

Oracle Database In-Memory

ORACLE WHITE PAPER | JULY 2015





Table of Contents

Executive Overview	3
Intended Audience	3
Introduction	4
Oracle Database In-Memory Option	5
Row Format vs. Column Format	5
The In-Memory Column Store	6
In-Memory Compression	8
In-Memory Scans	10
In-Memory Joins	12
In-Memory Aggregation	13
DML And The In-Memory Column Store	15
Bulk Data Loads	15
Partition Exchange Loads	16
Transaction Processing	17
The In-Memory Column Store On RAC	18
In-Memory Fault Tolerance	19
The In-Memory Column Store in a Multitenant Environment	20
Controlling the use of Oracle Database In-Memory	21



Core Initialization Parameters	21
Additional Initialization Parameters	22
Optimizer Hints	23
Monitoring and Managing Oracle Database In-Memory	24
Monitoring What Objects Are In The In-Memory Column Store	24
Managing IM Column Store Population CPU consumption	25
Conclusion	27



Executive Overview

Oracle Database In-Memory transparently accelerates analytic queries by orders of magnitude, enabling real-time business decisions. Using Database In-Memory, businesses can instantaneously run analytics and reports that previously took hours or days. Businesses benefit from better decisions made in real-time, resulting in lower costs, improved productivity, and increased competitiveness.

Oracle Database In-Memory accelerates both Data Warehouses and mixed workload OLTP databases, and is easily deployed under any existing application that is compatible with Oracle Database 12c. No application changes are required. Database In-Memory uses Oracle's mature scale-up, scale-out, and storage-tiering technologies to cost effectively run any size workload. Oracle's industry leading availability and security features all work transparently with Database In-Memory, making it the most robust offering on the market.

The ability to easily perform real-time data analysis together with real-time transaction processing on all existing applications enables organizations to transform into Real-Time Enterprises that quickly make data-driven decisions, respond instantly to customer demands, and continuously optimize all key processes.

Intended Audience

Readers are assumed to have hands-on experience with Oracle Database technologies from the perspective of a DBA or performance specialist.



Introduction

Today's information architecture is much more dynamic than it was just a few years ago. Business users now demand more decision-enabling information, sooner. In order to keep up with increases in demand, companies are being forced to run analytics on their operational systems, in addition to their data warehouses. This leads to a precarious balancing act between transactional workloads, subject to frequent inserts and updates, and reporting style queries that need to scan large amounts of data.

With the introduction of Oracle Database In-Memory, a single database can now efficiently support mixed workloads, delivering optimal performance for transactions while simultaneously supporting real-time analytics and reporting. This is possible due to a unique "dual-format" architecture that enables data to be maintained in both the existing Oracle row format, for OLTP operations, and a new purely in-memory column format, optimized for analytical processing. In-Memory also enables both datamarts and data warehouses to provide more ad-hoc analytics, giving end-users the ability to ask multiple business driving queries in the same time it takes to run just one now.

Embedding the in-memory column format into the existing Oracle Database software ensures that it is fully compatible with ALL existing features, and requires no changes in the application layer. Companies striving to become real-time enterprises can more easily achieve their goals, regardless of what applications they are running. This paper describes the main components of Oracle Database In-Memory and provides simple, reproducible examples to make it easy to get acquainted with them. It also outlines how Database In-Memory can be integrated into existing operational systems and data warehouse environments to improve both performance and manageability.

This whitepaper is the first in a two part series on Oracle Database In-Memory. It describes the main components and the key concepts of Oracle Database In-Memory, while the second part outlines the best practices for Implementing it.

Oracle Database In-Memory Option

Row Format vs. Column Format

Oracle Database has traditionally stored data in a row format. In a row format database, each new transaction or record stored in the database is represented as a new row in a table. That row is made up of multiple columns, with each column representing a different attribute about that record. A row format is ideal for online transaction systems, as it allows quick access to all of the columns in a record since all of the data for a given record are kept together in-memory and on-storage.

A column format database stores each of the attributes about a transaction or record in a separate column structure. A column format is ideal for analytics, as it allows for faster data retrieval when only a few columns are selected but the query accesses a large portion of the data set.

But what happens when a DML operation (insert, update or delete) occurs on each format? A row format is incredibly efficient for processing DML as it manipulates an entire record in one operation i.e. insert a row, update a row or delete a row. A column format is not so efficient at processing row-wise DML: In order to insert or delete a single record in a column format all of the columnar structures in the table must be changed.

Up until now you have been forced to pick just one format and suffer the tradeoff of either sub-optimal OLTP or sub-optimal analytics performance.

Oracle Database In-Memory (Database In-Memory) provides the best of both worlds by allowing data to be simultaneously populated in both an in-memory row format (the buffer cache) and a new in-memory column format.

Note that the dual-format architecture **does not** double memory requirements. The in-memory column format should be sized to accommodate the objects that must be stored in memory, but the buffer cache has been optimized for decades to run effectively with a much smaller size than the size of the database. In practice it is expected that the dual-format architecture will impose less than a 20% overhead in terms of total memory requirements. This is a small price to pay for optimal performance at all times for all workloads.



Figure 1. Oracle's unique dual-format architecture.

With Oracle's unique approach, there remains a single copy of the table on storage, so there are no additional storage costs or synchronization issues. The database maintains full transactional consistency between the row and the columnar formats, just as it maintains consistency between tables and indexes. The Oracle Optimizer is fully aware of the column format: It automatically routes analytic queries to the column format and OLTP

operations to the row format, ensuring outstanding performance and complete data consistency for all workloads without any application changes.

The In-Memory Column Store

Database In-Memory uses an In-Memory column store (IM column store), which is a new component of the Oracle Database System Global Area (SGA), called the In-Memory Area. Data in the IM column store does not reside in the traditional row format used by the Oracle Database; instead it uses a new column format. The IM column store does not replace the buffer cache, but acts as a supplement, so that data can now be stored in memory in both a row and a column format.

The In-Memory area is a static pool within the SGA, whose size is controlled by the initialization parameter `INMEMORY_SIZE` (default 0). The current size of the In-Memory area is visible in `V$SGA`. As a static pool, any changes to the `INMEMORY_SIZE` parameter will not take effect until the database instance is restarted. It is also not impacted or controlled by Automatic Memory Management (AMM). The In-Memory area must have a minimum size of 100MB.

