# SQL Plan Management

NYOUG Meeting 2011

**Sameer Malik**
Tata Consultancy Services
sameer18july@gmail.com

# Introduction

- Optimizer not being able to guarantee a plan will change always for the better has led some customers to freeze their execution plans (using Stored Outlines) or lock their optimizer statistics. However, doing so prevents such environments from ever taking advantage of new optimizer functionality or access paths, which would improve the SQL statements performance.

- Being able to preserve the current execution plan amidst environment changes and **allowing changes only for the better would be the ultimate solution.**

- Oracle database 11g solves this problem with introduction of brand new feature “SQL Plan Management" (SPM). It provides a framework for completely transparent controlled execution plan evolution. With SPM the optimizer automatically manages execution plans and ensures 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 by the database to have comparable or better performance than the current plan.

# SQL Plan Management

The SQL Plan Management has three main phases:

## 1. SQL plan baseline capture:

You can capture SQL Plan baselines by either having the database automatically capture the plan or you can manually load them using various techniques described later in the presentation. The baselines contain 2 sets of plans, one which are accepted are stored in SQL Plan baseline, and non-accepted plan are stored in the SQL Plan history.

## SQL plan baselines: -

A SQL plan baselines are set of all accepted plans in the plan history that the database maintains for each repeatable SQL statement that the database executes. The database defines a plan as acceptable when it verifies that the plan doesn't lead to performance regression when compared to the other plan in the plan history. The very first execution plan the database generates for a SQL statement is always considered acceptable by the optimizer and becomes the original SQL plan baseline as well as the plan history for the statement. Later execution plans will not be included in the SQL baseline unless the database verifies that they don't lead to performance regression.

## Plan history: -

Database maintains plan history for repeatable SQL statements and not for adhoc SQL statements, the Plan history contains all the plans generated for the specific SQL over time, it includes all information used by the optimizer when figuring out an optimal execution plan, including information regarding the SQL text, bind variables and the environment in which the SQL is being executed.

