Oracle Database In-Memory Quick Start Guide

May 2021 | Version 1.7

PURPOSE STATEMENT

This document provides information about how to get started with using Oracle Database In-Memory. It is intended solely to help you assess the business benefits of upgrading and using Oracle Database In-Memory.

DISCLAIMER

This document in any form, software or printed matter, contains proprietary information that is the exclusive property of Oracle. Your access to and use of this confidential material is subject to the terms and conditions of your Oracle software license and service agreement, which has been executed and with which you agree to comply. This document and information contained herein may not be disclosed, copied, reproduced or distributed to anyone outside Oracle without prior written consent of Oracle. This document is not part of your license agreement nor can it be incorporated into any contractual agreement with Oracle or its subsidiaries or affiliates.

This document is for informational purposes only and is intended solely to assist you in planning for the implementation and upgrade of the product features described. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described in this document remains at the sole discretion of Oracle.

Due to the nature of the product architecture, it may not be possible to safely include all features described in this document without risking significant destabilization of the code.

Table of Contents

INTRODUCTION

Oracle Database In-Memory (Database In-Memory) was introduced in the first patch set for Oracle Database 12c Release 1 (12.1.0.2). Since then it has been actively developed and enhanced to make it even more performant, manageable and scalable. Database In-Memory is available with Oracle Database Enterprise Edition on-premises, Oracle Cloud and Cloud at Customer. It adds in-memory functionality to Oracle Database for transparently accelerating analytic queries by orders of magnitude, enabling real-time business decisions without needing application code changes. It accomplishes this using a "dual-format" architecture to leverage columnar formatted data for analytic queries while maintaining full compatibility with all of Oracle's existing technologies. This "dual-format" architecture provides the best of both worlds. The traditional row format for incredibly efficient on-line transaction processing (OLTP) and the columnar format for super-fast analytic reporting.

The purpose of this technical brief is to provide a set of guidelines that can enable most customers to quickly get started with Database In-Memory with a minimum amount of effort. For more detailed guidelines on evaluating or implementing Database In-Memory see the Oracle Database In-Memory Implementation Guidelines. The information provided is based on our experience of participating in many implementations and working directly with customers and should work well in most situations.

This technical brief assumes that you are familiar with Database In-Memory fundamentals as outlined in the Oracle Database In-Memory whitepaper.

PLANNING

Identify Analytic Workloads

Database In-Memory is not a one size fits all solution. It is specifically targeted at analytical workloads, which is why the IM column store is populated in a columnar format. Columnar formats are ideal for scanning and filtering a relatively small number of columns very efficiently. It is important to understand the fundamental difference between Database In-Memory and the traditional Oracle Database row-store format. The row format is excellent for OLTP type workloads, but the columnar format is not and Database In-Memory will not speed up pure OLTP workloads. Workloads that involve pure OLTP, that is inserts, updates and deletes (i.e. DML) along with queries that select a single row, or just a few rows, will generally not benefit from Database In-Memory. Workloads that do mostly ETL where the data is only written and read once are also not very good candidates for Database In-Memory.

The ideal workload for Database In-Memory is analytical queries that scan large amounts of data, access a limited number of columns and use aggregation and filtering criteria to return a small number of aggregated values. Queries that spend the majority of their time scanning and filtering data see the most benefit. Queries that spend the majority of their time on complex joins (e.g. where result sets flow through the plan or that use windowing functions), sorting or returning millions of rows back to the client will see less benefit. A good way to think about this is that the goal of Database In-Memory queries is to perform as much of the query as possible while scanning the data. By taking advantage of the ability to push predicate filters directly into the scan of the data, the use of Bloom filters to transform hash joins into scan and filter operations, and the use of VECTOR GROUP BY to perform group by aggregations as part of the in-memory scan, Database In-Memory can in the best cases perform the majority of a query's processing while scanning the data. Other Database In-Memory features complement these primary attributes and include the use of Single Instruction Multiple Data (SIMD) vector processing, In-Memory storage index pruning, In-Memory Expressions (IME) to avoid re-computing commonly used expressions, Join Groups (JG) to further enhance join performance and In-Memory Dynamic Scans (IMDS) to dynamically parallelize In-Memory compression unit (IMCU) scans to provide even more scan performance gains.