The In-Memory area is sub-divided into two pools: a 1MB pool used to store the actual column formatted data populated into memory, and a 64K pool used to store metadata about the objects that are populated into the IM column store. The amount of available memory in each pool is visible in the `V$INMEMORY_AREA` view. The relative size of the two pools is determined by internal heuristics, the majority of the In-Memory area memory is allocated to the 1MB pool.

```
SQL> Select pool, alloc_bytes, used_bytes, populate_status
       2 From V$INMEMORY_AREA;
```

POOL	ALLOC_BYTES	USED_BYTES	POPULATE_STATUS
1MB POOL	1710227456	16777216	DONE
64KB POOL	419430400	1900544	DONE

Figure 2. Details of the space allocation within the INMEMORY_AREA as seen in V\$INMEMORY_AREA


Populating The In-Memory Column Store

Unlike a pure In-Memory database, not all of the objects in an Oracle database need to be populated in the IM column store. The IM column store should be populated with the most performance-critical data in the database. Less performance-critical data can reside on lower cost flash or disk. Of course, if your database is small enough, you can populate all of your tables into the IM column store. Database In-Memory adds a new `INMEMORY` attribute for tables and materialized views. Only objects with the `INMEMORY` attribute are populated into the IM column store. The `INMEMORY` attribute can be specified on a tablespace, table, (sub)partition, or materialized view. If it is enabled at the tablespace level, then all new tables and materialized views in the tablespace will be enabled for the IM column store by default.

```
ALTER TABLESPACE ts_data DEFAULT INMEMORY;
```

Figure 3. Enabling the In-Memory attribute on the ts_data tablespace by specifying the INMEMORY attribute

By default, all of the columns in an object with the `INMEMORY` attribute will be populated into the IM column store. However, it is possible to populate only a subset of columns if desired. For example, the following



statement sets the In-Memory attribute on the table `SALES`, in the `SH` sample schema, but it excludes the column `PROD_ID`.

```
ALTER TABLE sales INMEMORY NO INMEMORY(prod_id);
```

Figure 4. Enabling the In-Memory attribute on the sales table but excluding the `prod_id` column

Similarly, for a partitioned table, all of the table's partitions inherit the in-memory attribute but it's possible to populate just a subset of the partitions or sub-partitions.

To indicate an object is no longer a candidate, and to instantly remove it from the IM column store, simply specify the `NO INMEMORY` clause.

```
ALTER TABLE sales MODIFY PARTITION SALES_Q1_1998 NO INMEMORY;
```

Figure 5. Disabling the In-Memory attribute on one partition of the sales table by specifying the `NO INMEMORY` clause

The IM column store is populated by a set of background processes referred to as *worker processes* (`ora_w001_orcl`). The database is fully active / accessible while this occurs. With a pure in-memory database, the database cannot be accessed until all the data is populated into memory, which causes severe availability issues.

Each worker process is given a subset of database blocks from the object to populate into the IM column store. Population is a streaming mechanism, simultaneously columnizing and compressing the data.

Just as a tablespace on disk is made up of multiple extents, the IM column store is made up of multiple In-Memory Compression Units (IMCUs). Each worker process allocates its own IMCU and populates its subset of database blocks in it. Data is not sorted or ordered in any specific way during population. It is read in the same order it appears in the row format.

Objects are populated into the IM column store either in a prioritized list immediately after the database is opened or after they are scanned (queried) for the first time. The order in which objects are populated is controlled by the keyword `PRIORITY`, which has five levels (see figure 7). The default `PRIORITY` is `NONE`, which means an object is populated only after it is scanned for the first time. All objects at a given priority level must be fully populated before the population for any objects at a lower priority level can commence. However, the population order can be superseded if an object without a `PRIORITY` is scanned, triggering its population into IM column store.

```
ALTER TABLE customers INMEMORY PRIORITY CRITICAL;
```

Figure 6. Enabling the In-Memory attribute on the customers table with a priority level of critical

PRIORITY	DESCRIPTION
CRITICAL	Object is populated immediately after the database is opened
HIGH	Object is populated after all CRITICAL objects have been populated, if space remains available in the IM column store
MEDIUM	Object is populated after all CRITICAL and HIGH objects have been populated, and space remains available in the IM column store
LOW	Object is populated after all CRITICAL, HIGH, and MEDIUM objects have been populated, if space remains available in the IM column store
NONE	Objects only populated after they are scanned for the first time (Default), if space is available in the IM column store

Figure 7. Different priority levels controlled by the PRIORITY sub clause of the INMEMORY clause

Restrictions

Almost all objects in the database are eligible to be populated into the IM column but there are a small number of exceptions. The following database objects cannot be populated in the IM column store:

- Any object owned by the SYS user and stored in the SYSTEM or SYSAUX tablespace
- Index Organized Tables (IOTs)
- Clustered Tables

The following data types are also not supported in the IM column store:

- LONGS (deprecated since Oracle Database 8i)
- Out of line LOBS

All of the other columns in an object that contains these datatypes are eligible to be populated into the IM column store. Any query that accesses only the columns residing in the IM column store will benefit from accessing the table data via the column store. Any query that requires data from columns with a non-supported column type will be executed via the buffer cache.

Objects that are smaller than 64KB are not populated into memory, as they will waste a considerable amount of space inside the IM column store as memory is allocated in 1MB chunks.

The IM column store cannot be used on an Active Data Guard standby instance in the current release. However it can be used in a Logical Standby instance and in an instance maintained using Oracle Golden Gate.

In-Memory Compression

Typically compression is considered only as a space-saving mechanism. However, data populated into the IM column store is compressed using a new set of compression algorithms that not only help save space but also improve query performance. The new Oracle In-Memory compression format allows queries to execute directly against the compressed columns. This means all scanning and filtering operations will execute on a much smaller amount of data. Data is only decompressed when it is required for the result set.

In-memory compression is specified using the keyword `MEMCOMPRESS`, a sub-clause of the `INMEMORY` attribute. There are six levels, each of which provides a different level of compression and performance.

COMPRESSION LEVEL	DESCRIPTION
<code>NO MEMCOMPRESS</code>	Data is populated without any compression
<code>MEMCOMPRESS FOR DML</code>	Minimal compression optimized for DML performance
<code>MEMCOMPRESS FOR QUERY LOW</code>	Optimized for query performance (default)
<code>MEMCOMPRESS FOR QUERY HIGH</code>	Optimized for query performance as well as space saving
<code>MEMCOMPRESS FOR CAPACITY LOW</code>	Balanced with a greater bias towards space saving
<code>MEMCOMPRESS FOR CAPACITY HIGH</code>	Optimized for space saving

Figure 8. Different compression levels controlled by the `MEMCOMPRESS` sub-clause of the `INMEMORY` clause

By default, data is compressed using the `FOR QUERY LOW` option, which provides the best performance for queries. This option utilizes common compression techniques such as Dictionary Encoding, Run Length Encoding and Bit-Packing. The `FOR CAPACITY` options apply an additional compression technique on top of `FOR QUERY` compression, which can have a significant impact on performance as each entry must be decompressed before the `WHERE` clause predicates can be applied. The `FOR CAPACITY LOW` option applies a proprietary compression technique called OZIP that offers extremely fast decompression that is tuned specifically for Oracle Database. The `FOR CAPACITY HIGH` option applies a heavier-weight compression algorithm with a larger penalty on decompression in order to provide higher compression.

