQL execution plans with sub-optimal performance will be used.

This behavior is controlled using two parameters:

- ALTERNATE_PLAN_BASELINE new default AUTO
- ALTERNATE_PLAN_SOURCE new default AUTO

ALTERNATE_PLAN_SOURCE set to AUTO means that SPM evolution will automatically identify where to search for alternative SQL execution plans for SQL statements. The ALTERNATE_PLAN_BASELINE parameter determines which SQL statements are eligible for plan capture. AUTO means that any SQL statement will be a candidate (subject to its presence in AWR). These parameters were available prior to Oracle Database 19c, but internal functionality has been enhanced for this release.

Auto SPM captures and accepts SQL plan baselines so it can be used as an alternative to auto plan capture. Nevertheless, it can be used along-side existing plan capture methods if there is a requirement (for example) to capture and plan baseline entire application workloads.

These parameters were available prior to Oracle Database 19c, but internal functions have been developed and enhanced in this release. The pre-19c defaults for the parameters are:

- ALTERNATE_PLAN_BASELINE default 'EXISTING'
- ALTERNATE_PLAN_SOURCE default 'CURSOR_CACHE+AUTOMATIC_WORKLOAD_REPOSITORY'

The parameter settings can be changed for the SQL evolve advisor task as follows:

```
BEGIN
   DBMS_SPM.SET_EVOLVE_TASK_PARAMETER(
      task_name => 'SYS_AUTO_SPM_EVOLVE_TASK' ,
      parameter => 'ALTERNATE_PLAN_BASELINE',
      value     => 'EXISTING');
END;
/
```

Parameter values can be viewed as follows:

```
SELECT PARAMETER_NAME, PARAMETER_VALUE
FROM   DBA_ADVISOR_PARAMETERS
WHERE  TASK_NAME = 'SYS_AUTO_SPM_EVOLVE_TASK';
```

MANUAL PLAN EVOLUTION

Alternatively, it is possible to evolve an unaccepted plan manually using Oracle Enterprise Manager or the supplied package `DBMS_SPM`. From Oracle Database 12c Release 1 onwards, the original SPM evolve function `(DBMS_SPM.EVOLVE_SQL_PLAN_BASELINE)` has been deprecated in favor of a new API that calls the SPM evolve advisor. Figure 6 shows the steps needed to invoke the SPM evolve advisor. It is typically a three step processes beginning with the creation of an evolve task. Each task is given a unique name, which enables it to be executed multiple times. Once the task has been executed, you can review the evolve report by supplying the *task name* and *execution name* to the `DBMS_SPM.REPORT_EVOLVE_TASK` function.

```
variable tname   varchar2(50)
variable exename varchar2(50)

tname  := DBMS_SPM.create_evolve_task(sql_handle => 'SQL_34afe9373762137a');
exename:= DBMS_SPM.execute_evolve_task(task_name => :tname);

SELECT DBMS_SPM.report_evolve_task(
               task_name      => :tname,
               execution_name => :exename) AS output
FROM   dual;
```

Figure 6: Invoking the SPM evolve advisor manually

When the SPM evolve advisor is manually invoked, the unaccepted plan(s) is not automatically accepted even if it meets the performance criteria. The plans must be manually accepted using the `DBMS_SPM.ACCEPT_SQL_PLAN_BASELINE` procedure. The evolve report contains detailed instructions, including the specific syntax, to accept the plans.

```
execute dbms_spm.accept_sql_plan_baseline(task_name => :tname);
```

Figure 7: Manually accepting a plan

Note that the ‘Administer SQL Management Object’ privilege is required to manually evolve plans.

If you want to prevent automatic evolution from accepting plans (i.e. you want to manage SQL plan baselines manually), then you can prevent plans from being accepted automatically as follows:

```
BEGIN
   DBMS_SPM.SET_EVOLVE_TASK_PARAMETER(
      task_name => 'SYS_AUTO_SPM_EVOLVE_TASK' ,
      parameter => 'ACCEPT_PLANS',
      value     => FALSE);
END;
/
```

Additionally, if you are managing SQL plan baselines manually in Oracle Database 19c, then you may want to avoid capturing SQL plan history entries for SQL statements that don't have SQL plan baselines already:

```
BEGIN
   DBMS_SPM.SET_EVOLVE_TASK_PARAMETER(
      task_name => 'SYS_AUTO_SPM_EVOLVE_TASK' ,
      parameter => 'ALTERNATE_PLAN_BASELINE',
      value     => 'EXISTING');                    /* The default is AUTO */
END;
/
```

EXISTING allows the SQL evolve advisor to find alternative plans for SQL plan baselines in SQL repositories, but it will only do this for SQL statements that have existing SQL plan baselines. If ACCEPT_PLANS is set to FALSE (for the auto evolve advisor, as shown above), any alternative plans found will be kept in the SQL plan history and they will not be accepted automatically.

