Almost Automagic: Adaptive SQL Plan Management, Automatic Dynamic Sampling and Extended Column Statistics in Oracle 12*c*R1

By Jim Czuprynski

Arup Nanda, Editor

The past articles in this series have concentrated on several of the newest Oracle Database 12cR1 feature sets that have directly improved the Oracle query optimizer, especially Adaptive Execution Plans, Dynamic Plans, SQL Plan Directives, and the new Top Frequency and Hybrid histograms. But there are a few other 12c features that augment those already introduced in earlier releases, especially Oracle 11gR1. In this article, I'll delve into the enhancements that Oracle 12c has made to SQL Plan Management (SPM), Extended Statistics and Dynamic Sampling. I'll start with a quick review of SPM's newest features in 12cR1, especially the ability to automatically evolve a SQL Plan Baseline during the database's nightly maintenance tasks.

SQL Plan Management (SPM) Automation

Oracle Database 11*g* Release 1 (11*g*R1) implemented significant changes in the way an Oracle DBA would tune the performance of repeatable SQL statements. Adaptive Cursor Sharing (ACS) now meant that SQL statements with bind variables could become bind-aware; it also meant that different ranges of bind variable values could be directly tied to their appropriate SQL execution plans, which would now be stored as SQL profiles within the SQL Management Base (SMB). Even better — perhaps after a database upgrade, or once better optimizer statistics were available — a SQL plan that eventually demonstrated better performance could be evolved over time and be recognized as the best plan for the SQL statement's next execution.

These features — collectively termed SQL Plan Management (SPM) — have been enhanced even more dramatically in Oracle Database 12*c*R1:

- SPM will now store both non-accepted as well as accepted execution plans for SQL statements that have been executed more than once.
- The SQL Management Base now captures these execution plans so that it's no longer necessary to re-parse a SQL statement to re-evaluate its cost. So if DBMS_XPLAN.DISPLAY_PLAN_BASELINE is called to display a plan stored with the SMB, that information is returned much faster.
- Since the execution plan data is retained in the SMB for some time, this also yields another significant advantage: the ability to see an execution plan's statistics and "footprint" at a prior historical point in time.

Adaptive SQL Plan Management

During the normal nightly/weekend maintenance window, a new maintenance task named SYS_AUTO_SPM_EVOLVE_TASK will be automatically executed. This new task looks at any non-accepted plans and will attempt to evolve them automatically, beginning with the most recently added plans and then proceeding to evaluate older plans:

- If a non-accepted plan still appears to perform poorly after re-evaluation, then Oracle 12*c*R1 will tag the "poorer" plan to make sure it doesn't get re-evaluated until another 30 days have elapsed (and even then, only if the plan's statement is still currently active).
- On the other hand, if a non-accepted plan now appears to perform better than an existing accepted or non-accepted plan, then Oracle 12*c*R1 automatically enables that plan, and thus the optimizer can take immediate advantage of it.

This automatic SPM evolution process's results can be generated and viewed via a call to the new REPORT_AUTO_EVOLVE_TASK procedure of package DBMS_SPM.

To illustrate these new features, I first set the OPTIMIZER_CAPTURE_SQL_PLAN_BASELINES initialization parameter to a value of TRUE for my session, and then executed the query in Listing 1 twice so that its current execution plan (shown in the EXPLAIN PLAN in Figure 1) is captured within my Oracle 12*c*R1 database's SMB. Figure 2 shows the resulting evidence of its capture.

```
SELECT /* AUTOSPM_1 */
     C.cust_last_name
    ,SUM(S.amount_sold)
    ,SUM(S.quantity_sold)
  FROM
     sh.sales S
    ,sh.customers C
    ,sh.products P
 WHERE S.cust_id = C.cust_id
   AND S.prod_id = P.prod_id
   AND C.cust_last_name IN ('Farmer','York')
 GROUP BY C.cust_last_name
 ORDER BY C.cust_last_name;
```

Listing 1: SPM Demonstration Query

Now that I've got a suitable plan that's both ENABLED and ACCEPTED in the SMB for the query, I'll add a descending, non-unique index (see Listing 2) on the CUST_LAST_NAME column of table SH.CUSTOMERS to significantly improve the query's performance.

```
DROP INDEX sh.cust_ln_desc;
CREATE INDEX sh.cust_ln_desc
    ON sh.customers (cust_last_name DESC)
    TABLESPACE example;
```

Listing 2: Adding a New Index to Improve Query Performance

I'll then rerun the query a few more times to load the new plan in the SMB. However, note that even though it will be ENABLED, the plan will not be ACCEPTED until that plan has been evolved, either by automatic or manual means. Figure 3 shows its new execution plan, and Figure 4 shows the new state of the SMB.

