Changes

Technology changes, our careers change and the industry changes — so we learn, adapt and grow.

Features:

Calling SQL from PL/SQL the Right Way

Oracle vs. NoSQL – A Primer (Part II)

In-Memory Column Store in Oracle 12.1.0.2

DBA 201: Database LISTENER Registration

www.ioug.org
I’ll start with a brief review of how Oracle databases handle retrieving and updating data in prior releases.

**Row-Major Storage: The Sword That Cuts Both Ways**

The Oracle Database — as well as other traditional RDBMs like DB2, SQLServer, Sybase and Teradata — typically store table and index data as separate rows within a logical structure that’s usually called a database block or page. In the case of Oracle, each database block’s logical structure is superimposed upon several physical blocks stored on the OS platform’s underlying file system. For example, EXT4 for Linux, ZFS for Solaris, or even Automatic Storage Management (ASM) for post-Oracle 10g databases, if desired. Nothing much happens within an Oracle Database until a database block is accessed through either a query (SELECT) or a DML statement (INSERT, UPDATE or DELETE), at which point the database will retrieve the needed blocks into the database instance’s database buffer cache, so that it’s now possible for the actual row pieces stored within each database block to be read and/or modified. (Granted, direct-path reads work completely differently; starting in Oracle Database Release 11gR1, the results from scanning a large table are placed directly into the SQL work area of the requesting session’s PGA instead of the database buffer cache.)

Prior to Oracle 12.1.0.2, the only possible way for Oracle to present the contents of each database block was in row major format. For many database operations — especially DML — row major storage offers excellent advantages. For example, referential integrity is crucial to a transactional, online transaction processing (OLTP) application, so the ability to quickly verify that a row is truly unique within a database table during its initial INSERT is critical. Likewise, row-major format makes short work of constructing valid parent-child relationships, like creating a new invoice line item that’s associated with an invoice during manual accounts payable processing, or determining if a current customer entry exists via a FOREIGN KEY reference while constructing a new invoice for that customer.

But like an extremely sharp two-edged sword, row major storage unfortunately cuts both ways. It’s extremely beneficial for OLTP activity and queries that retrieve a relatively small subset of data from tables, especially when that data can be quickly accessed through index structures, or when the majority of the data in the row itself needs to be accessed. On the other hand, row-major storage is quite detrimental for queries that need to process only a small number of columns from the majority of rows in a table, or that need to retrieve a significant number of rows — but not all of them — in the table. To make matters worse, analytical queries that access these OLTP tables often need to retrieve a relatively small subset of data. Oracle DBAs
typically added multiple indexes on columns accessed most frequently to speed data retrieval during FILTER operations, or to bypass a full table scan of the entire table by using the indexes as an alternative row source. Of course, those indexes need to be maintained as well whenever the base table’s rows were added or modified.

The performance of queries against non-OLTP tables fared little better with row-major storage. Consider the practice of constructing materialized views — essentially, denormalized tables built specifically for a data warehouse — or a table ported directly to Oracle from a non-Oracle legacy system like table AP.EST_SALES shown in Online Listing 1. If a query accessed just 10 of the 60 columns from AP.EST_SALES, the columnar data that needs to be accessed comprises just one-sixth (16 percent) of the table’s columns, but because the table’s data is returned in row-major format, the additional columns for each row piece must be returned even though they are unnecessary to answer the query.

Click here to view the listing.

The Oracle Exadata Database Machine attempts to alleviate the performance bottleneck of row major storage through its smart scan features, which pushes much of the overhead of full table scans down to the individual storage cells, enables columnar projection (i.e., filtering only the columns needed to answer the query) and, in many cases, eliminates the need for unselective indexes. Exadata’s Hybrid Columnar Compression (HCC) features also enables dramatic compression of data on physical disk — especially valuable for historical data that’s hardly ever accessed — and also reduces the number of physical I/Os needed to retrieve data.

Of course, Exadata technology is not inexpensive. Many IT organizations reluctant to adopt Exadata have attempted to alleviate the row major storage bottleneck by shifting their analytical processing and reporting to other technology entirely, adopting federated databases like Hadoop or using SAP’s HANA data storage model. But, at a minimum, the data needed for analysis must be reloaded into these federated structures, to say nothing of the costs to train application developers and DBAs to use this new technology.