Managing and Monitoring SQL Plan Baselines

All aspects of managing and monitoring SQL plan baselines can be done through Oracle Enterprise Manager or the PL/SQL packages `DBMS_SPM`, `DBMS_XPLAN` and the DBA view `DBA_SQL_PLAN_BASELINES`.

To get to the SQL plan baseline page:

- Access the Database Home page in Enterprise Manager.
- At the top of the page, click on the performance tab and select the SQL Plan Control from the drop down list.
- The SQL Plan Control page appears.
- At the top of the page, click on the SQL Plan Baseline table to display the SQL plan baseline subpage.

Figure 8: SPM home page in Oracle Enterprise Manager

INITIALIZATION PARAMETERS

There are two initialization parameters that control SPM.

OPTIMIZER_CAPTURE_SQL_PLAN_BASELINES Controls the automatic creation of new SQL plan baselines for repeatable SQL statements. This parameter is set to FALSE by default. Note it is not necessary for this parameter to be set to TRUE in order to have a newly found plan added to an existing SQL plan baseline.

If a SQL statement has multiple plans then all of them will be captured, but only the first will be accepted. If you do not which to use plan baselines during auto capture, you can set *optimizer_use_sql_plan_baselines* to FALSE.

OPTIMIZER_USE_SQL_PLAN_BASELINES controls the use of SQL plan baselines. When enabled, the optimizer checks to see if the SQL statement being compiled has a SQL plan baseline before executing the cost-based plan determined during parse. If a SQL plan baseline is found and the cost-based plan is an accepted plan in that baseline, then the optimizer will go ahead and use that plan. However, if a SQL plan baseline is found and the cost-based plan is not an accepted plan in that baseline, then it will be added to the SQL plan baseline but not executed. The optimizer will cost each of the accepted plans in the SQL plan baseline and pick the one with the lowest cost. This parameter is TRUE by default. When set to FALSE the optimizer will only use the cost-based plan determined during parse (SQL plan baselines will be “ignored”) and no new plans will be added to existing SQL plan baselines.

These parameter values can be changed on the command line either at a session or system level using an alter session or alter system command. It is also possible to adjust the parameter setting on the upper left hand side of the main SQL plan baseline page (the Settings section) in Enterprise Manager.

MANAGING THE SPACE CONSUMPTION OF SQL MANAGEMENT BASE

The statement log and all SQL plan baselines are stored in the SQL Management Base. The SQL Management Base is part of the database dictionary, stored in the SYSAUX tablespace; this is the tablespace for all internal persistent information outside the dictionary and cannot be changed. By default, the space limit for the SQL Management Base is no more than 10% of the size of the SYSAUX tablespace. However, it is possible to change the limit to any value between 1% and 50% using the PL/SQL procedure DBMS_SPM.CONFIGURE or Enterprise Manager. A weekly background process measures the total space occupied by the SQL Management Base, and when the defined limit is exceeded, the process will generate a warning in the alert log, for example:

```
SPM: SMB space usage (99215979367) exceeds 10.000000% of SYSAUX size (1018594954366).
```

Reaching the limit will not prevent new plans from being added to existing SQL plan baselines or new SQL plan baselines from being added to the SQL Management Base.

There is also a weekly scheduled purging task (operated by MMON) that manages the disk space used by SPM inside the SQL Management Base. The task runs automatically and purges any plans that have not been used for more than 53 weeks, by running the DBMS_SPM.DROP_SQL_PLAN_BASELINE function on each one. It is possible to change the unused plan retention period using either using DBMS_SPM.CONFIGURE or Enterprise Manager; its value can range from 5 to 523 weeks (a little more than 10 years).

The SQL Management Base is stored entirely within the SYSAUX tablespace, so SPM will not be used if this tablespace is not available.