Automatic SQL Plan Baseline Evolution

Even though I've now populated two execution plans for the same query in the SMB, Oracle 12*c*R1 will not evolve the new, better plan until the nightly maintenance window opens for my database. In prior releases, an Oracle DBA usually had to manually pursue any execution plan evolution via calls to DBMS_SPM.EVOLVE_SQL_PLAN_BASELINE. That's because SPM didn't offer any mechanism to automatically evolve an existing plan.

Figure 5 shows the results of the next evening's execution of the Automatic SQL Plan Baseline Evolution task (SYS_AUTO_SPM_EVOLVE_TASK) that's new in Oracle 12*c*R1. Note that the new plan (ccaf39cd) is now marked as both ENABLED and ACCEPTED, so it can now be leveraged for any future executions of the SQL statement.

SPM Evolve Advisor

If I don't want to wait until the next maintenance cycle for the Automatic SPM Evaluation task to be scheduled and executed, I can alternatively force an immediate evaluation of a newer, possibly more efficient plan in Oracle 12*c*R1 through the new SPM Evolve Advisor. This new advisor allows me to create a specific scheduled task that can be either executed immediately or at another regularly scheduled time outside of the normal maintenance cycle, and it comprises four new subprograms that are now part of the DBMS_SPM package in Oracle 12*c*R1:

- CREATE_EVOLVE_TASK builds a new SPM plan evolution task.
- EXECUTE_EVOLVE_TASK executes an existing SPM plan evolution task.
- REPORT_EVOLVE_TASK generates a report of all proposed SPM plan evolutions discovered.
- IMPLEMENT_EVOLVE_TASK performs the implementation recommendations that have been discovered and recommended earlier.

The PL/SQL code in Listing 3 demonstrates the manual creation and execution of a SPM Evolve Advisor task instead of waiting for the next execution of the Automatic SPM Evolve Advisor task:

```
-----
-- Invoking CREATE_EVOLVE_TASK procedure to build a new SPM plan evolution task
-- for a single SQL statement and SQL Plan Name
-----
SET SERVEROUTPUT ON
DECLARE
    task_name VARCHAR2(32) := 'SPME_1';
    exec_name VARCHAR2(32) := 'SPME_1';
BEGIN
    BEGIN
        DBMS_SPM.DROP_EVOLVE_TASK(task_name);
    EXCEPTION
        WHEN OTHERS THEN NULL;
    END;

    task_name :=
        DBMS_SPM.CREATE_EVOLVE_TASK(
            sql_handle => NULL
           ,plan_name => 'SQL_PLAN_9xwy49zgdcnptccaf39cd'
           ,time_limit => DBMS_SPM.AUTO_LIMIT
           ,task_name => task_name
           ,description => 'Build task to evolve +all+ non-accepted plans'
        );

    exec_name :=
        DBMS_SPM.EXECUTE_EVOLVE_TASK(
            task_name => task_name
           ,execution_name => exec_name
           ,execution_desc => 'Execute task to evolve +all+ non-accepted plans'
        );

EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Unexpected error during CREATE/EVOLVE SPM task: '
|| SQLERRM);
END;
/
```

Listing 3: Requesting Manual SQL Plan Evolution

The report in Figure 6 shows that Oracle 12*c*R1 is smart enough to recognize that the new plan is more efficient than its predecessor, so it automatically selects the more efficient SQL plan so it can be used immediately by any requesting server process. Listing 4 demonstrates how to use the IMPLEMENT_EVOLVE_TASK procedure to complete the acceptance of the recommended improved execution plan.

```
-----
-- IMPLEMENT_EVOLVE_TASK performs the implementation recommendations that have
-- been recommended by a particular SPM Advisor task
-----
VARIABLE acpt_plans NUMBER
BEGIN
    :acpt_plans :=
        DBMS_SPM.IMPLEMENT_EVOLVE_TASK(
         task_name => 'SPME_1'
        ,task_owner => NULL
        ,execution_name => 'SPME_1'
        ,force => TRUE
    );
    DBMS_OUTPUT.PUT_LINE(:acpt_plans || ' pending plan(s) have been successfully
accepted.');
END;
/
```

Listing 4: Using DBMS_SPM. IMPLEMENT_EVOLVE_TASK to Manually Accept Recommendations

Like the licensing requirements in earlier releases for SQL Plan Management, be aware that automatic evolution of SQL Plan Baselines does require licensing the Oracle 12*c*R1 Database Tuning Pack. Finally, note that whenever an Oracle 11*g*R1 or 11*g*R2 database is upgraded to Oracle 12*c*R1, all new SQL Plan Management data is captured automatically during execution of the plan(s) and loaded directly into the SMB.

Extending Extended Statistics

One really neat feature in Oracle 11*g*R1 was the capability to create extended statistics for column subset in a database table whenever queries constantly access those columns in consistent patterns. For example, it's pretty obvious to just about every user who queries a table that contains customer address information in specific columns like CITY, PROVINCE, POSTAL_CODE and

continued on page 8

COUNTRY that these columns have an implicit relationship; however, the Oracle query optimizer is simply not smart enough to recognize the relationships between those columns without some help, and that's when capturing extended statistics can yield much better query performance.

