An Oracle White Paper
November 2010

Upgrading from Oracle Database 10g to 11g: What to expect from the Optimizer

Introduction

The purpose of the optimizer is to determine the most efficient execution plan for your queries. It makes these decisions based on the structure of the query, the statistical information it has about your data, and by leveraging Oracle database features. After an upgrade, the optimizer is expected to generate the same or a better performing execution plan for most SQL statements. Still, it is possible that the optimizer may generate a sub-optimal plan for some SQL statements in the new release compared to the prior release. Determining the root cause of these plan-related performance regressions can be a daunting task.

This paper aims to dispel the mystery that surrounds the optimizer and to prepare you to upgrade from Oracle Database 10g to Oracle Database 11g. It will introduce the new optimizer features, outline what steps you should take before and after the upgrade to avoid performance regressions, related to plan changes that do occur. The paper is divided into three sections:

- The first section introduces the 11g optimizer features including statistics management.
- The second section explains the pre-upgrade steps you need to take.
- Finally, the third section covers what to expect after the upgrade and how to address any performance regression caused by plan changes.

New Optimizer and Statistics Features in 11g

Init.ora parameters

There are several new initialization parameters that govern the optimizer and its new features in Oracle Database 11g. Below are the details on the new parameters.

OPTIMIZER_USE_INVISIBLE_INDEXES

In Oracle Database 11g a new feature called Invisible indexes was introduced. This feature allows a DBA to create an index on a table without affecting any execution plan and thus not impacting the performance of the application. An invisible index cannot be used by the optimizer as an access path to that table. In order to test such indexes as a potential access path, the DBA needs to set `OPTIMIZER_USE_INVISIBLE_INDEXES` to `TRUE` (default `FALSE`) within a session, and then execute any SQL statement that could benefit from using the index to test its performance. If the index proves to be useful, the DBA can mark it visible; if it does not, it can be dropped without impacting the performance of the application.

OPTIMIZER_USE_PENDING_STATISTICS

Traditionally, when optimizer statistics are gathered they are published (written) immediately to the appropriate dictionary tables and begin to be used by the optimizer. However, in Oracle Database 11g it is possible to gather optimizer statistics but not have them published immediately. Instead of going into the usual dictionary tables the statistics are stored in pending tables so that they can be tested before they are published. A DBA can test these pending statistics by doing an `alter session command` to set `OPTIMIZER_USE_PENDING_STATISTICS` to `TRUE`. Then they should run any SQL statements they believe might be affected by the new statistics. By testing the statistics before they are published, the DBA gets an opportunity to prevent plan regressions that result from incomplete or incorrect statistics.

More information on pending statistics can be found in the new features section below.

OPTIMIZER_CAPTURE_SQL_PLAN_BASELINES

In Oracle Database 11g a new feature called SQL Plan Management (SPM) has been introduced to guarantees any plan changes that do occur lead to better performance. When `OPTIMIZER_CAPTURE_SQL_PLAN_BASELINES` is set to `TRUE` (default `FALSE`) Oracle will automatically capture a SQL plan baseline for every repeatable SQL statement on the system. The execution plan found at parse time will be added to the SQL plan baseline as an accepted plan.

More information on SPM can be found in the new features section below.

OPTIMIZER_USE_SQL_PLAN_BASELINES

Each SQL statement captured in SPM has a SQL plan baseline, which contains one or more known or verified execution plans. When `OPTIMIZER_USE_SQL_PLAN_BASELINES` is set to `TRUE` (default) the optimizer will only use one of these known plans even if a different plan is found during SQL-compilation. This guarantees that any plan change is verified to have better performance before using it.

SUMMARY OF NEW INIT.ORA PARAMETERS

PARAMETER NAME	11G DEFAULT VALUE
OPTIMIZER_USE_INVISIBLE_INDEXES	FALSE
OPTIMIZER_USE_PENDING_STATISTICS	FALSE
OPTIMIZER_CAPTURE_SQL_PLAN_BASELINES	FALSE
OPTIMIZER_USE_SQL_PLAN_BASELINES	TRUE

Changes to Optimizer statistics and the DBMS_STATS package

The ANALYZE **command has been officially obsolete for gathering statistics.**

In Oracle 8i a new PL/SQL package, called `DBMS_STATS` was introduced to gather and manage optimizer statistics. `DBMS_STATS` is Oracle's preferred method for gathering statistics. The `DBMS_STATS` package has been extended in Oracle Database 11g to accommodate new types of statistics and monitoring data that can now be collected. Changes have also been made to the automatic statistics-gathering job that is enabled by default in 11g.

New subprograms in the DBMS_STATS package

Setting preferences for parameter values used by the DBMS_STATS procedures

In previous releases you had to use the `DBMS_STATS.SET_PARM` procedure to change the default value for the parameters used by the `DBMS_STATS.GATHER_*_STATS` procedures. The scope of any changes that were made was all subsequent operations. In Oracle Database 11g, the

DBMS_STATS.SET_PARAM procedure has been deprecated and it has been replaced with a set of procedures that allow you to set a preference for each parameter at a table, schema, database, and global level. These new procedures are called DBMS_STATS.SET_*_PREFS and offer a much finer granularity of control. The list of parameters you can change are as follows:

```
AUTOSTATS_TARGET (SET_GLOBAL_PREFS only)
CASCADE
DEGREE
ESTIMATE_PERCENT
METHOD_OPT
NO_INVALIDATE
GRANULARITY
PUBLISH
INCREMENTAL
STALE_PERCENT
```

The SET_TABLE_PREFS procedure allows you to change the default values of the parameters used by the DBMS_STATS.GATHER_*_STATS procedures for the specified table only.

The SET_SCHEMA_PREFS procedure allows you to change the default values of the parameters used by the DBMS_STATS.GATHER_*_STATS procedures for all of the existing tables in the specified schema. This procedure actually calls SET_TABLE_PREFS for each of the tables in the specified schema. Since it uses SET_TABLE_PREFS calling this procedure will not affect any new objects created after it has been run. New objects will pick up the GLOBAL preference values for all parameters.

The SET_DATABASE_PREFS procedure allows you to change the default values of the parameters used by the DBMS_STATS.GATHER_*_STATS procedures for all of the user-defined schemas in the database. This procedure actually calls SET_TABLE_PREFS for each of the tables in each of the user-defined schemas. Since it uses SET_TABLE_PREFS this procedure will not affect any new objects created after it has been run. New objects will pick up the GLOBAL preference values for all parameters. It is also possible to include the Oracle owned schemas (sys, system, etc) by setting the ADD_SYS parameter to TRUE.

The SET_GLOBAL_PREFS procedure allows you to change the default values of the parameters used by the DBMS_STATS.GATHER_*_STATS procedures for any object in the database that does not have an existing table preference. All parameters default to the global setting unless there is a table preference set or the parameter is explicitly set in the GATHER_*_STATS command. Changes made by this procedure **will** affect any new objects created after it has been run. New objects will pick up the GLOBAL_PREF values for all parameters. With GLOBAL_PREFS it is also possible to set a default value for one additional parameter, called AUTOSTAT_TARGET. This additional parameter controls what objects the automatic statistic gathering job (that runs in the nightly maintenance window) will look after. The possible values for this parameter are `ALL`, `ORACLE`, and `AUTO`. The default value is `AUTO`.

`DBMS_STATS.GATHER*_STATS` obeys the following hierarchy for parameter values; parameter values explicitly set in the command overrule everything. If the parameter has not been set in the command, we check for a table level preference. If there is no table preference set, we use the `GLOBAL` preference. For example, if you wanted to switch off histogram creation for the SALES table in the SH schema you can use:

```
BEGIN
DBMS_STATS.SET_TABLE_PREFS(‘SH’,'SALES','METHOD_OPT',’FOR ALL COLUMN SIZE
1);
END;
```

Copying table statistics

Very often with partitioned tables, a new empty partition will be added to the table and data will begin to be loaded into it immediately. If the newly loaded data is queried before statistics can be gathered for this partition, then the Optimizer will have to prorate its cardinality estimates for these queries. Prorated cardinality estimates can lead to sub-optimal plans. By using `DBMS_STATS.COPY_TABLE_STATS` procedure, it is possible to copy the statistics from one of the other partitions in the table to the new partition. Column statistics (min, max, NDV, histogram, etc), partition statistics (number of rows, blocks, etc) and statistics for local indexes will be copied. The minimum and maximum values for the partitioning columns will be adjusted to reflect the correct values for the new partition. In the following example, SALES_Q3_2000 is the source partition and SALES_Q4_2000 is the target partition.

```
BEGIN
DBMS_STATS. COPY_TABLE_STATS (‘SH’,’SALES’,'SALES_Q3_2000',
'SALES_Q4_2000', FORCE=>TRUE);
END;
```

Extended statistics

In real-world data, there is often a relationship or correlation between the data stored in different columns of the same table. For example, in the `CUSTOMERS` table, the values in the `CUST_STATE_PROVICE` column are influenced by the values in the `COUNTRY_ID` column, as the state of California is only going to be found in the United States. Until now, the Optimizer had no real way of knowing about these real-world relationships and could potentially miscalculate the selectivity if multiple columns from the same table are used in the where clause of a statement. With extended statistics you now have an opportunity to tell the Optimizer about these real-world relationships.

By creating statistics on a group of columns, the Optimizer can be given a more accurate selectivity guideline for the columns used together in a where clause of a SQL statement. Not all of the columns in the column group need to be present in the SQL statement for the Optimizer to use extended statistics; only a subset of the columns is necessary. Use `DBMS_STATS.CREATE_EXTENDED_STATS` to define the column group you want to have statistics gathered on as a whole. Once the group has been established Oracle will automatically maintain the statistics on that column group when statistics are gathered on the table.

```
SELECT DBMS_STATS.CREATE_EXTENDED_STATS(null,'customers',
      '(country_id, cust_state_province)');
FROM dual;
```

After creating the column group, you will see an additional column, with a system-generated name, in `user_tab_col_statistics` table. This new column represents the column group.

