Adaptability

SQL That (Almost) Tunes Itself: Oracle Database 12cR1 Optimizer Enhancements
by Jim Czuprynski

Are Oracle Linux and Red Hat Linux the Same?
by Gary Gordhamer

ODSM: Oracle Directory Services Manager for Names Resolution
by Ray Smith
SQL That (Almost) Tunes Itself: Oracle Database 12cR1 Optimizer Enhancements

By Jim Czuprynski
Arup Nanda, Editor

In my dozen-plus years as an Oracle DBA and as an Oracle University instructor for several hundred of my fellow DBAs since 2005, I’ve noticed that my colleagues and I always agree on one stark observation regardless of our backgrounds and shared experiences: Even though the Oracle query optimizer continues to improve after every new Oracle Database release, its effectiveness is still quite dependent on the presence of accurate and timely optimizer statistics.

Each database release has brought welcome, improved effectiveness to the optimizer. Here are the highlights of each release:

- Oracle Database 9i Release 2 (9/R2) introduced bind variable peeking so that a more efficient execution plan would be created as a child cursor when a different range of bind variable values indicated an alternate plan would be more appropriate.

- 9/R2 also introduced the optimizer’s ability to employ adaptive sampling so that if crucial statistics were missing for a particular database object chosen as a row source to complete a query, it could gather statistics immediately based on a sufficiently representative sample of the row source’s data to avoid unnecessarily guessing at the best access path and join method for that row source. This tended to alleviate unnecessary table scans when an index range scan or even a fast full index scan would be more efficient.

- The next big improvement to more effective execution of complex SQL statements, adaptive cursor sharing (ACS), was introduced in Oracle Database 11g Release 1 (11gR1). ACS significantly extends bind variable peeking by capturing and retaining the specific spheres of selectivity based on the ranges of bind variable values for each individual child cursor. ACS therefore tends to reduce the number of child cursors needed, especially in a data warehousing environment. (For a deeper discussion on how ACS works, including a much closer look at the selectivity spheres it generates and retains, see my article on the topic: http://www.databasejournal.com/features/oracle/article.php/3774606/Oracle-Database-11g-Adaptive-Cursor-Sharing.htm).

On the other hand, an Oracle DBA could still do very little if one or more subcomponents of the execution plan for an especially complex query was still suboptimal at execution time — for example, because one or more of its row sources had significantly changed its content since statistics for those row source(s) were most recently last gathered, or if non-equijoin methods were required to answer the query’s question.

Adaptive Query Optimization

The good news here is that this all changes in Oracle Database 12c Release 1 (12cR1) because the Oracle optimizer is significantly upgraded to handle these exact circumstances with the introduction of the adaptive query optimization (AQO) feature set. AQO offers distinct sets of features for two radically different challenges to the optimizer’s effectiveness.

Adaptive Plans

Adaptive plans are aimed at statements that can benefit from delaying the optimizer’s decisions on the best final execution plan until the statement is executed. For example, an adaptive plan makes it possible for the optimizer to choose a different join method between two row sources until the query is actually running and real-time execution statistics prove out that a different subplan — perhaps, say, a hash join instead of a nested-loop join — is the better choice for this statement’s execution. While I’m going to focus in this article primarily on the join methods that adaptive plans can provide, the 12cR1 optimizer can also implement different parallel execution methods when it recognizes during statement execution that there is a more effective way to distribute parallel processing requests to multiple parallel servers.

Adaptive Statistics

On the other hand, adaptive statistics are aimed at situations that adaptive plans can’t effectively intercept, such as deciding to join result sets in a different order to obtain a better query response time. Unfortunately, capturing additional optimizer statistics in these scenarios doesn’t generally help during the initial execution of the statement, but the statistics gathered during that execution can be of benefit during the next time the statement is executed. Adaptive statistics encompasses three key sets of features — Dynamic Statistics, Automatic Re-Optimization and SQL Plan Directives — that can yield equally impressive performance improvements for subsequent statement execution(s).

Let’s get started with a few examples of how adaptive plans can automatically identify candidate SQL statements for immediately better performance.