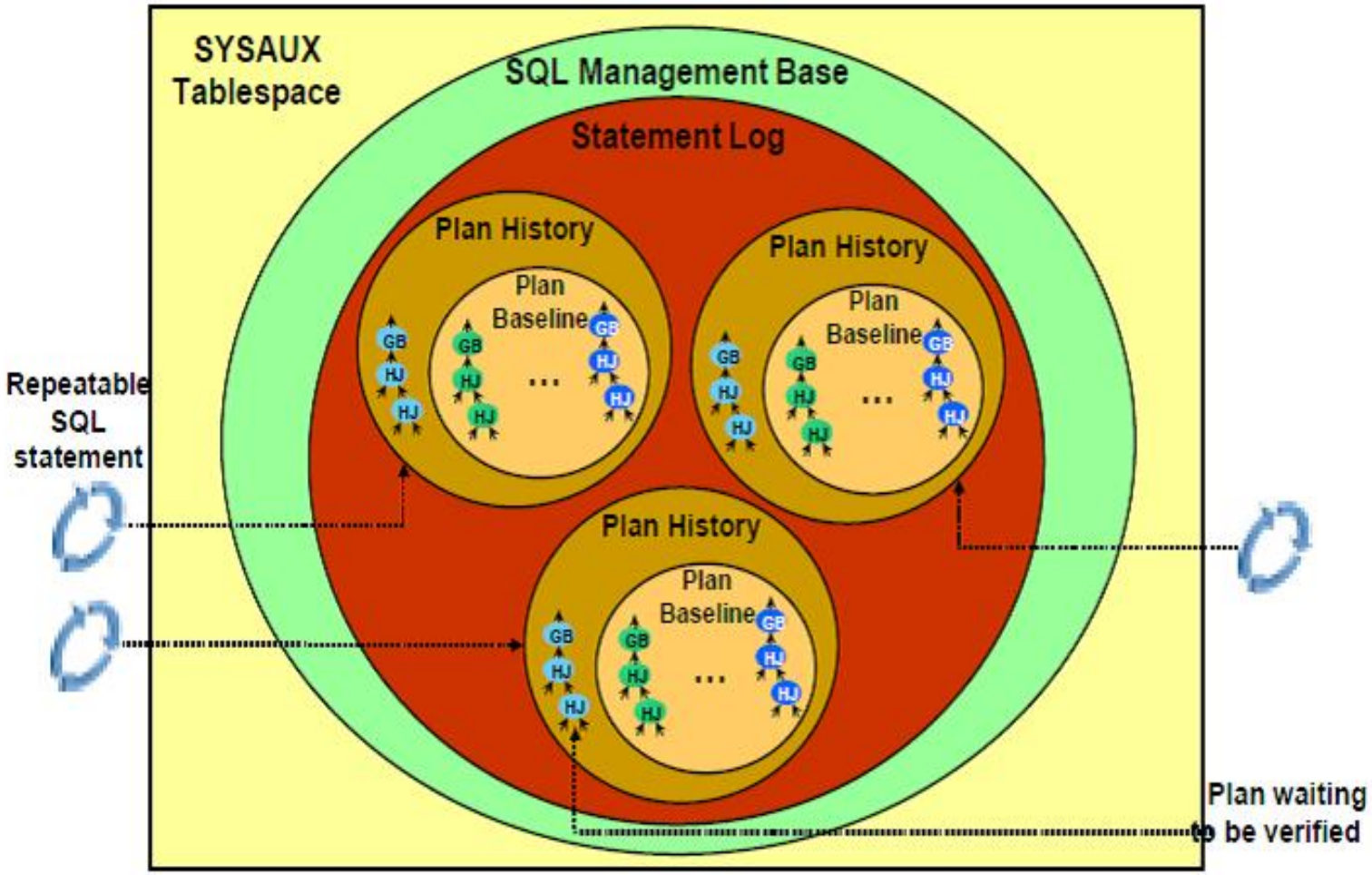
## 2. SQL plan baseline selection :-

Once you collect the SQL plans either from the AWR or from the cursor cache, the next steps in SPM is to use those SQL Plan baselines.
The use of SQL Plan baselines is controlled by initialization parameter "optimizer_use_sql_plan_baselines" which has the default value set to true, which means the use of SQL Plan baselines are enabled by default.

## 3. SQL plan baseline evolution :-

Evolving SQL Plan baseline is the critical phase when the database changes a non-accepted plan in the plan history to an accepted plan and makes it as part of SQL Plan baselines. Only the plans which are captured automatically on the fly are subject for the Plan evolution, the manually loaded plans are always added as an accepted plan.

# SQL Plan Management Components



# Scenarios for SQL Plan Management

You can use SPM to preserve SQL performance when you encounter the following types of system changes.

1) Database upgrades
2) New optimizer versions
3) Changes in optimizer parameters
4) Changes in system settings.
5) Changes in schema and metadata definitions.
6) Deployment of new application modules.

# Capturing Sql Plan Baselines

Automatic Plan Capture:-

Set the new initialization parameter "optimizer_capture_sql_plan_baselines" to true to let the database automatically create and maintain a SQL plan history for each repeatable SQL statements. By default the parameter is set to false. This is a dynamic parameter and hence can be changed on the fly.

The very first SQL plan that the database generates for any SQL statements is automatically integrated into the corresponding SQL plan baseline. You can use the automatic SQL plan capturing mechanism to retain good execution plan for use after database upgrade to Oracle 11g as shown below:-

1) Upgrade the Oracle database to Oracle 11g.

2) Set initialization parameter optimizer_capture_sql_plan_baselines=TRUE.

3) Set initialization parameter “optimizer_features_enable”=10.2.

4) The SQL plan management feature will collect the pre-Oracle 11g execution plans and store them as SQL plan baselines in the upgraded database.

5) Once the database goes through a complete workload and captures all possible SQLs and hence SQL plan baselines, and then set the optimizer_features_enable to 11.x.x.x.

## Manual Plan Loading

There is one significant difference between the manual plan loading and automatic plan loading, that when you load plan manually the database doesn't verify them for performance regression and it immediately adds the plans you added manually as accepted plans to the SQL Plan baseline. Manual loading of execution plans is especially useful when a database is being upgraded from a previous version to Oracle Database 11g or when a new application is being deployed.

# Techniques for bulk loading plans into SQL Management Base

1. Bulk load using SQL Tuning Set (STS) and AWR snapshots.
2. Bulk load from Stored Outlines.
3. Bulk load using Cursor Cache.
4. Bulk load using SQL plan baselines.

## SQL Tuning set (STS) and AWR snapshot load

Bulk loading execution plans from a STS is an excellent way to guarantee no plan changes as part of a database upgrade.