MONITORING SQL PLAN BASELINES

The view DBA_SQL_PLAN_BASELINES displays information about the current SQL plan baselines in the database. The same information is displayed at the bottom of the SQL plan baseline page in Oracle Enterprise Manager.

```
SQL> SELECT sql_text,sql_handle, plan_name, enabled, accepted
     FROM   dba_sql_plan_baselines;

SQL_TEXT                                  SQL_HANDLE           PLAN_NAME                      ENA ACC
----------------------------------------- -------------------- ------------------------------ --- ---
select * from (select  s_store_name       SQL_c12de7a8908e48eb SQL_PLAN_c2bg7p288wk7b18498f6e YES YES
      ,sum(ss_net_profit

select * from (select  s_store_name       SQL_c12de7a8908e48eb SQL_PLAN_a2bg9p243ee7b27218e19 YES NO
      ,sum(ss_net_profit
```

Figure 9: Monitoring SQL plan baselines using the dictionary view DBA_SQL_PLAN_BASELINES

In the example in Figure 9, the same SQL statement has two plans in its SQL plan baseline. Both plans were automatically captured but only one of the plans (SQL_PLAN_c2bg7p288wk7b18498f6e) will be used by the optimizer, as it is the only plan that is both enabled and accepted. The other plan is an unaccepted plan, which means it won't be used until it has been verified.

To check the detailed execution plan for any SQL plan baseline, click on the plan name on the SQL plan baseline page in Enterprise Manager or use the procedure DBMS_XPLAN.DISPLAY_SQL_PLAN_BASELINE. In Oracle Database 11g, executing this function will trigger a compilation of the SQL statement using the information stored about the plan in the SQL Management Base. This is also the case in Oracle Database 19c if the plan was created in Oracle Database 11g. From Oracle Database 12c Release 1 onwards, Oracle captures the actual plan rows when adding a new plan to SPM. This means that DISPLAY_SQL_PLAN_BASELINE will display the actual plan recorded when the plan was added to the SQL plan baseline.

Capturing the actual execution plans ensures that if a SQL plan baseline is moved from one system to another, the plans in the SQL plan baseline can still be displayed even if some of the objects used in it or the parsing schema itself does not exist on the new system. This means that a plan can still be displayed even if it cannot be reproduced.

Figure 10 shows the execution plan for the accepted plan from Figure 9. The plan shown is the actual plan that was captured when this plan was added to the SQL plan baseline because the attribute 'Plan rows' is set to 'From dictionary'. For plays displayed based on an outline, the attribute 'Plan rows' is set to 'From outline'.

```
SQL> select *
     from
         dbms_xplan.display_sql_plan_baseline(
                                       sql_handle=>'SQL_c12de7a8908e48eb',
                                       plan_name=>' SQL_PLAN_c2bg7p288wk7b18498f6e');

PLAN_TABLE_OUTPUT
-----------------
--------------------------------------------------------------------------------
SQL handle: SQL_3abe0ddc4497528e
SQL text: select avg(ss_quantity)      ,avg(ss_ext_sales_price)
      ,avg(ss_ext_wholesale_cost)         ,sum(ss_ext_wholesale_cost)   from
...
--------------------------------------------------------------------------------

--------------------------------------------------------------------------------
Plan name: SQL_PLAN_3pghdvj29fnnf9b76af9e      Plan id: 2608246686
Enabled: YES     Fixed: NO    Accepted: YES      Origin: AUTO-CAPTURE
Plan rows: From dictionary
--------------------------------------------------------------------------------
```

Figure 10: Displaying one of the accepted plans from a SQL plan baseline

It is also possible to check whether a SQL statement is using a SQL plan baseline by looking in V$SQL. If the SQL statement is using a SQL plan baseline, the plan_name, from the SQL plan baseline, will appear in the sql_plan_baseline column of V$SQL. You can join the V$SQL view to the DBA_SQL_PLAN_BASELINES view using the plan_name columns or use the exact_matching_signature column.

```
SQL> Select s.sql_text, b.plan_name, b.origin, b.accepted
  2  From dba_sql_plan_baselines b, v$sql s
  3  Where s.exact_matching_signature = b.signature
  4  And   s.SQL_PLAN_BASELINE = b.plan_name;

SQL_TEXT             PLAN_NAME                      ORIGIN         ACC
-------------------- ------------------------------ -------------- ---
select /*LOAD_AUTO*/ SQL_PLAN_6zsnd8f6zsd9g54bc8843 AUTO-CAPTURE   YES
 * from sh.sales whe
re quantity_sold > 4
0 order by prod_id
```

Figure 11: Joining V$SQL to DBA_SQL_PLAN_BASELINES

INTERACTION WITH OTHER PERFORMANCE FEATURES

SPM is not the only feature in the Oracle Database that influences the optimizer's choice of execution plan. This section describes in detail how SPM interacts with the other plan-influencing features in the Oracle Database. Bear in mind that SPM is a very conservative approach for guaranteeing plan stability and that level of conservatism doesn't always allow other features to fully exert themselves.