It is also possible to create extended statistics for an expression (including functions), as it is difficult for the Optimizer to estimate the cardinality of a where clause predicate that has columns embedded inside expressions. For example, if it is common to have a where clauses predicate `UPPER(LastName)=:B1`, then it would be beneficial to create extended statistics for `UPPER(LastName)`.

```
BEGIN
DBMS_STATS.GATHER_TABLE_STATS(null,'customers',method_opt =>

    'for all columns size skewonly for columns(upper(cust_last_name))');
END;
```

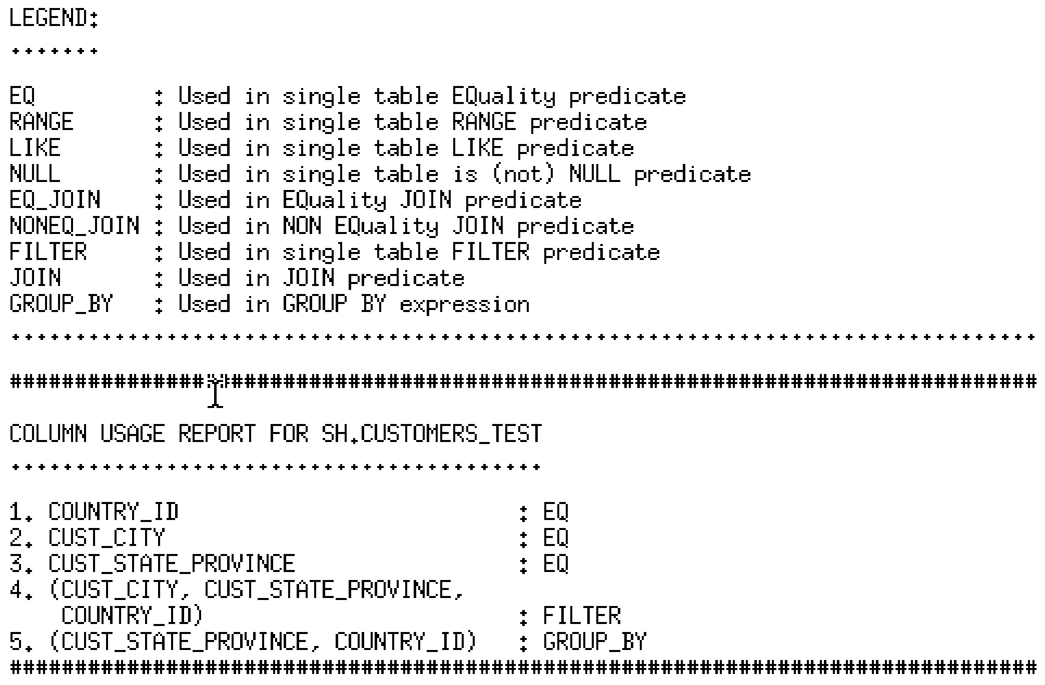
In Oracle Database 11g Release 2 (11.2.0.2) it is possible to have Oracle automatically determine which extended statistics are required for a table based on a given workload. It is a simple three step process:

1. Seed column usage. Oracle must observe a representative workload, in order to determine the appropriate extended statistics. Using the new procedure `DBMS_STATS.SEED_COL_USAGE`, you tell Oracle how long it should observe the workload. The following example turns on monitoring for 5 minutes or 300 seconds.

```
BEGIN

DBMS_STATS.SEED_COL_USAGE(null,null,300);

END;
```

2. Review column usage report. Once the monitoring window has finished, it is possible to review the column usage information recorded using the new procedure DBMS_STATS.REPORT_COL_USAGE.

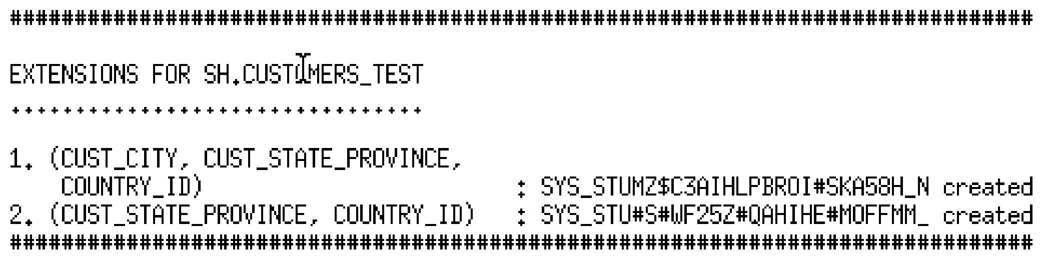
```
SELECT DBMS_STATS.REPORT_COL_USAGE(user, ‘customer_test’) FROM dual;
```



```
LEGEND:
.......

EQ         : Used in single table EQuality predicate
RANGE      : Used in single table RANGE predicate
LIKE       : Used in single table LIKE predicate
NULL       : Used in single table is (not) NULL predicate
EQ_JOIN    : Used in EQuality JOIN predicate
NONEQ_JOIN : Used in NON EQuality JOIN predicate
FILTER     : Used in single table FILTER predicate
JOIN       : Used in JOIN predicate
GROUP_BY   : Used in GROUP BY expression
...............................................................................

###############################################################################

COLUMN USAGE REPORT FOR SH.CUSTOMERS_TEST
.........................................

1. COUNTRY_ID                           : EQ
2. CUST_CITY                            : EQ
3. CUST_STATE_PROVINCE                  : EQ
4. (CUST_CITY, CUST_STATE_PROVINCE,
    COUNTRY_ID)                         : FILTER
5. (CUST_STATE_PROVINCE, COUNTRY_ID)    : GROUP_BY
###############################################################################
```

3. Create the extended statistics. The next time statistics are gathered on the table the column usage information captured during the monitoring window will be use to create the extended statistics. From then on the extended statistics will be maintained for the table whenever statistics are gathered on the table.

```
SELECT DBMS_STATS.CREATE_EXTENDED_STATS(null,'customers_test') FROM dual;
```



```
###############################################################################

EXTENSIONS FOR SH.CUSTOMERS_TEST
................................

1. (CUST_CITY, CUST_STATE_PROVINCE,
    COUNTRY_ID)                         : SYS_STUMZ$C3AIHLPBROI#SKA58H_N created
2. (CUST_STATE_PROVINCE, COUNTRY_ID)    : SYS_STU#S#WF25Z#QAHIHE#MOFFMM_ created
###############################################################################
```

Incremental Statistics for Partitioned Tables

Gathering statistics on partitioned tables consists of gathering statistics at both the table level and partition level. Prior to Oracle Database 11g, adding a new partition or modifying data in a few partitions required scanning the entire table to refresh table-level statistics. Scanning the entire table can be very expensive as partitioned tables are generally very large. However, in Oracle Database 11g this issue has been addressed with the introduction of incremental global statistics. Typically with partitioned tables, new partitions are added and data is loaded into these new partitions. After the partition is fully loaded, partition level statistics need to be gathered and the global statistics need to be updated to reflect the new data. If the `INCREMENTAL` preference for the partitioned table is set to `TRUE`, and the `DBMS_STATS GRANULARITY` parameter is set to `AUTO`, Oracle will gather statistics on the new partition and update the global table statistics by scanning only those partitions that have been added or modified and not the entire table. Below are the steps necessary to use incremental global statistics.

```
BEGIN
DBMS_STATS.SET_TABLE_PREFS('SH','SALES','INCREMENTAL','TRUE');
END;

BEGIN
DBMS_STATS.GATHER_TABLE_STATS('SH','SALES');
END;
```

Incremental Global Stats works by storing a *synopsis* for each partition in the table. A synopsis is statistical metadata for that partition and the columns in the partition. Each synopsis is stored in the SYSAUX tablespace and takes approximately 200KB. Global statistics are generated by aggregating the synopses from each partition, thus eliminating the need to scan the entire table to gather table level statistics (see *Figure 1).* When a new partition is added to the table you only need to gather statistics for the new partition. The global statistics will be automatically updated by aggregating the new partition synopsis with the existing partitions synopses.

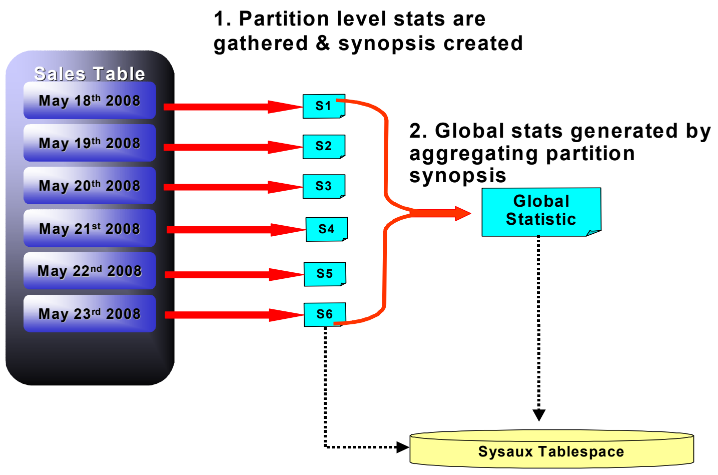


Figure 1 Incremental Global Statistics

Comparing Statistics

When it comes to deploying a new application or application module it is standard practice to test and tune the application in a test environment before it is moved to production. However, even with testing it is possible that SQL statements in the application will have different execution plans in production then they did on the test system. One of the key reasons an execution plan can differ from one system to another (from test and production) is because the optimizer statistics on each system are different. In Oracle Database 10g Release 2, the `DIFF_TABLE_STATS_*` functions can be used to compare statistics for a table from two different sources. The statistics can be from:

- A user statistics table and current statistics in the dictionary
- A single user statistics table containing two sets of statistics that can be identified using statids
- Two different user statistics tables
- Two points in history
- Current statistics and a point in history
- Pending Statistics with the current statistics in the dictionary
- Pending Statistics with a user statistics table

The function also compares the statistics of the dependent objects (indexes, columns, partitions). The function displays statistics for the object(s) from both sources if the difference between the statistics exceeds a specified threshold. The threshold can be specified as an argument to the function; the default value is 10%. The statistics corresponding to the first source will be used as the basis for computing the differential percentage.