Another issue that often arises with a set of columns like those that I've just described occurs whenever those columns are used for filtering results, especially when used in equality conditions. One of the optimizer's main tasks is to make accurate determinations of a row set's cardinality, but even as smart as it is, it sometimes makes an erroneous assumption that a row set's cardinality will be significantly reduced because multiple equality conditions are present in a SQL statement. This is exactly the situation in which extended statistics can make a significant impact on performance because the optimizer will have much more accurate counts of the number of column value combinations.

Once I've determined which columns for a table could benefit from extended statistics, I would still have to use the Oracle-supplied DBMS_STATS.CREATE_EXTENDED_STATS procedure to identify those columns to the database. However, note that this procedure doesn't actually create extended statistics; that is still performed via procedure DBMS_STATS.GATHER_TABLE_STATS via either a manual execution or during the normally scheduled automatic statistics gathering process.

While extended statistics are obviously valuable, one glaring issue remains: The effective identification and collection of extended statistics requires an Oracle DBA to be intimately familiar with how applications are using columns that have implicit relationships. And given the breadth of an Oracle DBA's responsibilities these days, it's not unlikely that she may miss all but the most obvious of these relationships.

Automatically Capturing Extended Statistics Targets

Fortunately, Oracle 12*c*R1 expands the ability to detect just about any implied relationship between columns because it provides a new DBMS_STATS procedure, SEED_COL_USAGE, that allows the database to identify these relationships during a specific timeframe and capture them as targets for gathering extended statistics. Listing 5 shows several queries against two tables, SH.CUSTOMERS and AP.VENDORS that are potential candidates for extended statistics. Note that Oracle 12*c* does not require these queries to actually execute in order to capture this information; it is sufficient to simply ask the optimizer to make its best guess at an execution plan via EXPLAIN PLAN:

```
EXPLAIN PLAN FOR
SELECT *
  FROM sh.customers
 WHERE cust_city = 'San Jose'
   AND cust_state_province = 'CA'
   AND country_id = 52790;

EXPLAIN PLAN FOR
SELECT country_id, cust_state_province, COUNT(cust_city)
  FROM sh.customers
 GROUP BY country_id, cust_state_province
 ORDER BY country_id, cust_state_province;

EXPLAIN PLAN FOR
SELECT *
  FROM ap.vendors
 WHERE credit_card = 3728*
   AND credit_limit BETWEEN 250,000 and 500,000;

EXPLAIN PLAN FOR
SELECT *
  FROM ap.invoices INV,
       ap.invoice_items ITM
 WHERE INV.invoice_id = ITM.invoice_id
   AND INV.indicator 'A'
   and INV.credit_limit = 'Z';

EXPLAIN PLAN FOR
SELECT *
 FROM ap.vendors
 WHERE credit_card = 3728*
   AND credit_limit BETWEEN 250,000 and 500,000;
BETWEEN 250,000 and 500,000;
```

Listing 5: Extended Statistics Use Cases for SH.CUSTOMERS and AP.VENDORS

Listing 6 shows how I used the SEED_COL_USAGE procedure to monitor any implied relationships between columns as well as capture columns that are typically used in equality predicates, both of which could benefit from extended statistics, over a 900-second (15-minute) period specified for the procedure's third argument. (Note that the first two arguments could also specify the name of a SQL Tuning Set and its owner, respectively, if I chose to utilize it as a source of input instead of the database's actual SQL workload.)

```
BEGIN
    DBMS_STATS.SEED_COL_USAGE(
      ARG1 => NULL
     ,ARG2 => NULL
     ,ARG3 => 900);
END;
/
```

Listing 6: Activating Automatic Capture of "Best" Columns for Column and Expression Statistics

Listing 7 shows how Oracle 12*c*'s new DBMS_STATS procedure, REPORT_COL_USAGE, displays exactly how columns in each table had been used during the 15-minute observation period that just concluded. Note that REPORT_COL_USAGE not only identifies potential groups of columns for effective filtering, but also points to sets of columns that are used during aggregation like GROUP BY:

```
SQL> DBMS_STATS.REPORT_COL_USAGE('AP','VENDORS')

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 AP.VENDORS
..................................

1. ACTIVE_IND                      : EQ
2. CITY                            : EQ
3. COUNTRY                         : EQ
4. STATE                           : EQ
5. VENDOR_ID                       : EQ_JOIN
6. (CITY, COUNTRY)                 : FILTER
#####################################################################
```

```
SQL> DBMS_STATS.REPORT_COL_USAGE('SH','CUSTOMERS')

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
.....................................
1. COUNTRY_ID                        : EQ EQ_JOIN
2. CUST_CITY                         : EQ
3. CUST_CITY_ID                      : EQ_JOIN
4. CUST_ID                           : EQ_JOIN
5. CUST_STATE_PROVINCE               : EQ
6. CUST_STATE_PROVINCE_ID            : EQ_JOIN
7. CUST_TOTAL_ID                     : EQ_JOIN
8. (CUST_CITY, CUST_STATE_PROVINCE)  : FILTER
9. (CUST_CITY, CUST_STATE_PROVINCE,
    COUNTRY_ID)                      : FILTER
10. (CUST_STATE_PROVINCE, COUNTRY_ID)  : GROUP_BY
############################################################
```