SQL Plan Baselines and Adaptive Plans

Adaptive plans, introduced in Oracle Database 12c, enable the optimizer to defer the final plan decision for a statement until execution time. The optimizer instruments its chosen plan (the default plan) with statistics collectors so that it can detect at runtime, if its cardinality estimates differ greatly from the actual number of rows seen by the operations in the plan. If there is a significant difference, then the plan or a portion of it will be automatically adapted to avoid suboptimal performance on the first execution of a SQL statement.

More detail can be found in Reference 1, *Optimizer with Oracle Database 19c*.

SPM Plan Capture and Adaptive Plans

When automatic plan capture is enabled and a SQL statement that has an adaptive plan is executed, only the final plan used will be captured in the SQL plan baseline.

However, if the optimizer finds a new adaptive plan for a SQL statement that has an existing SQL plan baseline, then only the initial or default plan will be recorded in the SQL plan baseline. The recorded plan will be marked adaptive, so the optimizer can take that into consideration during the evolve process.

```
SQL> select sql_handle, sql_text, plan_name, origin, adaptive, accepted from dba_sql_plan_baselines;

SQL_HANDLE           SQL_TEXT                                 PLAN_NAME                        ORIGIN         ADA ACC
-------------------- ---------------------------------------- -------------------------------- -------------- --- ---
SQL_68c2d91a38d0d1f9 select /*+ gather_plan_statistics*/      SQL_PLAN_6jhqt38wd1ngt17328d86   AUTO-CAPTURE   YES NO
                      product_name
                     from order_items2 o, product_in

SQL_68c2d91a38d0d1f9 select /*+ gather_plan_statistics*/      SQL_PLAN_6jhqt38wd1ngt8aa6928d   MANUAL-LOAD    NO  YES
                      product_name
                     from order_items2 o, product_in
```

Figure 12: DBA_SQL_PLAN_BASELINES shows a newly added plan with the adaptive attribute set to TRUE

SPM Plan Selection and Adaptive Plans

Accepted plans in a SQL plan baseline are never adaptive. Thus, the plan selected for a SQL statement with a SQL plan baseline will never be adaptive.

SPM Plan Evolution and Adaptive Plans

When evolving a plan that has been marked adaptive, the optimizer determines all of the possible subplans before the test executions begin. Once executing, the optimizer will decide which subplan to use based on the execution statistics recorded in the statistics collectors. The performance of the final plan used during the test execution is compared to the existing accepted plan, in the SQL plan baseline. If the final plan performs better than the existing accepted plan, it will be added to the plan baseline as an accepted plan, and will replace the non-accepted adaptive plan that was verified. Since the newly added plan is the final plan, it is no longer marked adaptive.

SQL Plan Baselines and Adaptive Cursor Sharing

Since Oracle Database 11g, the optimizer has allowed multiple execution plans to be used for a single statement with bind variables at the same time. This functionality is called Adaptive Cursor Sharing (ACS)[1]and relies on the monitoring of execution statistics to ensure the correct plan is used for each bind value.

On the first execution the optimizer will peek the bind value(s) and determine the execution plan based on the bind value's selectivity. The cursor will be marked bind sensitive if the optimizer believes the optimal plan may depend on the value of the bind variable (for example, a histogram is present on the column or the predicate is a range, or <, >). When a cursor is marked bind sensitive, Oracle monitors the behavior of the cursor using different bind values, to determine if a different plan is called for.

Oracle's adaptive cursor sharing is built on an optimistic approach: if a different bind value is used in a subsequent execution, the optimizer will use the existing cursor and execution plan because Oracle initially assumes the cursor can be shared. However, the execution statistics for the new bind value will be recorded and compared to the execution statistics for the previous value. If Oracle determines that the new bind value caused the data volumes manipulated by the query to be significantly different it "adapts" and hard parses based on the new bind value on its next execution and the cursor is marked bind-aware. Each bind-aware cursor is associated with a selectivity range of the bind so that the cursor is only shared for a statement when the bind value in the statement is believed to fall within the range.