In the example below, we compare the current dictionary statistics for the `EMP` table with the statistics for `EMP` in the statistics table `TAB1`; the SQL statement will generate a report-like output on the screen.

```
SQL> select report, maxdiffpct from
 table(DBMS_STATS.DIFF_TABLE_STATS_IN_STATTAB(‘SCOTT’,’EMP’,’TAB1’ ));
```

Concurrent Statistics gathering

Oracle Database 11g Release 2 (11.2.0.2) introduces a new statistics gathering mode, 'concurrent statistic gathering', which should reduce the time it takes to gather statistics sufficiently. The goal of this new mode is to enable statistics to be gathered on multiple tables in a schema / database and multiple (sub)partitions within a table concurrently. Gathering statistics on multiple tables concurrently can reduce the overall time it takes to gather statistics by allowing Oracle to fully utilizing mutli-processor environments.

Concurrent statistic gathering is controlled by a global parameter, CONCURRENT, in the DBMS_STATS package. The CONCURRENT parameter is Boolean so it can be set to TRUE or FALSE. By default it is set to FALSE. When CONCURRENT is set to TRUE, Oracle employs Oracle Job Scheduler and Advanced Queuing components to create and manage multiple statistics gathering jobs concurrently.

If you call dbms_stats.gather_table_stats on a partitioned table when CONCURRENT is set to true, Oracle will create a separate statistic gather job for each (sub)partition in the table. The job schedule will decide how many of these jobs will execute concurrently and how many will be queued based on available system resources. As the currently running jobs complete more jobs will be dequeued and executed until all (sub)partitions have had statistics gathered on them.

If you gather statistics using `DBMS_STATS.GATHER_DATABASE_STATS` or `DBMS_STATS.GATHER_SCHEMA_STATS` or `DBMS_GATHER_DICTIONARY_STATS` then Oracle will create a separate statistics gathering job for each non-partitioned table and each (sub)partition in the partitioned tables. The database will then run as many concurrent jobs as possible and queue the remaining jobs until the executing jobs complete. However, there can only be one active statistic collection job for a partitioned table. If the database / schema / dictionary contain multiple partitioned tables then the jobs created to collect statistics on the other partition tables will be queued until the current partitioned table has completed. There is no such restriction for non-partitioned tables. The maximum number of concurrent statistics gathering jobs is bounded by the job_queue_processes initialization parameter.

For example assume the parameter job_queue_processes is set to 30 and you issued a `DBMS_STATS.GATHER_SCHEMA_STATS` on the SH schema, Oracle would create a statistics gathering job for each of the non-partitioned tables
SUPPLEMENTARY_DEMOGRAPHICS,
COUNTRIES,
CUSTOMERS,
PROMOTIONS,
CHANNELS,
PRODUCTS,
TIMES.
And a statistics gathering job for each of the partitions in the SALES and COSTS tables.

Then the Oracle Job Scheduler would allow start 30 statistic gather job to start and would queue the rest. Let's assume that the 28 jobs (one for each partition) for the SALES table get scheduled, two non-partitioned table statistics gathering jobs will also be started. The statistics gather jobs for the COSTS table will be automatically queued because only one partitioned table statistics gathering job can be running at any one time. As each job finishes another job will be dequeued and started until all 63 jobs have been completed.

Locking partition level statistics

In previous releases you could lock statistics for a table or a schema. Once statistics are locked, no modifications can be made to those statistics until the statistics have been unlocked. In Oracle

Database 11g the `DBMS_STATS` package has two new procedures for locking and unlocking statistics at the partition level. These additional procedures allow for a finer granularity of control.

```
BEGIN
DBMS_STATS.LOCK_PARTITION_STATS('SH','SALES', 'SALES_Q3_2000');
END;
```

Pending Statistics

As discussed in the previous section, it is now possible to gather optimizer statistics but not have them published immediately. To activate pending statistics you need to use `DBMS_STATS.SET_*_PREFS` procedure to change value of the parameter PUBLISH from `TRUE` (default) to `FALSE` for the object(s) you wish to create pending statistics for.

```
BEGIN
DBMS_STATS.SET_TABLE_PREFS('SH','SALES','PUBLISH','FALSE');
END;
```

Then simple gather statistics on the object(s) as normal.

```
BEGIN
DBMS_STATS.GATHER_TABLE_STATS('SH','SALES');
END;
```

The statistics gathered for these objects can be displayed using the dictionary views called `USER_*_PENDING_STATS`. These statistics can be tested by issuing an alter session command to set the initialization parameter `OPTIMIZER_USE_PENDING_STATS` to `TRUE` and running the necessary queries. Once you are happy with the pending statistics you can publish them using the new procedure `PUBLISH_PENDING_STATS`.

```
BEGIN
DBMS_STATS.PUBLISH_PENDING_STATS('SH','SALES');
END;
```

New Sampling Algorithm

It is critical to provide the Optimizer with accurate statistics and traditionally the most accurate statistics are gathered when all rows in the table are processed. However, gathering statistics by doing a full scan of larger tables is extremely time consuming and is rarely an option for most customers. In fact most customers use sampling to gather statistics as it executes quickly. But without detailed testing, it is difficult to know the sample size to use to get accurate statistics. In the past, Oracle has tried to address this issue by introducing `AUTO_SAMPLE_SIZE`. With `AUTO_SAMPLE_SIZE` Oracle would automatically determine the sample size to give good statistics. However, `AUTO_SAMPLE_SIZE` had an Achilles heel; the sample size determined by `AUTO_SAME_SIZE` may be inaccurate for computing statistics when there is an extreme skew in the data. In Oracle Database 11g the sampling algorithm has been completely rewritten. The new algorithm is hash based and provides deterministic statistics, which have the accuracy of computing the statistics but with the speed of a 10% sample. This new algorithm is used when estimate_percent is AUTO_SAMPLE_SIZE (default) in any of the DBMS_STATS.GATHER_*_STATS procedures.

Automatic statistics gathering job

Oracle will automatically collect statistics for all database objects, which are missing statistics or have stale statistics by running an Oracle AutoTask during a predefined maintenance window (10pm to 2am weekdays and 6am to 2am at the weekends).

This AutoTask gathers optimizer statistics by calling the internal procedure DBMS_STATS.GATHER_DATABASE_STATS_JOB_PROC. This procedure operates in a very similar fashion to the DBMS_STATS.GATHER_DATABASE_STATS procedure using the GATHER AUTO option. The primary difference is that Oracle internally prioritizes the database objects that require statistics, so that those objects, which most need updated statistics, are processed first. You can verify that the automatic statistics gathering job exists by viewing the DBA_AUTOTASK_CLIENT_JOB view:

```
SELECT client_name, status
FROM DBA_AUTOTASK_TASK
WHERE client_name like 'auto optimizer %';
```

Statistics on a table are considered stale when more than STALE_PERCENT (default 10%) of the rows are changed (total # of inserts, deletes, updates) in the table. Oracle monitors the DML activity for all objects and records it in the SGA. The monitoring information is periodically flushed to disk and is exposed in the *_tab_modifications view.

```
SELECT TABLE_NAME, INSERTS, UPDATES, DELETES
FROM USER_TAB_MODIFICATIONS;
```

It is also possible to manually flush this data by calling the procedure DBMS_STATS.FLUSH_MONITORING_INFO.

The automatic statistics-gathering job uses the default parameter values for the DBMS_STATS procedures. If you wish to change these default values you can use the DBMS_STATS.SET_GLOBAL_PREFS procedure. Remember these values will be used for all schemas including 'SYS'. To change the 'STALE_PERCENT' you can use

```
BEGIN
DBMS_STATS.SET_GLOBAL_PREFS('STALE_PERCENT','5');
END;
```

If you already have a well-established statistics gathering procedure or if for some other reason you need to disable automatic statistics gathering altogether, the most direct approach is to disable the GATHER_STATS_JOB as follows:

```
BEGIN
  DBMS_AUTO_TASK_ADMIN.DISABLE(
    client_name => 'auto optimizer stats collection',
    operation => NULL,
    window_name => NULL);
END;
```

If you choose to switch off the automatic statistics gathering job for your main application schema, consider leaving it on for the dictionary tables. You can do this by changing the value of `AUTOSTATS_TARGET` to `ORACLE` instead of `AUTO` using `DBMS_STATS.SET_GLOBAL_PREFS`.

SQL Plan Management

Execution plan stability has always been somewhat of a holy grail in the optimizer space and several features have been introduced in previous releases to improve it, such as Stored Outlines and SQL Profiles. However, these methods would also prevent the optimizer from finding better plans when data volume changes. In Oracle Database 11g plan stability has been addressed once and for all with the introduction of SQL Plan Management (SPM).

SQL plan management (SPM) ensures that runtime performance will never degrade due to the change of an execution plan. To guarantee this, only accepted (trusted) execution plans will be used; any plan will be tracked and evaluated at a later point in time and only accepted as verified if the new plan performs better than an accepted plan. SQL Plan Management has three main components:

1. SQL plan baseline capture:
   Create SQL plan baselines that represents accepted execution plans for all relevant SQL statements. The SQL plan baselines are stored in a plan history inside the SQL Management Base in the `SYSAUX` tablespace.
2. SQL plan baseline selection
   Ensure that only accepted execution plans are used for statements with a SQL plan baseline and track all new execution plans in the history for a statement as unaccepted plan. The plan history consists of accepted and unaccepted plans. An unaccepted plan can be unverified (newly found but not verified) or rejected (verified but not found to performant).
3. SQL plan baseline evolution
   Evaluate all unverified execution plans for a given statement in the plan history to become either accepted or rejected.

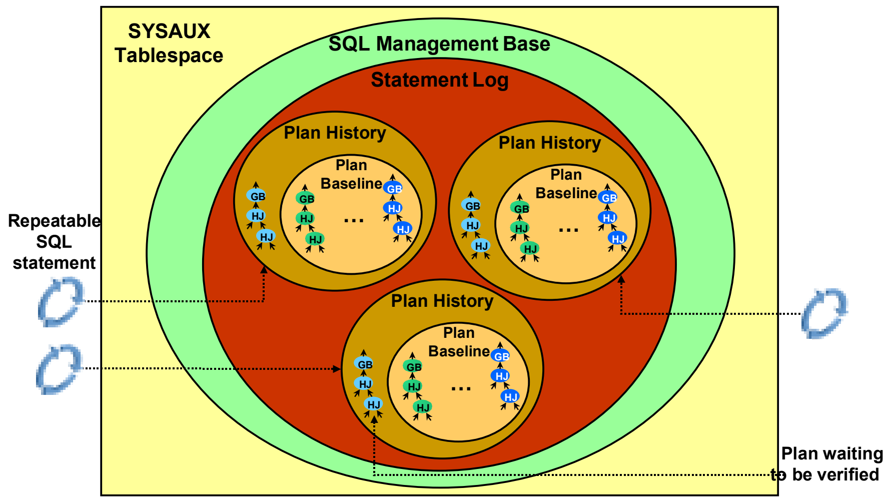


Figure 2 SQL Management base, consisting of the statement log and plan histories for repeatable SQL Statements.