Understanding How Database In-Memory Works

It is important to understand how Database In-Memory works since it does not benefit all workload types. As was stated earlier, Database In-Memory benefits queries that spend the majority of their run time scanning and filtering data, performing joins and group by aggregations. This will be reflected in execution plans showing inmemory full table scans (i.e. TABLE ACCESS INMEMORY FULL), hash joins with Bloom filters or nested loops joins with inmemory access and aggregations using VECTOR GROUP BY operations. In addition, the execution plan may also show predicate push down and filtering for inmemory scans[1]. All of these operations benefit from the optimizations included in Database In-Memory. Other database operations do not benefit from Database In-Memory. This includes DML operations (i.e. insert, update, delete), sorting, row fetching, other types of joins, index accesses, etc.

[1] See the Database In-Memory blog for a two part blog post on predicate push down.

Identify Success Criteria

It is important to define success criteria when implementing Database In-Memory. We have seen many implementations get bogged down in trying to get improvement from every transaction or SQL statement. The reality is that most SQL statements that are analytical in nature will improve. By how much is highly dependent on how much time is being spent scanning, filtering, joining and aggregating data, and whether the Optimizer can further optimize the execution plan based on an in-memory access path. Another important consideration is whether any SQL statements regress in performance.

Success criteria will be dependent on what the customer is trying to achieve, but in general will consist of one or more of the following criteria:

- Transaction or SQL response time
- Resource usage
- System level workload

Although transaction response time is often the most important consideration, transaction throughput or an increase in capacity headroom can be just as valuable. Consider the environment where a relatively small reduction in transaction resource usage can translate into a huge gain in capacity headroom because of the enormous volume of transactions executed. And all without having to change application code.

An important way to create measurable criteria for success is to establish a baseline for the performance of the application or selected SQL statements. If focusing on SQL response time then the simplest way to create a baseline is to identify the SQL statements that are analytical in nature and are the target of the implementation. This can be done based on knowledge of the application or with the help of the In-Memory Advisor as mentioned earlier. No matter what the success criteria, once identified, performance characteristics can be measured and recorded as the baseline. Whether upgrading from an earlier release of Oracle Database, or implementing a new application, a baseline should be created in the target Oracle Database version prior to implementing Database In-Memory.

As part of creating a baseline, it is important to consider the hardware platform and software involved. For example, you cannot expect to see the same performance if migrating to different hardware platforms or CPU generations. As with any benchmark-like project it is also important to ensure that the testing is repeatable. In the case of Oracle Database this means that execution plans need to have stabilized and that the application environment has reached a steady state. A single SQL execution is not a realistic baseline, there are just too many variables involved in SQL executions. This will probably result in the need for many repeated executions and multi-user testing to replicate a representative environment.

Once a baseline is created, a comparison can then be made to determine how much benefit Database In-Memory provided. It is also important to properly define expectations. How much improvement needs to occur? For example, do all the targeted SQL statements or transactions need to improve, and if so, by how much? Is a 10X performance improvement acceptable and how much business value can be derived from the improvement? How much improvement is really practical? There are limits to performance improvements based on the technology used.

These are the types of questions that can be used as input to determining the success of a Database In-Memory implementation.

Identify How to Measure Improvement

It is important to determine how changes and benefits will be identified and measured. For individual SQL we recommend the use of SQL Monitor active reports to accomplish this. SQL Monitor requires the Oracle Diagnostics and Tuning packs and provides a visual, time based method of analyzing the execution of SQL statements. The SQL Monitor active report is used to display this information and provides a simple, accurate way of comparing SQL statement execution. More information can be found in the Oracle Database SQL Tuning Guide.

If measuring success based on application throughput then that throughput has to be quantified. This may be available in application tracking information or perhaps Database system statistics. For database workload measurement the Automatic Workload Repository (AWR) and it's reporting capability may be the best tool.

