Advance

Take on challenges and watch what happens.

Features:

Customizing WebCenter Content
Supercharged Query Performance
Upgrade Oracle GoldenGate
DBA 201: Oracle B-Tree and Bitmap Indexes

www.ioug.org
What Happens When IMCS Is Exhausted?
In a word: nothing! Should its entire space capacity be consumed, any new tables or table partitions simply will not be populated into the IMCS. The fact that the IMCS is completely occupied is registered within the database’s alert log once every six minutes.

This error message ceases as soon as enough free space exists again within the IMCS — usually because an object with a higher priority but a smaller size, or even with a higher compression ratio (and thus likely to consume more memory), has requested residence within the IMCS. Of course, the Oracle DBA could attempt to intervene in an out-of-memory situation by changing attributes of an existing table or table partition, but she should remember one other important point: Issuing the ALTER TABLE ... INMEMORY COMPRESSON clause to reduce the size of a table doesn’t impact the space already allocated to that object in the IMCS. The modified compression state will only be reflected in the object’s data dictionary definition, and the new compression level won’t be enforced until the next time the object is populated within the IMCS.

IMCS: What Happens When DML Is Executed?
When a DML statement is executed against a table or table partition that’s already present in the IMCS, Oracle 12.1.0.2 handles the change quite differently than if the object was only present in the database buffer cache, as Figure 1 illustrates.

Figure 1: DML Statement Execution
Essentially, IMCS retains a special set of change vectors — one set per each IMCU with modified data — within a separate part of the IMCS called the Transaction Journal, but it is only for the row pieces that have been modified via DML. These change vectors are not online redo log entries; rather, they are stored in an entirely different format for quick access by any query that may need to obtain a read consistent version of the data. Note that even if the Database Writer process (DBWn) should write out dirty blocks back to disk, this has no impact on Transaction Journal entries; they are flushed and synchronized on a totally independent schedule based on a predetermined algorithm. When enough Transaction Journal entries become stale, the IMC0 (in-memory coordinator) background process flushes the journal automatically via calls to the SMC0 background process, which allocates one or more W000 worker processes to complete the flush and repopulate the IMCS-resident object’s IMCUs with the latest data.

To demonstrate exactly how this processing works, I’ll issue the DML statement in Listing 1 to apply some changes to the AP.

```
SQL> UPDATE ap.randomized_sorted
    2     SET key_sts = 50
    3     WHERE key_sts <> 50
    4     AND key_id BETWEEN 1100000 and 1300000;
```

60087 rows updated.

```
SQL> COMMIT;
```

Committed.

Listing 1: Impact of Updates on IMCS Transaction Journal

Shortly after running the UPDATE statement and committing the transaction, I ran the query in Listing 2. This query filters only the current session’s instance level statistics for any IMCS activity. Listing 3 shows the corresponding session-level instance statistics that record the repopulation of the IMCS Transaction Journal.

```
SQL> SELECT
    2     A.name,
    3     B.value
    4     FROM
    5     v$sysstat A,
    6     v$mystat B
    7     WHERE A.statistic# = B.statistic#
    8     AND (A.name LIKE '%IM%' OR A.name LIKE '%In memory%')
    9     ORDER BY A.name;
```

Listing 2: Querying Session-Level Instance Statistics for IMCS Transaction Journal

Listing 3: Session-Level Instance Statistics for IMCS Transaction Journal

```
SQL> SELECT
    2     A.name,
    3     B.value
    4     FROM
    5     v$sysstat A,
    6     v$mystat B
    7     WHERE A.statistic# = B.statistic#
    8     AND (A.name LIKE '%IM%' OR A.name LIKE '%In memory%')
    9     ORDER BY A.name;
```

Listing 4: Test Case #1: Query and Resulting Execution Plan

The corresponding EXPLAIN PLAN shows that this relatively tiny table (50,000 rows) was accessed within the IMCS as shown by the TABLE ACCESS INMEMORY FULL operation. The execution times for this query to complete with various degrees of in-memory compression are shown in Table 1.

```
<table>
<thead>
<tr>
<th>Method</th>
<th>Baseline Initial Population into IMCS</th>
<th>After IMCS Population</th>
</tr>
</thead>
<tbody>
<tr>
<td>SH.CUST_PREFS</td>
<td>1.65s</td>
<td>0.87s</td>
</tr>
<tr>
<td>Compression Method</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DML</td>
<td>1.82s</td>
<td>0.15s</td>
</tr>
<tr>
<td>QUERY LOW</td>
<td>1.44s</td>
<td>0.15s</td>
</tr>
<tr>
<td>QUERY HIGH</td>
<td>1.34s</td>
<td>0.10s</td>
</tr>
<tr>
<td>CAPACITY LOW</td>
<td>1.38s</td>
<td>0.14s</td>
</tr>
<tr>
<td>CAPACITY HIGH</td>
<td>2.77s</td>
<td>0.13s</td>
</tr>
</tbody>
</table>
```

Table 1: Test Case #1: Impact of In-Memory Compression at Various Compression Levels

In-Memory Compression: Impacts on Query Performance

I demonstrated how to estimate the approximate expected in-memory compression ratios for tables and table partitions in the previous article. However, I devoted very little time to show the impact of its different compression algorithms against tables that varied in size, data types composition and sparseness of data, all of which can have a dramatic impact on the compressibility of data within the IMCS.