1 More information on Adaptive Cursor Sharing can be found in Closing the Query Loop in Oracle 11g

When another new bind value is used, the optimizer tries to find a cursor it thinks will be a good fit, based on similarity in the bind value's selectivity. If it cannot find such a cursor, it will create a new one. If the plan for the new cursor is the same as an existing cursor, the two cursors will be merged to save space in the shared pool. And the selectivity range for that cursor will be increased to include the selectivity of the new bind.

If a SQL plan baseline exists for a SQL statement that benefits from ACS, then all possible execution plans for that SQL statement should be accepted and enabled in the SQL plan baseline. Otherwise the presence of the SQL plan baseline may prevent the SQL statement from taking full advantage of ACS. Below is a brief description of how the different aspects of SPM interact with ACS.

SPM Plan Capture and Adaptive Cursor Sharing

When automatic plan capture is enabled and a SQL statement that benefits from ACS is executed twice, then only one of the possible plans for that statement will be captured initially. Any additional plans found for the statement won't be used because they will not be accepted. Note, new plans are only added to the SQL plan baseline at hard parse and ACS will not trigger additional hard parses for a SQL statement that has only one accepted plan in its SQL plan baseline. (See SPM plan selection and Adaptive Cursor Sharing section below for more details).

It is highly recommended that all possible plans for a SQL statement that benefits from ACS are manually loaded in to a SQL plan baseline directly from the cursor cache (in the shared pool). This ensures that all of the plans will be automatically accepted and available for use. Figure 13 shows a SQL statement that has been executed multiple times, using different bind variable values, that has two distinct plans (child cursor 1 & 2) in the cursor cache. The procedure `DBMS_SPM.LOAD_PLANS_FROM_CACHE` is then used to capture both plans as accepted plans in the SQL plan baseline.

```
SQL> Select sql_id, child_number, is_bind_sensitive S, is_bind_aware A
  2  From   v$sql
  3  Where  sql_text like 'select /*ACS_1%';

SQL_ID        CHILD_NUMBER S A
------------- ------------ - -
272gr4hapc9w1            0 Y N
272gr4hapc9w1            1 Y Y
272gr4hapc9w1            2 Y Y

SQL>
SQL> exec :cnt :=dbms_spm.load_plans_from_cursor_cache(sql_id=>'272gr4hapc9w1');

PL/SQL procedure successfully completed.

SQL>
SQL> Select sql_handle, sql_text, plan_name, origin, accepted
  2  From   dba_sql_plan_baselines;

SQL_HANDLE           SQL_TEXT                  PLAN_NAME                      ORIGIN        ACC
-------------------- ------------------------- ------------------------------ ------------- ---
SQL_9463acadb5d62fd9 select /*ACS_1*/ count(*) SQL_PLAN_98sxcpquxcbyt10205ee7 MANUAL-LOAD   YES
                     , max(empno)
                     from emp
                     where deptno = :deptno

SQL_9463acadb5d62fd9 select /*ACS_1*/ count(*) SQL_PLAN_98sxcpquxcbytc392520a MANUAL-LOAD   YES
                     , max(empno)
                     from emp
                     where deptno = :deptno
```


Figure 13: Manual capture of multiple plans found by ACS for a SQL statement

SPM Plan Selection and Adaptive Cursor Sharing

When a SQL statement that is a candidate for ACS has a SQL plan baseline, which contains all desirable plans for the statement and they are all accepted, then ACS will behave as expected. The correct plan will be used for the different bind variables used in the statement.

However, if the SQL plan baseline has only one accepted plan, then ACS is automatically disabled for this statement. Recall that a cursor will be marked bind sensitive only if the optimizer believes the optimal plan may depend on the value of the bind variable. If SPM allows only one plan to be chosen for the query, then the cursor will not be marked bind sensitive.

SQL Plan Baselines versus SQL Profiles

A SQL profile contains auxiliary information that prevents a sub-optimal plan from otherwise being chosen due to defects in the usual inputs (statistics, bind variable values etc.) used by the optimizer . SQL profiles don't constrain the optimizer to any specific plan, which is why they can be shared. SQL plan baselines, on the other hand, constrain the optimizer to only select from a set of accepted plans. The cost-based approach is still used to choose a plan, but only within this set of plans. SQL plan baselines are a more conservative plan selection strategy than SQL profiles.

The presence of a SQL profile affects all three components of SPM. Below is a brief description of each of these interactions.