Compression ratios can vary from 2X – 20X, depending on the compression option chosen, the datatype, and the contents of the table. The compression technique used can vary across columns, or partitions within a single table. For example, you might optimize some columns in a table for scan speed, and others for space saving.

```
CREATE TABLE employees
( c1 NUMBER,
  c2 NUMBER,
  c3 VARCHAR2(10),
  c4 CLOB )
INMEMORY MEMCOMPRESS FOR QUERY
NO INMEMORY(c4)
INMEMORY MEMCOMPRESS FOR CAPCITY HIGH(c2);
```

Figure 9. A create table command that indicates different compression techniques for different columns

Oracle Compression Advisor

Oracle Compression Advisor (`DBMS_COMPRESSION`) has been enhanced to support in-memory compression. The advisor provides an estimate of the compression ratio that can be realized through the use of `MEMCOMPRESS`. This estimate is based on analysis of a sample of the table data and provides a good estimate of the actual results obtained once the table is populated into the IM column store. As the advisor actually applies the new `MEMCOMPRESS` algorithms to the data it can only be run in an Oracle Database 12.1.0.2 (or later) environment.

```

DECLARE
    l_blkcnt_cmp          PLS_INTEGER;
    l_blkcnt_uncmp       PLS_INTEGER;
    l_row_cmp            PLS_INTEGER;
    l_row_uncmp          PLS_INTEGER;
    cmp_ratio            PLS_INTEGER;
    l_comptype_str       VARCHAR2(100);
    comp_ratio_allrows   NUMBER := -1;
BEGIN
    dbms_compression.Get_compression_ratio (
        -- Input parameters
        scratchtbsname => 'TS_DATA',
        ownname         => 'SSB',
        objname         => 'LINEORDER',
        subobjname      => NULL,
        comptype        => dbms_compression.comp_inmemory_query_low,
        -- Output parameter
        blkcnt_cmp      => l_blkcnt_cmp,
        blkcnt_uncmp    => l_blkcnt_uncmp,
        row_cmp         => l_row_cmp,
        row_uncmp       => l_row_uncmp,
        cmp_ratio       => cmp_ratio,
        comptype_str    => l_comptype_str,
        subset_numrows => dbms_compression.comp_ratio_allrows);
    dbms_output.Put_line('The IM compression ratio is '|| cmp_ratio);

    dbms_output.Put_line('Size in-mem 1 byte for every '|| cmp_ratio ||
'bytes on disk');
);
END;
/

```

Figure 10. Using the Oracle Compression Advisor (DBMS_COMPRESSION) to determine the size of the MY_SALES table in memory

Note when you set the comptype to an of the MEMCOMPRESS types the blkcnt_cmp output value is always set to 0 as there are on data blocks in the IM column store.

Also changing the compression clause of columns with an ALTER TABLE statement results in a repopulation of any existing data in the IM column store.

In-Memory Scans

Analytic queries typically reference only a small subset of the columns in a table. Oracle Database In-Memory accesses only the columns needed by a query, and applies any WHERE clause filter predicates to these columns directly without having to decompress them first. This greatly reduces the amount of data that needs to be accessed and processed.

In-Memory Storage Index

A further reduction in the amount of data accessed is possible due to the In-Memory Storage Indexes that are automatically created and maintained on each of the columns in the IM column store. Storage Indexes allow data pruning to occur based on the filter predicates supplied in a SQL statement. An In-Memory Storage Index keeps track of minimum and maximum values for each column in an IMCU. When a query specifies a WHERE clause predicate, the In-Memory Storage Index on the referenced column is examined to determine if any entries with the specified column value exist in each IMCU by comparing the specified value(s) to the minimum and maximum values maintained in the Storage Index. If the column value is outside the minimum and maximum range for an IMCU, the scan of that IMCU is avoided.

For equality, in-list, and some range predicates an additional level of data pruning is possible via the metadata dictionary created for each IMCU when dictionary-based compression is used. The metadata dictionary contains a list of the distinct values for each column within that IMCU. Thus dictionary based pruning allows Oracle Database to determine if the value being searched for actually exists within an IMCU, ensuring only the necessary IMCUs are scanned.

SIMD Vector Processing

For the data that does need to be scanned in the IM column store, Database In-Memory uses SIMD vector processing (Single Instruction processing Multiple Data values). Instead of evaluating each entry in the column one at a time, SIMD vector processing allows a set of column values to be evaluated together in a single CPU instruction.

The columnar format used in the IM column store has been specifically designed to maximize the number of column entries that can be loaded into the vector registers on the CPU and evaluated in a single CPU instruction. SIMD vector processing enables the Oracle Database In-Memory to scan billion of rows per second.

For example, let's use the `SALES` table in the `SH` sample schema (see Figure 11), and let's assume we are asked to find the total number of sales orders that used the `PROMO_ID` value of 9999. The `SALES` table has been fully populated into the IM column store. The query begins by scanning just the `PROMO_ID` column of the `SALES` table. The first 8 values from the `PROMO_ID` column are loaded into the SIMD register on the CPU and compared with 9999 in a single CPU instruction (the number of values loaded will vary based on datatype & memory compression used). The number of entries that match 9999 is recorded, then the entries are discarded and another 8 entries are loaded into the register for evaluation. And so on until all of the entries in the `PROMO_ID` column have been evaluated.

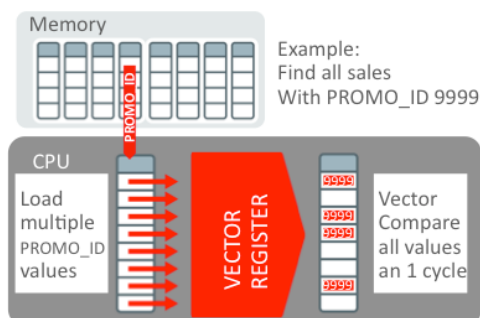


Figure 11. Using SIMD vector processing enables the scanning of billions of rows per second

To determine if a SQL statement is scanning data in the IM column examine the execution plan.

Id	Operation	Name
0	SELECT STATEMENT	
1	SORT AGGREGATE	
2	PARTITION RANGE ALL	
* 3	TABLE ACCESS INMEMORY FULL	SALES

Figure 12. New IN MEMORY keyword in the execution plan indicates operations that are candidates for In-Memory

You will notice that the execution plan shows a new set of keywords “IN MEMORY”. These keywords indicate that the LINEORDER table has been marked for IN MEMORY and Oracle Database **may** use the column store in this query.

In-Memory Joins

SQL statements that join multiple tables can also be processed very efficiently in the IM column store as they can take advantage of **Bloom Filters**. A Bloom filter transforms a join into a filter that can be applied as part of the scan of the larger table. Bloom filters were originally introduced in Oracle Database 10g to enhance hash join performance and are not specific to Oracle Database In-Memory. However, they are very efficiently applied to column format data via SIMD vector processing.

When two tables are joined via a hash join, the first table (typically the smaller table) is scanned and the rows that satisfy the WHERE clause predicates (for that table) are used to create an in-memory hash table (stored in Process Global Area - PGA). During the hash table creation, a bit vector or Bloom filter is also created based on the join column. The bit vector is then sent as an additional predicate to the scan of the second table. After the WHERE clause predicates have been applied to the second table scan, the resulting rows will have their join column hashed and it will be compared to values in the bit vector. If a match is found in the bit vector that row will be sent to the hash join. If no match is found then the row will be discarded

It's easy to identify Bloom filters in the execution plan. They will appear in two places, at creation time and again when it is applied. Let's take a simple two-table join between the DATE_DIM and LINEORDERS table as an example.

```
SELECT SUM(lo_extendedprice * lo_discount) revenue
FROM   lineorder l,
       date_dim d
WHERE  l.lo_orderdate = d.d_datekey
AND    l.lo_discount BETWEEN 2 AND 3
AND    d.d_date='December 24, 2013';
```

Figure 13. Simple two-table join that will benefit from Bloom filters in the In-Memory column store

Below is the plan for this query with the Bloom filter highlighted. The first step executed in this plan is actually line 4; an in-memory full table scan of the DATE_DIM table. The Bloom filter (:BF0000) is created immediately after the scan of the DATE_DIM table completes (line 3). The Bloom filter is then applied as part of the in-memory full table scan of the LINEORDER table (line 5 & 6).