Here are four brief test cases illustrating how in-memory compression positively influences the execution times of even relatively simple queries once their underlying objects have been placed within IMCS.

Test Case #1: Small Table, No Filtering

This first test is a simple query against the SH.CUST_PREFS table. This table is relatively small — just 16MB in size — with just 37 columns whose data types are primarily VARCHAR2; in addition, many of these columns are not populated with data at all. Listing 4 shows the query and its corresponding execution plan.

```
SQL> SELECT active_ind ,COUNT(*)
    2     FROM sh.cust_prefs GROUP BY active_ind
    3     ORDER BY active_ind;
```

Plan hash value: 2844705231

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (%CPU)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1171K</td>
<td>0</td>
<td>00:00:01</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>SORT GROUP BY</td>
<td></td>
<td>96</td>
<td>23 (5)</td>
<td>00:00:01</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>TABLE ACCESS INMEMORY FULL</td>
<td>CUST_PREFS</td>
<td>50000</td>
<td>1171K</td>
<td>21 (0)</td>
<td>00:00:01</td>
</tr>
</tbody>
</table>

Listing 4: Test Case #1: Query and Resulting Execution Plan

Click here to view Listing 3.
In this case, the resulting performance improvement is most noticeable when applying the QUERY HIGH compression method, with the query’s execution time decreasing by a factor of 16.5X as compared to the worst-case scenario for non-IMCS retrieval (1.65s / 0.1s = 16.5X). This is certainly a respectable result, considering that not one line of application code needed to be changed to accomplish this performance improvement.

**Test Case #2: Huge Fact Table, No Filtering**

This next test is yet another simple query executed against the TPCH.H_LINEITEM table. However, this is the largest table against which I tested IMCS performance; it is not wide (about a dozen columns with mixed NUMBER, DATE and VARCHAR2 data types) but it is extremely deep (approximately 24 million rows). The query and its resulting execution plan are shown in Listing 5.

```
SQL> SELECT     l_linestatus   ,l_returnflag   ,SUM(l_quantity)   ,SUM(l_extendedprice)
FROM tpch.h_lineitem
GROUP BY l_linestatus, l_returnflag
ORDER BY l_linestatus, l_returnflag;
```

Listing 5: Test Case #2: Query and Resulting Execution Plan

The corresponding EXPLAIN PLAN shows that the entire contents of the table were accessed inside the IMCS. The execution times for this query to complete with various degrees of in-memory compression are shown in Table 2.

<table>
<thead>
<tr>
<th>Baseline</th>
<th>Initial Population into DBC</th>
<th>After DBC Population</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPCH.H_LINEITEM</td>
<td>114.77s</td>
<td>104.97s</td>
</tr>
<tr>
<td>Compression Method</td>
<td>Initial Population into IMCS</td>
<td>After IMCS Population</td>
</tr>
<tr>
<td>DML</td>
<td>258.39s</td>
<td>11.72s</td>
</tr>
<tr>
<td>QUERY LOW</td>
<td>224.91s</td>
<td>9.64s</td>
</tr>
<tr>
<td>QUERY HIGH</td>
<td>206.56s</td>
<td>12.87s</td>
</tr>
<tr>
<td>CAPACITY LOW</td>
<td>212.30s</td>
<td>10.05s</td>
</tr>
<tr>
<td>CAPACITY HIGH</td>
<td>197.58s</td>
<td>10.65s</td>
</tr>
</tbody>
</table>

Table 2: Test Case #2: Impact of In-Memory Compression at Various Compression Levels

In this case, the QUERY LOW compression method showed the most improvement, with the query’s execution time decreasing by a factor of 11.9X as compared to the worst-case scenario for non-IMCS retrieval (114.77s / 9.64s = 11.9X). This is certainly a respectable result, but not in the range of 100X that Oracle has often claimed can be expected from queries against IMCS-resident tables. This is probably the case because the target table (AP.EST_SALES) is actually quite “wide” with over 190 columns, and every numeric column for that table is populated with data, so in-memory compression has a limited capability to compress this object and therefore influence a positive result.

**Test Case #3: Huge Fact Table, In-Memory Filtering**

Listing 6 shows yet another query against the TPCH.H_LINEITEM table and its resulting execution plan. This test case applied in-memory filtering for the first time, and it needed to retrieve data against about a quarter of the table before it performed its aggregation processing.

```
SQL> SELECT     l_linestatus   ,l_returnflag   ,SUM(l_quantity)   ,SUM(l_extendedprice)
FROM tpch.h_lineitem
WHERE     l_linestatus <> 'O'
AND     l_returnflag IN ('A','R')
GROUP BY l_linestatus, l_returnflag
ORDER BY l_linestatus, l_returnflag;
```

Listing 6: Test Case #3: Query and Resulting Execution Plan

In this case, the EXPLAIN PLAN shows in-memory filtering was definitely applied against the target table to aggregate counts within two columns and returning just three rows in the final result set. The results of this query’s execution with various degrees of in-memory compression are shown in Table 3.

