19c ORACLE Database

Oracle Database In-Memory with Oracle Database 19c

Technical Overview

ORACLE WHITE PAPER | FEBRUARY 2019

ORACLE

Disclaimer

Table of Contents

Executive Overview

Oracle Database In-Memory adds in-memory functionality to Oracle Database for transparently accelerating 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. 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 Oracle 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 database workloads makes Oracle Database In-Memory ideally suited for the Cloud and on-premises because it requires no additional changes to the application. Oracle Database In-Memory 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 striving to run analytics directly 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.

Oracle introduced Database In-Memory in Oracle Database Enterprise Edition with the first patch set (12.1.0.2) for Oracle Database 12c Release 1. Database In-Memory has been significantly enhanced in subsequent releases of Oracle Database with additional performance, scalability and manageability features.

With 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 columnar format, optimized for analytical processing. Oracle Database In-Memory also enables data marts and data warehouses to provide more ad-hoc analytics, giving end-users the ability to run multiple business-driven queries in the same time it previously took to run just one query.

Embedding the in-memory column format into the existing Oracle Database software ensures full compatibility with ALL existing features, and no changes in the application. This makes it an ideal analytics platform in the Cloud. Applications can be moved to the Cloud and seamlessly take advantage of the performance of Oracle Database In-Memory's ability to provide real-time analytics. 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 Oracle Database In-Memory can be integrated into existing operational systems and data warehouse environments to improve both performance and manageability.

Oracle Database In-Memory Overview

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 as efficient at processing DML, to insert or delete a single record in a column format all the columnar structures in the table must be changed. That could require one or more I/O operations per column. Database systems that support only one format 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 columnar format: a dual-format architecture.

Note that the dual-format architecture **does not** double memory requirements. The in-memory columnar format should be sized to accommodate the objects that must be stored in memory. This is different than the buffer cache which 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% additional memory overhead. This is a small price to pay for optimal performance at all times for all workloads.

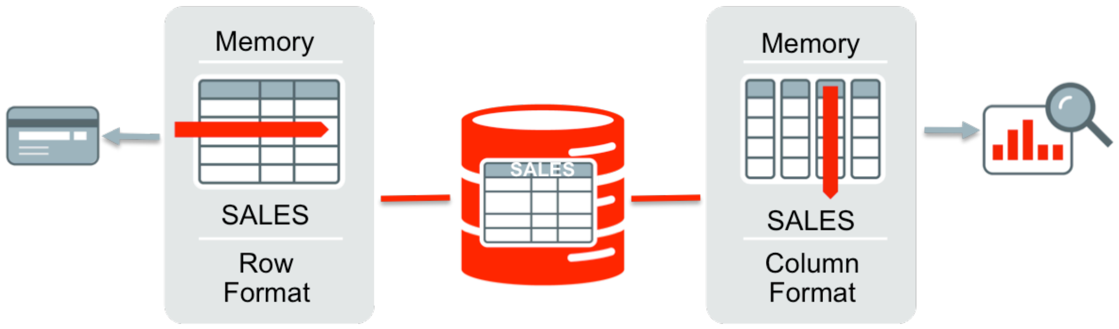


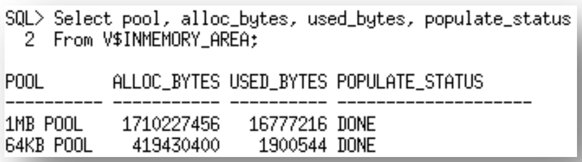
Figure 1. Oracle's unique dual-format architecture.

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 columnar format: It automatically routes analytic queries to the columnar 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 columnar 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 columnar format.

The In-Memory area is sub-divided into two pools: a 1MB pool used to store the actual columnar 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`

Dynamic Resizing and Automatic Memory Management

The size of the In-Memory area, within the SGA, is controlled by the initialization parameter `INMEMORY_SIZE` (default 0). The In-Memory area must have a minimum size of 100MB. The current size of the In-Memory area is visible in the view `V$SGA`. Starting in 12.2, it is possible to increase the size of the In-Memory area on the fly, by increasing the `INMEMORY_SIZE` parameter via an `ALTER SYSTEM` command, assuming there is spare memory within the SGA. The `INMEMORY_SIZE` parameter must be increased by 128MB or more in order for this change to take effect. It is not possible to shrink the size of the In-Memory area on the fly. A reduction in the size of the `INMEMORY_SIZE` parameter will not take effect until the database instance is restarted. It is important to note that the In-Memory area is not impacted or controlled by Oracle Automatic Memory Management (AMM).

Populating the In-Memory Column Store

Not all of the objects in an Oracle database need to be populated in the IM column store. This is an advantage over so-called "pure" in-memory databases that require the entire database to be memory-resident. With Oracle Database In-Memory, 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, partition, subpartition, 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 `INMEMORY` 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 is possible to populate just a subset of the partitions or subpartitions.

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* (e.g. ora_w001_orcl). The database is fully active and accessible while this occurs. With a pure in-memory database, the database cannot be accessed until all of the data is populated into memory, which blocks availability until the population is complete.

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 of 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 contain 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 row store.

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.

Populating Using the DBMS_INMEMORY.POPULATE_WAIT Function

In Oracle Database 19c a new POPULATE_WAIT function has been added to the DBMS_INMEMORY package. This function processes all INMEMORY objects with a PRIORITY setting greater than or equal to the PRIORITY specified as input to the function (the default is LOW). The function also accepts a population percentage and a timeout interval allowing applications to know when the IM column store has been populated. At database startup, or when changing the contents of the IM column store, applications can prevent user access until all application objects are populated. This can ensure stable performance and prevent sub-optimal query performance if not all of the application data has been populated. This can be particularly useful if no analytic indexes exist.

In-Memory Compression

In general, 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
NO MEMCOMPRESS	Data is populated without any compression
MEMCOMPRESS FOR DML	Minimal compression optimized for DML performance

MEMCOMPRESS FOR QUERY LOW	Optimized for query performance (default)
MEMCOMPRESS FOR QUERY HIGH	Optimized for query performance as well as space saving
MEMCOMPRESS FOR CAPACITY LOW	Balanced with a greater bias towards space saving
MEMCOMPRESS FOR CAPACITY HIGH	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 CAPACITY 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 for Database In-Memory in an Oracle Database 12c 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 compressed size and compression ratio of the LINEORDER table in memory

Note: When you set the comptype input parameter to any of the MEMCOMPRESS types the blkcnt_cmp output parameter value is always set to 0 as there are no 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 FastStart

In-Memory population is a CPU bound operation, involving reformatting data into a columnar format and compressing that data before placing it in memory. With In-Memory FastStart (IM FastStart), it is possible to checkpoint IMCUs to disk to relieve the CPU overhead of population, at the cost of additional disk space and IO bandwidth.

When IM FastStart is enabled, the system checkpoints the IMCUs from the IM column store to the FastStart area on disk. On subsequent database restarts, data is populated via the FastStart area rather than from the base tables.

The FastStart area is a designated tablespace where In-Memory objects are stored and managed. The IM FastStart service is database specific, such that only one FastStart area is permitted for each database or Pluggable Database (PDB) in a Container Database (CDB) environment and is automatically enabled for all In-Memory objects except for objects compressed with “NO MEMCOMPRESS”, “MEMCOMPRESS FOR DML” or with Join Groups defined on them.

The following PL/SQL procedure enables IM FastStart, and designates the tablespace FS_TBS as the FastStart area.

```
BEGIN
  dbms_inmemory_admin.faststart_enable('FS_TBS');
END;
```

Figure 11. New PL/SQL procedure FASTSTART_ENABLE to turn on In-Memory FastStart

When IM FastStart is enabled, the IMCO (In-Memory Coordinator) background process designates one of the background worker processes as the FastStart coordinator process. The FastStart coordinator maintains an ordered list of IMCUs to be written to the FastStart area. IMCUs that have not been written to the FastStart area and ones that are not changing frequently are given the highest positions on the list. If one or more of the IMCUs of an object are changing rapidly then the writing out of those IMCUs will be delayed until the frequency of the changes slows down.