How To Get Started

Two questions come up most frequently when implementing Database In-Memory:

- How do we identify the objects to populate into the IM column store?
- How much memory should we allocate to the IM column store?

The answers to both questions are highly dependent on each application environment. Since Database In-Memory does not require that the entire database be populated into memory, customers get to decide how much memory to allocate and which objects to

populate. While there is no substitute for understanding the application workload being targeted for Database In-Memory, it is not always practical to know which objects would benefit the application the most from being populated in the IM column store.

In the case of which objects to populate into the IM column store, there are basically two ways to tackle the issue. There is a utility available called the Oracle Database In-Memory Advisor (In-Memory Advisor). It uses Automatic Workload Repository (AWR), Active Session History (ASH) and other meta-data to help with determining which objects will benefit from Database In-Memory. Since the In-Memory Advisor is available as a separate utility it can be used in database versions 11.2.0.3 or higher. This allows for advanced planning for existing database environments before implementing Database In-Memory. More information can be obtained about the In-Memory Advisor via My Oracle Support (MOS) Note 1965343.1 and the In-Memory Advisor technical white paper. The other way of tackling the issue is to identify the objects that are accessed by the application's analytic queries and enable those objects for population. This can be accomplished by mining the SQL statements being run or based on application knowledge. If there is enough memory available then it is also possible to just populate all of an application's objects.

In Oracle Database 18c and higher there is a feature of Database In-Memory called Automatic In-Memory (AIM). This feature leverages Heat Map like data to track actual segment usage and can be used to populate, evict and even compress segments to a higher level based on usage.

For the second question about how much memory to allocate to the IM column store, the Compression Advisor (i.e. the DBMS_COMPRESSION PL/SQL package) can be used. The Compression Advisor has been enhanced in Oracle Database 12.1.0.2 and above to recognize the different Database In-Memory compression levels and to measure how much memory would be consumed in the IM column store by the object(s) in question based on the target compression level.

Database Upgrades

Since Database In-Memory requires Oracle Database 12.1.0.2 or higher it is still very common that a Database In-Memory implementation will also require an upgrade to Oracle Database. If this is the case then the two activities, that is the database upgrade and the Database In-Memory implementation, should be performed separately. It can be very difficult to identify the root cause of a performance regression, or attribute performance improvement, when multiple things change at once. By separating the upgrade activity from the Database In-Memory implementation it will be much easier to identify performance changes.

Execution Plan Stability

An important consideration when implementing Database In-Memory is that execution plans will change when the IM column store is accessed. For example, a TABLE ACCESS BY INDEX ROWID could change to a TABLE ACCESS INMEMORY FULL. The *Identifying In-Memory Usage* section later in this document has some specific examples of the types of execution plan changes that can occur when using Database In-Memory. As part of the implementation process outlined in the *Implementation* section optional steps have been added that specify the use of SQL Plan Baselines to help identify changed execution plans. SQL Plan Baselines are one part of Oracle's SQL Plan Management feature, and while not required, can be very useful in controlling SQL execution plans.

The steps jn the *Implementation* section have been made optional since SQL Plan Baselines do have limitations and it is beyond the scope of this paper to provide a step by step guide as to their usage. However, there is an abundance of information about SQL Plan Management and the use of SQL Plan Baselines and other methods of ensuring controlled plan execution. A good place to start for more information is with the Oracle Database SQL Tuning Guide and the Optimizer Blog.

CONFIGURATION

Apply the Latest Database Release Update

Before starting any Database In-Memory implementation you should ensure that the latest available Database Release Update (RU) (formerly known as the Database Proactive Bundle Patch) has been applied. Database In-Memory fixes are only distributed through the Database Release Update process and this will ensure that you have the latest performance enhancing fixes. Database Release Updates are delivered on a pre-defined quarterly schedule and are cumulative.