SPM Plan Capture and SQL Profiles

When a SQL statement is executed it will be hard parsed and a cost based plan will be generated. This plan will be influenced by the SQL profile. Once the cost based plan is determined it will be compared to the plans that exist in the SQL plan baseline. If the plan matches one of the accepted plans in the SQL plan baseline, the optimizer will use it. However, if the cost based plan doesn't match any of the accepted plans in the SQL plan baseline it will be added to the plan baseline as an unaccepted plan.

SPM Plan Selection and SQL profiles

When a SQL statement with a SQL plan baseline is parsed, the accepted plan with the best cost will be chosen. This process uses the optimizer's cost model. The presence of a SQL profile will affect the estimated cost of each of these plans and thus potentially the plan that is finally selected.

SPM Plan evolution and SQL profiles

The evolution process test-executes the unaccepted plan against the best of the accepted plans. The best accepted plan is selected based on cost. If a SQL profile exists for the statement, it will influence the estimated cost and thus the accepted plan chosen for comparison against the unaccepted plan.

SQL Plan Baselines versus Stored Outlines

Stored outlines were the predecessors to SQL plan baselines and due to this they share some similarities. A stored outline consists of a set of hints that specifies the execution plan to generate for a particular SQL statement. When a SQL statement with a stored outline is executed, the optimizer uses the hints in the outline to reproduce the plan, rather than using the standard cost-based approach.

Stored outlines were deprecated in Oracle Database 11g in favor of SQL plan baselines and are no longer supported.

Integration with Automatic SQL Tuning Advisor

From Oracle Database 11g onwards, the SQL Tuning Advisor automatically runs during the maintenance window. This automatic SQL tuning task targets high-load SQL statements. These statements are identified by the execution performance data collected in the Automatic Workload Repository (AWR) snapshots. If the SQL Tuning Advisor finds a better execution plan for one of the high-load SQL statements, it will recommend a SQL profile for that statement.

If a SQL profile recommendation made by the automatic SQL tuning task is implemented, and the SQL statement already has a SQL plan baseline, then the execution plan found by the SQL Tuning Task will be added as an accepted plan in the SQL plan baseline.

The SQL Tuning Advisor can also be invoked manually, by creating a SQL Tuning Set for a given SQL statement. If the SQL Tuning Advisor recommends a SQL profile for the statement and it is manually implemented then the tuned plan will be added as an accepted plan to the SQL statement's plan baseline if one exists.

SQL PLAN MANAGEMENT USE-CASES

The controlled plan evolution offered by SPM can be extremely useful when dealing with some of the more daunting tasks a DBA must face on a production database environment. This final section of the whitepaper describes, with the use of clear how-to examples, how SPM can be used to tackle database upgrades, new application or module roll-outs, and the correction of regressed SQL statements.

Using SPM for Upgrades

Bulk loading execution plans from a SQL Tuning Set (STS) is an excellent way to guarantee no plan changes as part of a database upgrade. The following four steps are all it takes:

1. Before upgrading create an STS that includes the execution plan for each of the critical SQL statements in the current database environment. Note the attribute parameter, of the `DBMS_SQLTUNE` procedure used to populate the STS, must be set to ALL to capture the execution plans.

```
SQL> BEGIN
  2  sys.dbms_sqltune.create_sqlset(sqlset_name=>'SPM_STS', sqlset_owner=>'SPM');
  3  END;
  4  /

PL/SQL procedure successfully completed.

SQL>
SQL> DECLARE
  2     stscur   dbms_sqltune.sqlset_cursor;
  3  BEGIN
  4     OPEN stscur FOR
  5       SELECT VALUE(P)
  6       FROM   TABLE(dbms_sqltune.select_cursor_cache(
  7              'sql_text like ''select /*LOAD_STS*/%''',
  8              null, null, null, null, null, null, 'ALL')) P;
  9
 10     -- populate the sqlset
 11     dbms_sqltune.load_sqlset(sqlset_name      => 'SPM_STS',
 12                              populate_cursor  => stscur,
 13                              sqlset_owner     => 'SPM');
 14  END;
 15  /

PL/SQL procedure successfully completed.
```


Figure 14: Set attribute parameter to ALL when creating STS to capture the execution plan

2. Pack the STS into a staging table and export the staging table into a flat file.

3. Import the staging table into latest version of the Oracle Database and unload the STS or simple upgrade the existing database. An STS will persist across an upgrade.