In order to reduce the overhead of IM FastStart, the FastStart coordinator schedules the writing of IMCUs to the FastStart area based on the ordered list as described above. Additionally, the IMCUs are written lazily to the FastStart area with the new version of an IMCU replacing its previous version in the FastStart area. This helps ensure that the overhead to maintain the FastStart area is balanced with the benefit of having the most up to date copy of each IMCU for the object in the FastStart area.

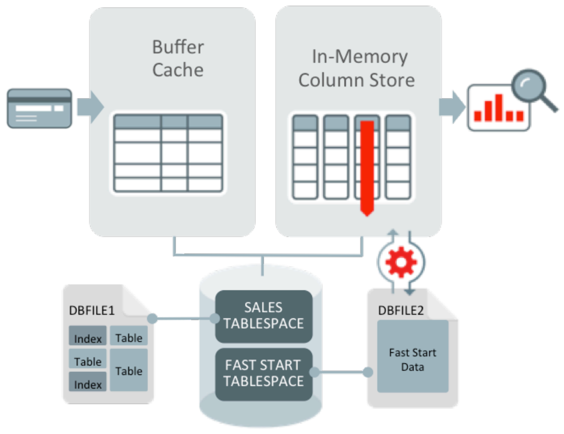


Figure 12. Populating the column store from the FastStart area

In order to populate the IM column store from the FastStart area, all transactional consistency checks need to be performed. This ensures that the data populated into the IM Column Store is consistent as of the population time. Transactional consistency checks involve comparing the System Change Number (SCN) at which the IM FastStart checkpoint was taken for the IMCU with the most recent modification SCN. Depending on the result of this check and internal thresholds, the IMCU will be populated into the IM Column Store entirely from the FastStart area, populated from the FastStart area with some rows marked invalid (due to data modification after the IMCU was written to the FastStart area) or completely discarded and populated from disk.

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. 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 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 13), 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.

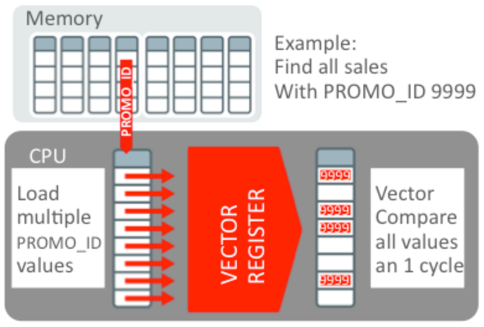
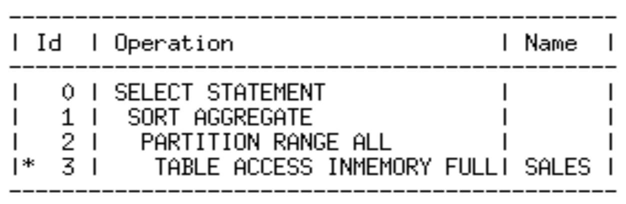


Figure 13. Using SIMD vector processing enables the scanning of billions of rows per second

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



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

Figure 14. New INMEMORY 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 Dynamic Scans

In-Memory Dynamic Scans (IM dynamic scans) are available in 18c to further increase scan performance. When additional CPU is available, IM dynamic scans accelerate In-Memory table scans. IM dynamic scans automatically use idle CPU resources to scan IMCUs in parallel and maximize CPU usage. Because in-memory scans tend to be CPU bound, IM dynamic scans can provide a significant increase in performance of in-memory scans. IM dynamic scans are more flexible than traditional Oracle parallel execution, and the two can work together. IM dynamic scans use multiple lightweight threads of execution within a process and this helps keep the performance overhead low. IM dynamic scans are controlled by Oracle Database Resource Manager and require that a CPU resource plan is enabled (for example, `RESOURCE_MANAGER_PLAN=DEFAULT_PLAN`). In Oracle Database 19c Resource Manager is automatically enabled when the `inmemory_size` initialization parameter is set greater than 0.

IM dynamic scans are considered when a query accesses an object(s) that is currently populated in the IM column store. A serial or parallel query is a candidate for IM dynamic scans when the following characteristics exist:

- Accesses a large number of IMCUs or columns
- Consumes all rows in a table
- Is CPU-intensive

In-Memory Optimized Arithmetic

In-Memory Optimized Arithmetic are available in 18c and encodes the `NUMBER` data type as a fixed-width native integer scaled by a common exponent. This enables faster calculations using SIMD hardware. The Oracle Database NUMBER data type has high fidelity and precision. However, `NUMBER` can incur a significant performance overhead for queries because arithmetic operations cannot be performed natively in hardware. The In-Memory optimized number format enables native calculations in hardware for segments compressed with the `QUERY LOW` compression option.

Not all row sources in the query processing engine have support for the In-Memory optimized number format so the IM column store stores both the traditional Oracle Database `NUMBER` data type and the In-Memory optimized number type. This dual storage increases the space overhead, sometimes up to 15%.

In-Memory Optimized Arithmetic is controlled by the initialization parameter `INMEMORY_OPTIMIZED_ARITHMETIC`. The parameter values are `DISABLE` (the default) or `ENABLE`. When set to `ENABLE`, all `NUMBER` columns for tables that use `FOR QUERY LOW` compression are encoded with the In-Memory optimized format when populated (in addition to the traditional Oracle Database NUMBER data type). Switching from ENABLE to DISABLE does not immediately drop the optimized number encoding for existing IMCUs. Instead, that happens when the IM column store repopulates affected IMCUs.

In-Memory External Tables

Database In-Memory in 18c supports populating external tables into the IM column store. This can be useful for short-term data that must be scanned repeatedly in a short time span, data aggregated by NoSQL tools that must be joined to relational data, and data that must be queried by both Oracle Database and NoSQL tools, and which must not be duplicated in the database. Unlike internal tables, external tables do not use the automatic repopulation mechanism. To refresh, or repopulate, and external table you must use the DBMS_INMEMORY.REPOPULATE procedure. In Oracle Database 19c support has been added for the ORACLE_HIVE and ORACLE_BIGDATA drivers.

In-Memory Expressions

Analytic queries often contain complex expressions in the select list or where clause predicates that need to be evaluated for every row processed by the query. The evaluation of these complex expressions can be very resource intensive and time consuming.

In-Memory Expressions provide the ability to materialize commonly used expressions in the IM column store. Materializing these expressions not only improves the query performance by preventing the re-computation of the expression for every row but it also enables us to take advantage of all of the In-Memory query performance optimizations when we access them.

An In-Memory Expression can be a combination of one or more values, operators, and SQL or PL/SQL functions (deterministic only) that resolve to a value. They must be derived only from the table they are associated with, which means that they cannot access column values in a different table. In-Memory Expressions can be created either manually via virtual columns or automatically via the Expression Statistics Store (ESS).

In-Memory Virtual Columns

User-defined virtual columns can now be populated in the IM column store. Virtual columns will be materialized as they are populated and since the expression is evaluated at population time it can be retrieved repeatedly without re-evaluation. The initialization parameter INMEMORY_VIRTUAL_COLUMNS must be set to ENABLE or MANUAL to create user-defined In-Memory virtual columns. When set to ENABLE all user-defined virtual columns on a table with the INMEMORY attribute, will be populated into the IM column store. However, it is possible to have just a subset of virtual columns be populated.