In-Memory Column Store (IMCS): Features and Concepts

The solution to this issue is truly a sea change for the Oracle 12c Database: Let the database buffer cache do what it’s best at, which is representing a database block in memory for quick access during DML, and instead simultaneously store an object’s data that must be accessed quickly for reporting in a complementary, but completely different, format in memory. This new feature is called the In-Memory Column Store (IMCS) and, as its name implies, it stores data for specific database objects in columnar format for fastest access for analytic processing as well as for simple and complex queries. In essence, IMCS makes many of the best features of Exadata — columnar projection, columnar compression, Smart Flash Cache and über-fast data retrieval of huge datastores — directly available to database instance. We’ll take a much closer look at how columnar store data is retained within the IMCS in a later section, but here are some of the additional intrinsic benefits of IMCS.

In-Memory Compression

Data captured within the IMCS is not only stored in the more appropriate columnar format for reporting, but it can also be compressed quite tightly — between 200 percent and 1,000 percent — so that more row source data can be accessed much more quickly. Data retained within the IMCS can be compressed at one of three compression levels (HIGH, MEDIUM or LOW), and the Oracle DBA has precise control over which database objects — tables, partitions, subpartitions and even a table’s individual columns — she has determined will most benefit from IMCS features, and apply appropriate levels of compression to those objects. And, as I’ll demonstrate, the potential compression ratio can be determined beforehand via the Oracle 12c Compression Advisor so that objects targeted for compression and an appropriate compression level can be chosen wisely.

Reducing Unnecessary Indexes

Because data placed within the IMCS can be accessed quickly and filtered readily, many of the indexes that have been inflicted upon tables in an attempt to increase the selectivity of its column(s) become completely unnecessary. (Of course, indexes that enforce a PRIMARY KEY or UNIQUE KEY constraint or that are needed to enforce foreign key references between parent and child entities are not unnecessary.) Because maintenance of indexes during DML invocation is one of the most expensive operations that any RDBMS must perform, this also means those CPU cycles can be freed up for other, more meaningful purposes. And OLTP application workload operations performed against database objects with fewer indexes are likely to benefit dramatically once the indexes that had been present only for faster data selection during queries are permanently removed.

SQL Tuning Impacts

Each new release of the Oracle optimizer tends to improve query performance dramatically — especially with the introduction of Adaptive Execution Plans and Automatic Re-Optimization features in Oracle 12.1.0.1. However, many Oracle DBAs still spend an inordinate amount of their day tuning pesky, poorly written or complex queries. The good news about IMCS is that in almost all cases, it will have no negative effects on existing application code; in fact, the performance improvement for most queries may be so dramatic that there will be no need to perform any additional SQL tuning. (One small caveat regarding deciding to place only some of a table’s columns within IMCS: Because of the way IMCS works, there is a small probability that additional tuning may be required. We’ll demonstrate this scenario in more detail in another article.)

Other Key Advantages

The following are some other key advantages of implementing IMCS:
- Even though a partitioned table’s partitions can definitely take advantage of IMCS, not all partitions necessarily need to be placed within it. For example, partitions mainly containing older, “cold” data that’s only being retained for regulatory purposes are likely to gain little from placement within IMCS.

- It’s also possible to decorate a database object with an IMCS priority that’s different than another object’s priority. For example, a table partition that contains data that’s constantly being accessed for either queries or DML would likely get the most benefit from an INMEMORY_PRIORITY setting of CRITICAL versus data that’s hardly accessed at all, which can be assigned a setting of MEDIUM or LOW.

- The operation of Real Application Cluster (RAC) databases is unchanged, and the removal of unnecessary indexes may even significantly improve performance of OLTP workloads in a RAC environment.

- The new Oracle 12c multitenant database architecture is fully supported; database objects stored in non-CDBs, CDBs and PDBs can all take advantage of IMCS features.

- All other database operations (e.g., backup and recovery) are unchanged.