4. Use EM or `DBMS_SPM.LOAD_PLANS_FROM_SQLSET` to load the execution plans into the SQL Management Base

```
SQL> variable cnt number
SQL>
SQL> execute :cnt := dbms_spm.load_plans_from_sqlset( -
>                    sqlset_name  => 'SPM_STS', -
>                    basic_filter => 'sql_text like ''select /*LOAD_STS*/%''');

PL/SQL procedure successfully completed.
```

Figure 15: Loading a plan for a SQL Tuning Set into a SQL plan baselines

Using SPM for New Application or Module Deployments

The deployment of a new application module means the introduction of a completely new set of SQL statements into the database. From Oracle Database 11g onwards, any 3rd party software vendor can ship their application software along with the appropriate SQL plan baselines for the new SQL being introduced. This guarantees that all SQL statements that are part of the SQL plan baseline will initially run with the plans that are known to give good performance under a standard test configuration. Alternatively, if an application is developed or tested in-house, the correct plans can be exported from the test system and imported into production using the following steps:

1. On the test or development system, create a staging table using the `DBMS_SPM.CREATE_STGTAB_BASELINE` procedure.

```
SQL> exec DBMS_SPM.CREATE_STGTAB_BASELINE(table_name => 'MY_STGTAB', table_owner => 'SPM');

PL/SQL procedure successfully completed.
```

2. Pack the SQL plan baselines you want to export from the SQL management base into the staging table using the `DBMS_SPM.PACK_STGTAB_BASELINE` function.

```
SQL> exec :cnt := dbms_spm.pack_stgtab_baseline(table_name => 'MY_STGTAB', table_owner => 'SPM', enabled=>'YES', accepted=>'YES');

PL/SQL procedure successfully completed.
```

3. Export the staging table into a flat file using the export command or Oracle Data Pump.

4. Transfer this flat file to the target system.

5. Import the staging table from the flat file using the import command or Oracle Data Pump.

6. Unpack the SQL plan baselines from the staging table into the SQL management base on the target system using the `DBMS_SPM.UNPACK_STGTAB_BASELINE` function.

```
SQL> exec :cnt := dbms_spm.unpack_stgtab_baseline(table_name => 'MY_STGTAB', table_owner => 'SPM', enabled=>'YES', accepted=>'YES');

PL/SQL procedure successfully completed.
```

Using SPM to Correct Regressed SQL Statements

Loading plans directly from the cursor cache can be extremely useful if an existing application has to be tuned manually using hints. It is unlikely the application code can be changed to incorporate the necessary hints, but any tuned execution plan can be captured as a SQL plan baseline, outside the application context, so you can ensure that the application SQL will use the desired tuned plan in the future without the need to change the application.

By using the simple steps below you can use SPM to capture the hinted execution plan and associate it with the non-hinted SQL statement. You begin by capturing a SQL plan baseline for the non-hinted SQL statement.

1. Find the `SQL_ID` for the statement in the `V$SQL` view and use it to create a SQL plan baseline for the statement using `DBMS_SPM.LOAD_PLAN_FROM_CURSOR_CACHE`.

```
SQL> SELECT sql_id, sql_fulltext
  2  FROM   v$sql
  3  where  sql_text like '%SELECT      p.prod_name%';

SQL_ID        SQL_FULLTEXT
------------- ---------------------------------------------
bn5p8hp266tah SELECT   p.prod_name, sum(s.amount_sold) amt
              FROM     Sales s, Products p
              WHERE

SQL> variable cnt number;
SQL>
SQL> execute :cnt :=dbms_spm.load_plans_from_cursor_cache(sql_id=>'bn5p8hp266tah');

PL/SQL procedure successfully completed.
```

2. The plan that was captured is the sub-optimal plan and it will need to be disabled using `DBMS_SPM.ALTER_SQL_PLAN_BASELINE`. The `SQL_HANDLE & PLAN_NAME` are required to disable the plan and can found by looking in `DBA_SQL_PLAN_BASELINE` view.

```
SQL> exec :cnt :=DBMS_SPM.ALTER_SQL_PLAN_BASELINE(SQL_HANDLE =>'SQL_10ed3803a09c8fe1', -
>                                                  PLAN_NAME =>'SQL_PLAN_11v9s0fh9t3z1c47b6be0', -
>                                                  ATTRIBUTE_NAME => 'enabled', -
>                                                  ATTRIBUTE_VALUE => 'NO');

PL/SQL procedure successfully completed.

SQL>
SQL> select sql_handle, sql_text, plan_name, enabled from dba_sql_plan_baselines where sql_text like '%SELECT   p.prod_name%';

SQL_HANDLE                     SQL_TEXT                                    PLAN_NAME                      ENA
------------------------------ ------------------------------------------- ------------------------------ ---
SQL_10ed3803a09c8fe1           SELECT   p.prod_name, sum(s.amount_sold) amt SQL_PLAN_11v9s0fh9t3z1c47b6be0 NO
                               FROM     Sales s, Products p
                               WHERE
```