<table>
<thead>
<tr>
<th>Baseline</th>
<th>Initial Population into DBC</th>
<th>After DBC Population</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPCH.H_LINEITEM</td>
<td>130.78s</td>
<td>112.45s</td>
</tr>
<tr>
<td>Compression Method</td>
<td>Initial Population into IMCS</td>
<td>After IMCS Population</td>
</tr>
<tr>
<td>DML</td>
<td>175.80s</td>
<td>10.10s</td>
</tr>
<tr>
<td>QUERY LOW</td>
<td>190.18s</td>
<td>4.34s</td>
</tr>
<tr>
<td>QUERY HIGH</td>
<td>141.31s</td>
<td>4.68s</td>
</tr>
<tr>
<td>CAPACITY LOW</td>
<td>226.45s</td>
<td>7.60s</td>
</tr>
<tr>
<td>CAPACITY HIGH</td>
<td>203.09s</td>
<td>6.07s</td>
</tr>
</tbody>
</table>

Table 3: Test Case #3: Impact of In-Memory Compression at Various Compression Levels
The secret to this extremely positive result is that there are nearly 180 columns in table AP.EST_SALES between those named in the query (sls_2014q3 and sls_1970q2) that Oracle 12.1.02 can simply ignore as it performs its accumulations. This shows the real potential of IMCS: the elimination of columns from consideration during complex queries, especially when the row sets being processed have large numbers of columns that IMCS doesn’t need to consider at all when it attempts to provide answers.

Test Case #4: Large Table, In-Memory Filtering
In this final test case, the query in Listing 8 was executed against table AP.EST_SALES, which had been fully populated into IMCS.

SQL> SELECT
    2    zipcode,
    2    SUM(sls_2014q4) "2014_Q4", SUM(sls_2014q3) "2014_Q3"
    2    SUM(sls_1970q2) "1970_Q2", SUM(sls_1970q1) "1970_Q1"
FROM ap.est_sales
WHERE zipcode LIKE '606%'
GROUP BY zipcode ORDER BY zipcode;

Plan hash value: 2076845129

Listing 7: Test Case #4: Query and Resulting Execution Plan

Note that the EXPLAIN PLAN shows in-memory filtering was definitely applied against the target table to retrieve only the data for ZIPCODE values that begin with 606 (a large metropolitan region near Chicago, Illinois). The results of this query’s execution with various degrees of in-memory compression are shown in Table 4.

Impressive … most impressive! This time, the QUERY HIGH compression method showed the most improvement in execution time, which decreased by a factor of 137.1X as compared to the worst-case scenario for non-IMCS retrieval (19.2s / 0.14s = 137.1X).

The QUERY LOW compression method once again showed the most improvement in execution time, which decreased by a factor of 30.1X as compared to the worst-case scenario for non-IMCS retrieval (130.78s / 4.34s = 30.1X). This is an eminently respectable result, especially considering that no indexes were needed to filter out the approximately 75 percent of all rows in the table.

The QUERY LOW compression method once again showed the most improvement in execution time, which decreased by a factor of 30.1X as compared to the worst-case scenario for non-IMCS retrieval (130.78s / 4.34s = 30.1X). This is an eminently respectable result, especially considering that no indexes were needed to filter out the approximately 75 percent of all rows in the table.

Test Case #4: Large Table, In-Memory Filtering
In this final test case, the query in Listing 8 was executed against table AP.EST_SALES, which had been fully populated into IMCS.

SQL> SELECT
    2    zipcode,
    2    SUM(sls_2014q4) "2014_Q4", SUM(sls_2014q3) "2014_Q3"
    2    SUM(sls_1970q2) "1970_Q2", SUM(sls_1970q1) "1970_Q1"
FROM ap.est_sales
WHERE zipcode LIKE '606%'
GROUP BY zipcode ORDER BY zipcode;

Plan hash value: 2076845129

Listing 7: Test Case #4: Query and Resulting Execution Plan

Note that the EXPLAIN PLAN shows in-memory filtering was definitely applied against the target table to retrieve only the data for ZIPCODE values that begin with 606 (a large metropolitan region near Chicago, Illinois). The results of this query’s execution with various degrees of in-memory compression are shown in Table 4.

Impressive … most impressive! This time, the QUERY HIGH compression method showed the most improvement in execution time, which decreased by a factor of 137.1X as compared to the worst-case scenario for non-IMCS retrieval (19.2s / 0.14s = 137.1X).

What’s Next for 12.1.0.2?
Now that I’ve covered a majority of the internal mechanisms of the IMCS, I’ll turn my attention to three other brand-new special features of Oracle Database Release 12.1.0.2, including:

- How in-memory joins (IMJ) speed the application of bloom filters for star schema queries
- How in-memory aggregation (IMA) works in concert with IMJ, and how the query optimizer decides to choose between these methods
- How the new advanced index compression (AIC) features offer the ability to compress indexes for significant space savings and faster access to data when an index can still be leveraged for data retrieval

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 9i, 10g and 11g.