Id	Operation	Name
0	SELECT STATEMENT	
1	SORT AGGREGATE	
* 2	HASH JOIN	
3	JOIN FILTER CREATE	:BF0000
* 4	TABLE ACCESS INMEMORY FULL	DATE_DIM
5	JOIN FILTER USE	:BF0000
* 6	TABLE ACCESS INMEMORY FULL	LINEORDER

Figure 14. Creation and use of a Bloom filter in a two-table join between the DATE_DIM and LINEORDER tables

It is possible to see what join condition was used to build the Bloom filter by looking at the predicate information under the plan. Look for 'SYS_OP_BLOOM_FILTER' in the filter predicates. You may be wondering why a HASH JOIN appears in the plan (line 2) if the join was converted to a Bloom filter. The HASH JOIN is there because a Bloom filter has the potential to return a false positive. The HASH JOIN confirms that all of the rows returned from the scan of the LINEORDER table are true matches for the join condition. Typically this consumes very little work.

What happens for a more complex query where there are multiple tables being joined? This is where Oracle's 30+ years of database innovation kicks in. By seamlessly building the IM column store into Oracle Database we can take advantage of all of the optimizations that have been added to the database since the first release. Using a series of optimizer transformations, multiple table joins can be rewritten to allow multiple Bloom filters to be created and used as part of the scan of the large table or fact table.

Note: from Oracle Database 12.1.0.2 onward, Bloom filters can be used on serial queries when executed against a table that is populated into the IM column store. Not all of the tables in the query need to be populated into the IM column store in order to create and use Bloom filters.

In-Memory Aggregation

Analytic style queries often require more than just simple filters and joins. They require complex aggregations and summaries. A new optimizer transformation, called *Vector Group By*, has been introduced with Oracle Database 12.1.0.2 to ensure more complex analytic queries can be processed using new CPU-efficient algorithms.

The Vector Group By transformation is a two-part process not dissimilar to that of star transformation. Let's take the following business query as an example: Find the total sales of footwear products in outlet stores.

Phase 1

1. The query will begin by scanning the two dimension tables (smaller tables) STORES and PRODUCTS (lines 5 & 10 in the plan below).
2. A new data structure called a *Key Vector* is created based on the results of each of these scans (lines 4, 9, & 13 in the plan below). A key vector is similar to a Bloom filter as it allows the join predicates to be applied as additional filter predicates during the scan of the SALES table (largest table). Unlike a Bloom filter a key vector will not return a false positive.

- The key vectors are also used to create an additional structure called an In-Memory Accumulator. The accumulator is a multi-dimensional array built in the PGA that enables Oracle Database to conduct the aggregation or `GROUP BY` during the scan of the `SALES` table instead of having to do it afterwards.
- At the end of the first phase temporary tables are created to hold the payload columns (columns referenced in the `SELECT` list) from the smaller dimension table (lines 2, 6, & 11 in the plan below). Note this step is not depicted in Figure 15 below.

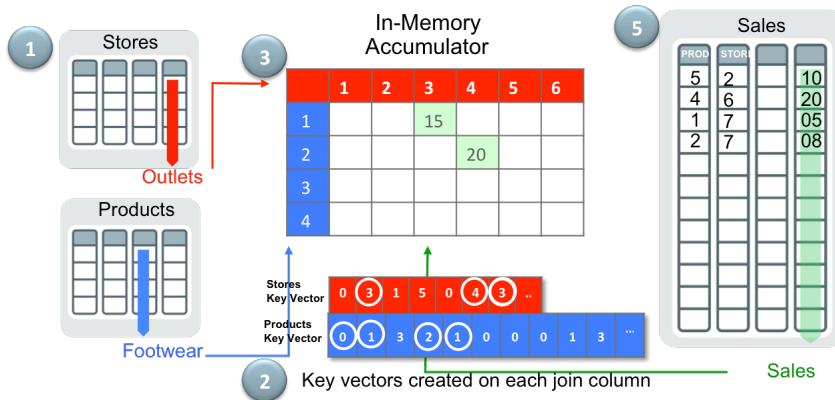


Figure 15. In-Memory aggregation example - Find the total sales of footwear in our outlet stores

Phase 2

- The second part of the execution plan begins with the scan of the `SALES` table and the application of the key vectors (line 24-29 in the plan below). For each entry in the `SALES` table that matches the join conditions (is an outlet store and is a footwear product), the corresponding sales amount will be added to the appropriate cell in the In-Memory Accumulator. If a value already exists in that cell, the two values will be added together and the resulting value will be put in the cell.
- Finally the results of the large table scan are then joined back to the temporary tables created as part of the scan of the dimension tables (lines 16, 18, & 19). Remember these temporary tables contain only the payload columns. Note this step is not depicted in Figure 15 above.

The combination of these two phases dramatically improves the efficiency of a multiple table join with complex aggregations.

Id	Operation	Name	
0	SELECT STATEMENT		
1	TEMP TABLE TRANSFORMATION		
2	LOAD AS SELECT	SYS_TEMP_0FD9D6635_4F9FC7	
3	VECTOR GROUP BY		
4	KEY VECTOR CREATE BUFFERED	:KV0000	
5	TABLE ACCESS INMEMORY FULL	STORES	PHASE 1
6	LOAD AS SELECT	SYS_TEMP_0FD9D6636_4F9FC7	
7	VECTOR GROUP BY		
8	HASH GROUP BY		
9	KEY VECTOR CREATE BUFFERED	:KV0001	
10	TABLE ACCESS INMEMORY FULL	PRODUCTS	
11	SORT GROUP BY		
12	HASH JOIN		
13	MERGE JOIN CARTESIAN		
14	TABLE ACCESS FULL	SYS_TEMP_0FD9D6636_4F9FC7	
15	BUFFER SORT		
16	TABLE ACCESS FULL	SYS_TEMP_0FD9D6635_4F9FC7	
17	VIEW	VW_VT_80F21617	
18	VECTOR GROUP BY		
19	HASH GROUP BY		
20	KEY VECTOR USE	:KV0000	PHASE 2
21	KEY VECTOR USE	:KV0001	
22	TABLE ACCESS INMEMORY FULL	SALES	

Figure 16. Execution plan for query that benefits from In-Memory aggregation

The `VECTOR GROUP BY` transformation is a cost based transformation, which means the optimizer will cost the execution plan with and without the transformation and pick the one with the lowest cost. For example, the `VECTOR GROUP BY` transformation may be selected in the following scenarios:

- The join columns between the tables contain "mostly" unique keys or numeric keys
- The fact table (largest table in the query) is at least 10X larger than the other tables
- The tables are populated into the IM column store

The `VECTOR GROUP BY` transformation is unlikely to be chosen in the following scenarios:

- Joins are performed between two or more very large tables
- The dimension tables contain more than 2 billion rows
- The system does not have sufficient memory resources

DML And The In-Memory Column Store

It's clear that the IM column store can dramatically improve the performance of all types of queries but very few database environments are read only. For the IM column store to be truly effective in modern database environments it has to be able to handle both bulk data loads **AND** online transaction processing.

Bulk Data Loads

Bulk data loads occur most commonly in Data Warehouse environments and are typically conducted as a direct path load. A direct path load parses the input data, converts the data for each input field to its corresponding Oracle data type, and then builds a column array structure for the data. These column array structures are used to format Oracle data blocks and build index keys. The newly formatted database blocks are then written directly to the database, bypassing the standard SQL processing engine and the database buffer cache.