Adaptive Plans

One of the really great things about adaptive plans is that it’s possible to see the potential impact of the alternate subplans within an EXPLAIN PLAN before the query is executed. The query in Figure 1 demonstrates a simple example of an adaptive plan. It searches for all customers in table SH that live in either Chicago or Los Angeles and who have at least one invoice in table AP.INVOICES. I’ll then leverage the new ADAPTIVE directive for the FORMAT argument of DBMS_XPLAN.DISPLAY to generate its corresponding EXPLAIN PLAN output captured in Listing 1.1.

continued on page 10
SELECT /*+ MONITOR dyn_1 */ 
  C.cust_last_name 
FROM ap.invoices I, ap.invoice_items LI, sh.customers C 
WHERE I.invoice_id = LI.invoice_id 
  AND I.customer_id = C.cust_id 
  AND LI.extended_amt > 22500.00 
  AND C.cust_last_name LIKE 'Aa%';

Figure 2: Adaptive Plans, Example 2

In this case, the 12cR1 optimizer was able to construct the adaptive plan shown in Listing 2.1 that decides a hash join between the SH.CUSTOMERS and AP.INVOICE_ITEMS tables is actually more effective than using a nested loops join method between a completely different set of index row sources.

The optimizer can also take advantage of adaptive plans when it detects during execution that a particular set of values falls into a domain that would be best served by an alternative subplan rather than the original, default subplan. Once the optimizer has picked the appropriate alternate subplan, there’s no need to continue buffering the row source results, so the statistics collector automatically stops any further row collection and just passes the row source results through to the rest of the query’s execution. When those same values or another set of similar values fall into the same range, the optimizer will be able to immediately choose the proper alternate plan without any additional row source buffering. Adaptive plans are therefore most valuable for statements operating against row sources that tend to be more volatile and whose statistics may even become stale more frequently than expected — perhaps even before the next scheduled capture of more valid statistics.

---

**Listing 1.1. Adaptive Plans, Example 1: Results of Pre-Execution EXPLAIN PLAN**

When this query is actually executed for the first time, a dynamic statistics collector buffers the row set that results from the intersection of these two tables. The 12cR1 optimizer then determines that for this execution of the statement, it’s more efficient to use a nested loop join method instead of a hash join; DBMS_XPLAN reflects this by placing leading dashes before each operation that’s bypassed.

In a slightly more complex scenario, the query in Figure 2 is searching for all customers whose last name starts with the letters Aa and whose total invoice line amount exceeds $22,500.00:

```sql
SELECT /*+ MONITOR DYN_1 */ 
  C.cust_last_name 
FROM ap.invoices I, ap.invoice_items LI, sh.customers C 
WHERE I.invoice_id = LI.invoice_id 
  AND I.customer_id = C.cust_id 
  AND LI.extended_amt > 22500.00 
  AND C.cust_last_name LIKE 'Aa%';
```

---

**Listing 2.1. Adaptive Plans, Example 2: Results of Pre-Execution EXPLAIN PLAN**

**Identifying Candidates for Adaptive Plans and ARO**

Oracle 12cR1 makes it easy to locate candidate statements for either adaptive plans and/or automatic re-optimization via the V$SQL dynamic view:

- Column IS_RESOLVED_ADAPTIVE_PLAN will be set to a value of (Y)es when a statement offers at least one opportunity for an adaptive plan.
- Likewise, column IS_REOPTIMIZABLE will contain a value of (Y)es when a statement can potentially utilize automatic re-optimization (ARO).

I’ve shown a sample query against V$SQL in Figure 3, and an example of the output for the prior adaptive plan queries in Listing 3.1.
Oracle 12cR1 augments cardinality feedback with the introduction of statistics feedback. Statistics feedback is at the heart of how ARO works because it allows the optimizer to compare the estimated cardinality for statement operations against the actual cardinality of those operations after they have been executed, and then take appropriate action during the next execution of the same statement. When optimizer statistics are simply missing or stale for row sources, when the statement employs complex non-equijoins, or when selection criteria resolves into multiple conjunctive or disjunctive filter predicates, statistics feedback provides a valuable alternative to dynamic plans.