Let’s look at an analytic query, which contains a number of expressions:

```
SELECT
      l_returnflag, l_linestatus,
      SUM(l_extendedprice * (1 - l_discount)) AS sum_disc_price,
      SUM(l_extendedprice * (1 - l_discount) * (1 + l_tax)) AS sum_charge,
      COUNT(*) as count_order
FROM  lineitem
WHERE l_shipdate <= to_date ('1998-12-01','YYYY-MM-DD') - 90
GROUP BY l_returnflag, l_linestatus;
```

Figure 15. Analytic query containing a number of commonly used expressions

Imagine the expressions `SUM(l_extendedprice * (1 - l_discount))` and `SUM(l_extendedprice * (1 - l_discount) * (1 + l_tax))` are commonly used expressions for a given application, which makes them good candidates for In-Memory Expressions. Below are the steps necessary to define these expressions as virtual columns and have them populated in the IM column store:

```
-- Create virtual columns for the two expressions
ALTER TABLE lineorder ADD sum_disc_price AS (lo_extendedprice * (1 - lo_discount))

ALTER TABLE lineorder ADD sum_charge AS (lo_extendedprice * (1 - lo_discount) * (1 +
lo_tax))

-- Enable the INMMEORY attribute on the Lineorder table
ALTER TABLE lineorder INMEMORY PRIORITY HIGH
```

Figure 16. Steps required to manually populate two commonly used expressions into the IM column store

Automatically Detected In-Memory Expressions

In-Memory Expressions can also be automatically detected using the ESS and the new procedure in the `DBMS_INMEMORY_ADMIN` package. When you execute the `IME_CAPTURE_EXPRESSIONS` procedure, the 20 most frequently executed expressions, as determined by the Optimizer, are captured from the ESS and populated automatically into the IM column store. Automatically added expressions are created as hidden virtual columns and a full list of the expressions captured can be found in the view `USER_IM_EXPRESSIONS`. Below is an example demonstrating the steps for capturing the 20 most frequently executed expressions from the ESS for the past 24 hours (i.e. the `CURRENT` parameter value) and then populating these expressions into the IM column store. Note, if the second command is not used then the database will not populate the captured In-Memory Expressions until the associated table is repopulated.

```
-- Capture the expressions for ESS
BEGIN
  dbms_inmemory_admin.ime_capture_expressions('CURRENT');
END;

-- Check what expressions were captured
SELECT * FROM user_im_expressions


-- Populate the captured expression in the IM column store
BEGIN
  dbms_inmemory_admin.ime_populate_expressions;
END;
```

Figure 17. Steps required to populate the 20 most frequently executed expressions from the ESS into the IM column store

This feature also requires the setting of the initialization parameter `INMEMORY_EXPRESSIONS_USAGE`, to determine what type of In-Memory Expressions are eligible to be populated. See the Managing and Monitoring section below for more details on this parameter and other parameters used to manage In-Memory Expressions.

Currently In-Memory Expressions and In-Memory Virtual Columns are not candidates to be check-pointed to disk using In-Memory FastStart. Only the user defined columns of a table are written to the FastStart area.

JSON Document Support

Although JSON documents have always been supported in the IM column store, JSON documents can now be stored in a special binary JSON format, which enables JSON functions, such as `JSON_TABLE`, `JSON_VALUE` or `JSON_EXISTS` to perform much more efficiently.

The example below shows how to enable JSON documents in the IM column store using a table called `RETAIL`, which contains a column, `JDOC`, which contains JSON documents (not shown) and then add a new column `TAXAMOUNT` based on the `JSON_VALUE` function to enable fast retrieval of the taxable amount in the JSON document.

```
-- Create the RETAIL table with a JSON column

CREATE TABLE retail
(jdoc VARCHAR2(2000) INMEMORY CONSTRAINT json_con_1 CHECK (jdoc IS json) )

-- Load JSON documents into the RETAIL table (not shown)

-- Create binary JSON column

ALTER TABLE retail ADD
(taxamount as (json_value(jdoc,'$.TaxSummary.Summaries.TaxableAmount')))
```

Figure 18. Steps required for adding a JSON binary column to a table and then populating the table into the IM column store

The following query that uses the `JSON_VALUE` function to aggregate taxable amounts from the `RETAIL` table will now be automatically rewritten to take advantage of the JSON binary column in the IM column store, dramatically improving the performance of the query:

```
SELECT MAX(JSON_VALUE(jdoc, '$.TaxSummary.Summaries.TaxableAmount')) AS  max_tax_amount,
       SUM(JSON_VALUE(jdoc, '$.TaxSummary.Summaries.TaxableAmount')) AS  sum_tax_amount,
       AVG(JSON_VALUE(jdoc, '$.TaxSummary.Summaries.TaxableAmount')) AS  avg_tax_amount
FROM   retail
```

Figure 19. Example of a query that uses the JSON_VALUE function

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 Database In-Memory. However, they are very efficiently applied to column format data via SIMD vector processing.

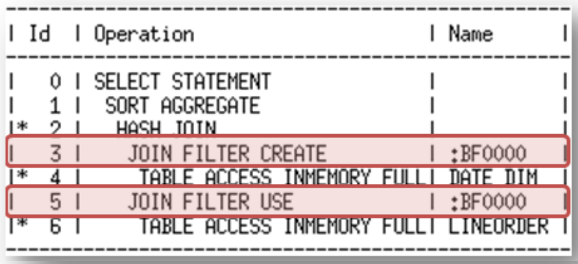
When two tables are joined via a hash join, the first table (typically the smaller table) is scanned and the rows that satisfy the `WHERE` clause predicates (for that table) are used to create an in-memory hash table stored in the Program Global Area (PGA). During the hash table creation, a bit vector 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 20. 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 21. 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: With Database In-Memory, 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.

Join Groups

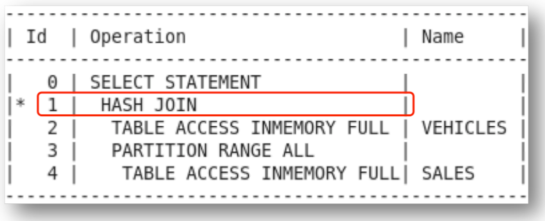
If there is no filter predicate on the dimension table (smaller table on the left hand side of the join) then a bloom filter won't be generated and the join will be executed as a standard HASH JOIN. Join Groups have been added to improve the performance of standard HASH JOINS in the IM column store. Join Groups allow the join columns from multiple tables to share a single compression dictionary, enabling the HASH JOINS to be conducted on the compressed values in the join columns rather than having to decompress the data and then hash it before conducting the join.

Let's look at an example of a two table join between the `VEHICLES` and `SALES` table to see the impact of Join Groups:

```
SELECT v.year,
       v.name,
       s.sales_price
FROM   vehicles v,
       sales s
WHERE  v.name = s.name
```

Figure 22. Simple two-table join that won't benefit from Bloom filters in the In-Memory column store

In this statement there is no filter predicate on the `VEHICLES` table so a Bloom filter won't be generated. There are also no indexes on these tables, so the optimizer will select a standard `HASH JOIN` plan (shown below):



```
------------------------------------------------------
| Id  | Operation                        | Name     |
------------------------------------------------------
|   0 | SELECT STATEMENT                 |          |
|*  1 |  HASH JOIN                       |          |
|   2 |   TABLE ACCESS INMEMORY FULL     | VEHICLES |
|   3 |   PARTITION RANGE ALL            |          |
|   4 |    TABLE ACCESS INMEMORY FULL    | SALES    |
------------------------------------------------------
```

Figure 23. Standard Hash Join Plan

This plan starts on line 2, with a full table scan of the `VEHICLES` table via the IM column store. The data from the `VEHICLES` table will be read from the IM column store, decompressed and the values in the `NAME` column (the join column) will be hashed to create a hash table in memory to help complete the `HASH JOIN` on line 1 of the plan. Next, line 4 of the plan will be executed resulting in a full table scan of the `SALES` table. Again the data will be read from the IM column store, decompressed and values in the `NAME` column will be hashed so they can be used to probe into the hash table and complete the join.