- Best of all, no data needs to be reloaded to take advantage of these features. From my perspective, this is truly the most dramatic feature of IMCS because many other solutions to this issue — for example, federated databases like Hadoop or other reporting-centric solutions like SAP HANA — essentially require completely reloading the required data into entirely different structures.

**IMCS Architecture: IMCU and IM Data Segments**

Whenever a database object is placed into IMCS, the rows are “chunked” into a series of In-Memory Column Units (IMCUs). IMCUs are stored within a separate, contiguous region of SGA memory. Unlike database block images stored in row major format in the database buffer cache, IMCUs are much larger in size — usually many orders of magnitude larger! — to enable faster scan speeds. The number of rows per IMCU is determined at runtime based on an internal algorithm that takes into account the size and structure of each database object as well as the amount of free memory within the IMCS. And because IMCUs are purely an in-memory format and are never used for DML, there are no additional writes to disk, which means no additional overhead of writing to online redo logs or to an UNDO segment.

IMCUs are organized into In-Memory Data Segments, which are collections of used IMCUs. A database object is converted from its on-disk format to in-memory columnar format within an IMCU when either an object is first placed within the IMCS by altering its state to be included there, or during instance startup, during which any object that’s already been defined for residence in the IMCS is converted. When conversion takes place, it’s possible the whole object may not initially fit in memory, or it may take a considerable amount of time to convert the object. While the conversion proceeds, a query accessing the data that’s not yet fully populated within the IMCS will continue to access it on disk, thus alleviating the wait for full availability.

It’s important to note that the IMCS buffer pool is a static pool, so like other optional static buffer pools in the SGA — for example, the KEEP and RECYCLE database buffer caches — Automatic Shared Memory Management (ASMM) will not auto-tune its size, and the memory allocated to IMCS is still part of the memory bound allocated via the SGA_TARGET initialization parameter. And like other static memory pools, IMCS does not use a least-recently used (LRU) algorithm to manage its content, so candidate objects essentially obtain memory on a first-come, first-served basis and will not be populated into the IMCS unless free memory is available.

**Populating the IMCS: An Example**

The new In-Memory Coordinator background process (IMCO) handles the creation of population and refresh of all data stored within IMCUs in the IMCS. IMCO is responsible for scheduling IMCS object population and repopulation, and it coordinates that activity via the Space Management Coordinator (SMCO) background process and multiple “minion” worker processes (Wnnn) that perform the actual population and repopulation of the IMCUs within the IMCS.

So what happens when a database object that’s been marked as a candidate for retention within the IMCS is accessed?

- If the object hasn’t yet been populated into the IMCS, the IMCO, SMCO and Wnnn background processes convert the on-disk row major format data into columnar store format within IMCUs in the IMCS.

- If a specific compression level has been requested for either the entire object, its sub-objects (partitions and sub-partitions), or any of its columns, compression will be applied as the columnar store format data is converted.

- Database objects that have been marked with a higher priority (i.e., CRITICAL or HIGH) get first consideration over objects with a lower priority (i.e., MEDIUM or LOW).

- As much data as possible will be converted, but it’s possible that there won’t be enough free memory in the IMCS to fit the candidate object’s data in its entirety.

- Finally, the query will retrieve as much data as possible from the columnar store format first, and then proceed to retrieve data from any data resident for the database object in the database buffer cache, the database flash cache and then finally from disk. (Note that just because an Oracle DBA has requested an object’s presence within the IMCS doesn’t mean it has already been populated completely.)

- IMCS employs columnar projection (i.e., it reads only the needed columns and ignores the rest. If the query has selected all of the object’s columns, then all data is accessed directly from IMCS memory; if only some columns have been populated, however,
then those columns' data have to be retrieved from the database blocks in the database buffer cache).

- Once the database object has been fully populated into the IMCS, the query optimizer can take immediate advantage of the speed at which data in the columnar store format is accessed and execute the statement as an In-Memory Query.

**IMCS and Modified Data**

Finally, what happens if data is modified for a database object that's already been populated into the IMCS?

- The columnar entries in the corresponding IMCUs are marked as stale, then the IMCO, SMCO and Wnnn processes capture the row versions that are recorded in the transaction journal from the modified data in the database buffer cache.