For more information on patching see the following Oracle support documents:

- Oracle Database - Overview of Database Patch Delivery Methods for 12.2.0.1 and greater (MOS Note: 2337415.1)
- Oracle Database - Overview of Database Patch Delivery Methods - 12.1.0.2 and older (MOS Note: 1962125.1)

There are also specific Proactive Patch Information notes available based on release:

- Oracle Database 19c Proactive Patch Information (MOS Note: 2521164.1)

- Database 18 Proactive Patch Information (MOS Note: 2369376.1)
- Database 12.2.0.1 Proactive Patch Information (MOS Note: 2285557.1)
- Database 12.1.0.2 Proactive Patch Information (MOS Note: 2285558.1)

Memory Allocation

When adding Database In-Memory to an existing database environment, it is important to add additional memory to support the IM column store. You should not plan on sacrificing the size of the other System Global Area (SGA) components to satisfy the sizing requirements for the IM column store. IM column store sizing can be done using the In-Memory Advisor and the Compression Advisor which are described in prior sections of this document, but knowledge of the application workload will make this process much easier and more accurate.

When using Database In-Memory in a Real Applications Cluster (RAC) environment additional memory must also be added to the shared pool. In RAC environments, every time a database block is populated in the IM column store as part of an IMCU, Database In-Memory allocates a RAC-wide lock for that database block. This RAC lock ensures that any attempt to do a DML on a database block on any of the RAC instances will invalidate the database block from the column store of all other instances where it is populated. Additional memory is also required for an IMCU home location map which is used to track the IMCU locations across the RAC instances.

The amount of memory allocated from the shared pool by Database In-Memory depends on several factors:

- Size of the IM column store
- Compression ratio of the data populated in the IM column store
- Database block size
- Size of the RAC lock (approximately 300 bytes)

The allocation of memory for the In-Memory column store should follow the general formula below in Figure 1 and should be considered a minimum recommendation for the additional amount of memory that should be allocated to the SGA. See the MOS Note, Oracle Database In-Memory Option (DBIM) Basics and Interaction with Data Warehousing Features (1903683.1).

These recommendations are made assuming Automatic Shared Memory Management (ASMM) and the use of SGA_TARGET and PGA_AGGREGATE_TARGET initialization parameters. See the Database Administrator's Guide for more information about managing memory.

Type of Database	New SGA_TARGET with DBIM	New PGA_AGGREGATE_TARGET** with DBIM
Single-instance Databases	SGA_TARGET + INMEMORY_SIZE	PGA_AGGREGATE_TARGET
RAC Databases	SGA_TARGET + (INMEMORY_SIZE * 1.1)	PGA_AGGREGATE_TARGET

Figure 1. Memory Allocation

** Note that Database In-Memory queries tend to perform large aggregations and can use additional Program Global Area (PGA) memory. If not enough PGA memory is available then space in the temporary tablespace will be used. The maximum amount of PGA memory that a single database process can use is 2GB, but for parallel queries it can be as high as 2GB * PARALLEL_MAX_SERVERS. PGA_AGGREGATE_TARGET should be sized with this in mind.

Sufficient PGA memory should also be allocated to ensure that any joins, aggregations, or sorting operations remain in memory and spilling to disk is avoided. For existing systems, a good way to identify the initial SGA and PGA sizes is to use AWR reports. In the Advisory Statistics section there is a Buffer Pool Advisory section and a PGA Memory Advisory section.

Planning for Additional Memory Requirements

You should also not plan on allocating all of the memory in the IM column store memory. You need to allow extra room to accommodate new data and changes to existing data due to DML activity. Like database blocks in the row store, when IMCUs are re-populated they can change in size, or can be split thus taking up more memory. When new data is inserted new IMCUs may be created to populate the new data.

Database Parameter Settings

We strongly recommend that you evaluate all initialization parameters that have non-default settings. It is very common to keep parameter settings through different database versions because nobody knows why they were set or what they do. The problem with some initialization parameters is that they can negatively affect the use of Database In-Memory.

For example, customers on previous database releases often set the following parameters to non-default values:

- COMPATIBLE
- OPTIMIZER_INDEX_CACHING
- OPTIMIZER_INDEX_COST_ADJ
- OPTIMIZER_FEATURES_ENABLE

These particular parameters can prevent the use of Database In-Memory plans or severely limit their usage. This is not to say that these are the only parameters to look for, or that their usage will necessarily prevent the use of Database In-Memory execution plans, but these are examples of where you can get into trouble by just leaving parameters set because you don't know why they are set.