SQL plan baseline capture

For SPM to work, you must first seed the SQL Management Base with the execution plans, which will become the SQL plan baseline for each statement. There are two different ways to populate a SQL Management Base:

- Automatic capture
- Bulk load

Automatic capture – "on the fly"

Automatic capture can be switched on by setting the init.ora parameter `OPTIMIZER_CAPTURE_SQL_PLAN_BASELINES` to `TRUE` *(*default `FALSE`*)*. When automatic capture is enabled, the SPM repository will be automatically populated for any repeatable SQL statement. To identify repeatable SQL statements, the optimizer will log the identity (SQL Signature) of each SQL statement into a statement log the first time it is compiled. If the SQL statement is processed again (executed or compiled) the presence of its identity in the statement log will signify it to be a repeatable statement. A SQL plan history will be created for the

Capture plans “on the fly “or bulk load SPM with plans from the cursor cache, a SQL Tuning Set or import plans from another system.

statement, which will include information used by the optimizer to reproduce the current execution plan, such as the SQL text, outline, bind variables, and compilation environment. The current execution plan will be added as the first SQL plan baseline and this plan will be marked as accepted. Only accepted plans will be used; if some time in the future a new plan is found for this SQL statement, the execution plan will be added to the plan history and will be marked for verification. It will only be marked accepted if its performance is better than that of a plan chosen from the current SQL plan baseline.

Bulk Load

Bulk 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. Bulk loading can be done in conjunction with, or instead of, automatic plan capture. Execution plans that are bulk loaded are automatically used to create new SQL plan baselines or to add to an existing one. The new SQL plan baselines are marked as accepted. The SQL Management Base can be bulk loaded using four different techniques:

1. Populate the execution plans for a given SQL Tuning Set (STS)
2. Populate the execution plans from Stored Outlines
3. Use the execution plans currently in the Cursor Cache
4. Unpack existing SQL plan baselines from a staging table

More information on using bulk loading during upgrade can be found in the Preparing to Upgrade Section.

SQL Plan Baseline 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_PLAN_BASELINES` is set to `TRUE` (default) then before the cost based plan is executed the optimizer will try to find a matching plan in the SQL statement’s SQL plan baseline; this is done as in-memory operation, thus introducing no measurable overhead to any application. If a match is found then it proceeds with this plan. Otherwise, if no match is found, the newly generated plan will be added to the plan history; it will have to be verified before it can be accepted as a SQL plan baseline. 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 verified/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.

SQL Plan Baseline Evolution

Plans can be manually evolved or verified at any time or you can schedule a database job to run the evolve process.

When the optimizer finds a new plan for a SQL statement, the plan is added to the plan history as a non-accepted plan that needs to be verified before it can become an accepted plan. It is possible to evolve a SQL statement's execution plan using Oracle Enterprise Manager or by running the command-line function `DBMS_SPM.EVOLVE_SQL_PLAN_BASELINE`. Using either of these methods you have three choices:

1. Accept the plan only if it performs better than the existing SQL plan baseline
2. Accept the plan without doing performance verification
3. Run the performance comparison and generate a report without evolving the new plan.

If you choose option 1, it will trigger the new plan to be evaluated to see if it performs better than a selected plan baseline. If it does, then the new plan will be added to the SQL plan baseline, as an accepted plan. If not the new plan will remain in the plan history as a non-accepted plan but its LAST_VERIFIED attribute will be updated with the current timestamp. A formatted text report is returned by the function, which contains the actions performed by the function as well as side-by-side display of performance statistics of the new plan and the original plan.

More information on SQL Plan Management can be found in the Preparing to Upgrade section.

Adaptive Cursor Sharing (Bind Peeking)

Oracle introduced the bind peeking feature in Oracle 9i. With bind peeking, the Optimizer peeks at the values of user-defined bind variables on the first invocation of a cursor. This allows the optimizer to determine the selectivity of any WHERE clause condition as if literals have been used instead of bind variables, thus improving the quality of the execution plan generated for statements using bind variables.

With Adaptive Cursor Sharing a Statement with binds can have multiple execution plans

However, there was a problem with this approach, when the column used in the WHERE clause with the bind contained a data skew. If there is data skew in the column, it is likely that a histogram has been created on this column during statistics gathering. When the optimizer peeks at the value of the user-defined bind variable and chooses a plan, it is not guaranteed that this plan will be good for all possible values for the bind variable. In other words, the plan is optimized for the peeked value of the bind variable, but not for all possible values.

In Oracle Database 11g, the optimizer has been enhanced to allow multiple execution plans to be used for a single statement that uses bind variables. This ensures that the best execution plan will be used depending on the bind value.

A cursor will be marked bind sensitive if the optimizer believes the optimal plan may depend on the value of the bind variable. When a cursor is marked bind sensitive, Oracle monitors the behavior of the cursor using different bind values, to determine if a different plan for different bind values is called for. A cursor is typically marked bind sensitive, because there is a histogram on the column with the bind variable. Since the presence of the histogram indicates that the column is skewed, different values of the bind variable may call for different plans.

If a different bind value is used in a subsequent execution, it will use the same execution plan because Oracle initially assumes it can be shared. However, the execution statistics for this 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" its behavior so that the same plan is not always shared for this query. Hence a new plan will be generated based on the new bind value and the cursor is marked bind-aware.

A bind-aware cursor may use different plans for different bind values, depending on how selective the predicates containing the bind variable are. Note that the original cursor generated for the statement will be discarded when the cursor switches to bind aware mode. This is a one-time overhead. The cursor is marked as not shareable (`is_shareable` in `V$SQL` is "N"), which means that this cursor will be among the first to be aged out of the cursor cache, and that it will no longer be used.

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 cursor cache. And the selectivity range for that cursor will be increased to include the selectivity of the new bind. This will result in one cursor being left behind that is in a not shareable state. This cursor will again be aged out if there is crowding in the cursor cache, and will not be used for future executions.

SQL Test Case Builder

If you ever need to contact Oracle Support about a SQL issue obtaining a reproducible test case is the single most important factor to ensure a speedy resolution. This can also be the longest and most painful step. A new tool called the SQL Test Case Builder has been introduced in Oracle Database 11g to help customers to gather as much information as possible relating to a SQL incident and package it up ready to send to Oracle. This package of information will allow a developer at Oracle to reproduce the problem standalone on a different Oracle instance and resolve the issue sooner. You can access the SQL Test Case Builder through Oracle Enterprise Manager or the `PL/SQL` package `DBMS_SQLDIAG`. There are two procedures relating to SQL Test Case Builder; `DBMS_SQLDIAG.EXPORT_SQL_TESTCASE`, which enables you to export a SQL test case for a given SQL statement into a given directory and `DBMS_SQLDIAG.IMPORT_SQL_TESTCASE`, which enables you to import a given SQL test case from a given directory.

To use SQL Test Case Builder, create an Oracle directory pointing to OS directory where the output files will go.

```
SQL> CREATE DIRECTORY EXPDP AS '/scratch/mcolgan/spm/tc';
```

Then call the `DBMS_SQLDIAG.EXPORT_SQL_TESTCASE` package and pass it the directory you just created and the `SQL_ID` for the statement in question. You will also need to supply a name for the testcase; in this example we chose tc.

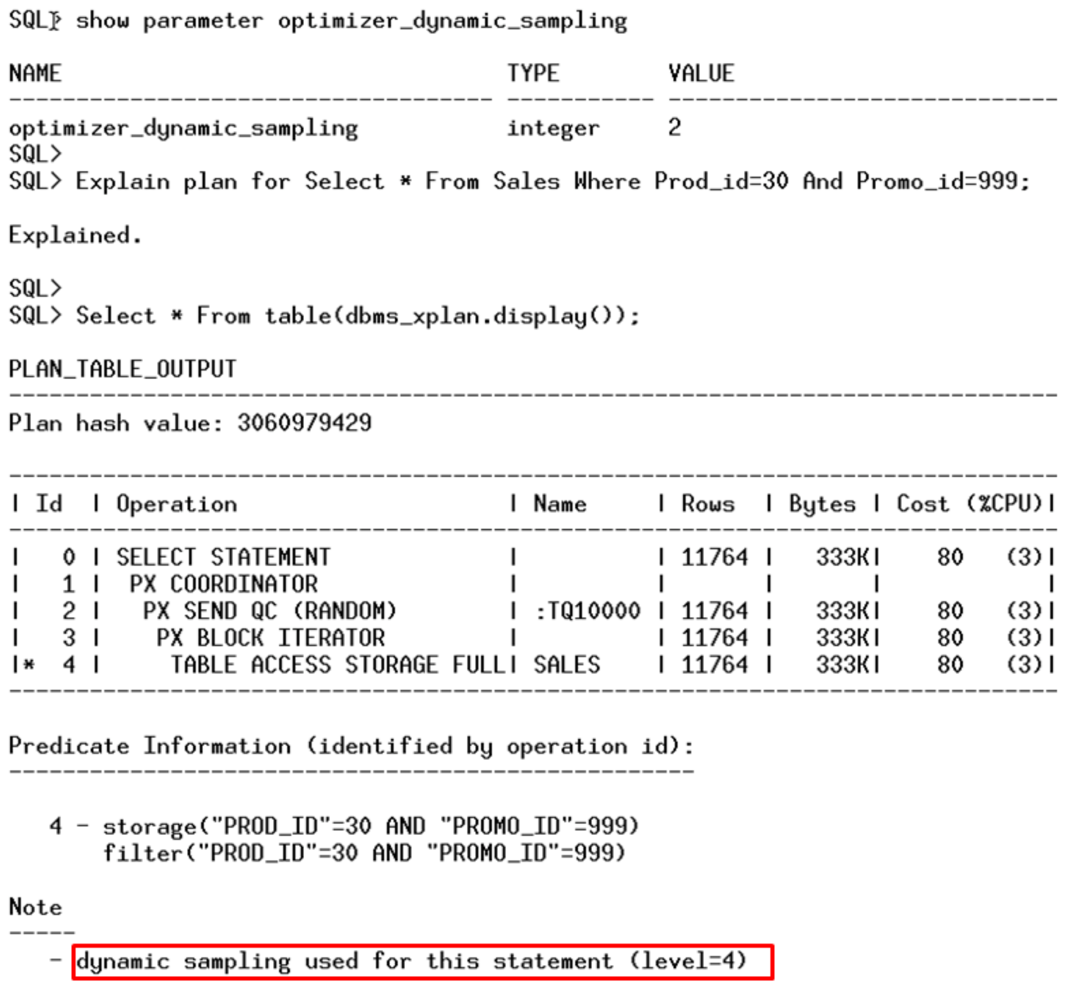
```
DECLARE tc clob;
BEGIN
DBMS_SQLDIAG.EXPORT_SQL_TESTCASE(
DIRECTORY=>'EXPDP',
SQL_ID=>'aqa1r0ca84rs1',
TESTCASE=>tc);
END;
```

After calling `EXPORT_SQL_TESTCASE` you will find several trace, dump and .sql files in the specified directory, along with a `README.txt`. You should create a single compressed file from this directory and upload it to Oracle Support. For security reason, the user data is not exported by default. You have the option to set `exportData` to `TRUE` to include the data.