- This keeps transaction activity where it belongs — the database buffer cache — and only the affected data needs to be synchronized with the data already present in the IMCUs.

- The resynchronization is handled automatically and is dependent upon an internal algorithm based on a combination of factors, including the number of invalidated rows per each IMCU, the IMCS transaction journal running low on memory, and the number of invalidated of block images due to activity in Real Application Cluster (RAC) database instances.

**Planning an IMCS Implementation**

As I’ll demonstrate in the next section, it is almost ridiculously simple to activate IMCS for any Oracle 12.1.0.2 Database. However, that doesn’t mean that some preplanning isn’t warranted. Here are some prerequisites to consider.

**IMCS: Potential Beneficiaries**

Just as it’s rarely useful to pin an entire table permanently into the database buffer cache with the CACHE storage attribute, IMCS needs to be used with some intelligence as well. First, the following database objects can take advantage of columnar storage within the IMCS:

- Heap-organized tables
- Table partitions and subpartitions
- In-line LOBs
- Materialized views, materialized view logs and materialized join views

However, the following data objects and datatypes cannot be placed into IMCS:

- Clustered tables and index-organized tables (IOTs)
- Column datatypes of LONG, LONG RAW, out-of-line LOBs, VARRAYs or nested columns
- Any object that’s owned by SYS or that resides in either the SYSTEM or SYSAUX tablespaces

**Estimating IMCS Space Requirements**

If the in-memory compression features of IMCS appear attractive, the good news is that it’s relatively easy to estimate how much compression will benefit which database objects through the enhanced Compression Advisor (DBMS_COMPRESSION) in 12.1.0.2. I will demonstrate how to leverage these new capabilities in the next section.

**Database Bounce Required**

Finally, remember that activating IMCS does, unfortunately, require a bounce of the database instance before the IMCS is activated because the INMEMORY_SIZE initialization parameter is non-dynamic. The minimum setting for INMEMORY_SIZE is 100MB; any attempt to allocate less than that amount of memory may cause the instance restart to fail.

**Viewing IMCS Metadata**

There are two types of IMCS metadata: static information specific to database object attributes, and dynamic information that changes over time as objects enter and leave the boundaries of IMCS. Online Listing 2 contains the queries that return the SQL*Plus formatted reports about IMCS.

To see the impact of placing these tables and partitions into IMCS, I executed queries against three data dictionary views — DBA_TABLES, DBA_SEGMENTs, and DBA_TAB_PARTITIONS — to see which database objects in the AP schema are now enabled or disabled for IMCS. The results are shown in Online Listing 3.

Oracle 12.1.0.2 also provides two new dynamic views — V$IM_SEGMENT and V$IM_COLUMN_LEVEL — that provide information about what database objects are actually deployed within IMCS.

**A Brief Proof of Concept: Query Performance With Versus Without IMCS**

The key premise of IMCS is that it will enhance query performance without changing any application code. I’ll demonstrate the performance of a simple query (see Listing 1) against the AP.RANDOMIZED_SORTED table.

```
SELECT key_sts, COUNT(*)
FROM ap.randomized_sorted
WHERE key_sts IN(10,20,30)
GROUP BY key_sts;
```

Listing 1: A Simple Query
To ensure that IMCS is not used for the first invocation of this query, I’ve set the INMEMORY_QUERY initialization parameter to FALSE at a session level. The resulting execution statistics and execution plan before the table was added to IMCS are shown in Listing 2.

Listing 2: Query Performance Without Utilizing IMCS

Listing 3 shows the same query’s execution statistics and execution plan after the table has been fully populated within IMCS:

Listing 3: Query Performance Utilizing IMCS

As shown in Table 1, even though this query is relatively simple, note that its performance has increased significantly — 5.96 seconds versus 0.10 seconds, or nearly 60 times faster — with absolutely no changes to the application code.

Let’s walk through an example.

What’s Next for IMCS and Oracle 12.1.0.2?

This article has just scratched the surface of the potentially immense performance improvements that IMCS can provide. Now that I’ve hopefully explained the basic concepts of IMCS and demonstrated how simple it is to deploy and managed, the next article in this series will delve much deeper into the features of IMCS.

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.