A direct path load operation is an all or nothing operation. This means that the operation is not committed until all of the data has been loaded. Should something go wrong in the middle of the operation, the entire operation will be aborted. To meet this strict criterion, a direct path loads inserts data into database blocks that are created above the segment high water mark (maximum number of database blocks used so far by an object or segment). Once the direct path load is committed, the high water mark is moved to encompass the newly created blocks into the segment and the blocks will be made visible to other SQL operations on the same table. Up until this point the IM column store is not aware that any data change occurred on the segment.

Once the operation has been committed, the IM column store is instantly aware it does not have all of the data populated for the object. The size of the missing data will be visible in the `BYTES_NOT_POPULATED` column of the `v$IM_SEGMENTS` view (see monitoring section). If the object has a `PRIORITY` specified on it then the newly added data will be automatically populated into the IM column store. Otherwise the next time the object is queried, the background worker processes will be triggered to begin populating the missing data, assuming there is free space in the IM column store.

Partition Exchange Loads

It is strongly recommended that the larger tables or fact tables in a data warehouse be partitioned. One of the benefits of partitioning is the ability to load data quickly and easily with minimal impact on users by using the exchange partition command. The exchange partition command allows the data in a non-partitioned table to be swapped into a particular partition in a partitioned table. The command does not physically move data; instead it updates the data dictionary to exchange a pointer from the partition to the table and vice versa. Because there is no physical movement of data, an exchange does not generate redo and undo, making it a sub-second operation and far less likely to impact performance than any traditional data-movement approaches such as `INSERT`.

As with a direct path operation the IM column is not aware of a partition exchange load until the operation has been completed. At that point the data in the temporary table is now part of the partitioned table. If the temporary table had the `INMEMORY` attribute set and all of its data has been populated into the IM column store, nothing else will happen. The data that was in the temporary table will simply be accessed via the IM column store along with the rest of the data in the partitioned table the next time it is scanned.

However, if the temporary table did not have the `INMEMORY` attribute set, then all subsequent accesses to the data in the newly exchanged partition will be done via the buffer cache. Remember the `INMEMORY` attribute is a physical attribute of an object. If you wish the partition to have that attribute after the exchange it must be specified on the temporary table before the exchange takes place. Specifying the attribute on the empty partition is not sufficient.

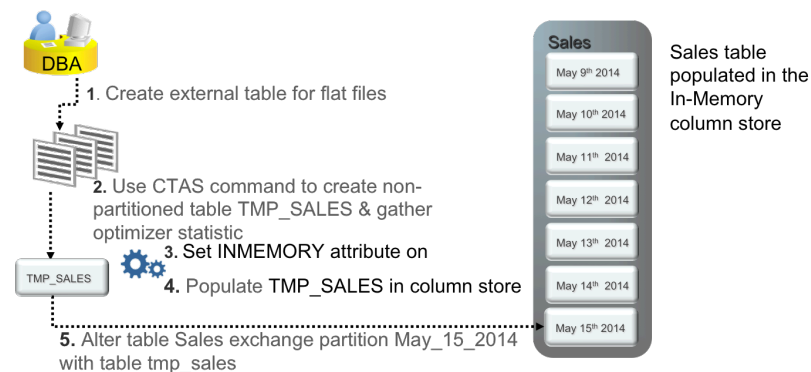


Figure 17. Five steps necessary to complete a partition exchange load on an `INMEMORY` table

Transaction Processing

Single row data change operations (DML) execute via the buffer cache (OLTP style changes), just as they do without Database In-Memory enabled. If the object in which the DML operations occurs is populated in the IM column store, then the changes are reflected in the IM column store as they occur. The buffer cache and the column store are kept transactionally consistent via the In-Memory Transaction Manager. All logging is done on the base table just as it was before, no logging is needed for the In-Memory Column store.

For each IMCU in the IM column store, a transaction journal is automatically created and maintained (see figure 18). When a DML statement changes a row in an object that is populated into the IM column store, the corresponding entries for that row is marked stale in the IMCU and a copy of the new version of the row is added to the in-memory transaction journal. The original entries in the IMCU are not immediately replaced in order to provide read consistency and maintain data compression. Any transaction executing against this object in the IM column store that started before the DML occurred, needs to see the original version of the entries. Read consistency in the IM column store is managed via System Change Numbers (SCNs) just as it is without Database In-Memory enabled.

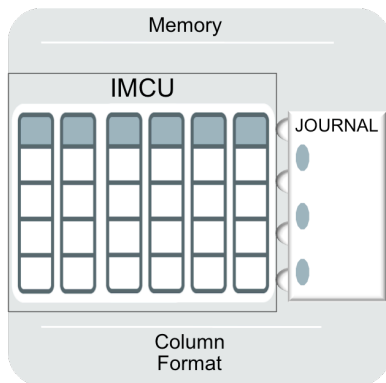


Figure 18. Each IMCU in the IM column store contains a subset of rows from an object & a transaction journal

When a query with a newer SCN is executed against the object, it will read all of the entries for the columns in the IMCU except the stale entries. The stale entries will be retrieved either from the transaction journal or from the base table (buffer cache).

Repopulation

The more stale entries there are in an IMCU, the slower the scan of the IMCU will become. Therefore Oracle Database will repopulate an IMCU when the number of stale entries in an IMCU reaches a staleness threshold. The staleness threshold is determined by heuristics that take into account the frequency of IMCU access and the number of stale rows in the IMCU. Repopulation is more frequent for IMCUs that are accessed frequently or have a higher percentage of stale rows. The repopulation of an IMCU is an online operation executed by the background worker processes. The data is available at all times and any changes that occur to rows in the IMCU during repopulation are automatically recorded.

In addition to the standard repopulation algorithm, there is another algorithm that attempts to clean all stale entries using a low priority background process. The IMCO (In-Memory Coordinator) background process may also instigate *trickle repopulation* for any IMCU in the IM column store that has some stale entries but does not currently meet the staleness threshold. Trickle repopulate is a constant background activity.

The IMCO wakes up every two minutes and checks to see if any population tasks need to be completed. For example, the `INMEMORY` attribute has just been specified with a `PRIORITY` sub-clause on a new object. The IMCO will also check to see if there are any IMCUs with stale entries in the IM column store. If it finds some it will trigger the worker processes to repopulate them. The number of IMCUs repopulated via trickle repopulate in a given 2 minute window is limited by the new initialization parameter `INMEMORY_TRICKLE_REPOPULATE_SERVERS_PERCENT`. This parameter controls the maximum percentage of time that worker processes can participate in trickle repopulation activities. The more worker processes that participate, the more IMCUs that can be trickle repopulated, however the more worker processes that participate the higher the CPU consumption. You can disable trickle repopulation altogether by setting `INMEMORY_TRICKLE_REPOPULATE_SERVERS_PERCENT` to 0.

Overhead of Keeping IM Column Store Transactionally Consistent

The overhead of keeping the IM column store transactionally consistent will vary by application based on a number of factors including: the rate of change, the in-memory compression level chosen for a table, the location of the changed rows, and the type of operations being performed. Tables with higher compression levels will incur more overhead than tables with lower compression levels.

Changed rows that are co-located in the same block will incur less overhead than changed rows that are spread randomly across a table. Examples of changed rows that are co-located in the same blocks are newly inserted rows since the database will usually group these together. Another example is data that is loaded using a direct path load operation.