Dynamic Sampling

Dynamic sampling (DS) was introduced in Oracle Database 9i Release 2 to improve the optimizer's ability to generate good execution plans. The goal of DS is to augment the optimizer statistics; it is used when regular statistics are not sufficient to get good quality cardinality estimates. During the compilation of a SQL statement, the optimizer decides whether to use DS or not by considering whether the available statistics are sufficient to generate a good execution plan. If the available statistics are not enough, dynamic sampling will be used. It is typically used to compensate for missing or insufficient statistics that would otherwise lead to a very bad plan.

From Oracle Database 11g Release 2 onwards the optimizer will automatically decide if dynamic sampling will be useful and what dynamic sampling level will be used for SQL statements executed in parallel even if statistics do exist for all objects accessed in the SQL statement. This decision is based on size of the tables in the statement and the complexity of the predicates. However, if the OPTIMIZER_DYNAMIC_SAMPLING parameter is explicitly set to a non-default value, then that specified value will be honored. You can tell if dynamic sampling kicks in by looks in the note section of the execution plan.



```
SQL> show parameter optimizer_dynamic_sampling

NAME                                 TYPE        VALUE
------------------------------------ ----------- ------------------------------
optimizer_dynamic_sampling           integer     2
SQL>
SQL> Explain plan for Select * From Sales Where Prod_id=30 And Promo_id=999;

Explained.

SQL>
SQL> Select * From table(dbms_xplan.display());

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------
Plan hash value: 3060979429

------------------------------------------------------------------------------
| Id  | Operation                   | Name     | Rows  | Bytes | Cost (%CPU)|
------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |          | 11764 |   333K|    80   (3)|
|   1 |  PX COORDINATOR             |          |       |       |            |
|   2 |   PX SEND QC (RANDOM)       | :TQ10000 | 11764 |   333K|    80   (3)|
|   3 |    PX BLOCK ITERATOR        |          | 11764 |   333K|    80   (3)|
|*  4 |     TABLE ACCESS STORAGE FULL| SALES    | 11764 |   333K|    80   (3)|
------------------------------------------------------------------------------

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

   4 - storage("PROD_ID"=30 AND "PROMO_ID"=999)
       filter("PROD_ID"=30 AND "PROMO_ID"=999)

Note
-----
   - dynamic sampling used for this statement (level=4)
```

New Cost-Based Transformations

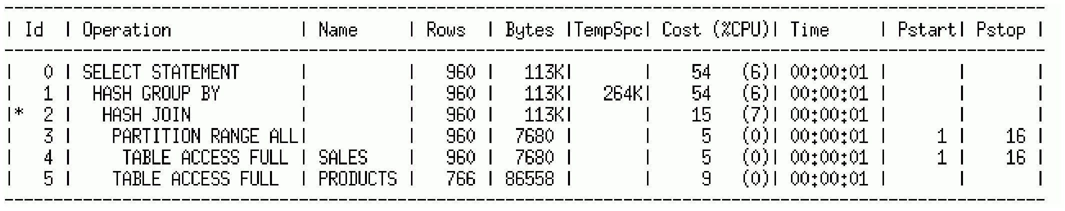
Oracle transforms SQL statements using a variety of sophisticated techniques during query optimization. The purpose of this phase of query optimization is to transform the original SQL statement into a semantically equivalent SQL statement that can be processed more efficiently. In Oracle Database 11g, several new cost-based transformations were introduced. We discuss below three of these transformations.

Group-By Placement

Group-by placement allows the optimizer to rewrite queries in order to minimize the number of rows necessary for subsequent joins by performing the group-by operation before some of the joins. For example, consider a query shown below.

```
SELECT p.prod_id, sum(s.quantity_sold)
FROM Products p, Sales s
WHERE p.prod_id = s.prod_id
GROUP BY p.prod_id;
```

In Oracle Database 10g the optimizer would have chosen a traditional plan of a hash join followed by a group by for this query.

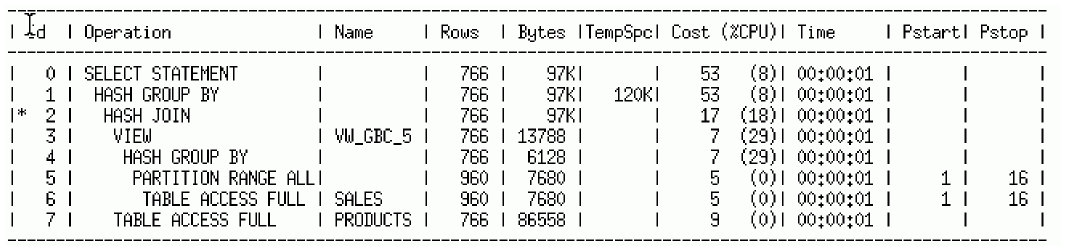


```
-----------------------------------------------------------------------------------------------------
| Id  | Operation             | Name     | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     | Pstart| Pstop |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |          |   960 |   113K|       |    54   (6)| 00:00:01 |       |       |
|   1 |  HASH GROUP BY        |          |   960 |   113K|   264K|    54   (6)| 00:00:01 |       |       |
|*  2 |   HASH JOIN           |          |   960 |   113K|       |    15   (7)| 00:00:01 |       |       |
|   3 |    PARTITION RANGE ALL|          |   960 |  7680 |       |     5   (0)| 00:00:01 |     1 |    16 |
|   4 |     TABLE ACCESS FULL | SALES    |   960 |  7680 |       |     5   (0)| 00:00:01 |     1 |    16 |
|   5 |    TABLE ACCESS FULL  | PRODUCTS |   766 | 86558 |       |     9   (0)| 00:00:01 |       |       |
-----------------------------------------------------------------------------------------------------
```

In Oracle Database 11g the Optimizer can transform this query to perform the group-by operation before the join. The statement will be rewritten as follows:

```
SELECT V.sumv, p.prod_id
FROM Products p,
      (SELECT sum(s.quantity_sold) as sumv, s.prod_id
       FROM Sales s
       GROUP BY s.prod_id) V
WHERE V.prod_id = p.prod_id
```

By rewriting the statement, the group by will now happens inside the view (VW_GBC_5 in the plan below) thus reducing the number of rows that must be processed by the hash join between `SALES` and `PRODUCTS`.



```
-------------------------------------------------------------------------------------------------------
| Id  | Operation               | Name     | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     | Pstart| Pstop |
-------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT        |          |   766 |    97K|       |    53   (8)| 00:00:01 |       |       |
|   1 |  HASH GROUP BY          |          |   766 |    97K|   120K|    53   (8)| 00:00:01 |       |       |
|*  2 |   HASH JOIN             |          |   766 |    97K|       |    17  (18)| 00:00:01 |       |       |
|   3 |    VIEW                 | VW_GBC_5 |   766 | 13788 |       |     7  (29)| 00:00:01 |       |       |
|   4 |     HASH GROUP BY       |          |   766 |  6128 |       |     7  (29)| 00:00:01 |       |       |
|   5 |      PARTITION RANGE ALL|          |   960 |  7680 |       |     5   (0)| 00:00:01 |     1 |    16 |
|   6 |       TABLE ACCESS FULL | SALES    |   960 |  7680 |       |     5   (0)| 00:00:01 |     1 |    16 |
|   7 |    TABLE ACCESS FULL    | PRODUCTS |   766 | 86558 |       |     9   (0)| 00:00:01 |       |       |
-------------------------------------------------------------------------------------------------------
```

Extended Join Predicate Push Down

In previous releases when you had a SQL statement where a view V and a table T were joined by a join predicate T.x = V.y, the Optimizer had only two possible join methods, a hash join or a sort merge join to join T and V.

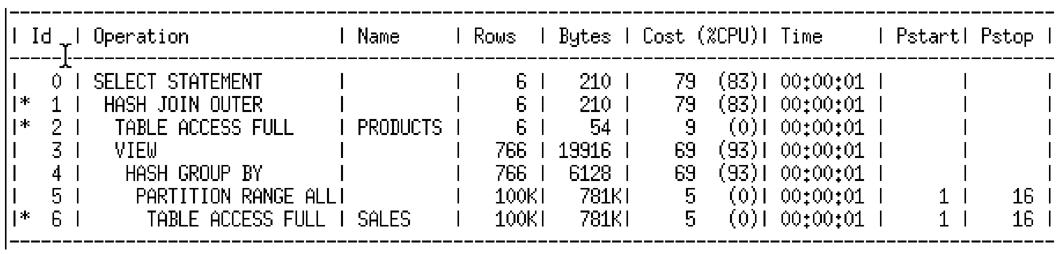
In Oracle Database 10g, we introduced the join predicate push down transformation, which enabled the optimizer to push the join predicate into the view. So the join T.x = V.y becomes

T.x = T2.y (where T2 is the table inside view V, which has the column y in it) thereby opening up the possibility of using a nested-loops join if an index is present on T2.y.