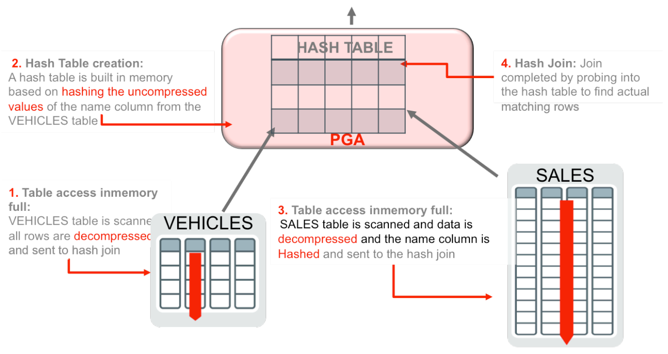

HASH TABLE
PGA
2. Hash Table creation: A hash table is built in memory based on hashing the uncompressed values of the name column from the VEHICLES table
4. Hash Join: Join completed by probing into the hash table to find actual matching rows
1. Table access inmemory full: VEHICLES table is scanne all rows are decompressec and sent to hash join
VEHICLES
3. Table access inmemory full: SALES table is scanned and data is decompressed and the name column is Hashed and sent to the hash join
SALES

Figure 24. Steps necessary to complete a standard Hash Join Plan

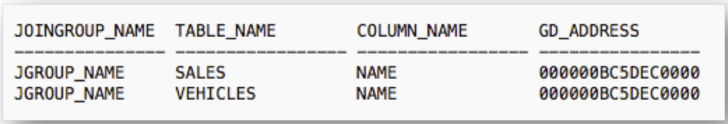
Let's now create a Join Group to help improve the performance of this `HASH JOIN`. Below is the syntax you need to create the join group:

```
CREATE INMEMORY JOIN GROUP jgroup_name (sales(name), vehicles(name))
```

Figure 25. Syntax for creating an In-Memory Join Group

The Join Group tells the IM column store that the `NAME` column in both the `VEHICLES` and `SALES` tables should share the same compression dictionary. If you recall, the compression dictionary contains a list of the distinct values for a column and the corresponding compression symbol. By using the same compression dictionary for the join column in both the `VEHICLES` and `SALES` tables, we are able to conduct the join on the compressed values, saving the effort of decompressing and hashing the data and reducing the amount of memory required to complete the join.

Although the Join Group has been created it won't begin to help until both the `VEHICLES` and `SALES` tables are repopulated into the IM column store, as we need to create and use the shared compression dictionary, known as a common dictionary. You can confirm that a Join Group has been created and common dictionary exists by querying the view `USER_JOINGROUPS`. Note that the Join Group details will be available along with the dictionary address once the tables involved in the Join Group have been repopulated:



```
JOINGROUP_NAME  TABLE_NAME         COLUMN_NAME        GD_ADDRESS
--------------- ------------------ ------------------ ----------------
JGROUP_NAME     SALES              NAME               000000BC5DEC0000
JGROUP_NAME     VEHICLES           NAME               000000BC5DEC0000
```

Figure 26. Query to confirm that a Join Group has been created and populated

If we return our focus to our simple query (Figure 22), we will see that the execution plan (figure 23) won't change even with the presence of the Join Group but how it's executed will. The execution begins just as it did before, with a full table scan of the `VEHICLES` table via the IM column store. The data from the `VEHICLES` table will be read from the IM column store, and the values in the `NAME` column (the join column) will be used to create an array of compressed values to help complete the `HASH JOIN` on line 1 of the plan. Next, line 4 of the plan will be executed resulting in a full table scan of the `SALES` table via the IM column store. The compressed values in the `NAME` column will be used to probe into the array to see if there is a match, thus completing the join.

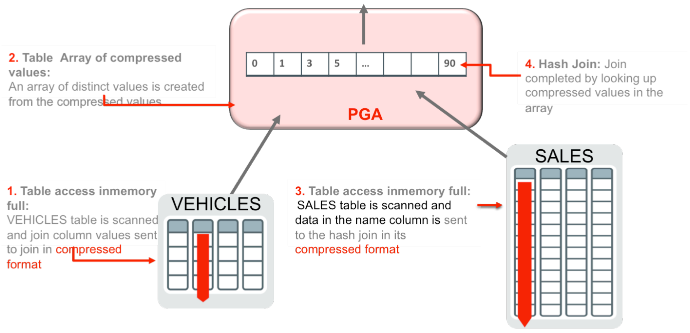

2. Table Array of compressed values:
An array of distinct values is created from the compressed values
0
1
3
5
...
90
PGA
4. Hash Join: Join completed by looking up compressed values in the array
SALES
1. Table access inmemory full:
VEHICLES table is scanned and join column values sent to join in compressed format
VEHICLES
3. Table access inmemory full:
SALES table is scanned and data in the name column is sent to the hash join in its compressed format

Figure 27. Steps necessary to complete a Hash Join Plan with a Join Group

Currently if a Join Group has been defined between tables, these tables are not candidates to be check-pointed to disk using In-Memory FastStart.

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*, was introduced with Database In-Memory 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.

3. 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.

4. 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). Starting with 12.2, the temporary tables used are in-memory only and no IO is required. Note this step is not depicted in Figure 28 below.

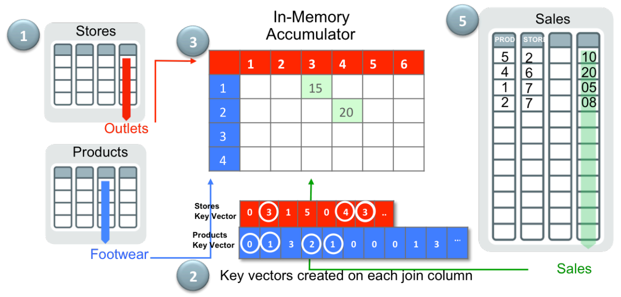


Figure 28. In-Memory aggregation example - Find the total sales of footwear in our outlet stores

Phase 2

5. 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.