Briefly, here’s how statistics feedback and ARO work together to construct better execution plans. For each step of the execution plan during the statement’s initial execution, ARO buffers the actual cardinality for each row set retrieved as well as statistics on different joins performed. These actual cardinality and join statistics are then compared with their estimated counterparts, and if the optimizer detects a significant variance between them, it then determines the best method to improve the statement’s execution of the plan during its next invocation — perhaps by selecting a different join order, or gathering statistics for a specific set of columns via dynamic sampling — as well as the frequency at which these required statistics should be collected. And if a SQL statement uses bind variables, these additional statistics can also be leveraged by adaptive cursor sharing.

ARO: A Simple Example. I’ve illustrated this scenario with the query in Figure 4. This query retrieves data from several row sources in the AP schema into an in-line view, and then combines that row source with other data in the SH schema:

```
SELECT /*+ MONITOR GATHER_PLAN_STATISTICS ARO_1 */
  INV.cust_id,
  .c.cust_last_name cust_name
  ,SUM(INV.xtd_amt) xtd_amt
  ,SUM(S.quantity_sold) qty_sold
  ,SUM(S.amt_sold) amt_sold
FROM
  sh.customers C
  ,sh.products P
  ,sh.sales S
  ,
  SELECT
    I.customer_id cust_id
    ,LI.product_id prod_id
    ,SUM(LI.extended_amt) xtd_amt
  FROM
    ap.invoices I
  ,ap.invoice_items LI
WHERE I.invoice_id = LI.invoice_id
  AND I.taxable_amt > 1000.00
  AND LI.extended_amt < 100.00
GROUP BY
  I.customer_id
  ,LI.product_id
) INV
WHERE INV.cust_id = C.cust_id
  AND S.cust_id = C.cust_id
  AND S.prod_id = P.prod_id
GROUP BY
  INV.cust_id
  ,C.cust_last_name
ORDER BY
  INV.cust_id
  ,C.cust_last_name;
```

Figure 4: Automatic Re-Optimization

Oracle 12cR1's new adaptive plans are quite powerful medicine for statements that perform poorly, but they still do have limitations, because while adaptive plans can modify individual subplans within an execution plan for better performance, they still cannot implement global changes to that plan. For example, what about a query that performs poorly because the order in which row sets are joined results in a less than optimal execution plan? The good news is that, even in this case, the 12cR1 optimizer may still be able to apply automatic re-optimization (ARO) to generate a better plan immediately after statement execution from which all future executions of the same statement can benefit. And each time a similar version of the same query is run against different data domains, ARO will capture additional sets of statistics specific to those domains, thus accumulating more and more accurate actual execution statistics over time for improved future plan selection.

As noted previously, the introduction of adaptive cursor sharing (ACS) in Oracle 11gR1 made it possible for a SQL statement's cursor to become first bind-aware and then bind-sensitive during later executions of the same statement when identical or nearly identical sets of bind variable values were used. Oracle 11gR2 enhanced ACS and then reclassified them as cardinality feedback (CF), making it possible to review the impact of — but not yet act upon — the estimated statistics for row sources at an individual statement level during the initial execution of that statement.
Unfortunately, the selection criteria against the three tables in the AP schema use non-equality conditions. To make matters worse, the TAXABLE_IND column APINVOICE_ITEMS table is skewed significantly toward one value; of 47,500 invoice line items, 40,000 (84.2 percent) are taxable (i.e., a value of Y) while 7,500 are (N)ot. The counts of estimated rows (E-Rows) and actual rows (A-Rows) are 47,500 invoice line items, 40,000 (84.2 percent) are taxable (i.e., a value of Y) while 7,500 are (N)ot.

Listing 4.1. Automatic Re-Optimization: Results of Initial Execution

As the EXPLAIN PLAN output for this statement’s initial execution shows in Listing 4.1, under these conditions the optimizer is unable to estimate cardinality correctly. The counts of estimated rows (E-Rows) and actual rows (A-Rows) columns diverge significantly during this statement’s first execution. But because the GATHER_PLAN_STATISTICS hint has been added for this statement, the 12cR1 optimizer recognizes that while it can’t improve this statement’s initial execution, it can certainly capture the actual cardinality to assist in more accurate subsequent execution(s). Indeed, it’s the next execution of the same statement that shows the real promise and power of ARO. The resulting EXPLAIN PLAN in Listing 4.2 demonstrates that the statement is now able to leverage the new execution plan and corresponding child cursor that’s already been generated using the actual statistics from the statement’s initial execution. Finally, it’s important to note that ARO and statistics feedback tends to be most useful for statements executed against row sources whose data volumes and compositions over time remain stable, so they are an excellent counterbalance to dynamic plans, which, as previously noted, are more likely to be useful for extremely volatile row sources (e.g., a work table that’s constantly “torn down” and refreshed from scratch every few hours).