In Oracle Database 11g the join predicate push down capabilities have been extended to include group by, distinct, anti-join, and semi-joins. For the following query;

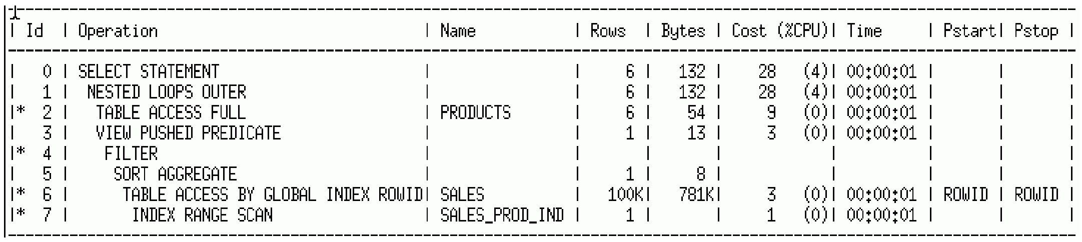
```
SELECT  p.prod_id, v1.row_count
FROM products p,
     (SELECT  s.prod_id, count(*) row_count
     FROM sales s
     WHERE s.quantity_sold BETWEEN 1 AND 47
     GROUP BY s.prod_id) v1
WHERE p.supplier_id = 12
AND   p.prod_id = v1.prod_id(+);
```

Although join predicate push down exists in Oracle Database 10g we can not use it due to the group by. In the 10g we see the view v1 being evaluated followed by a hash join to the `PRODUCTS` tables.



```
-------------------------------------------------------------------------------------------
| Id  | Operation              | Name     | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |          |     6 |   210 |    79  (83)| 00:00:01 |       |       |
|*  1 |  HASH JOIN OUTER       |          |     6 |   210 |    79  (83)| 00:00:01 |       |       |
|*  2 |   TABLE ACCESS FULL    | PRODUCTS |     6 |    54 |     9   (0)| 00:00:01 |       |       |
|   3 |   VIEW                 |          |   766 | 19916 |    69  (93)| 00:00:01 |       |       |
|   4 |    HASH GROUP BY       |          |   766 |  6128 |    69  (93)| 00:00:01 |       |       |
|   5 |     PARTITION RANGE ALL|          |  100K|   781K|     5   (0)| 00:00:01 |     1 |    16 |
|*  6 |      TABLE ACCESS FULL | SALES    |  100K|   781K|     5   (0)| 00:00:01 |     1 |    16 |
-------------------------------------------------------------------------------------------
```

However, in 11g join predicate pushdown has become possible and we are now taking advantage of the index on the sales table to do a nested-loops join instead of a hash join. The cost of the new plan has come down from 79 to 28 because of join predicate pushdown.



```
----------------------------------------------------------------------------------------------------------
| Id  | Operation                               | Name          | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
----------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                        |               |     6 |   132 |    28   (4)| 00:00:01 |       |       |
|   1 |  NESTED LOOPS OUTER                     |               |     6 |   132 |    28   (4)| 00:00:01 |       |       |
|*  2 |   TABLE ACCESS FULL                     | PRODUCTS      |     6 |    54 |     9   (0)| 00:00:01 |       |       |
|   3 |   VIEW PUSHED PREDICATE                 |               |     1 |    13 |     3   (0)| 00:00:01 |       |       |
|*  4 |    FILTER                               |               |       |       |            |          |       |       |
|   5 |     SORT AGGREGATE                      |               |     1 |     8 |            |          |       |       |
|*  6 |      TABLE ACCESS BY GLOBAL INDEX ROWID| SALES         |  100K|   781K|     3   (0)| 00:00:01 | ROWID | ROWID |
|*  7 |       INDEX RANGE SCAN                  | SALES_PROD_IND |     1 |       |     1   (0)| 00:00:01 |       |       |
----------------------------------------------------------------------------------------------------------
```

Null-Aware Anitjoin

A SQL statements that contains a NOT IN or NOT EXISTS subquery can often be rewritten into a query containing an anti-join. An anti-join produces matches if the key from table on the left hand side has no matches in the table on right hand side. Unnesting such a subquery can provide an order of magnitude performance improvement. For example

```
SELECT C.CUST_ID, C.CUST_FIRST_NAME, C.CUST_LAST_NAME
FROM CUSTOMERS C
```

```
WHERE C.CUST_ID NOT IN (SELECT ST.CUST_ID
                        FROM SALES_TRANSACTIONS_EXT ST
                        WHERE ST.UNIT_PRICE < 15);
```

If the columns C.CUST_ID or ST.CUST_ID are nullable (allows a null value), then the Optimizer cannot unnest the subquery in Oracle Database 10g using the existing (regular) anti-join. Therefore, the execution plan effectively becomes a Cartesian product.

```
-------------------------------------------------------------------------------------------------
| Id  | Operation                  | Name                   | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT           |                        |   629 | 13838 |   639   (1)| 00:00:08 |
|*  1 |  FILTER                    |                        |       |       |            |          |
|   2 |   TABLE ACCESS FULL        | CUSTOMERS              |   630 | 13860 |     5   (0)| 00:00:01 |
|*  3 |   EXTERNAL TABLE ACCESS FULL| SALES_TRANSACTIONS_EXT |     1 |    26 |     2   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------
```

In Oracle Database 11g, a new variant of anti-join, called null-aware anti-join, has been introduced; this allows the Optimizer to unnest the subquery, which means several new join methods are possible including nested-loops join, a hash join or a sort-merge join. In this example the 11g plan uses a hash join but in the execution plan it appears as a `HASH JOIN RIGHT ANTI NA`, where "NA' stands for null-aware. Note that the plan cost has come down from 639 to 38.

```
-------------------------------------------------------------------------------------------------
| Id  | Operation                  | Name                   | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT           |                        |   323 | 15504 |    38  (11)| 00:00:01 |
|*  1 |  HASH JOIN RIGHT ANTI NA   |                        |   323 | 15504 |    38  (11)| 00:00:01 |
|*  2 |   EXTERNAL TABLE ACCESS FULL| SALES_TRANSACTIONS_EXT |   408 | 10608 |    32  (10)| 00:00:01 |
|   3 |   TABLE ACCESS FULL        | CUSTOMERS              |   630 | 13860 |     5   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------
```

Preparing to Upgrade

Undertaking a database upgrade is a daunting task for any DBA. Once the database has been successfully upgraded you run the gauntlet of possible database behavior changes. On the top of every DBA's list of potential behavior changes are execution plan changes. In order to easily detect these changes, and rectify any execution plans that may have regressed, you need to have a very good understanding of the execution plans and Optimizer statistics you had before you began the upgrade. You also need to test your applications against the new release before upgrading your production system.

Capturing existing Execution plans

With the introduction of SQL Plan Management (SPM) in 11g you have an additional safety net to ensure execution plans do not change after the upgrade. In order to take full advantage of this safety net, you need to capture your existing execution plans before you upgrade so they can be used to seed SPM.

Using SQL Tuning Sets

If you have access to SQL Tuning Sets (STS) in the diagnostics pack then this is the easiest way to capture your existing 10g execution plans. An STS is a database object that includes one or more SQL statements along with their execution statistics, execution context and their current execution plan. An STS in Oracle Database 10gR1 will not capture the execution plans for the SQL statements so it can't be used to seed SPM. Only a 10gR2 STS will capture the plans.

To begin you will need to create a new STS. You can either do this through Oracle Enterprise Manager (EM) or using the `DBMS_SQLTUNE` package. In this example we will use `DBMS_SQLTUNE`

```
BEGIN
 SYS.DBMS_SQLTUNE.CREATE_SQLSET (
             sqlset_name  => 'SPM_STS',
             description => '10g plans');
END;
```

Once the STS has been created you need to populate it. You can populate an STS from the workload repository, another STS, or from the cursor cache. In this case we will capture the SQL statements and their execution plans from the cursor cache. This is a two-step process. In the first step we create a ref cursor to select the specified SQL from the cursor cache (in this case all non sys SQL statements). Then we use that ref cursor to populate the STS.

```
DECLARE
   stscur   dbms_sqltune.sqlset_cursor;
BEGIN
   OPEN stscur FOR
     SELECT VALUE(P)
     FROM   TABLE(dbms_sqltune.select_cursor_cache(
            'parsing_schema_name <> ''SYS''',
            null, null, null, null, 1, null, 'ALL')) P;

-- populate the sqlset
   dbms_sqltune.load_sqlset(sqlset_name      => 'SPM_STS',
                            populate_cursor  => stscur);
END;
```

Using Stored Outlines

If you don't have access to SQL Tuning Sets you can capture your existing execution plan using Stored Outlines. 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. Start a new session and issue the following command to 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.

   ```
   SQL > ALTER SYSTEM set CREATE_STORED_OUTLINES=OLDPLAN;
   ```

NOTE: Ensure that the user for which the Stored Outlines are to be created has the `CREATE ANY OUTLINE` privilege. If they don't the Stored Outlines will not be captured.

2. Now execute your workload either by running your 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.

3. Once you have executed your critical SQL statements you should turn off the automatic capture by issuing the following command:

```
SQL > ALTER SYSTEM set CREATE_STORED_OUTLINES=false;
```

4. To confirm you have captured the necessary Stored Outlines issue the following SQL statement.

```
SQL> SELECT name, sql_text, category FROM user_outlines;
```

NOTE: Each Stored Outline should have the OLDPLAN category.

5. The actual Stored Outlines are stored in the OUTLN schema. Before you upgrade you should export this schema as a backup.

```
exp outln/outln file=soutline.dmp owner=outln rows=y
```

Note: If you are not planning on doing an in-place upgrade you will have to move the STS, SQL Trace files or Stored Outlines to the Oracle Database 11g system.

Capturing existing Optimizer Statistics

Before doing the upgrade you need to capture the current set of optimizer statistics. You will use this set of statistics after the upgrade until the system is stable. You want to change as little as possible during the upgrade in order to make it easier to diagnose any changes that may occur. Since statistics have the biggest impact on the optimizer it is advisable for them to stay constant during the upgrade. The best way to keep a backup of the statistics is to export a complete set of Optimizer statistics into a statistics table using `DBMS_STATS.EXPORT_*_STATS`.