6. 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 28 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 (CURSOR DURATION MEMORY)| SYS_TEMP_0FD9D75EA_4AB70D |
|   3 |    HASH GROUP BY                          |                           |
|   4 |     KEY VECTOR CREATE BUFFERED            | :KV0000                   |
|   5 |      TABLE ACCESS INMEMORY FULL          | STORES                    | <== PHASE 1
|   6 |   LOAD AS SELECT (CURSOR DURATION MEMORY)| SYS_TEMP_0FD9D75EB_4AB70D |
|   7 |    HASH GROUP BY                         |                           |
|   8 |     KEY VECTOR CREATE BUFFERED           | :KV0001                   |
|   9 |      TABLE ACCESS INMEMORY FULL          | PRODUCTS                  |
|  10 |   HASH GROUP BY                          |                           |
|* 11 |    HASH JOIN                             |                           |
|  12 |     TABLE ACCESS FULL                    | SYS_TEMP_0FD9D75EA_4AB70D |
|* 13 |     HASH JOIN                            |                           |
|  14 |      TABLE ACCESS FULL                   | SYS_TEMP_0FD9D75EB_4AB70D |
|  15 |      VIEW                                | VW_VT_0737CF93            |
|  16 |       VECTOR GROUP BY                    |                           |
|  17 |        HASH GROUP BY                     |                           |
|  18 |        KEY VECTOR USE                    | :KV0000                   |
|  19 |         KEY VECTOR USE                   | :KV0001                   | <== PHASE2
|  20 |          PARTITION RANGE ALL             |                           |
|* 21 |           TABLE ACCESS INMEMORY FULL     | SALES                     |
-----------------------------------------------------------------------------
```

Figure 29. 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 compare 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 load inserts data into database blocks that are created above the segment high water mark (i.e. the 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 the 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 any 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 row store. Remember that 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.

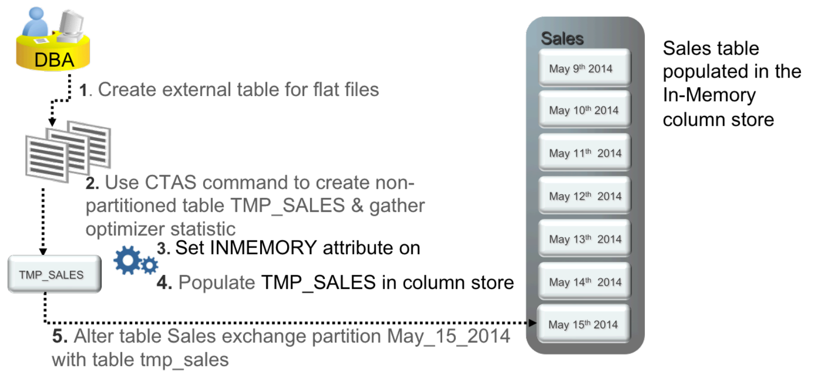

DBA
1. Create external table for flat files
2. Use CTAS command to create non-partitioned table TMP_SALES & gather optimizer statistic
3. Set INMEMORY attribute on
4. Populate TMP_SALES in column store
TMP_SALES
5. Alter table Sales exchange partition May_15_2014 with table tmp_sales
Sales
May 9th 2014
May 10th 2014
May 11th 2014
May 12th 2014
May 13th 2014
May 14th 2014
May 15th 2014
Sales table populated in the In-Memory column store

Figure 30. Five steps necessary to complete a partition exchange load on an `INMEMORY` table

Transaction Processing

Single row data change operations (DML) execute via the row store (OLTP style changes), just as they do without Database In-Memory enabled. If the object in which the DML operations occur is populated in the IM column store, then the changes are reflected in the IM column store as they occur. The row store and the column store are always kept transactionally consistent, similarly to the way indexes are kept consistent. All serialization and logging is done on the base table just as it was before. No additional locks or logging are needed for the In-Memory Column store.

For each IMCU in the IM column store, a Snapshot Metadata Unit (SMU) is automatically created and maintained (see figure 31). When a DML statement changes a row in an object that is populated into the IM column store, the corresponding column entries for that row are marked stale in the IMCU and the rowid is added to the metadata in the SMU. The original entries in the IMCU are not immediately replaced in order to provide read consistency and maintain data compression. Any transaction executing against the object in the IM column store, that started before the DML occurred, can still see the originial version of the entries in the IMCU. Read consistency in the IM column store is managed via System Change Numbers (SCNs) just as it is without Database In-Memory enabled.

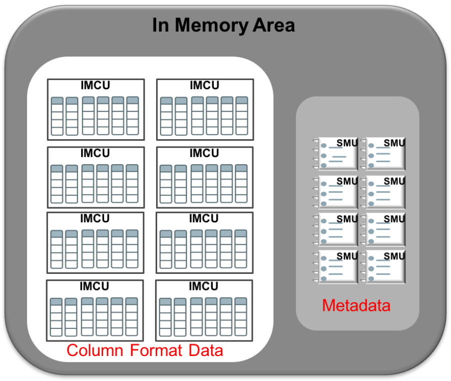


Figure 31. Each IMCU in the IM column store contains a subset of rows from an object and a corresponding SMU

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 from the base table (i.e. the row store).

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 the 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 making the IM column store a shared-nothing architecture. 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 32. This command distributes the lineorder table 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 tables where `DISTRIBUTE BY PARTITION` would lead to 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, the location of the data needs to be known. Previously, Automatic Degree of Parallelism (Auto DOP) was required so that the query coordinator could ensure that the degree of parallelism (DOP) was at least as great as the number of IM column stores involved in the query based on IMCU locations. In 12.2 this restriction has been lifted, which means the onus is now on the user to ensure that the DOP of the query is greater than or equal to the number of IM column stores involved. If this is not the case, then the data residing in IM column stores that do not get a parallel server process assigned to them will have to be read from the row store since IMCUs are not shipped across a RAC cluster.

If a DML statement is issued in a RAC environment, then the mechanism to provide read consistency is essentially the same as what was described earlier in the Transaction Processing section. The main difference is that in a RAC environment the row values that are changed are marked stale in the corresponding IMCU, whether it resides in the local node (i.e. where the DML is issued) or in another node in the cluster. In either case the IM column store is always kept transactionally consistent and no IMCUs are shipped between nodes. This ensures that DML in a RAC environment is as efficient as possible.

Distribute For Service

In addition to distributing data across all IM column stores in the RAC cluster, in 12.2 it is now possible to selectively distribute objects to specific IM column stores using the `FOR SERVICE` subclause of the `DISTRIBUTE` clause. This now makes it simpler to distribute an object to a subset of IM column stores based on service. This also facilitates the placement of objects between primary and standby databases in an Active Data Guard environment (see section below for more details). If the service is stopped then the objects distributed for that service will be removed from the IM column store(s).

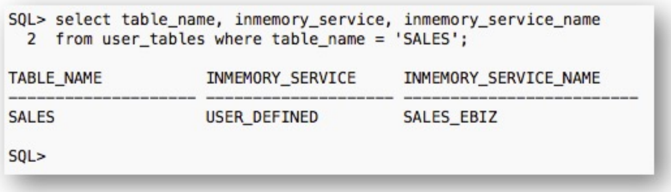
The following syntax shows adding a service to the SALES table:

```
ALTER TABLE sales INMEMORY DISTRIBUTE FOR SERVICE sales_ebiz
```

Figure 33. This command distributes sales table across the IM column stores in the cluster by service.

The SALES table will now be distributed in IM column stores for instances that run the sales_ebiz service.

The *_TABLES views have been modified to add in-memory service information.



```
SQL> select table_name, inmemory_service, inmemory_service_name
  2  from user_tables where table_name = 'SALES';

TABLE_NAME           INMEMORY_SERVICE     INMEMORY_SERVICE_NAME
-------------------- -------------------- -------------------------
SALES                USER_DEFINED         SALES_EBIZ