1.In an Oracle Database 10gR2 create an STS that includes the execution plan for each of the SQL statements.

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

3. Import the staging table from a flat file into an 11g database and unload the STS.

4. DBMS_SPM.LOAD_PLANS_FROM_SQLSET to load the execution plans into the SMB.

# Creating SQL Tuning sets (STS)

```
BEGIN
  DBMS_SQLTUNE.create_sqlset (sqlset_name => 'SPM_TEST_1',description =>
'A test SQL tuning set.');
END;
```

Statements are added to the set using the LOAD_SQLSET procedure which accepts a REF CURSOR of statements retrieved using one of the following pipelined functions:

SELECT_WORKLOAD_REPOSITORY - Retrieves statements from the (AWR).

SELECT_CURSOR_CACHE - Retrieves statements from the cursor cache.

SELECT_SQLSET - Retrieves statements from another SQL tuning set.

SELECT_SQL_TRACE Function – using the SQL trace files. This table function reads the content of one or more trace files and returns the SQL statements it finds in the format of sqlset_row

# Loading SQL Tuning sets

## STS load from the AWR for snapshot range.

```
DECLARE
  l_cursor  DBMS_SQLTUNE.sqlset_cursor;
BEGIN
 OPEN l_cursor FOR
    SELECT VALUE(p)
    FROM   TABLE (DBMS_SQLTUNE.select_workload_repository (
                  998,  -- begin_snap
                  999,  -- end_snap
NULL, -- basic_filter (The SQL predicate to filter the SQL from the workload repository defined on attributes of the SQLSET_ROW)
NULL, -- object_filter (The objects to selected  from the SWRF)
NULL, -- ranking_measure1(An order-by clause on the selected SQL)
NULL, -- ranking_measure2
NULL, -- ranking_measure3
NULL, -- result percentage ((A filter which picks the top N% as per ranking measure given.it
              applies only if one ranking measure is given)
10,   -- result_limit (The top L(imit) SQL from the (filtered) source ranked by the ranking measure)
NULL) --attribute_list (List of SQL statement attributes to return in the result. The possible values
              are: Basic,Typical,ALL) ) p;

 DBMS_SQLTUNE.load_sqlset (sqlset_name => 'SPM_TEST_1',  populate_cursor => l_cursor);
END;
```

## STS load from AWR for predefined baseline

```
SELECT VALUE(p)
    FROM    TABLE (DBMS_SQLTUNE.select_workload_repository ('peek_baseline',) p;
```

## STS load from the cursor cache.

```
 DECLARE
  l_cursor  DBMS_SQLTUNE.sqlset_cursor;
BEGIN
  OPEN l_cursor FOR
    SELECT VALUE(p)
FROM    TABLE (DBMS_SQLTUNE.select_cursor_cache (
) p;          --parameters as shown above with the AWR snapshot load

  DBMS_SQLTUNE.load_sqlset (sqlset_name => 'SPM_TEST_1',populate_cursor => l_cursor);
END;
e.g. DBMS_SQLTUNE.SELECT_CURSOR_CACHE ('parsing_schema_name <> ''SYS'')) P;
```

## STS load from the CAPTURE_CURSOR_CACHE_SQLSET

```
 DBMS_SQLTUNE.CAPTURE_CURSOR_CACHE_SQLSET(
 sqlset_name =>'SPM_TEST_1', -- The SQL tuning set name
 time_limit => '1800',       -- The total amount of time, in seconds, to execute
 repeat_interval=>'300',     -- The amount of time, in seconds, to pause between sampling
 capture_option =>'MERGE'    --either insert new statements, update existing ones, or both.
                             'INSERT', 'UPDATE', or 'MERGE'
 sqlset_owner =>'SYS');
END;
```

CAPTURE_CURSOR_CACHE_SQLSET collects SQL statements from the cursor cache over a specified time interval, attempting to build a realistic picture of system workload.