Listing 4.2. Automatic Re-Optimization: Results of Second Execution

SQL Plan Directives (SPD)

I’ve demonstrated the effectiveness of ARO to capture actual statistics for future executions of statements, but it’s important to note that only individual table cardinalities are examined, and that this information is stored only in the corresponding cursor for the statement, so if the cursor were to be aged out of the shared pool, this valuable performance metadata would be lost forever. To alleviate this, Oracle 12cR1 offers the ability to retain these valuable statistics in a new object called an SQL Plan Directive (SPD). A single SPD isn’t linked directly to any individual SQL statement or execution plan; rather, as its name suggests, it contains directives to obtain appropriate optimizer statistics so the optimizer can utilize them when a particular execution plan operation might benefit from gathering additional statistics before the optimizer completes construction of the final execution plan.

SPDs are only created when the 12cR1 optimizer recognizes they are needed. They are maintained automatically and, best of all, they are automatically flushed to disk every 15 minutes and are retained in the SQL Management Base (SMB) in the SYSAUX tablespace for later re-use. And much like SQL Plan Baselines that are also stored within the SMB, if a SPD hasn’t been used within one year of its initial creation, Oracle 12cR1 will automatically purge the SPD from the SMB. It’s also possible to flush SPDs manually into the SMB using the new Oracle 12cR1 DBMS_SPD package. Once they’ve been flushed to the SMB, new data dictionary views describe the metadata about all available SPDs.
To demonstrate how SPDs can be utilized, I’ll first flush any existing SPDs from the shared pool and delete any existing SPDs to provide a “clean slate.” I’ll then execute the simple query in Figure 6 three times consecutively, only changing the query’s comments to force a hard parse so that it’s simpler to see its corresponding SPD evolve. During each iteration of this demonstration, I’ll also flush all SPDs to the SMB via procedure DBMS_SPD.FLUSH_SQL_PLAN_DIRECTIVE so it will be easier to see their progression via the query in Figure 8.

```
-- Flush shared pool
ALTER SYSTEM FLUSH SHARED_POOL;

-- Purge existing SPDs
DECLARE
    CURSOR cur_spd_ids IS
        SELECT DISTINCT directive_id
        FROM dba_sql_plan_directive_objects
        WHERE owner IN ('AP', 'HR', 'OE', 'SH')
        AND object_type = 'TABLE';
BEGIN
    DBMS_SPD.FLUSH_SQL_PLAN_DIRECTIVE;
    LOOP
        DBMS_SPD.DROP_SQL_PLAN_DIRECTIVE(directive_id => spds.directive_id);
    END LOOP;
END;
/

-- Increase dynamic sampling rate to facilitate SPD creation
ALTER SESSION SET
    OPTIMIZER_DYNAMIC_SAMPLING = 4
/
```

Figure 5: Preparing to Test SQL Plan Directives

```
SELECT /*+ MONITOR GATHER_PLANSTATISTICS SP0_ZA */
    C.cust_city, C.cust_state_province
, SUM(S.quantity_sold) qty_sold
, SUM(S.amount_sold) amt_sold
FROM
    sh.customers C
, sh.products P
WHERE S.cust_id = C.cust_id
    AND S.prod_id = P.prod_id
    AND S.quantity_sold < 3
    AND S.amount_sold > 1599.99
GROUP BY C.cust_city, C.cust_state_province
HAVING SUM(S.quantity_sold) > 30
ORDER BY C.cust_city, C.cust_state_province;
```