SQL>
```

Figure 34. In-memory service information for the SALES table.

Other benefits of the `DISTRIBUTE FOR SERVICE` subclause include support for rolling patches and upgrades, and application affinity.

Support for rolling patches and upgrades

Using the `DUPLICATE` subclause and the `FOR SERVICE` subclause allows a node to be taken out of service without affecting application availability, assuming that there is enough capacity to support the workload on the remaining nodes.

Application affinity

Some applications require one or more dedicated nodes and the `FOR SERVICE` subclause makes it simpler to direct specific objects to a specific node(s).

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.

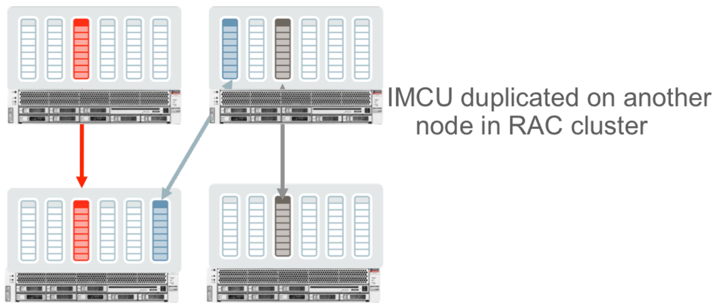


Figure 35. Objects in the IM column store on Engineered Systems can be mirrored to improve fault tolerance

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 36. This command ensures each IMCU of the lineorder table will appear in all IM column stores 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.

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 of the other nodes in the cluster.

Note that data is not 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. In order to avoid this, the system waits for a few minutes before initiating data redistribution, allowing 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.

In-Memory FastStart on RAC

In a RAC environment, in 12.2 the FastStart area is global (visible by all instances). The IMCUs from one instance can be used by another instance to populate its IM column store. When the `DUPLICATE ALL` option is enabled, only the primary instance persists the IMCUs to the FastStart area.

On instance restarts or IMCU population on a different RAC instance, the FastStart area is checked for availability of the IMCU before it is populated from disk. If the IMCU exists in the FastStart area it can be used to efficiently populate the IMCU into the IM column store. Alternatively, the regular population mechanism will populate the data from disk.

Controlling the Contents of the In-Memory Column Store

Automatic Data Optimization

Automatic Data Optimization [1](ADO) was introduced in Oracle Database 12c Release 1 to enable the automation of Information Lifecycle Management (ILM) tasks. ADO supports both compression tiering and storage tiering using policies defined at the row or segment level on tables and partitions. The Heat Map feature of ADO tracks the access of segments (reads & writes) at the row level (aggregated to block-level statistics) and at the segment level. This allows the automatic management of database segments using policies based on how database segments are being used.

Starting in 12.2, ADO has been extended to encompass the IM column store. ADO manages the content of the IM column store by executing user-defined policies to move tables or partitions in and out of the IM column store and adjusting the compression level of objects within the IM column store from a lower compression level to a higher compression level.

Three new policies have been added that enable managing objects in the IM column store:

POLICY NAME	DESCRIPTION
SET INMEMORY	Enables the INMEMORY attribute on a specified segment
MODIFY INMEMORY	Changes the compression level of an object from a lower level of compression to a higher level
NO INMEMORY	Removes, or evicts, an object from the IM column store

Figure 37. ADO policies for the IM column store

The criteria for ADO policy evaluation remains the same as it is for segment-based policies which is the number of days since the object was modified, accessed or created, or with a user-defined function. Policies will be executed as part of the automated database maintenance task's maintenance window, and since all ADO policies for Database In-Memory are segment level policies they execute only one time and are then disabled.

[1] More information Automatic Data Optimization can be found in the paper Automatic Data Optimization with Oracle Database 12c Release 2

Let's look at an example of how to specify a policy on the SALES table to enable the INMEMORY attribute 5 days after it was created. Delaying the population of newly created objects can be useful when those objects experience a high rate of change initially, but then are used mostly for queries.

```
ALTER TABLE sales ilm ADD policy SET INMEMORY AFTER 5 days OF creation
```

Figure 38. Command to specify the INMEMORY attribute on the SALES table via an ADO policy

Alternatively, we could have enabled the INMEMORY attribute on the SALES table using the MEMCOMPRESS FOR DML sub-clause and then specified an ADO policy to increase the compression level, 3 days after it stops being modified.

```
ALTER TABLE sales ilm ADD policy MODIFY INMEMORY memcompress FOR query high AFTER 3 days
OF no modification
```

Figure 39. Command to increase the in-memory compression level attribute on the SALES table via an ADO policy

Using the second approach we can maximize the space allocated within the IM column store without incurring additional compression overhead for data that is being changed frequently.

Finally let's look at a policy that will evict the SALES table from the IM column store after it has not been accessed for 30 days.

```
ALTER TABLE sales ilm ADD policy NO INMEMORY SEGMENT AFTER 30 days OF no ACCESS
```

Figure 40. Command to specify the NO INMEMORY attribute on the SALES table via an ADO policy

This type of policy provides the ability to automatically remove unused objects from the IM column store based on Heat Map data, and eliminates the chances that a frequently accessed object will be inadvertently removed from the IM column store.

User-Defined ADO Policy

You can also customize policies with the ON PL/SQL FUNCTION option using customized PL/SQL to determine when the policy should be executed. The function must return a BOOLEAN value and accept a NUMBER as an input parameter. The following is a simple example that always returns TRUE and then creates a policy based on the function:

```
CREATE OR REPLACE FUNCTION custom_im_ado (objn IN NUMBER) RETURN BOOLEAN;
ALTER TABLE sales ilm ADD policy NO INMEMORY SEGMENT ON custom_im_ado;
```

Figure 41. An example of a PL/SQL function used for an ADO eviction policy on the SALES table

Automatic In-Memory

In Oracle Database 18c, Automatic In-Memory is available to automatically manage the contents of the IM column store. If the sum of the space of the segments that have been enabled for in-memory exceeds the available memory in the IM column store then Automatic In-Memory will kick in and manage the IM column store space using heat map statistics. Using access tracking, column statistics, and other relevant statistics segments can be automatically evicted from the IM column store to make room for the population of more active segments.

This feature is controlled by the parameter INMEMORY_AUTOMATIC_LEVEL and can be left disabled (OFF – the default) or enabled with the LOW or MEDIUM options.

Currently only objects with a priority of NONE are eligible to be evicted by Automatic In-Memory. In LOW mode, if a population would fail due to a lack of sufficient space within the IM column store, then Automatic In-Memory will evict eligible populated segments before the population begins based on heat map statistics. In MEDIUM mode, if there is insufficient space in the IM column store then all population (i.e. segments enabled for INMEMORY) will be blocked and population will be fully managed by Automatic In-Memory based on heat map statistics.

The In-Memory Column Store in a Multitenant Environment

Oracle Multitenant[2] is a 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.

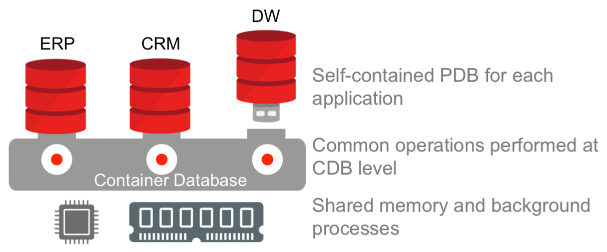

DW
ERP
CRM
Container Database
Self-contained PDB for each application
Common operations performed at CDB level
Shared memory and background processes

Figure 42. Three PDBs in a single Oracle Database 12c Container Database

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 that valuable space in the IM column store is not wasted should one of the pluggable databases be shutdown or unplugged.

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.

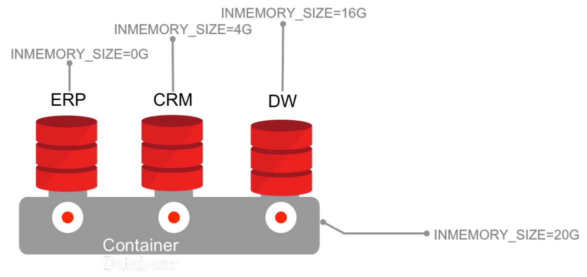


Figure 43. PDBs specify how much of the shared IM column store they can use by setting `INMEMORY_SIZE` parameter

Each pluggable database (PDB) is a full Oracle database in its own right, so the data populated into the IM column store by one PDB is not visible or accessible by another PDB and each PDB will have its own priority list. When a

[2] More information on Oracle Multitenant can be found in the white paper Oracle Multitenant: New Capabilities in Release 12.2

PDB starts up the objects on its priority list will be populated into the In-Memory column store in order of its own priority list.

The In-Memory Column Store in an Active Data Guard Environment

Oracle Active Data Guard[3] is the most comprehensive solution available to eliminate single points of failure for mission critical Oracle Databases. It prevents data loss and downtime in the simplest and most economical manner by maintaining a synchronized physical replica of a production database at a remote location. If the production database is unavailable for any reason, client connections can quickly, and in some configurations transparently, failover to the synchronized replica to restore service. It also eliminates the high cost of idle redundancy by allowing reporting applications, ad-hoc queries, and data extracts to be offloaded to read-only copies of the production database.

Active Data Guard is unique in using a highly parallelized process to apply changes to a standby database for best performance while enforcing the same read consistency model as the primary database. In 12.2, Active Data Guard has been tightly integrated with Database In-Memory, providing users the ability to enable the IM column store on the primary, standby or both environments.

With synchronized physical replication and read-consistency, in-memory processing on Active Data Guard is a viable solution for running read-only workloads instead of running those on the primary. It makes it possible to run real-time analytics on the standby database with no impact on the production database, making productive use of the standby database resources, and at the same time increasing the total columnar capacity of the system.

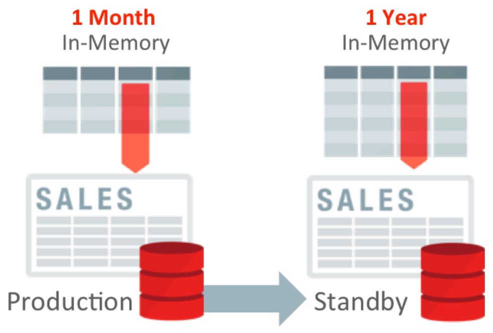


Figure 44. Example of how the IM column store on the standby database can have very different content to the primary

By considering the standby environment as a separate database, Database In-Memory makes it possible to populate the same or a different set of tables or table partitions in-memory on the primary and on the standby database. Just as Active Data Guard maintains a synchronized physical replica of the production database, it also maintains the contents of the IM column store ensuring transactionally consistent results as of the query SCN.

As described in the RAC section above, the `DISTRIBUTE FOR SERVICE` clause can be used to specify the placement and population of the in-memory objects across instances in a RAC cluster. This clause can also be used for populating objects into the IM column store on the standby database by providing the appropriate service name, active only on the standby.

When a service name is specified, the table or table partition is populated into memory only on the database instances where the specified service is active. In an event of a role change or switchover, the tables will be repopulated on the instance(s) where the new services are active.

[3] More information on Active Data Guard can be found in the paper Oracle Active Data Guard

Below is an example of syntax required to populate the EMPLOYEES table into memory only on the database instances where the "REPORTING_STANDBY_SVC" is allowed to run.