## STS load using existing STS

```
DECLARE
l_cursor  DBMS_SQLTUNE.sqlset_cursor;
BEGIN
DBMS_SQLTUNE.create_sqlset(sqlset_name  => 'SPM_TEST_2',
description  => 'Another test SQL tuning set.');
OPEN l_cursor FOR
SELECT VALUE(p)
FROM   TABLE (DBMS_SQLTUNE.select_sqlset ('SPM_TEST_1', -- sqlset_name ) p;
--parameters as shown above with the AWR snapshot load

DBMS_SQLTUNE.load_sqlset (sqlset_name => 'SPM_TEST_2',  populate_cursor => l_cursor);
END;
```

## Load the STS into a staging table and export the staging table

```
BEGIN
DBMS_SQLTUNE.create_stgtab_sqlset(table_name => 'SQLSET_TAB',
schema_name => 'SPMUSER', tablespace_name => 'USERS');
END;
```

Use the PACK_STGTAB_SQLSET procedure to export SQL tuning set into the staging table.

```
BEGIN
  DBMS_SQLTUNE.pack_stgtab_sqlset (sqlset_name => 'SPM_TEST_1‘,sqlset_owner => 'SYS',
staging_table_name   => 'SQLSET_TAB‘, staging_schema_owner => 'SPMUSER');
END;
```

## Import the staging table and unload the STS

Once the SQL tuning set is packed into the staging table, the table can be transferred to the test system using Datapump, Export/Import or via a database link. Once on the test system, the SQL tuning set can be imported using the UNPACK_STGTAB_SQLSET procedure.

```
BEGIN
  DBMS_SQLTUNE.unpack_stgtab_sqlset(sqlset_name => '%', sqlset_owner => 'SYS',
replace => TRUE, staging_table_name => 'SQLSET_TAB', staging_schema_owner => 'SPMTESTUSER');
END;
```

## Load the plan from STS to SQL Plan Baseline

To load plans from a STS, use the LOAD_PLANS_FROM_SQLSET function of the DBMS_SPM.

```
 DECLARE
my_plans pls_integer;
```

```
 BEGIN
  my_plans := DBMS_SPM.LOAD_PLANS_FROM_SQLSET(sqlset_name =>SPM_TEST_1');
END;
```

basic_filter => 'sql_text like "select%p.prod_name%"'-- 'sql_text like "select /*LOAD_STS*/%"'

The filter can take the form of any WHERE clause predicate that can specified against the view DBA_SQLSET_STATEMENTS

fixed=>NULL -- Default 'NO' means loaded plans will be used as 'non-fixed' plans.

enable=>NULL --Default 'YES' means loaded plans are enabled for use by optimizer.

commit_rows=> NULL --Number of SQL plans to load before doing a periodic commit. This helps to shorten the undo log.);

## From Stored Outlines

If you don't have access to SQL Tuning Sets or if you are upgrading from an earlier version than Oracle Database 10gR2 you can capture your existing execution plan using Stored Outlines. Stored Outlines can be loaded into the SQL Management Base as SQL plan baselines using the PL/SQL procedure DBMS_SPM.MIGRATE_STORED_OUTLINE or through Oracle Enterprise Manager (EM). Next time these statements are executed the SQL plan baselines will be used.

There are two ways to capture Stored Outlines, you can either manually create one for each SQL statement using the CREATE OUTLINE command or let Oracle automatically create a Stored Outline for each SQL statement that is executed. Below are the steps needed to let Oracle automatically create the Stored Outlines for you.

1. You should begin by starting a new session and switch on the automatic capture of a Stored Outline for each SQL statement that gets parsed from now on until you explicitly turn it off.

2. Then execute the workload either by running the application or manually issuing SQL statements. NOTE: if you manually issue the SQL statements ensure you use the exact SQL text used by the application, if it uses bind variables you will have to use them too. Once you have executed your critical SQL you should turn off the automatic capture.

3. The actual Stored Outlines are stored in the OUTLN schema. You can either export the schema and import it into the 11g database or upgrade your existing database to 11g.

4. Use DBMS_SPM.MIGRATE_STORED_OUTLINE to load the Stored Outlines into the SMB.

After migrating the stored outlines to SQL plan Baselines we need to ensure, that the database creates plan baselines but does not create stored outlines. The database only uses stored outlines when the equivalent SQL plan baselines do not exist.

For example, the following SQL statements instruct the database to create SQL plan baselines instead of stored outlines when a SQL statement is executed. The example also instructs the database to apply a stored outline in category all_rows or DEFAULT only if it exists and has not been migrated to a SQL plan baseline. In other cases, the database applies SQL plan baselines instead.

```
SQL> ALTER SYSTEM SET CREATE_STORED_OUTLINE = false;
SQL> ALTER SESSION SET USE_STORED_OUTLINES = all_rows;
```

# Using the execution plans currently in the Cursor Cache

Starting in Oracle Database 11g it is possible to load plans for statements directly from the cursor cache into the SMB . By applying a filter – on the module name, the schema, or the SQL_ID – you can identify the SQL statement or set of SQL statement you wish to capture.