3. Now you need to modify the SQL statement using the necessary hints & execute the modified statement.

```
SQL> SELECT      /*+ INDEX(p) */ p.prod_name, sum(s.amount_sold) amt
  2  FROM        Sales s, Products p
  3  WHERE       s.prod_id=p.prod_id
  4  AND         p.supplier_id = :sup_id
  5  group by p.prod_name;

PROD_NAME                                                     AMT
-------------------------------------------------- ----------
Bounce                                                  244595.65
Comic Book Heroes                                        101214.6
Envoy External 6X CD-ROM                                645586.12
Finding Fido                                             78881.08
Model K8822S Cordless Phone Battery                     582640.54
```

4. Find the `SQL_ID` and `PLAN_HASH_VALUE` for the hinted SQL statement in the `V$SQL` view.

```
SQL> SELECT sql_id, plan_hash_value, sql_fulltext
  2  FROM   v$sql
  3  where  sql_text like '%SELECT      /*+ INDEX(p)%';

SQL_ID        PLAN_HASH_VALUE SQL_FULLTEXT
------------- --------------- ------------------------------------------------------------
cn29d9b5wp9u7       903671040 SELECT sql_id, plan_hash_value, sql_fulltext
                              FROM   v$sql
                              where  sql_text like '

ac7jyxhg9mj0c       187119048 SELECT   /*+ INDEX(p) */ p.prod_name, sum(s.amount_sold) amt
                              FROM     Sales s, P
```

5. Using the `SQL_ID` and `PLAN_HASH_VALUE` for the modified plan, create a new accepted plan for original SQL statement by associating the modified plan to the original statement's `SQL_HANDLE`.

```
SQL> exec :cnt:=dbms_spm.load_plans_from_cursor_cache(sql_id =>'ac7jyxhg9mj0c', -
>                                                      plan_hash_value => 187119048, -
>                                                     sql_handle =>'SQL_10ed3803a09c8fe1');

PL/SQL procedure successfully completed.
```

6. Now if you query DBA_SQL_PLAN_BASELINES you will see two plans in the SQL plan baseline, the original non-hinted plan and the new hinted plan. Only the hinted plan is accepted, so this will be the plan chosen next time the SQL statement is executed.

```
SQL> SELECT sql_handle, sql_text, plan_name, enabled
  2  FROM   dba_sql_plan_baselines
  3  WHERE  sql_text like '%SELECT      p.prod_name%';

SQL_HANDLE                     SQL_TEXT                                     PLAN_NAME                      ENA
------------------------------ -------------------------------------------- ------------------------------ ---
SQL_10ed3803a09c8fe1           SELECT   p.prod_name, sum(s.amount_sold) amt SQL_PLAN_11v9s0fh9t3z1aa1ba510 YES
                               FROM     Sales s, Products p
                               WHERE

SQL_10ed3803a09c8fe1           SELECT   p.prod_name, sum(s.amount_sold) amt SQL_PLAN_11v9s0fh9t3z1c47b6be0 NO
                               FROM     Sales s, Products p
                               WHERE
```

CONCLUSION

The performance of any database application heavily relies on consistent query execution. SQL Plan Management enables you to preserve the current execution plans for your critical SQL statements amidst environment changes, only allowing them to change for the better.

With SPM, the optimizer automatically manages execution plans and uses only known or verified plans. When a new plan is found for a SQL statement, it will not be used until it has been verified to perform better than the existing accepted plan.

By taking such a conservative approach, SPM allows DBAs to tackle some of the more daunting tasks they must face on a production environment, such as upgrades; safe in the knowledge they will have consistent query plans throughout.

REFERENCES

1. Oracle white paper: *Optimizer with Oracle Database 19c*

ORACLE CORPORATION

Worldwide Headquarters
500 Oracle Parkway, Redwood Shores, CA 94065 USA

Worldwide Inquiries
TELE + 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.

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

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

ORACLE®