For tables that have a high rate of DML, `MEMCOMPRESS FOR DML` is recommended, and, where possible, it is also recommended to use partitioning to localize changes within the table. For example, range partitioning can be used to localize data in a table by date so most changes will be confined to data stored in the most recent partition. Date range partitioning also provides many other manageability and performance advantages.

The In-Memory Column Store On RAC

Each node in a RAC environment has its own IM column store. It is highly recommended that the IM column stores be equally sized on each RAC node. Any RAC node that does not require an IM column store should have the `INMEMORY_SIZE` parameter set to 0. By default all objects populated into memory will be distributed across all of the IM column stores in the cluster. It is also possible to have the same objects appear in the IM column store on every node (Engineered Systems only). The distribution of objects across the IM column stores in a cluster is controlled by two additional sub-clauses to the `INMEMORY` attribute: `DISTRIBUTE` and `DUPLICATE`.

In a RAC environment, an object that only has the `INMEMORY` attribute specified on it will be distributed across all of the IM column stores in the cluster, effectively makes the IM column store a share-nothing architecture in a RAC environment. How an object is distributed across the cluster is controlled by the `DISTRIBUTE` sub-clause. By default, Oracle decides the best way to distribute the object across the cluster given the type of partitioning used (if any). Alternatively, you can specify `DISTRIBUTE BY ROWID RANGE` to distribute by rowid range, `DISTRIBUTE BY PARTITION` to distribute partitions to different nodes, or `DISTRIBUTE BY SUBPARTITION` to distribute sub-partitions to different nodes.

```
ALTER TABLE lineorder INMEMORY DISTRIBUTE BY PARTITION;
```

Figure 19. This command distributes the lineorder tables across the IM column stores in the cluster by partition.

`DISTRIBUTE BY PARTITION` or `SUBPARTITION` is recommended if the tables are partitioned or sub-partitioned by `HASH` and a partition-wise join plan is expected. This will allow each partition join to be co-located

within a single node. `DISTRIBUTE BY ROWID RANGE` can be used for non-partitioned tables or for partitioned table where `DISTRIBUTE BY PARTITION` would lead to a data skew.

If the object is very small (consists of just 1 IMCU), it will be populated into the IM column store on just one node in the cluster.

Since data populated in-memory in a RAC environment is affinitized to a specific RAC node, parallel server processes must be employed to execute a query on each RAC node against the piece of the object that resides in that node's IM column store. The query coordinator aggregates the results from each of the parallel server processes together before returning them to the end user's session. In order to ensure the parallel server processes are distributed appropriately across the RAC cluster, you **must** use Automatic Degree of Parallelism (AutoDOP), so the query coordinator is IMCU instance location aware.

If AutoDOP is not used, and the parallel degree is specified manually (via a hint or parallel attribute on a table), its possible not all of the data will be read from the IM column store as the parallel server processes may not be started on the appropriate nodes and we do not ship IMCUs across a RAC cluster.

If a DML statement is issued against an object on the node where the object or that piece of the object resides in the IM column store, the corresponding row in the IM column store is marked stale and a copy of the new row is added to the transaction journal within that IMCU. However if the DML statement is issued on a different node, then cache fusion will be used to keep the IM column store transactionally consistent by marking the column entries in the database block with the changed row stale in the corresponding IM column store on the remote node.

In-Memory Fault Tolerance

Given the shared nothing architecture of the IM column store in a RAC environment, some performance sensitive applications may require a fault tolerant solution. On an Engineered System it is possible to mirror the data populated into the IM column store by specifying the `DUPLICATE` sub-clause of the `INMEMORY` attribute. This means that each IMCU populated into the IM column store will have a mirrored copy placed on one of the other nodes in the RAC cluster. Mirroring the IMCUs provides in-memory fault tolerance as it ensures data is still accessible via the IM column store even if a node goes down. It also improves performance, as queries can access both the primary and the backup copy of the IMCU at any time.

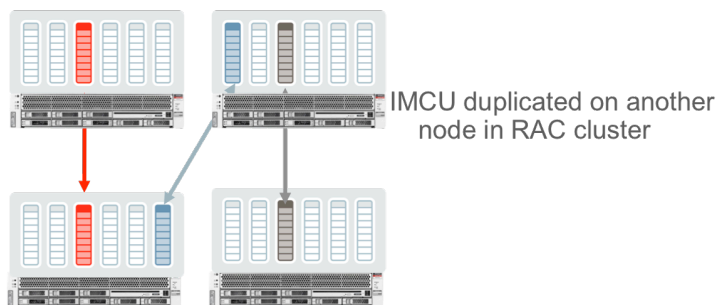


Figure 20. Objects in the IM column store on Engineered Systems can be mirrored to improve fault tolerance

Should a RAC node go down and remain down for some time, the only impact will be the re-mirroring of the primary IMCUs located on that node. Only if a second node were to go down and remain down for some time would the data have to be redistributed.

If additional fault tolerance is desired, it is possible to populate an object into the IM column store on each node in the cluster by specifying the `DUPLICATE ALL` sub-clause of the `INMEMORY` attribute. This will provide the

highest level of redundancy and provide linear scalability, as queries will be able to execute completely within a single node.

```
ALTER TABLE lineorder INMEMORY DUPLICATE ALL;
```

Figure 21. This command ensures each IMCU of the lineorder table will appear in all IM column store in the cluster

The `DUPLICATE ALL` option may also be useful to co-locate joins between large distributed fact tables and smaller dimension tables. By specifying the `DUPLICATE ALL` option on the smaller dimension tables a full copy of these tables will be populated into the IM column store on each node.

The `DUPLICATE` sub-clause is only applicable on an Oracle Engineered System and will be ignored if specified elsewhere.

If a RAC node should go down on a non-Engineered System, the data populated into the IM column store on that node will no longer be available in-memory on the cluster. Queries issued against the missing pieces of the objects will not fail. Instead they will access the data either from the buffer cache or storage, which will impact the performance of these queries. Should the node remain down for some time, the objects or pieces of the objects that resided in the IM column store on that node will be populated on the remaining nodes in the cluster (assuming there is available space). In order to minimize the impact on performance due to a downed RAC node, it is recommended that some space be left free in the IM column store on each node in the cluster.

Note that data is not immediately redistributed to other nodes of the cluster immediately upon a node or instance failure because it is very likely that the node or instance will be quickly brought back into service. If data was immediately redistributed, the redistribution process would add extra workload to the system that then would be undone when the node or instance returns to service. Therefore the system will wait for a few tens of minutes before initiating data redistribution. Waiting allows the node or instance time to rejoin the cluster.

When the node rejoins the cluster data will be redistributed to the newly joined node. The distribution is done on an IMCU basis and the objects are fully accessible during this process.

The In-Memory Column Store in a Multitenant Environment

Oracle Multitenant¹ is a new database consolidation model in which multiple Pluggable Databases (PDBs) are consolidated within a Container Database (CDB). While keeping many of the isolation aspects of single databases, it allows PDBs to share the system global area (SGA) and background processes of a common CDB. Therefore, PDBs also share a single IM column store.