Figure 6: Illustrating SQL Plan Directives: Example Query

```
COL spd_id FORMAT A08 HEADING “Abbrev|SPD ID”
COL used_dtm FORMAT A19 HEADING “Last|Used On”
COL owner FORMAT A08 HEADING “Owner”
COL object_name FORMAT A15 HEADING “Object Name”
COL subobject_name FORMAT A20 HEADING “SubObject Name”
COL object_type FORMAT A12 HEADING “Object Type”
COL state FORMAT A15 HEADING “State”
COL reason FORMAT A40 HEADING “Reason”
COL create_dtm FORMAT A19 HEADING “Created On”
COL update_dtm FORMAT A19 HEADING “Last Updated On”
TTITLE “Available SQL Plan Directives for Selected Schemas (from DBA_SQL_PLAN_DIR* Views)”
SELECT
    SUBSTR(TO_CHAR(SPD.directive_id),8,8) spd_id
, TO_CHAR(SPD.last_used,’yyyy-mm-dd hh24:mi:ss’) used_dtm
, SPD.owner
, SPD.object_name
FROM
    dba_sql_plan_directives SPD
, dba_sql_plan_dir_objects SPO
WHERE SPD.directive_id = SPO.directive_id
    AND SPO.owner IN ('AP', 'HR', 'OE', 'SH', 'SYSTEM')
ORDER BY 1, 2, 3, 4, 5 DESC;
```

Figure 7: Viewing SQL Plan Directive Metadata

The query’s EXPLAIN PLAN and the resulting SPDs after its first execution are shown in Listing 6.1. It’s easy to see that there is a massive difference between estimated and actual cardinality for several operations, and as a result an initial SPD has been built. However, note that the SPD’s state is initially recorded as NEW, which reflects that the need for better statistics has been recorded but none have been gathered yet.

```
Plan hash value: 92762162

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Starts</th>
<th>E-Rows</th>
<th>A-Rows</th>
<th>A-Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td>26</td>
<td>26</td>
<td>00:00:02</td>
</tr>
<tr>
<td>358</td>
<td>FILTER</td>
<td></td>
<td>1</td>
<td>26</td>
<td>26</td>
<td>00:00:02</td>
</tr>
<tr>
<td>1769</td>
<td>SORT GROUP BY</td>
<td></td>
<td>1</td>
<td>389</td>
<td>389</td>
<td>00:00:28</td>
</tr>
<tr>
<td>1769</td>
<td>MATCH?v=multiset1 (0)</td>
<td></td>
<td>1</td>
<td>3459</td>
<td>3459</td>
<td>00:00:15</td>
</tr>
<tr>
<td>358</td>
<td>NESTED LOOP</td>
<td></td>
<td>1</td>
<td>325</td>
<td>325</td>
<td>00:00:13</td>
</tr>
<tr>
<td>1769</td>
<td>NESTED LOOPS</td>
<td></td>
<td>1</td>
<td>325</td>
<td>325</td>
<td>00:00:12</td>
</tr>
<tr>
<td>358</td>
<td>PARTITION RANGE ALL</td>
<td></td>
<td>1</td>
<td>3459</td>
<td>3459</td>
<td>00:00:11</td>
</tr>
<tr>
<td>1769</td>
<td>INDEX UNIQUE SCAN</td>
<td></td>
<td>28</td>
<td>325</td>
<td>325</td>
<td>00:00:21</td>
</tr>
<tr>
<td>358</td>
<td>NESTED LOOPS</td>
<td></td>
<td>1</td>
<td>3459</td>
<td>3459</td>
<td>00:00:01</td>
</tr>
<tr>
<td>1769</td>
<td>NESTED LOOPS</td>
<td></td>
<td>1</td>
<td>3459</td>
<td>3459</td>
<td>00:00:02</td>
</tr>
</tbody>
</table>
```

Listing 6.1. SQL Plan Directives: First Iteration

Listing 6.2 shows the query’s EXPLAIN PLAN and the resulting SPDs after its second execution. Note that the presence of an SPD is now recognized and that the SPD’s state has been upgraded to HAS_STATS, which reflects that the optimizer has recognized that additional statistics are required to accurately build an effective execution plan.