Loading plans directly from the cursor cache can be extremely useful if application SQL has been tuned by hand using hints. Since it is unlikely the application SQL can be changed to include the hint, by capturing the tuned execution plan as a SQL plan baseline you can ensure that the application SQL will use that plan in the future. 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.Run the non-hinted SQL statement so we can begin the SQL plan baseline capture.

```
SQL> SELECT prod_name, SUM(amount_sold) FROM Sales s, Products p WHERE
    s.prod_id=p.prod_id AND prod_category = :ctgy GROUP BY prod_name;
```

2. Then find the SQL_ID for the statement in the V$SQL view.

```
SQL> SELECT sql_id, sql_fulltext FROM V$SQL WHERE sql_text LIKE '%SELECT prod_name, SUM(%';
SQL_ID SQL_FULLTEXT
------------- ------------------------------------
chj6q8z7ykbyy SELECT PROD_NAME, SUM(AMOUNT_SOLD)
```

3. Using the SQL_ID create a SQL plan baseline for the statement.

```
SQL> variable cnt number;
SQL> EXECUTE: cnt :=DBMS_SPM.LOAD_PLAN_FROM_CURSOR_CACHE(sql_id=>'chj6q8z7ykbyy');
```

4. The plan that was captured is the sub-optimal plan and it will need to be disabled. The SQL_HANDLE & PLAN_NAME are required to disable the plan.

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

SQL_HANDLE SQL_TEXT PLAN_NAME ENABLE

---------- ---------- ---------- ------

SYS_SQL_bf5c9b08f72bde3e SELECT PROD_NAME,  SQL_PLAN_byr4v13vkrrjy42949306 Y

5. Using DBMS_SPM.ALTER_SQL_PLAN_BASELINE disable the bad plan

SQL> exec :cnt :=DBMS_SPM.ALTER_SQL_PLAN_BASELINE(SQL_HANDLE => 'SYS_SQL_bf5c9b08f72bde3e',PLAN_NAME=> 'SQL_PLAN_byr4v13vkrrjy42949306', ATTRIBUTE_NAME => 'enabled', ATTRIBUTE_VALUE => 'NO');

6. Now you need to modify the SQL statement using the necessary hints

SQL> SELECT /*+ INDEX(p) */ prod_name, SUM(amount_sold) FROM Sales s, Products p --

7. 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, fulltext FROM V$SQL WHERE sql_text LIKE '%SELECT /*+ INDEX(p) */ prod_na%';

SQL_ID PLAN_HASH_VALUE SQL_FULLTEXT

------------- --------------- --------------------------

djkqjd0kvgmb5 3074207202 SELECT /*+ INDEX(p) */

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

```
exec cnt:=dbms_spm.load_plans_from_cursor_cache(sql_id=>'djkqjd0kvgmb5',
plan_hash_value => 3074207202,sql_handle => 'SYS_SQL_bf5c9b08f72bde3e');
```

## Bulk load using SQL plan baselines from a staging table

With Oracle Database 11g, any 3rd party software vendor can ship their application software along with the appropriate SQL plan baselines for 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. 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 source DB create a staging table using DBMS_SPM.CREATE_STGTAB_BASELINE procedure.
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.
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.

```
DECLARE
  l_plans_unpacked  PLS_INTEGER;
BEGIN
  l_plans_unpacked := DBMS_SPM.unpack_stgtab_baseline( table_name  => spm_stageing_tab',
    table_owner     => 'TEST',  creator         => 'TEST');
  DBMS_OUTPUT.put_line('Plans Unpacked: ' || l_plans_unpacked);
END;
```