Listing 7: Capturing the Results of Extended Statistics Recommendations

Now that REPORT_COL_USAGE has identified all potential extended statistics targets, invoking the CREATE_EXTENDED_STATS procedure for each table will tell Oracle 12*c* that they should be gathered during the next manual or automatic statistics gathering task, as shown in Listing 8.

```
SET LONG 7000
SET LONGCHUNKSIZE 7000
SET LINESIZE 500
SELECT DBMS_STATS.CREATE_EXTENDED_STATS('AP', 'VENDORS') FROM dual;
SELECT DBMS_STATS.CREATE_EXTENDED_STATS('SH', 'CUSTOMERS') FROM dual;

SQL> DBMS_STATS.CREATE_EXTENDED_STATS('AP','VENDORS')

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

EXTENSIONS FOR AP.VENDORS
.........................

1. (CITY, COUNTRY)                   : SYS_STU3UJN5IT#IIA5ASKY72Q8V6Z created
################################################################################



SQL> DBMS_STATS.CREATE_EXTENDED_STATS('SH','CUSTOMERS')

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

EXTENSIONS FOR SH.CUSTOMERS
...........................

1. (CUST_CITY, CUST_STATE_PROVINCE)   : SYS_STUWMBUN3F$#398R7BSOYVS86R created
2. (CUST_CITY, CUST_STATE_PROVINCE,
    COUNTRY_ID)                       : SYS_STUMZ$C3AIHLPBROI#SKA58H_N created
3. (CUST_STATE_PROVINCE, COUNTRY_ID)  : SYS_STU#S#WF25Z#QAHIHE#MOFFMM_ created
################################################################################
```

Listing 8: Establishing Relationships Between Columns in Selected Tables

As shown in Listing 9, Oracle 12c will automatically capture statistics for all identified statistics extensions that make the most sense to be gathered:

```
BEGIN
    DBMS_STATS.GATHER_TABLE_STATS('AP', 'VENDORS');
    DBMS_STATS.GATHER_TABLE_STATS('SH', 'CUSTOMERS');
END;
/
SQL> SQL> SQL>   2    3    4    5    6

PL/SQL procedure successfully completed.
```

Listing 9: Gather Statistics on Column Expressions

Finally, Listing 10 shows the original EXPLAIN PLAN before extended statistics were available for SH.CUSTOMERS, followed immediately by the actual execution plan once the extended statistics had been gathered.

```
EXPLAIN PLAN FOR
SELECT *
  FROM sh.customers
 WHERE cust_city = 'San Jose'
   AND cust_state_province = 'CA'
   AND country_id = 52790;
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY);


Plan hash value: 2008213504

-------------------------------------------------------------------------------
| Id  | Operation         | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |           |     1 |   208 |   423   (1)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| CUSTOMERS |     1 |   208 |   423   (1)| 00:00:01 |
-------------------------------------------------------------------------------

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

   1 - filter("CUST_CITY"='San Jose' AND "CUST_STATE_PROVINCE"='CA' AND
              "COUNTRY_ID"=52790)

. . .

SET AUTOTRACE TRACEONLY
SELECT *
  FROM sh.customers
 WHERE cust_city = 'San Jose'
   AND cust_state_province = 'CA'
   AND country_id = 52790;
SET AUTOTRACE OFF

Plan hash value: 2008213504

-------------------------------------------------------------------------------
| Id  | Operation         | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |           |    61 | 12688 |   423   (1)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| CUSTOMERS |    61 | 12688 |   423   (1)| 00:00:01 |
-------------------------------------------------------------------------------

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

   1 - filter("CUST_CITY"='San Jose' AND "CUST_STATE_PROVINCE"='CA' AND
              "COUNTRY_ID"=52790)
```

Listing 10: Impact of Extended Column Statistics: Pre- and Post-Gathering

continued on page 10

Now It Goes to 11: Automatic Dynamic Sampling

The final new Oracle 12*c*R1 optimizer-related feature I'll explore is a dramatic increase in the capabilities of dynamic sampling. In earlier releases, the optimizer only decided to invoke dynamic sampling when it detected that at least one table among the row set(s) being accessed lacked any statistics, in which case it would gather just enough statistics for the current session to construct a satisfactory execution plan. Starting in this release, however, it's possible to activate Automatic Dynamic Sampling (ADS) by setting the OPTIMIZER_DYNAMIC_SAMPLING initialization parameter to a value of 11. The optimizer will then decide whether dynamic sampling is advantageous for application workloads, and if so, what level of dynamic sampling should be applied.