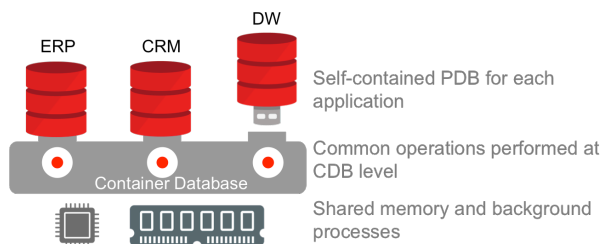


Figure 22. Three PDBs in a single Oracle Database 12c Container Database

¹ More information on Oracle Multitenant can be found in the white paper [Oracle Multitenant](#)

The total size of the IM column store is controlled by the `INMEMORY_SIZE` parameter setting in the CDB. Each PDB specifies how much of the shared IM column store it can use by setting the `INMEMORY_SIZE` parameter. Not all PDBs in a given CDB need to use the In-Memory column store. Some PDBs can have the `INMEMORY_SIZE` parameter set to 0, which means they won't use the In-Memory column store at all.

It is not necessary for the sum of the PDBs' `INMEMORY_SIZE` parameters to be less than or equal to the size of the `INMEMORY_SIZE` parameter on the CDB. It is possible for the PDBs to over subscribe to the IM column store. Over subscription is allowed to ensure valuable space in the IM column store is not wasted should one of the pluggable databases be shutdown or unplugged. Since the `INMEMORY_SIZE` parameter is static (requires a database instance restart for changes to be reflected) it is better to allow the PDBs to over subscribe, so all of the space in the IM column store can be used.

However, it is possible for one PDB to starve another PDB of space in the IM column store due to this over subscription. If you don't expect any PDBs to be shut down for extended periods or any of them to be unplugged it is recommended that you don't over subscribe.

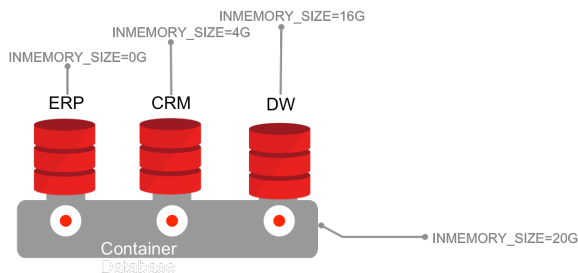


Figure 23. PDBs specify how much of the shared IM column store they can use by setting `INMEMORY_SIZE` parameter

Each pluggable database (PDB) is a full Oracle database in its own right, so each PDB will have its own priority list. When a PDB starts up the objects on its priority list will be populated into the In-Memory column store in order assuming there is available space.

Controlling the use of Oracle Database In-Memory

There are several new initialization parameters and optimizer hints that allow you to control when and how the IM column store will be used. This section describes them all and provides guidance on which ones are core and which are optional.

Core Initialization Parameters

Six new initialization parameters with the `INMEMORY` prefix have been introduced to directly control the different aspects of the new in-memory functionality. There is also a new optimizer parameter that can have an effect on whether queries use the IM column store or not.

```
SQL> Show Parameter INMEMORY
```

NAME	TYPE	VALUE
inmemory_clause_default	string	
inmemory_force	string	DEFAULT
inmemory_max_populate_servers	integer	4
inmemory_query	string	ENABLE
Inmemory_size	big integer	2G
inmemory_trickle_repopulate_servers_precent	integer	1

Figure 24. New In-Memory Initialization parameters

INMEMORY_SIZE

As described earlier in this document, the `INMEMORY_SIZE` parameter controls the amount of memory allocated to the IM column store. The default size is 0 bytes. This parameter is only modifiable at the system level and will require a database restart to take effect. The minimum size required for the `INMEMORY_SIZE` parameter is 100 MB.

INMEMORY_QUERY

The Oracle Optimizer is aware of the objects populated in the IM column store and will automatically direct any queries it believes will benefit from the in-memory column format to the IM column store. Setting `INMEMORY_QUERY` to `DISABLE` either at the session or system level disables the use of the IM column store completely. It will blind the Optimizer to what is in the IM column store and it will prevent the execution layer from scanning and filtering data in the IM column store. The default value is `ENABLE`.

INMEMORY_MAX_POPULATE_SERVERS

The maximum number of worker processes that can be started is controlled by the `INMEMORY_MAX_POPULATE_SERVERS`, which is set to $0.5 \times \text{CPU_COUNT}$ by default. Reducing the number of worker processes will reduce the CPU resource consumed during population but it will likely extend the amount of time it takes to do the population of the IM column store.

Additional Initialization Parameters

INMEMORY_CLAUSE_DEFAULT

The `INMEMORY_CLAUSE_DEFAULT` parameter allows you to specify a default mode for in-memory tables by specifying a valid set of values for all of the `INMEMORY` sub-clauses not explicitly specified in the syntax. The default value is an empty string, which means that only explicitly specified tables are populated into the IM column store.

```
ALTER SYSTEM SET inmemory_clause_default= 'INMEMORY PRIORITY LOW' ;
```

Figure 25. Using the `INMEMORY_CLAUSE_DEFAULT` parameter to mark all new tables as candidates for the IM column store

The parameter value is parsed in the same way as the `INMEMORY` clause, with the same defaults if one of the sub-clauses is not specified. Any table explicitly specified for in-memory will inherit any unspecified values from this parameter.

INMEMORY_TRICKLE_REPOPULATE_SERVERS_PERCENT

This parameter controls the maximum percentage of time that worker processes can perform trickle repopulation. The value of this parameter is a percentage of the `INMEMORY_MAX_POPULATE_SERVERS` parameter. Setting this parameter to 0 disables trickle repopulation; the default is 1 meaning that the worker processes will spend one percent of their time performing trickle repopulate.

INMEMORY_FORCE

By default any object with the `INMEMORY` attribute specified on it is a candidate to be populated into the IM Column Store. However, if `INMEMORY_FORCE` is set to `OFF`, then even if the in-memory area is configured, no tables are put in memory. The default value is `DEFAULT`.

OPTIMIZER_INMEMORY_AWARE

As mentioned above, the optimizer is aware of the IM column store and uses in-memory specific costs when it costs the alternative in-memory plans for a SQL statement. It is possible to disable all of the in-memory enhancements made to the optimizer's cost model by setting the `OPTIMIZER_INMEMORY_AWARE` parameter to `FALSE`. Please note that even with the Optimizer in-memory enhancements disabled, you may still get an In-Memory plan.

Optimizer Hints

The different aspects of In-Memory - in-memory scans, joins and aggregations - can be controlled at a statement or a statement block level via the use of optimizer hints. As with most optimizer hints, the corresponding negative hint for each of the hints described below is preceded by the word 'NO_'. Remember that an optimizer hint is a directive that will be followed when applicable.

INMEMORY Hint

The only thing the `INMEMORY` hint does is enables the IM column store to be used when the `INMEMORY_QUERY` parameter is set to `DISABLE`.

It won't force a table or partition without the `INMEMORY` attribute to be populated into the IM column store. If you specify the `INMEMORY` hint in a SQL statement where none of the tables referenced in the statement are populated into memory, the hint will be treated as a comment since its not applicable to this SQL statement.

Nor will the `INMEMORY` hint force a full table scan via the IM column store to be chosen, if the default plan (lowest cost plan) is an index access plan. You will need to specify the `FULL` hint to see that plan change take effect.

The `NO_INMEMORY` hint does the same thing in reverse. It will prevent the access of an object from the IM column store; even if the object is full populated into the column store and the plan with the lowest cost is a full table scan.