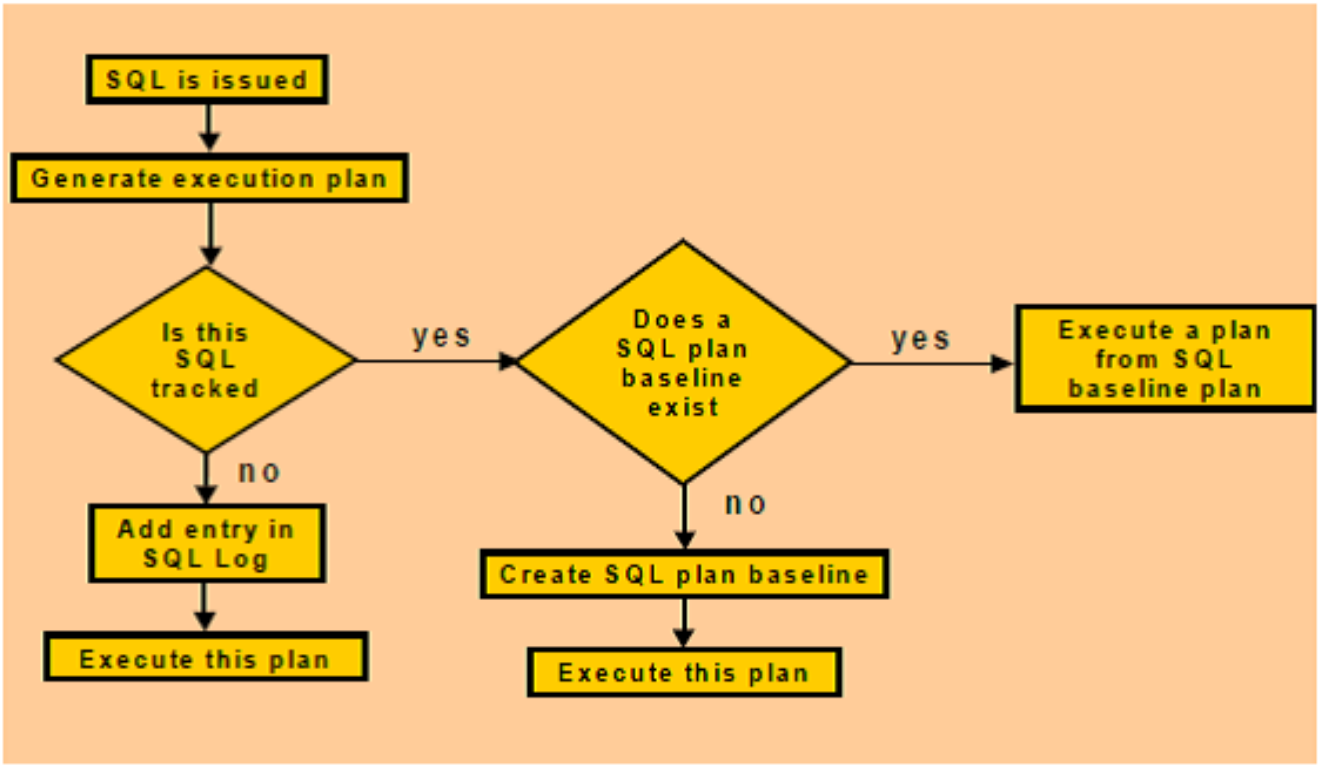
## Selecting SQL Plan baselines

To enable the database to use the SQL plan baselines captured by automatic capture or manual capture or both, set the initialization parameter "optimizer_use_sql_plan_baselines" to true. This parameter is set to true by default. When this parameter set to true either by default or manually will enable the use of SQL plan baselines stored by database in the SQL Management Base (SMB).

When a SQL statement is hard parsed, the cost based optimizer produces several execution plans and selects the one with the lowest cost. If a SQL plan baseline is present, the optimizer compares the plan it just produced with the plans in the SQL plan baseline. If a matching plan is found that is flagged as accepted the plan is used. If the SQL plan baseline doesn't contain an accepted plan matching the one it just created, the optimizer evaluates the accepted plans in the SQL plan baseline and uses the one with the lowest cost. If the execution plan originally produced by the optimizer has a lower cost than those present in the SQL plan baseline, it is added to the baseline as a not-accepted plan, so it is not used until it is verified not to cause a reduction in performance. If a system change affects all existing accepted plans, so they are considered non-reproducible, the optimizer will use the plan with the lowest cost.

Note:-Oracle only stores the outline for each plan in the SMB, therefore the optimizer will reproduce the actual execution plan from the stored outline of the selected plan and execute the sql using that plan.