Begin by creating the stats table.

```
BEGIN
DBMS_STATS.CREATE_STATS_TABLE('SYS','MY_STATS_TAB');
END;
/
```

Then export the statistics for your critical schemas and select a stats_id to make it easy to identify your 10g statistics.

```
BEGIN
DBMS_STATS.EXPORT_SCHEMA_STATS(‘SH’,'MY_STATS_TAB',’10g_stats’);
END;
/
```

Finally as an extra precaution, export the stats table as a backup.

Testing your application

It is absolutely vital that you test your complete application on Oracle Database 11g before you upgrade. Testing will give you an insight into what might change after the upgrade and provide you an opportunity to test your safety net (SPM). There is always the potential you will miss a critical SQL statement when you gather your 10g plans, doing a full application test will give you a chance to confirm you captured everything you need.

If your application currently uses optimizer hints then it is advisable to test the application, in Oracle Database 11g, without the hint. Typically hints are added to a statement or an application to work around a limitation or problem encountered in an earlier version of the Optimizer. Often we find removing the hints gives better execution plans in newer releases of the database. The easiest way to test without hints is to set the under score parameter `_OPTIMIZER_IGNORE_HINTS` to `TRUE`.

Pre-Upgrade Checklist

Before you upgrade your production system to Oracle Database 11g you must collect and save the following pieces of information to ensure you have a clear baseline that can be used for future comparison if the need arises.

1. Gather Instance-wide performance statistics from the Production database (during peak load times). These Instance-wide performance statistics include:
    a. Statspack or AWR data and reports. You will need hourly reports for at least 7 days. By default, AWR will take hourly snapshots that automatically capture the execution plans for the top SQL statement and these reports will be retained for the last 8 days. However if you have changed the `STATISTICS_LEVEL` parameter or the retention level you may need to change them back to default at least a week before the upgrade. If you are using Statspack, you will need to configure it to take level 7 snapshots so you can collect segment statistics and plan information.
    b. OS statistics including CPU, memory and IO (such as `sar, vmstat, iostat`)
2. Be sure to perform all business critical transactions as well as month-end processes and common ad-hoc queries during the baseline capture.
3. Export the Statspack schema owner, PERFSTAT and keep the export file as backup. Note this step is not necessary if you are using AWR.
4. Export a complete set of Optimizer statistics into a statistics table, and export the table as a backup.

5. Make a backup of your init.ora file.
6. Capture the execution plan for key statements include the current Top SQL statements, and any important SQL statements in the application. (See the capture existing execution plan section above for details)

Post Upgrade

Once you have completed the software upgrade, but before you restart the applications and allow users back on the system, you should populate SQL Plan Management (SPM) with the 10g execution plans you captured before the upgrade. Seeding SPM with the 10g execution plans ensures that the application will continue to use the same execution plans you had before the upgrade. Any new execution plans found in Oracle Database 11g will be recorded in the plan history for that statement but they will not be used. When you are ready you can evolve or verify the new plans and only implement those that perform better than the 10g plan.

Populating SQL Plan Management with 10g plans

You can bulk load execution plans into SPM using four different techniques:

1. Populate the execution plans for a given SQL Tuning Set (STS)
2. Populate the execution plans from Stored Outlines
3. Use the execution plans currently in the Cursor Cache
4. Unpack existing SQL plan baselines from a staging table

Regardless of where you bulk loaded the plans from, each plan will be automatically accepted to create a new SQL plan baseline or it will be added to an existing one. During an upgrade it's most likely you will be bulk loading plan using the first three options. We will discuss each of these options in detail below. More information on unpacking SQL plan baselines from a staging table can be found in Chapter 15 of the Oracle Database Performance and Tuning Guide.

Bulk loading from a SQL Tuning Set

Execution plans can be bulk loaded from an STS into SPM using the PL/SQL procedure `DBMS_SPM.LOAD_PLANS_FROM_SQLSET` or through Oracle Enterprise Manager (EM).

```
SQL> Variable cnt number
SQL> execute :cnt := DBMS_SPM.LOAD_PLANS_FROM_SQLSET( -
                           sqlset_name  => 'SPM_STS');
```

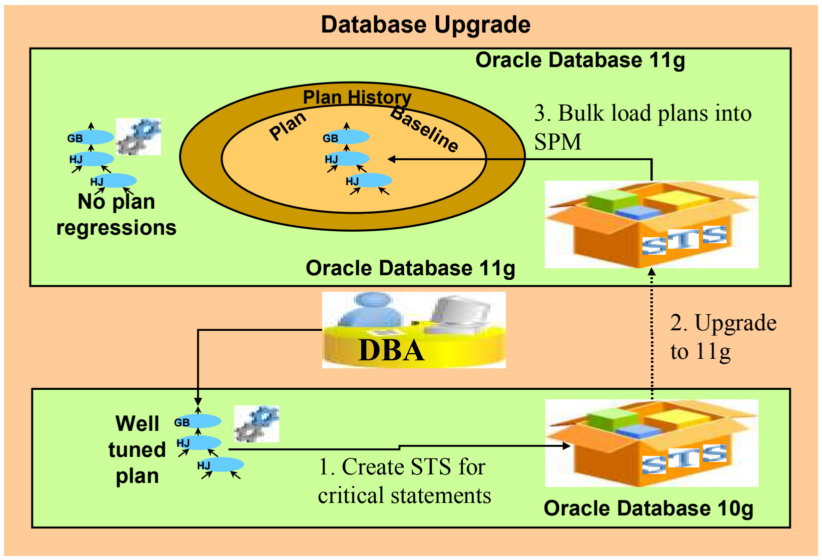


Figure 3 Upgrading using STS to seed SPM.

Bulk loading from Stored Outlines

In Oracle Database 11gR2 you can migrate stored outlines for one or more SQL statements to SQL plan baselines using `DBMS_SPM.MIGRATE_STORED_OUTLINE` or through Oracle Enterprise Manager (EM). You can specify which stored outline(s) to be migrated based on outline name, SQL text, or outline category, or migrate all stored outlines in the system to SQL plan baselines.

```
SQL> variable report clob;
-- Migrate a single Stored Outline by name
SQL> exec :report:=DBMS_SPM.MIGRATE_STORED_OUTLINE(
       attribute_name=>'OUTLINE_NAME',
       attribute_value => 'stmt01');

-- Migrate all Stored Outlines
SQL> exec :report:=DBMS_SPM.MIGRATE_STORED_OUTLINE(
       attribute_name=>'ALL');
```

Bulk loading from the Cursor Cache

It is also possible to load plans for statements directly from the cursor cache into SPM. 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. The plans can be loaded using the

PL/SQL procedure DBMS_SPM.LOAD_PLANS_FROM_CURSOR_CACHE or through Oracle Enterprise Manager. Loading plans directly from the cursor cache can be extremely useful if you were unable to capture plans for some or all of your statements prior to the upgrade.

By setting the parameter OPTIMIZER_FEATURES_ENABLE to the 10g version used before the upgrade, you should be able to revert back to the same execution plans you had prior to the upgrade. By capturing these 10g execution plans from the cursor cache you will be able to seed SPM with the 10g plans before setting OPTIMIZER_FEATURES_ENABLE to your 11g version. Note you must use the same Optimizer statistics you were using in 10g. Statistics should not be re-gathered until all of the 10g plans have been captured.

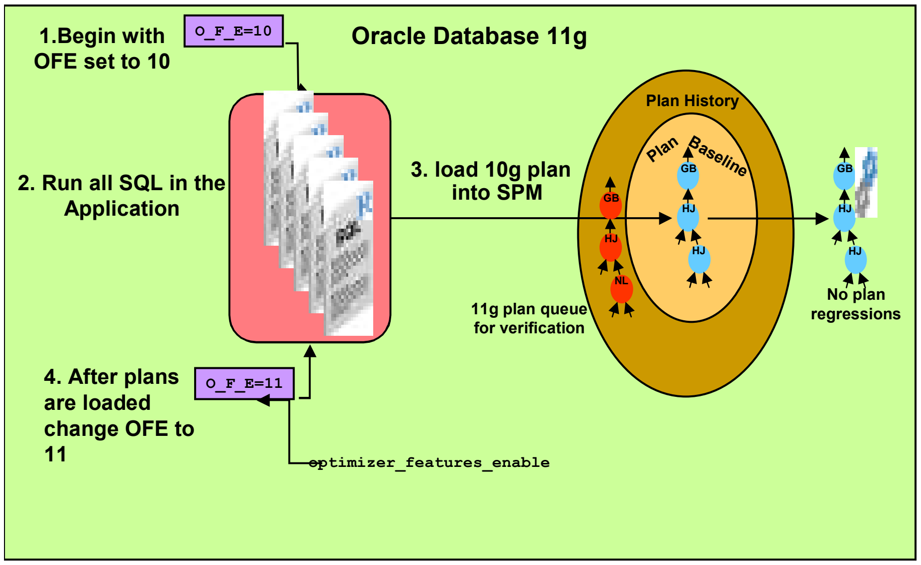

1.Begin with OFE set to 10
O_F_E=10
Oracle Database 11g
2. Run all SQL in the Application
3. load 10g plan into SPM
Plan History
Plan Baseline
GB
HJ
NL
11g plan queue for verification
No plan regressions
4. After plans are loaded change OFE to 11
O_F_E=11
optimizer_features_enable

Figure 4 Upgrading by capturing 10g plans from the cursor cache.