Once activated, ADS decides if there are sufficient optimizer statistics available for generating a good execution plan; if not, the optimizer will then leverage dynamic sampling to gather sufficient statistics. In Oracle 12*c*R1, the definition of sufficient statistics now includes enough information to effectively perform joins between row sets, construct GROUP BY aggregations efficiently, perform serial executions and even overcome a table's stale statistics. Best of all, these statistics are persisted within the database so that other queries and even other sessions can leverage them, thus eliminating the need for the optimizer to continually re-gather the identical statistics.

The query in Listing 11 below shows a simple example of ADS's power. The two row sets, AP.RANDOMIZED_PARTED and AP.RANDOMIZED_SORTED, contain significantly different data even though their columns are identical.

```
EXPLAIN PLAN FOR
SELECT rs.key_date, rs.key_desc, rp.KEY_STS
  FROM
     ap.randomized_sorted RS
    ,ap.randomized_parted RP
 WHERE RS.key_id = RP.key_id
   AND RS.key_desc LIKE 'aaa%'
   AND RS.key_sts > 40
   AND RP.key_date BETWEEN TO_DATE('2013-12-31','YYYY-MM-DD')
                       AND TO_DATE('2014-01-31','YYYY-MM-DD');
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY(NULL,NULL,'TYPICAL,+NOTE'));
```

Listing 11: Impact of Automatic Dynamic Sampling

Without ADS — in other words, when OPTIMIZER_DYNAMIC_SAMPLING is left at its 12*c*R1 default value of two — the optimizer is not able to accurately estimate the cardinality of these two row sets; as shown in Figure 7, it seriously overestimates the size of the final result set at 1,819 rows. However, when ADS is activated, the optimizer determines that dynamic sampling is beneficial to its estimation process and is thus able to determine a much more accurate estimate of the final size of the query's result set, as Figure 8 shows. Interestingly, the actual size of the result set is zero, so ADS was obviously instrumental in helping the optimizer to derive a much more accurate cardinality for the final result.

```
Plan hash value: 4191593406

-------------------------------------------------------------------------------------------------------
| Id  | Operation               | Name        | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
-------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT        |             |     2 |    84 |   941   (1)| 00:00:01 |       |       |
|   1 |  SORT GROUP BY          |             |     2 |    84 |   941   (1)| 00:00:01 |       |       |
|   2 |   NESTED LOOPS          |             |   102 |  4284 |   941   (1)| 00:00:01 |       |       |
|   3 |    VIEW                 | VW_GBC_9    |   102 |  3876 |   941   (1)| 00:00:01 |       |       |
|   4 |     HASH GROUP BY       |             |   102 |  3060 |   941   (1)| 00:00:01 |       |       |
|*  5 |      HASH JOIN          |             |  1992 | 59760 |   941   (1)| 00:00:01 |       |       |
|*  6 |       TABLE ACCESS FULL | CUSTOMERS   |   184 |  2392 |   423   (1)| 00:00:01 |       |       |
|   7 |       PARTITION RANGE ALL|            |  918K |   14M |   516   (2)| 00:00:01 |     1 |    28 |
|   8 |        TABLE ACCESS FULL| SALES       |  918K |   14M |   516   (2)| 00:00:01 |     1 |    28 |
|*  9 |    INDEX UNIQUE SCAN    | PRODUCTS_PK |     1 |     4 |     0   (0)| 00:00:01 |       |       |
-------------------------------------------------------------------------------------------------------

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

   5 - access("S"."CUST_ID"="C"."CUST_ID")
   6 - filter("C"."CUST_LAST_NAME"='Farmer' OR "C"."CUST_LAST_NAME"='York')
   9 - access("ITEM_1"="P"."PROD_ID")

Note
-----
   - statistics feedback used for this statementFigure 2: SMB Results After Loading First Plan Baseline
```

Figure 1: Original EXPLAIN PLAN output

```
                                                Current SQL Plan Baselines
                                               (From DBA_SQL_PLAN_BASELINES)

         SQL      Plan                                                 CBO Ena-           Auto Created     Last
Creator  Handle   Name     SQL Text                 Origin           Cost bled Acpt Fixd Purg On          Executed
-------- -------- -------- ------------------------ ------------ -------- ---- ---- ---- ---- ----------- -----------
SYS      4231c63b 57b5d5bb SELECT /* AUTOSPM_1 */    AUTO-CAPTURE      941 YES  YES  NO   YES  2014-06-21  2014-06-21
                             C.cust_last_name                                                  20:37:00    20:37:00
                            ,SUM(S.amount_sold)
                            ,S
```