# SQL Plan Management Flowchart



# Evolving SQL Plan Baselines:-

You can instruct database to evaluates new plan performance with a view to integrating plans with superior performance into SQL plan baseline for the corresponding SQL statement. Evolving SQL plan baseline changes a non-accepted plan in the plan history to an accepted plan and makes it a part of SQL plan baseline as an accepted plan. In order to deem a plan in the history as an accepted plan, its performance must be better than all of the already accepted plan in the baseline. Only the Plans which are captured automatically , are evolved .The plan which are loaded manually ,are loaded as accepted plan automatically.

# EVOLVE_SQL_PLAN_BASELINE function:-

The EVOLVE_SQL_PLAN_BASELINE function determines whether a new plan added to the plan history performs better than the plan from the corresponding SQL plan baseline. If it performs better then it adds the new plan to the SQL Plan baseline as an accepted plan.

```
DECLARE
report clob;
BEGIN
report := DBMS_SPM.EVOLVE_SQL_PLAN_BASELINE( sql_handle=> 'SYS_SQL_593bc74fca8e6738');
DBMS_OUTPUT.PUT_LINE (report);
END;
```

```
 DBMS_SPM.EVOLVE_SQL_PLAN_BASELINE ( sql_handle   IN VARCHAR2 := NULL,  plan_name    IN VARCHAR2 := NULL, time_limit  IN INTEGER  := DBMS_SPM.AUTO_LIMIT,  verify    IN VARCHAR2 := 'YES', commit IN VARCHAR2 := 'YES')
  RETURN CLOB;
```

DBMS_SPM.EVOLVE_SQL_PLAN_BASELINE (

plan_list IN DBMS_SPM.NAME_LIST, time_limit IN INTEGER := DBMS_SPM.AUTO_LIMIT, verify IN VARCHAR2 := 'YES', commit IN VARCHAR2 := YES')

parameters for EVOLVE_SQL_PLAN_BASELINE:-

**sql_handle**:-SQL statement identifier. Unless plan_name is specified, NULL means to consider all statements with non-accepted plans in their SQL plan baselines.

**plan_name** :-Plan identifier. Default NULL ,consider all non- accepted plans in the plan baseline of either the identified SQL statement or all sql statements if sql_handle is NULL.

**plan_list**:-A list of plan names. Each plan in the list can belong to same or different SQL statement.

**time_limit**:-Time limit in number of minutes. This applies only if verify = 'YES'. The time limit is global and it is used as follows: The time limit for first non- accepted plan verification is set equal to the input value; the time limit for second non-accepted plan verification is set equal to (input value - time spent in first plan verification); and so on . Possible values are shown below

DBMS_SPM.AUTO_LIMIT (Default) lets the system choose an appropriate time limit based on the number of plan verifications required to be done.

DBMS_SPM.NO_LIMIT means there is no time limit.

N (A positive integer value represents a user specified time limit).

verify:- Specifies whether to execute the plans and compare the performance before changing non-accepted plans into accepted plans. Possible values are shown below

'YES' (Default) will verify that a non-accepted plan gives better performance before changing it to an accepted plan.

'NO' means not to execute plans only change into accepted plans.

-commit :-Specifies whether to update the ACCEPTED status of non-accepted plans from 'NO' to 'YES'.

'YES' (Default) - perform updates of qualifying non-accepted plans and generate a report that shows the updates and the result of performance verification

'NO' - generate a report without any updates. Note that commit = 'NO' together with verify = 'NO' represents a no-op.

## Evolving SQL Plan with SQL Tuning Advisor:-

You can also evolve SQL plan baselines by running the SQL Tuning Advisor. This applies to both automatic and manual execution of SQL Tuning Advisor. SQL Tuning Advisor recommends accepting a SQL profile only if the Plan with SQL profile is better than the Original plan. Once you accept the recommended SQL profile, the advisor adds the plan to the SQL plan baseline for that SQL statement.

```
SQL> var tname varchar2(30);

SQL> exec :tname := dbms_sqltune.create_tuning_task(sql_id => 'abcdefghijkllss');

SQL> exec dbms_sqltune.execute_tuning_task(task_name => :tname);

SQL> select dbms_sqltune.report_tuning_task(:tname, 'TEXT', 'BASIC') FROM dual;

SQL> exec dbms_sqltune.accept_sql_profile(task_name => :tname);
```

# Fixed SQL Plan Baselines:-

If a SQL plan baselines contains one or more enabled plans for which the fixed attribute value is set to yes, the baseline is considered fixed. When the plan for a SQL gets fixed then the optimizer gives preference to the fixed plans over the nonfixed plans even if some of the nonfixed plans are cheaper with a lower cost of execution. The database doesn't evolve a fixed SQL plan baseline because the optimizer will not add any new execution plans to a fixed SQL plan baseline.