```
CREATE TABLE employees
  (c1 NUMBER,
   c2 NUMBER,
   c3 VARCHAR2(10),
   c4 CLOB )
INMEMORY MEMCOMPRESS FOR QUERY
DISTRIBUTE AUTO FOR SERVICE reporting_standby_svc
```

Figure 45. Using DISTRIBUTE AUTO FOR SERVICE sub-clause to populate a table on the standby instance in Active Data Guard

Restrictions on Active Data Guard

In an Active Data Guard environment, the standby is opened in a read-only mode, which has an impact on some of the new In-Memory functionality as outlined below:

- It is not possible to maintain the In-Memory FastStart area on the standby database, therefore In-Memory FastStart is not supported on the standby database.
- It is also not possible to maintain the Expression Statistics Store (ESS) on the standby database. Therefore, all automatically detected expressions will be based on the workload seen on the primary. If the workload on the primary database is not representative of the workload on the standby, we do not recommend using automatically detected In-Memory Expressions. Instead In-Memory virtual columns should be used to materialize the commonly used expressions on the standby into the IM column store.
- In-Memory Join Groups are not supported on the standby.
- With Automatic Data Optimization, INMEMORY policies are only evaluated on the primary database. However, since ILM INMEMORY policies are implemented with ALTER TABLE statements, an object residing in-memory on the standby database will be affected should a policy specified on it be executed. For example, if an object is only populated in-memory on the standby database and an ILM INMEMORY policy based on number of days since the object was accessed is specified on it, that policy would be evaluated on the primary database. If the object is only accessed on the standby and never on the primary, then the outcome of that policy may not be what was expected. It is for this reason that we recommend caution when specifying ILM INMEMORY policies based on days for objects that will reside in the standby database only.

Extending In-Memory Columnar Format to Flash on Exadata

The Oracle Exadata Database Machine uses a unique set of software algorithms to implement database intelligence in storage, PCI based flash, and InfiniBand networking. A full rack Exadata X7-2 Database Machine offers 358TB of flash, which is nearly 30X the capacity of DRAM and can deliver up to 350GB/second of bandwidth from flash.

It is now possible to store data in the In-Memory columnar format in the flash cache in an Exadata environment. This enables all of the In-Memory optimizations (accessing only the compressed columns required, SIMD vector processing, storage indexes, etc.) to be used on a potentially much larger amount of data.

When the `INMEMORY_SIZE` parameter is set to a non-zero value objects accessed using a Smart Scan will be brought into Exadata flash cache and will be automatically converted into the In-Memory columnar format. The data will initially be converted into a columnar cache format, but not Database In-Memory's columnar format. The data will be rewritten in the background into Database In-Memory columnar format. This will result in all subsequent accesses to the data benefitting from all of the In-Memory optimizations when that data is retrieved from the flash cache.

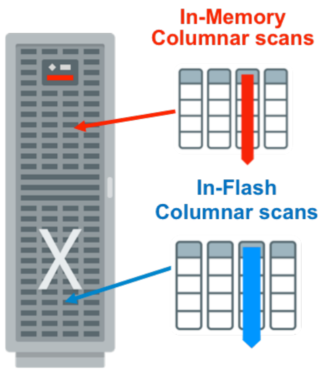


Figure 46. All of the benefit of In-Memory columnar now available on Exadata Flash

A new segment-level attribute, `CELLMEMORY`, has also been introduced to help control which objects should not be populated into flash using the In-Memory columnar format and which type of compression should be used. Just like the `INMEMORY` attribute you can specify different compression levels as sub-clauses to the `CELLMEMORY` attribute. However, not all of the `INMEMORY` compression levels are available; only `MEMCOMPRESS FOR QUERY LOW` and `MEMCOMPRESS FOR CAPACITY LOW` (default).

```
ALTER TABLE trades CELLMEMORY MEMCOMPRESS FOR QUERY LOW
```

Figure 47. New CELLMMEORY segment-level attribute indicates that the TRADES table should be populated into Exadata flash cache using MEMCOMPRESS FOR QUERY LOW compression

The `PRIORTY` sub-clause is also not available, as the concept of population by background worker processes is different on the Exadata storage cells (where the flash cache resides), as described above, than it is in DRAM in the IM column store.

Controlling the Use of Database In-Memory

There are multiple initialization parameters and optimizer hints that allow you to control when and how the IM column store will be used. This section describes the most important ones and provides guidance on which ones are key and which are optional.

Key Initialization Parameters

The following initialization parameters are key in that they directly control the different aspects of in-memory functionality.

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. Starting in 12.2 if the IM column store has been enabled (i.e. inmemory_size > 0) then it is possible to increase the size of the IM column store dynamically. The IM column store must be increased in increments of 128MB or more, and there must be enough memory available in the SGA to accommodate the increased size.

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 X 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

The following additional initialization parameters control additional features or behavior of in-memory functionality.

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 48. 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 is 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`.

INMEMORY_VIRTUAL_COLUMNS

The `INMEMORY_VIRTUAL_COLUMNS` parameter controls the use of virtual columns in the IM column store. Three parameter values are available (i.e. `ENABLE, MANUAL, DISABLE`). The `ENABLE` value specifies that all virtual columns for the table or partition will be populated in the IM column store at the default table or partition memcompress level unless the virtual column has been explicitly excluded or a different memcompress level has been specified. The `MANUAL` value is the default and specifies that no virtual columns will be populated in-memory unless they have been explicitly marked for inmemory or they have been marked for inmemory with a different memcompress level. The `DISABLE` value disables the use of virtual columns in the IM column store.

INMEMORY_EXPRESSIONS_USAGE

The `INMEMORY_EXPRESSIONS_USAGE` parameter controls which In-Memory Expressions are populated in the IM column store. There are four values for this parameter (i.e. `STATIC_ONLY, DYNAMIC_ONLY, ENABLE, DISABLE)`. The `STATIC_ONLY` value allows the population of JSON columns in a special binary JSON format. The `DYNAMIC_ONLY` value will populate automatically created In-Memory Expressions in the IM column store when used with the `DBMS_INMEMORY.IME_CAPTURE_EXPRESSIONS` procedure. The `ENABLE` value combines both the `STATIC_ONLY` and `DYNAMIC_ONLY` parameter values, and the `DISABLE` value prevents any In-Memory Expressions from being populated in the IM column store.

INMEMORY_AUTOMATIC_LEVEL

The `INMEMORY_AUTOMATIC_LEVEL` parameter is used to enable the Automatic In-Memory feature. There are three values for this parameter (i.e. `LOW`, `MEDIUM` and `OFF`). The `LOW` value enables the database to evict cold segments when the IM column store is under memory pressure. The `HIGH` value includes the ability to ensure that any hot segment that was not populated due to memory pressure is populated first. The `OFF` value disables Automatic In-Memory. This is the default value.

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 it will not be applicable to the 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 fully populated into the column store and the plan with the lowest cost is a full table scan.

In-Memory Scan

As stated 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 (i.e. table, partition or subpartition).

The `(NO_)INMEMORY_PRUNING` hint can also influence the performance of 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.

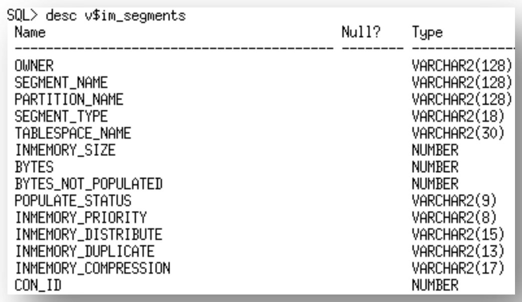
Conclusion

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.

Appendix A - Monitoring and Managing Oracle Database In-Memory

Monitoring Objects in the In-Memory Column Store

There are two 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
```