Figure 2: SMB Contents After Running First Query Iteration

```
Plan hash value: 2671087310

-----------------------------------------------------------------------------------------------------------------
| Id  | Operation                                | Name         | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
-----------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                         |              |     2 |    84 |   562   (2)| 00:00:01 |       |       |
|   1 |  SORT GROUP BY                           |              |     2 |    84 |   562   (2)| 00:00:01 |       |       |
|   2 |   NESTED LOOPS                           |              |   102 |  4284 |   562   (2)| 00:00:01 |       |       |
|   3 |    VIEW                                  | VW_GBC_9     |   102 |  3876 |   562   (2)| 00:00:01 |       |       |
|   4 |     HASH GROUP BY                        |              |   102 |  3060 |   562   (2)| 00:00:01 |       |       |
|*  5 |      HASH JOIN                           |              | 23907 |   700K|   562   (2)| 00:00:01 |       |       |
|   6 |       INLIST ITERATOR                    |              |       |       |            |          |       |       |
|   7 |        TABLE ACCESS BY INDEX ROWID BATCHED| CUSTOMERS    |   184 |  2392 |    43   (0)| 00:00:01 |       |       |
|*  8 |         INDEX RANGE SCAN                 | CUST_LN_DESC |   222 |       |     3   (0)| 00:00:01 |       |       |
|   9 |       PARTITION RANGE ALL                |              |   918K|    14M|   516   (2)| 00:00:01 |     1 |    28 |
|  10 |        TABLE ACCESS FULL                 | SALES        |   918K|    14M|   516   (2)| 00:00:01 |     1 |    28 |
|* 11 |    INDEX UNIQUE SCAN                     | PRODUCTS_PK  |     1 |     4 |     0   (0)| 00:00:01 |       |       |
-----------------------------------------------------------------------------------------------------------------

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

   5 - access("S"."CUST_ID"="C"."CUST_ID")
   8 - access(SYS_OP_DESCEND("CUST_LAST_NAME")=HEXTORAW('B99E8D929A8DFF') OR
              SYS_OP_DESCEND("CUST_LAST_NAME")=HEXTORAW('A6908D94FF'))
       filter(SYS_OP_UNDESCEND(SYS_OP_DESCEND("CUST_LAST_NAME"))='Farmer' OR
              SYS_OP_UNDESCEND(SYS_OP_DESCEND("CUST_LAST_NAME"))='York')
  11 - access("ITEM_1"="P"."PROD_ID")
```

Figure 3: New EXPLAIN PLAN output

```
                                                Current SQL Plan Baselines
                                               (From DBA_SQL_PLAN_BASELINES)

         SQL      Plan                                                 CBO Ena-           Auto Created     Last
Creator  Handle   Name     SQL Text                 Origin           Cost bled Acpt Fixd Purg On          Executed
-------- -------- -------- ------------------------ ------------ -------- ---- ---- ---- ---- ----------- -----------
SYS      4231c63b 57b5d5bb SELECT /* AUTOSPM_1 */    AUTO-CAPTURE      941 YES  YES  NO   YES  2014-06-21  2014-06-21
                             C.cust_last_name                                                  20:37:00    20:37:00
                            ,SUM(S.amount_sold)
                            ,S

SYS      4231c63b ccaf39cd SELECT /* AUTOSPM_1 */    AUTO-CAPTURE      562 YES  NO   NO   YES  2014-06-21
                             C.cust_last_name                                                  20:39:45
                            ,SUM(S.amount_sold)
                            ,S
```

Figure 4: SMB Results After Loading Second Plan Baseline

continued on page 12

```
                                                    Current SQL Plan Baselines
                                                  (From DBA_SQL_PLAN_BASELINES)

         SQL      Plan                                                  CBO Ena-           Auto Created     Last
Creator  Handle   Name     SQL Text                 Origin              Cost bled Acpt Fixd Purg On           Executed
-------- -------- -------- ------------------------ ------------ -------- ---- ---- ---- ---- ----------- -----------
SYS      ded652b9 57b5d5bb SELECT /* AUTOSPM_1 */    AUTO-CAPTURE      941 YES  YES  NO   YES  2014-06-22  2014-06-22
                             C.cust_last_name                                                  16:44:01    16:44:01
                            ,SUM(S.amount_sold)
                            ,S

SYS      ded652b9 ccaf39cd SELECT /* AUTOSPM_1 */    AUTO-CAPTURE      562 YES  YES  NO   YES  2014-06-22
                             C.cust_last_name                                                  16:45:07
                            ,SUM(S.amount_sold)
                            ,S
```

Figure 5: Automatic SQL Plan Evolution After Nightly Maintenance Window

```
-----
-- Show run of selection manual SPM Advisor report
-----
SET SERVEROUTPUT ON
SET LONG 40000
SET LONGCHUNKSIZE 32767
SET LINESIZE 150
SET PAGESIZE 20000
VARIABLE rptout CLOB;
BEGIN
:rptout := DBMS_SPM.REPORT_EVOLVE_TASK  (
     task_name => 'SPME_1'
    ,type => 'TEXT'
    ,level => 'TYPICAL'
    ,section => 'ALL'
    ,object_id => NULL
    ,execution_name => 'SPME_1');
END;
/
PRINT :rptout;

>>> Results:

GENERAL INFORMATION SECTION
-------------------------------------------------------------------------------------------

Task Information:
---------------------------------------------
Task Name            : SPME_1
Task Owner           : SYS
Description          : Build task to evolve +all+ non-accepted plans
Execution Name       : SPME_1
Execution Type       : SPM EVOLVE
Scope                : COMPREHENSIVE
Status               : COMPLETED
Started              : 06/25/2014 22:39:20
Finished             : 06/25/2014 22:39:23
Last Updated         : 06/25/2014 22:39:23
Global Time Limit    : 2147483646
Per-Plan Time Limit  : UNUSED
Number of Errors     : 0
-------------------------------------------------------------------------------------------

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