In-Memory Scan

As statement above if you wish to force an In-Memory full table scan you will need to use the `FULL` hint to change the access method for an object (table, or (sub)partition).

The `(NO_) INMEMORY_PRUNING` hint can also influence the performance on an In-Memory scan as it controls the use of In-Memory storage indexes. By default every query executed against the IM column store can take advantage of the In-Memory storage indexes, which enable data pruning to occur based on the filter predicates supplied in a SQL statement. As with most hints, the `INMEMORY_PRUNING` hint was introduced to help test the new functionality. In other words the hint was originally introduced to disable the IM storage indexes.

In-Memory Joins

The use of a Bloom filter to convert a join into a filter is a cost-based decision. If the Optimizer doesn't choose a Bloom filter, it is possible to force it by using the `PX_JOIN_FILTER` hint.

In-Memory Aggregation

The new in-memory aggregation feature (`VECTOR GROUP BY`) is a cost-based query transformation, which means it's possible to force the transformation to occur even when the Optimizer does not consider it to be the cheapest execution plan. A `VECTOR GROUP BY` plan can be forced by specifying the `VECTOR_TRANSFORM` hint.

Monitoring and Managing Oracle Database In-Memory

Monitoring What Objects Are In The In-Memory Column Store

There are two new v\$ views, `v$IM_SEGMENTS` and `v$IM_USER_SEGMENTS` that indicate what objects are currently populated in the IM column store.

```
SQL> desc v$im_segments
```

Name	Null?	Type
OWNER		VARCHAR2(128)
SEGMENT_NAME		VARCHAR2(128)
PARTITION_NAME		VARCHAR2(128)
SEGMENT_TYPE		VARCHAR2(18)
TABLESPACE_NAME		VARCHAR2(30)
INMEMORY_SIZE		NUMBER
BYTES		NUMBER
BYTES_NOT_POPULATED		NUMBER
POPULATE_STATUS		VARCHAR2(9)
INMEMORY_PRIORITY		VARCHAR2(8)
INMEMORY_DISTRIBUTE		VARCHAR2(15)
INMEMORY_DUPLICATE		VARCHAR2(13)
INMEMORY_COMPRESSION		VARCHAR2(17)
CON_ID		NUMBER

Figure 26. New v\$IM_SEGMENTS view

These views not only show which objects are populated in the IM column, they also indicate how the objects are distributed across a RAC cluster and whether the entire object has been populated (`BYTES_NOT_POPULATED`). It is also possible to use this view to determine the compression ratio achieved for each object populated in the IM column store, assuming the objects were not compressed on disk.

```
SELECT v.owner, v.segment_name,
       v.bytes          orig_size,
       v.inmemory_size  in_mem_size,
       v.bytes / v.inmemory_size comp_ratio
FROM   v$im_segments v;
```

Figure 27. Determining the compression ratio achieved for the objects populated into the IM column store

Another new view, `v$IM_COLUMN_LEVEL`, contains details on the columns populated into the column store, as not all columns in a table need to be populated into the column store.

```
SQL> SELECT table_name, column_name, inmemory_compression from v$im_column_level;
```

TABLE_NAME	COLUMN_NAME	INMEMORY_COMPRESSION
SALES	PROD_ID	NO_INMEMORY
SALES	CUST_ID	DEFAULT
SALES	TIME_ID	DEFAULT
SALES	CHANNEL_ID	DEFAULT
SALES	PROMO_ID	DEFAULT
SALES	QUANTITY_SOLD	DEFAULT
SALES	AMOUNT_SOLD	DEFAULT

Figure 28. The PROD_ID column was not populated into the IM column store

USER_TABLES

A new Boolean column called `INMEMORY` has been added to the `*_TABLES` dictionary tables to indicate which tables have the `INMEMORY` attribute specified on them.

```
SQL> Select table_name, inmemory From user_tables;
```

TABLE_NAME	INMEMORY
SALES	
COSTS	
SALES_TRANSACTIONS_EXT	DISABLED
TIMES	DISABLED
CHANNELS	ENABLED
PROMOTIONS	DISABLED
COUNTRIES	DISABLED
CUSTOMERS	ENABLED
PRODUCTS	ENABLED

Figure 29. New `INMEMORY` column added to `*_TABLES` to indicate which tables have `INMEMORY` attribute

In the example above you will notice that two of the tables – `COSTS` and `SALES` – don't have a value for the `INMEMORY` column. The `INMEMORY` attribute is a segment level attribute. Both `COSTS` and `SALES` are partitioned tables and are therefore logical objects. The `INMEMORY` attribute for these tables will be recorded at the partition or sub-partition level in `*_TAB_(SUB) PARTITIONS`.


Three additional columns – `INMEMORY_PRIORITY`, `INMEMORY_DISTRIBUTE`, and `INMEMORY_COMPRESSION` – have also been added to the `*_TABLES` views to indicate the current In-Memory attributes for each table.

Managing IM Column Store Population CPU consumption

The initial population of the IM column store is a CPU intensive operation, which can affect the performance of other workloads running concurrently. You can use Resource Manager² to control the CPU usage of IM column store population operations and change their priority as needed.

To do this, enable CPU Resource Manager by enabling one of the out-of-box resource plans, such as `default_plan`, or by creating your own resource plan. By default, in-memory population is run in the `ora$autotask` consumer group, except for on-demand population, which runs in the consumer group of the user that triggered the population. If the `ora$autotask` consumer group doesn't exist in the resource plan, then the population will run in `OTHER_GROUPS`. The other operations in `ora$autotask` include automated maintenance operations like gathering statistics and segment analysis.

² More information on using Oracle Database Resource Manager can be found in the white paper [Using Oracle Resource Manager](#)



The `SET_CONSUMER_GROUP_MAPPING` procedure can be used to change the consumer group for in-memory population.

```
BEGIN
    dbms_resource_manager.Set_consumer_group_mapping(
        attribute => 'ORACLE_FUNCTION',
        value     => 'INMEMORY',
        consumer_group => 'BATCH_GROUP');
END;
```

Figure 30. Changing the Resource Manager consumer group of the `INMEMORY` operation



Conclusion

Oracle Database In-Memory transparently accelerates analytic queries by orders of magnitude, enabling real-time business decisions. It dramatically accelerates data warehouses and mixed workload OLTP environments. The unique "dual-format" approach automatically maintains data in both the existing Oracle row format for OLTP operations, and in a new purely in-memory column format optimized for analytical processing. Both formats are simultaneously active and transactionally consistent. Embedding the column store into Oracle Database ensures it is fully compatible with ALL existing features, and requires absolutely no changes in the application layer. This means you can start taking full advantage of it on day one, regardless of the application



Oracle Corporation, World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065, USA

Worldwide Inquiries
Phone: +1.650.506.7000
Fax: +1.650.506.7200

Hardware and Software, Engineered to Work Together

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. This document is provided for information purposes only, and the contents hereof are subject to change without notice. This document is not warranted to be error-free, nor subject to any other warranties or conditions, whether expressed orally or implied in law, including implied warranties and conditions of merchantability or fitness for a particular purpose. We specifically disclaim any liability with respect to this document, and no contractual obligations are formed either directly or indirectly by this document. This document may not be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without our prior written permission.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group. 0415

White Paper Oracle Database In-Memory
July 2015
Author: Maria Colgan

CONNECT WITH US