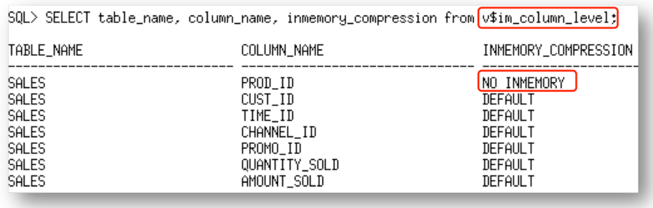
Figure 49. v$IM_SEGMENTS view

These views not only show which objects are populated in the IM column store, 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 50. Determining the compression ratio achieved for the objects populated into the IM column store

Another 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.



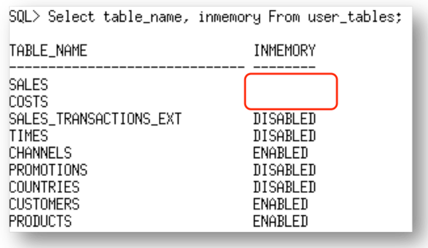
```
SQL> SELECT table_name, column_name, inmemory_compression from v$im_column_level;

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 51. The PROD_ID column was not populated into the IM column store

USER_TABLES

A 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 52. 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.

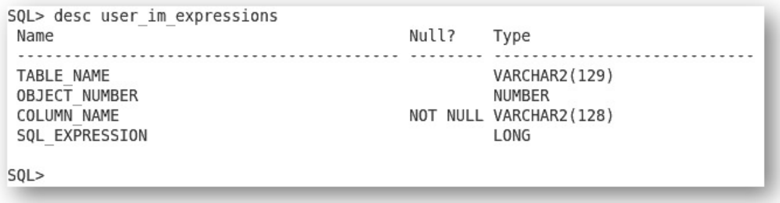
Three additional columns – INMEMORY_PRIORITY, INMEMORY_DISTRIBUTE, and INMEMORY_COMPRESSION – have also been added to the *_TABLES views to indicate the current In-Memory attributes for each table.

Two additional columns - INMEMORY_SERVICE, INMEMORY_SERVICE_NAME - have been added to the *_TABLES views to indicate the In-Memory attributes associated with the FOR SERVICE subclause of the DISTRIBUTE clause.

Finally, an additional column, CELLMEMORY has been added to the *_TABLES views to indicate that the table is a candidate to be brought into the flash cache on Exadata with non-default values.

USER_IM_EXPRESSIONS

Two views have been added USER_IM_EXPRESSIONS and DBA_IM_EXPRESSIONS to allow the easy display of In-Memory Expressions.



```
SQL> desc user_im_expressions
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 TABLE_NAME                                         VARCHAR2(129)
 OBJECT_NUMBER                                      NUMBER
 COLUMN_NAME                               NOT NULL VARCHAR2(128)
 SQL_EXPRESSION                                     LONG

SQL>
```

Figure 53. USER_IM_EXPRESSIONS view

In the example above, you will notice that the table and column name is available along with the SQL expression and object number. The DBA_IM_EXPRESSIONS view adds the OWNER column.

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 Manager[4] 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.

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 54. Changing the Resource Manager consumer group of the `INMEMORY` operation

Session Level Statistics

It is also possible to monitor what is happening with Database In-Memory by querying the session level statistics. Below is a list of the most commonly queried In-Memory session level statistics an explanation of what they represent.

Statistics Name	Description
IM scan rows optimized	Number of rows that were skipped (because of storage index pruning) or that weren't accessed due to aggregations with predicate push downs
IM scan rows projected	Number of rows returned to the upper layer
IM scan rows	Number of rows scanned in all IMCUs
IM scan rows valid	Number of rows scanned in all IMCUs after applying valid vector
IM scan CUs no memcompress IM scan CUs memcompress for *	Number of times IMCUs of each memcompress type were touched
IM scan CUs columns accessed	Number of columns accessed by a scan

[4] More information on using Oracle Database Resource Manager can be found in the white paper Using Oracle Resource Manager

IM scan CUs invalid or missing revert to on disk extent	Number of on disk extents accessed due to missing or invalid IMCUs
IM scan CUs pruned	Number of IMCUs with no rows passing min/max
IM scan segments minmax eligible	Number of IMCUs that are eligible for min/max pruning
IM scan segments disk	Number of times a segment marked for in-memory was accessed entirely from the buffer cache/direct read
IM scan CUs predicates applied	Number of min/max predicates applied
IM scan CUs predicates optimized	Number of IMCUs where either all rows passed min/max or no rows passed min/max
IM scan CUs predicates received	Number of min/max predicates received
IM scan EU rows	Number of rows scanned from EUs in the IM column store before where clause predicate applied
IM scan EUs no memcompress IM scan EUs memcompress for *	Number of times IMEUs of each memcompress type were touched
IM scan EUs columns accessed	Number of columns in the EUs accessed by scans
IM scan EUs columns decompressed	Number of columns in the EUs decompressed by scans
IM scan EU bytes in-memory	Size in bytes of in-memory EU data accessed by scans
IM scan EU bytes uncompressed	Uncompressed size in bytes of in-memory EU data accessed by scans
IM scan EUs columns theoretical max	Number of columns that would have been accessed from the EU if the scans looked at all columns
IM scan EUs split pieces	Number of split EU pieces among all IM EUs
IM populate segments requested	Number of population tasks for in-memory segments
table scan disk IMC fallback	Number of rows in blocks scanned from buffer cache/direct read where an IM scan was possible

table scan disk non-IMC rows gotten	Number of rows in blocks scanned from buffer cache/direct read where an IM scan was not possible
table scans (IM)	Number of segments scanned in-memory
session logical reads - IM	Number of blocks scanned in an IMCU

Figure 55. List of useful `INMEMORY` session level statistics

Oracle Database In-Memory: Technical Overview
February 2019, Revision 19.1
Author: Andy Rivenes