```
DETAILS SECTION
-------------------------------------------------------------------------------------------
 Object ID          : 2
 Test Plan Name     : SQL_PLAN_9xwy49zgdcnptccaf39cd
 Base Plan Name     : SQL_PLAN_9xwy49zgdcnpt57b5d5bb
 SQL Handle         : SQL_9ef3c44fded652b9
 Parsing Schema     : SYS
 Test Plan Creator  : SYS
 SQL Text           : SELECT /* AUTOSPM_1 */ C.cust_last_name
                    ,SUM(S.amount_sold) ,SUM(S.quantity_sold) FROM sh.sales S
                    ,sh.customers C ,sh.products P WHERE S.cust_id = C.cust_id
                    AND S.prod_id = P.prod_id AND C.cust_last_name IN
                    ('Farmer','York') GROUP BY C.cust_last_name ORDER BY
                    C.cust_last_name

Execution Statistics:
-----------------------------
                    Base Plan                    Test Plan
                    ---------------------------  ---------------------------
 Elapsed Time (s):  .031116                      .029343
 CPU Time (s):      .021097                      .02013
 Buffer Gets:       524                          274
 Optimizer Cost:    942                          562
 Disk Reads:        0                            0
 Direct Writes:     0                            0
 Rows Processed:    0                            0
 Executions:        6                            6

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

Findings (1):
-----------------------------
 1. The plan was verified in 2.40000 seconds. It passed the benefit criterion
    because its verified performance was 1.90157 times better than that of the
    baseline plan.

Recommendation:
-----------------------------
 Consider accepting the plan. Execute
 dbms_spm.accept_sql_plan_baseline(task_name => 'SPME_1', object_id => 2,
 task_owner => 'SYS');
```

Figure 6: Manual SQL Plan Evolution Results Report

```
Plan hash value: 2019123141

-----------------------------------------------------------------------------------------------------------------------------------
| Id  | Operation               | Name                | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |    TQ  |IN-OUT| PQ Distrib |
-----------------------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT        |                     |  1819 | 89131 |   191   (1)| 00:00:01 |       |       |        |      |            |
|   1 |  PX COORDINATOR         |                     |       |       |            |          |       |       |        |      |            |
|   2 |   PX SEND QC (RANDOM)   | :TQ10001            |  1819 | 89131 |   191   (1)| 00:00:01 |       |       |  Q1,01 | P->S | QC (RAND)  |
|*  3 |    HASH JOIN            |                     |  1819 | 89131 |   191   (1)| 00:00:01 |       |       |  Q1,01 | PCWP |            |
|   4 |     JOIN FILTER CREATE  | :BF0000             |  1819 | 60027 |   171   (1)| 00:00:01 |       |       |  Q1,01 | PCWP |            |
|   5 |      PX RECEIVE         |                     |  1819 | 60027 |   171   (1)| 00:00:01 |       |       |  Q1,01 | PCWP |            |
|   6 |       PX SEND BROADCAST | :TQ10000            |  1819 | 60027 |   171   (1)| 00:00:01 |       |       |  Q1,00 | S->P | BROADCAST  |
|   7 |        PX SELECTOR      |                     |       |       |            |          |       |       |  Q1,00 | SCWC |            |
|*  8 |         TABLE ACCESS FULL| RANDOMIZED_SORTED  |  1819 | 60027 |   171   (1)| 00:00:01 |       |       |  Q1,00 | SCWP |            |
|   9 |     JOIN FILTER USE     | :BF0000             | 13143 |  205K |    20   (0)| 00:00:01 |       |       |  Q1,01 | PCWP |            |
|  10 |      PX BLOCK ITERATOR  |                     | 13143 |  205K |    20   (0)| 00:00:01 |     4 |     4 |  Q1,01 | PCWC |            |
|* 11 |       TABLE ACCESS FULL | RANDOMIZED_PARTED   | 13143 |  205K |    20   (0)| 00:00:01 |     4 |     4 |  Q1,01 | PCWP |            |
-----------------------------------------------------------------------------------------------------------------------------------