Confirming your SQL Plan Baseline is being used

Once SPM has been seeded with the pre-upgrade plans, you can check that the new create SQL plan baselines are being used by looking at the notes section of the execution plan. If a SQL plan baseline is in use you will see the phrase “SQL plan baseline XXXXX used for this statement” in the note section.

```
SQL> explain plan for
  2  SELECT *
  3  FROM  sh.sales
  4  WHERE quantity_sold > 40
  5  ORDER BY prod_id;
```

```
Explained.
SQL> SELECT * FROM table(dbms_xplan.display(null, null, 'basic +note'));


PLAN_TABLE_OUTPUT
-----------------------------------------------------------------
Plan hash value: 1421641795
--------------------------------------
| Id  | Operation            | Name  |
--------------------------------------
|   0 | SELECT STATEMENT     |       |
|   1 |  SORT ORDER BY       |       |
|   2 |   PARTITION RANGE ALL|       |
|   3 |    TABLE ACCESS FULL | SALES |
--------------------------------------
Note
------
- SQL plan baseline "SQL_PLAN_2kgpw0an1uph654bc8843" used for this
statement
```

Handling Optimizer Statistics after Upgrade

After the upgrade you want to use the 10g statistics until the system is stable. After the upgrade and before you gather statistics for the first time you should switch on incremental statistics for all partitioned tables. This will allow you to generate global statistics for partitioned tables, from partition level statistics, and greatly reduce the amount of time it takes to gather statistics on larger tables.

```
BEGIN
DBMS_STATS.SET_GLOBAL_PREFS('INCREMENTAL','TRUE');
END;
```

As a precautionary step, you may want to temporarily switch on pending statistics for all objects. This will allow you to have an opportunity to test the new statistics before they are published and start to be used in your production environment.

```
BEGIN
DBMS_STATS.SET_GLOBAL_PREFS('PUBLISH','FLASE');
END;
```

At this point it is safe to gather 11g statistics. By not specifying any parameter values, the default values will be use, which means it will automatically use the new statistics algorithm. All of the statistics gathered will be stored as pending, thus allowing you to test them without impacting the current environment.

```
BEGIN
DBMS_STATS.GATHER_SCHEMA_STATS('SH');
END;
```

Alternatively you can export pending statistics from the production system and import them into a test system, then test their impact before publishing them. Pending statistics can be exported using procedure EXPORT_PENDING_STATS.

```
BEGIN
DBMS_STATS.EXPORT_PENDING_STATS(…);
END
```

You can now test your critical SQL statement with the pending statistics by changing the value of the `OPTIMIZER_USE_PENDING_STATISTICS` parameter to `TRUE` at a session level. The Optimizer will then use pending statistics for all SQL statements issued in this session.

```
ALTER SESSION set optimizer_use_pending_statistics=TRUE;
```

Finally when you have validated the 11g statistics you should turn off pending statistics and publish the new statistics.

```
BEGIN
DBMS_STATS.SET_GLOBAL_PREFS('PUBLISH','TRUE');
END

BEGIN
DBMS_STATS.PUBLISH_PENDING_STATS();
END
```

Post-Upgrade Checklist

Once you have successfully upgraded to Oracle Database 11g and your application is up and running again, you will have to monitor your environment carefully to ensure you do not encounter any performance issues or plan regressions. The steps below outline what you should do.

1. If you have not licensed the Diagnostic Pack, install or upgrade Statspack and set the level to 7. Follow Statspack instructions specially if upgrading it.
2. Schedule Statspack snapshots every hour. This will let Statspack capture expensive SQL in your 11g environment. If you have licensed the Diagnostic Pack then you can use the AWR reports, which will be automatically captured hourly.
3. Capture OS statistics, which coincide with your Statspack reports.
4. Identify the expensive SQL (top SQL) using Statspack or AWR reports. Compare these SQL statements to the top SQL statements you had prior to the upgrade. If they are not the same you will need to investigate why.

    a. If the pre-upgrade instance is still available, execute the source transaction in both instances (10g and 11g) and compare the execution plans, buffer gets, CPU time, and total elapse times.

   b. If pre-upgrade instance is no longer available, use the export of the PERFSTAT schema (you took as part of the pre-upgrade check list) to find the SQL statement and its execution plan (use script sprepsql.sql)

5. Determine root cause of sub optimal plans and take corrective action. Corrective action may be in the form of: re-gathering statistics with different parameter settings, use SQL Tuning Advisor, index creation, creation of a SQL Profile, use of Optimizer hints, research of known Bugs, logging an SR, etc.

Correcting Regressed SQL Statements

There is always a chance that you missed a SQL statement when you were recording the 10g execution plans or a new module gets rolled out and you discover some poorly performing SQL statement. In this cause you will need to correct the regressed SQL statement. If you don't have a known good plan for this statement you will have to do some investigation into the cause of the problem. In this section we will discuss the different options you have to correct this regressed SQL statement.

Examine statistics used for SQL statement

When the execution plan for a SQL statement is sub-optimal, one of the first places you should look is at the statistics used for that statement. The statistics have the biggest influence on the optimizer and are the one aspect that will change over time as the data set changes.

The first thing you should check is that the statistics are accurate and up to date for every object used in the query. Since Oracle Database 10g, the parameter `OPTIMIZER_MODE` is set to `ALL_ROWS` by default. That means the cost-based Optimizer will always be used and any object that does not have statistics will have statistics gathered at the time of hard parse using dynamic sampling. Dynamic sampling is a great way to augment existing statistics but it isn't a good substitute for genuine statistics. If dynamic sampling is being used, it will be shown in the note section of the plan.

If the statistics are both current and accurate, then its possible that the sub-optimal plan is being caused by real-world relationship in the data or by the presence of a function on one of the columns in the where clause. You can address both of these situations by using the new extended statistics feature in 11g.

Using SQL Tuning Advisor

You can invoke the SQL tuning advisor for a poor performing SQL statement, which will trigger the Optimizer to analyze the problem SQL statement and recommend a solution. The tuning process that is under taken is fully cost-based; it takes into account the past execution statistics of the SQL statement and customizes the optimizer settings for that statement. It also collects auxiliary information in conjunction with the regular statistics. The recommendation can fall into one of the following categories;

Statistics Analysis: Check each object in the query for missing or stale statistics, and makes recommendation to gather relevant statistics if they are needed.

SQL Profiling: The initial cardinality estimates used by the optimizer are checked and auxiliary information is collected to remove all estimation errors. If the estimates are founded to be lacking or if the auxiliary information proves to produce a more preformant execution plan a SQL Profile using the auxiliary information will be generated and the end-user will be recommend to create it.

Access Path Analysis: The Optimizer explores whether a new index can be used to significantly improve access to each table in the query, and when appropriate makes recommendations to create such indexes.

SQL Structure Analysis: The Optimizer tries to identify SQL statements that lend themselves to bad plans, and makes relevant suggestions to restructure them.

Using SQL Repair Advisor

If the statement you are investigating is failing with a critical error (ORA-600), then you run the SQL Repair Advisor from the Problem Details page in Oracle Enterprise manger or by using `DBMS_SQLDIAG.CREATE_DIAGNOSIS_TASKS` procedure. The SQL Repair Advisor will investigate the SQL statement to see if it can come up with an alternative execution plan that will not cause the critical error. If it finds such a plan it will recommend you apply a SQL patch for this statement. Accepting the patch will allow your SQL statement to execute without hitting the critical error but it may not be the most performant execution plan. Therefore you should still open a Service Request with Oracle support and remove the SQL patch once you get an official fix from Oracle.

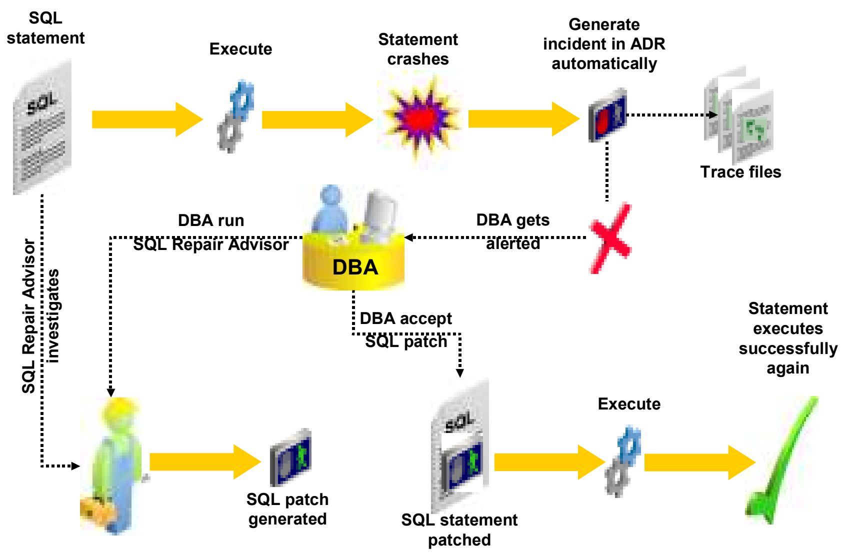


Figure 5 Workflow showing when SQL Repair Advisor can be used to correct regressed SQL

Using Hints and SPM

In a small number of cases it may become necessary to correct a regressed SQL statement by adding hints. However, for a third-party application it may not be possible to add hints to statement and if you enable automatic plan capture in SQL Plan Management (SPM) you will actually creating a new SQL plan baseline for the **modified** (hinted) statement and not for the original statement. What you most likely want is to add this hinted plan to the plan history of the **original** SQL statement. 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. In a SQL*Plus session 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
-------------     ---------------------------------------
74hnd835n81yv     select SQL_ID, SQL_FULLTEXT from v$SQL
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. These can found by looking in `DBA_SQL_PLAN_BASELINE` view.

```
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,SUM  SQL_PLAN_byr4v13vkrrjy42949306  YES
```

5. Using `DBMS_SPM.ALTER_SQL_PLAN_BASELINE` disable the bad plan

```
SQL> variable cnt number;
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');

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,SUM  SQL_PLAN_byr4v13vkrrjy42949306  NO
```

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

```
SQL> SELECT /*+ INDEX(p) */ 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;
```

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

9t5v8swp79svs   3262214722      select SQL_ID, SQL_FULLTEXT from v$SQL
djkqjd0kvgmb5   3074207202      SELECT /*+ INDEX(p) */ PROD_NAME,
```

8. 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`.

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

Conclusion

Since the introduction of the cost-based optimizer (CBO) in Oracle 7.0, people have been fascinated by the CBO and the statistics that it feeds on. It has long been felt that the internals of the CBO were shrouded in mystery and that a degree in wizardry was needed to work with it. In Oracle Database 11g the CBO's effectiveness and ease of use have been greatly improved. Most notably, the introduction of SQL Plan Management (SPM), which has simplified the daunting task of upgrading the database by ensuring that the Optimizer will use only the execution plans you had before the upgrade, if the new execution plans regress.

By outlining in detail the changes made in this release to the CBO and its statistics, we hope to remove some of the mystery that surrounds them and help make the upgrade process smoother, as being forewarned is being forearmed.

White Paper Title: Upgrading from Oracle Database 10g to 11g: what to expect from the Optimizer
November 2010
Author: Maria Colgan

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
oracle.com