You may however evolve even a fixed SQL plan baseline by manually loading a new plan either from the cursor cache or a STS.

```
DBMS_SPM.ALTER_SQL_PLAN_BASELINE (
   sql_handle       IN VARCHAR2 := NULL,  plan_name   IN VARCHAR2 := NULL,
   attribute_name    IN VARCHAR2,  attribute_value   IN VARCHAR2)
```

-attribute_name :- Name of plan attribute to set.

-attribute_value :-Value of plan attribute to use (see table below)

Enabled ('YES' means the plan is available for use. It may or may not be used depending on accepted status. Possible values 'YES' or 'NO')

fixed ('YES' means the SQL plan baseline is not evolved over time. Possible values 'YES' or 'NO')

autopurge ('YES' means the plan is purged if it is not used for a time period. 'NO' means it is never purged.possible values 'YES' or 'NO')

plan_name (Name of the plan. Possible values String of up to 30-characters)

## Managing SQL Plan Baselines:-

You can view the SQL plans stored in the SQL plan baseline for a specific sql/sqls

1)Using DBA_SQL_PLAN_BASELINES data dictionary

2) DBMS_XPLAN.DISPLAY_SQL_PLAN_BASELINE function

DBMS_XPLAN.DISPLAY_SQL_PLAN_BASELINE(sql_handle, NULL,'TYPICAL +NOTE')) XPLAN;

## SQL Management Base (SMB):-

The database stores all SPM related information such as statement logs,plan history and SQL profiles and SQL plan baselines in a new data dictionary component called the SQL Management Base (SMB). The database stores the SMB in the SYSAUX tablespace.

## Configuring the SQL Management Base (SMB):-

Space usage is controlled by altering two name-value attributes using the CONFIGURE procedure of the DBMS_SPM package.

-space_budget_percent (default 10) : Maximum size as a percentage of SYSAUX space. Allowable values 1-50.

-plan_retention_weeks (default 53) : Number of weeks unused plans are retained before being purged. Allowable values 5-523 weeks.

The current settings are visible using the DBA_SQL_MANAGEMENT_CONFIG view.

```
SQL> SELECT parameter_name, parameter_value  FROM   dba_sql_management_config;
```

Default settings can be changed by DBMS_SPM.configure.

```
DBMS_SPM.configure ('space_budget_percent', 11);
DBMS_SPM.configure ('plan_retention_weeks', 54);
```

## Purging policies:-

A weekly purging task that is the part of the automated tasks that run during the maintenance windows takes care of removing older unused baselines, to conserve space in the SMB. By default the database purges all SQL plan baselines which are not being used for 53 weeks, this setting can be change by adjusting the plan_retention_weeks attribute of the DBMS_SPM.configure procedure.

```
DBMS_SPM.configure('plan_retention_weeks', 10);
```

In addition manual purging can be done as well to purge the sql plan baseline for a specific sql statement by using PURGE_SQL_PLAN_BASELINE procedure

```
SQL>exec DBMS_SPM.drop_sql_plan_baseline (sql_handle => 'SYS_SQL_78888');
```

# References


Oracle® Database PL/SQL Packages and Types Reference 11*g* Release 1 (11.1)

http://download.oracle.com/docs/cd/B28359_01/server.111/b28274/optplanmgmt.htm
http://download.oracle.com/docs/cd/B28359_01/appdev.111/b28419/d_spm.htm#CACJJFDB

Oracle corporation white paper (SQL Plan Management in Oracle Database 11g)

Alapati, S. 2008. *OCP Oracle Database 11g New Features for Administrators Exam Guide*. Tata McGraw-Hill.

SQL Plan Management in Oracle Database 11g Release 1
http://www.oracle-base.com/articles/11g/SqlPlanManagement_11gR1.php

**Czuprynski, J. 2008. Oracle Database 11g: SQL Plan Management, Part 1 & 2**

http://www.databasejournal.com/features/oracle/article.php/3723676/Oracle-Database-11g-SQL-Plan-Management-Part-1.htm
http://www.databasejournal.com/features/oracle/article.php/3730391/Oracle-Database-11g-SQL-Plan-Management-Part-2.htm