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

   3 - access("RS"."KEY_ID"="RP"."KEY_ID")
   8 - filter("RS"."KEY_STS">40 AND "RS"."KEY_DESC" LIKE 'aaa%')
  11 - filter("RP"."KEY_DATE"<=TO_DATE(' 2014-01-31 00:00:00', 'syyyy-mm-dd hh24:mi:ss') AND "RP"."KEY_DATE">=TO_DATE(' 2013-12-31
              00:00:00', 'syyyy-mm-dd hh24:mi:ss') AND SYS_OP_BLOOM_FILTER(:BF0000,"RP"."KEY_ID"))
```

Figure 7: Impact of Dynamic Sampling Set to Default Value (OPTIMIZER_DYNAMIC_SAMPLING = 2)

continued on page 14

```
Plan hash value: 2019123141

-----------------------------------------------------------------------------------------------------------------------------------------
| Id  | Operation                | Name                | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |    TQ  |IN-OUT| PQ Distrib |
-----------------------------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT         |                     |     1 |    49 |   191   (1)| 00:00:01 |       |       |        |      |            |
|   1 |  PX COORDINATOR          |                     |       |       |            |          |       |       |        |      |            |
|   2 |   PX SEND QC (RANDOM)    | :TQ10001            |     1 |    49 |   191   (1)| 00:00:01 |       |       |  Q1,01 | P->S | QC (RAND)  |
|*  3 |    HASH JOIN             |                     |     1 |    49 |   191   (1)| 00:00:01 |       |       |  Q1,01 | PCWP |            |
|   4 |     JOIN FILTER CREATE   | :BF0000             |  1835 | 60555 |   171   (1)| 00:00:01 |       |       |  Q1,01 | PCWP |            |
|   5 |      PX RECEIVE          |                     |  1835 | 60555 |   171   (1)| 00:00:01 |       |       |  Q1,01 | PCWP |            |
|   6 |       PX SEND BROADCAST  | :TQ10000            |  1835 | 60555 |   171   (1)| 00:00:01 |       |       |  Q1,00 | S->P | BROADCAST  |
|   7 |        PX SELECTOR       |                     |       |       |            |          |       |       |  Q1,00 | SCWC |            |
|*  8 |         TABLE ACCESS FULL| RANDOMIZED_SORTED   |  1835 | 60555 |   171   (1)| 00:00:01 |       |       |  Q1,00 | SCWP |            |
|   9 |     JOIN FILTER USE      | :BF0000             | 13143 |  205K |    20   (0)| 00:00:01 |       |       |  Q1,01 | PCWP |            |
|  10 |      PX BLOCK ITERATOR   |                     | 13143 |  205K |    20   (0)| 00:00:01 |     4 |     4 |  Q1,01 | PCWC |            |
|* 11 |       TABLE ACCESS FULL  | RANDOMIZED_PARTED   | 13143 |  205K |    20   (0)| 00:00:01 |     4 |     4 |  Q1,01 | PCWP |            |
-----------------------------------------------------------------------------------------------------------------------------------------

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

   3 - access("RS"."KEY_ID"="RP"."KEY_ID")
   8 - filter("RS"."KEY_DESC" LIKE 'aaa%' AND "RS"."KEY_STS">40)
  11 - filter("RP"."KEY_DATE"<=TO_DATE(' 2014-01-31 00:00:00', 'syyyy-mm-dd hh24:mi:ss') AND "RP"."KEY_DATE">=TO_DATE(' 2013-12-31
              00:00:00', 'syyyy-mm-dd hh24:mi:ss') AND SYS_OP_BLOOM_FILTER(:BF0000,"RP"."KEY_ID"))

Note
-----
   - dynamic statistics used: dynamic sampling (level=AUTO)
```

Figure 8: Impact of Dynamic Sampling Set to Automatic (OPTIMIZER_DYNAMIC_SAMPLING = 11)

What's Next for 12cR1?

In my next article, I'll take a look at the highlights of the newest release of Oracle 12*c*R1, 12.1.0.2, especially the long-awaited arrival of the In-Memory Column Store.

About the Author

Jim Czuprynski has accumulated over 30 years of experience during his career in information technology. He has served diverse roles at several Fortune 1000 companies in those three decades — mainframe programmer, applications developer, business analyst, and project manager — before becoming an Oracle database administrator in 2001. He is an Oracle ACE Director and he currently holds OCP certification for Oracle 9*i*, 10*g* and 11*g*. Czuprynski teaches the core Oracle University database administration courses on behalf of Oracle and its education partners throughout the United States and Canada, instructing several hundred Oracle DBAs since 2005. He was selected as Oracle Education Partner Instructor of the Year in 2009. He continues to write a steady stream of articles that focus on the myriad facets of Oracle Database administration, with nearly 100 articles to his credit since 2003 at *databasejournal.com*. Jim's monthly blog, *Generally … It Depends* (http://jimczuprynski.wordpress.com), contains his regular observations on all things Oracle. Czuprynski is also a regular public speaker on Oracle Database technology features and has presented topics at Oracle OpenWorld, Hotsos, IOUG's COLLABORATE conferences and OUG Norway.