Finally, Listing 6.3 captures the results of the third and final iteration of the same query. Note that the query now uses PERMANENT optimizer statistics because between the second and third iterations, the nightly statistics maintenance job has already been submitted and more accurate optimizer statistics are at last available, so there is no further need for the SPD.
SQL That (Almost) Tunes Itself: Oracle Database 12c R1 Optimizer Enhancements

continued from page 13

R1 Optimizer Enhancements

SINGLE TABLE CARDINALITY MISESTIMATE
61430811 2013-11-25.14:11:22 SH       SALES           COLUMN       QUANTITY_SOLD        PERMANENT
SINGLE TABLE CARDINALITY MISESTIMATE
61430811 2013-11-25.14:11:22 SH       SALES           TABLE                             PERMANENT

--- ----------------------------------------
SQL Plan Directive Reason
SPD ID   Used On             Owner    Object Name     Object Type  SubObject Name       State
Abbrev   Last                                                                           SPD
(from DBA_SQL_PLAN_DIR* Views)
page    1 - dynamic statistics used: dynamic sampling (level=4)
-----
Note
9 - access("S"."CUST_ID"="C"."CUST_ID")
7 - filter(("S"."AMOUNT_SOLD">1599.99 AND "S"."QUANTITY_SOLD"<3))

|  10 |     TABLE ACCESS BY INDEX ROWID| CUSTOMERS    |   3459 |      1 |   3459 |00:00:00.02 |
|*  9 |      INDEX UNIQUE SCAN         | CUSTOMERS_PK |   3459 |      1 |   3459 |00:00:00.01 |
|   6 |       PARTITION RANGE ALL      |              |      1 |    325 |   3459 |00:00:00.01 |
|   5 |      NESTED LOOPS              |              |      1 |    325 |   3459 |00:00:00.01 |
|   4 |     NESTED LOOPS               |              |      1 |    325 |   3459 |00:00:00.03 |
|   3 |    NESTED LOOPS                |              |      1 |        |   3459 |00:00:00.04 |
|   0 | SELECT STATEMENT               |              |      1 |        |     26 |00:00:00.17 |

Plan hash value: 929743582

Listing 6.2. SQL Plan Directives: Second Iteration

Plan hash value: 929743582

SINGLE TABLE CARDINALITY MISESTIMATE
61430811 2013-11-25.14:10:43 SH       SALES           COLUMN       AMOUNT_SOLD          HAS_STATS
SINGLE TABLE CARDINALITY MISESTIMATE
61430811 2013-11-25.14:10:43 SH       SALES           COLUMN       QUANTITY_SOLD        HAS_STATS

--- ----------------------------------------
SQL Plan Directive Reason
SPD ID   Used On             Owner    Object Name     Object Type  SubObject Name       State
Abbrev   Last                                                                           SPD
(from DBA_SQL_PLAN_DIR* Views)
page    1 - dynamic statistics used: dynamic sampling (level=4)
-----
Note
8 - access("S"."PROD_ID"="P"."PROD_ID")
7 - filter(("S"."AMOUNT_SOLD">1599.99 AND "S"."QUANTITY_SOLD"<3))

|*  9 |      INDEX UNIQUE SCAN         | CUSTOMERS_PK |   3459 |      1 |   3459 |00:00:00.02 |
|  7 |        TABLE ACCESS FULL       | SALES        |     28 |    325 |   3459 |00:00:00.06 |
|  6 |       PARTITION RANGE ALL      |              |      1 |    325 |   3459 |00:00:00.01 |
|  5 |      NESTED LOOPS              |              |      1 |    325 |   3459 |00:00:00.01 |
|  4 |     NESTED LOOPS               |              |      1 |    325 |   3459 |00:00:00.03 |
|  3 |    NESTED LOOPS                |              |      1 |        |   3459 |00:00:00.04 |
|  0 | SELECT STATEMENT               |              |      1 |        |     26 |00:00:00.17 |

Plan hash value: 929743582

Listing 6.3. SQL Plan Directives: Third Iteration

What’s Next for 12cR1?

In next issue’s column, I’ll continue my review of Oracle 12cR1 SQL performance enhancements with a closer look at the two new types of histograms — top frequency histograms and hybrid histograms — as well as investigate and demonstrate some significant improvements to SQL Plan Management (SPM) in Oracle 12cR1.

About the Author

Jim Czuprynski has accumulated more than 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 currently holds OCP certification for Oracle 9i, 10g and 11g. Jim 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. Jim is also a regular public speaker on Oracle Database technology features, and has presented topics at Oracle OpenWorld (2008 and 2013), IOUG’s COLLABORATE conferences (2011 and 2013) and OUG Norway (2013).