This is not to say that all non-default parameters will cause problems, but in general, if you don't know why a parameter is set then unset it and see if there are any problems. Pay particular attention to any underscore parameters. Each new release of Oracle has many new features and bug fixes. An issue with an older release may have been corrected in a newer release and this is another reason that parameter settings should be re-evaluated.

Inmemory Parameter Settings

The following parameter enables Database In-Memory and should be set based on expected usage:

INMEMORY_SIZE - initially keep set at 0 for the baseline and then set based on the object space in the IM column store that is required plus room for growth.

Note that this may be an iterative process to get the size correct and since the IM column store is allocated from the SGA, other initialization parameters may have to modified to accommodate the increased size (i.e. SGA_TARGET or MEMORY_SIZE).

Also note that increasing the size of Oracle Database shared memory allocations can have operating system implications as well. Database In-Memory doesn't change the behavior of how Oracle Database uses shared memory, or how it implements OS optimizations.

For additional database specific information we recommend that you consult the appropriate installation and/or administration guides for your platform and MOS note(s) for any updates. The Oracle Database Upgrade PM team is also an excellent resource for Upgrade Best Practices.

Memory Allocation in Multitenant Databases

Oracle Multitenant is a database consolidation architecture first introduced in Oracle Database 12c in which multiple Pluggable Databases (PDBs) are consolidated within a single Container Database (CDB). While keeping many of the isolation aspects of single databases, Oracle Multitenant allows PDBs to share the SGA and background processes of a common CDB.

When used with Oracle Database In-Memory, PDBs also share a single In-Memory column store (IM column store) and hence the question, "How do I control how much memory each PDB can use in the IM column store?"

The total size of the IM column store is controlled by the INMEMORY_SIZE parameter setting in the CDB. By default, each PDB sees the entire IM column store size and has the potential to fully populate it and possibly starve other PDBs. In order to avoid this you can specify how much of the shared IM column store a PDB can use by setting the INMEMORY_SIZE parameter inside a specific PDB using the following command:

```
ALTER SYSTEM SET inmemory_size = 4G container = CURRENT scope = spfile
```

Not all PDBs in a given CDB may need to use the IM 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.

Population Considerations

There are several things to consider for population. Obviously having enough memory available in the IM column store is crucial. You do not want to have partially populated objects if at all possible as this will significantly affect the performance of the application. You also need to take into account the growth of segments populated in the IM column store. Just because everything fits today doesn't mean that they will still be fully populated after undergoing significant DML changes.

As mentioned in an earlier section, in order to gain memory for the IM column store you do not want to "steal" memory from other components of the SGA (i.e. buffer pool, shared pool, etc.) because that can affect existing application performance. This is especially important in mixed workload environments. Very often the easiest strategy is to populate the entire schema in the IM column store (reference the *How To Get Started* section earlier in the paper). The ability to achieve this will be dependent on the amount of memory that can be allocated to the IM column store and the compression level chosen.

Population is largely a CPU dominated process, and as with parallelism in general, the more worker processes that can be allocated to population, the faster the population will happen. This is a balancing act, and you typically don't want to saturate the machine with population if you have other workload needs. There are however, customers who do not want the database accessed until all critical segments are populated. For those customers, since the MAX_POPULATE_SERVERS parameter is dynamic, CPUs allocated to population can be adjusted higher during initial population. In addition, starting in Oracle Database 19c the POPULATE_WAIT function of the DBMS_INMEMORY_ADMIN package can be used to programmatically determine if all objects have been populated. This can be used to block connections from an application(s) until population is complete.

Another consideration is to only populate the data that will give you the biggest benefit. For many applications this means the most current data. Typically this is time based information and partitioning can help divide the data so that just the most beneficial data can be populated into the IM column store. Population of partitions can be handled in several different ways. The table can be enabled for in-memory and then all partitions will be populated, including newly created partitions since they will inherit the default attribute of inmemory enabled. If however a rolling window partitioning strategy is desired for population in the IM column store then individual partition/sub-partitions can be altered to be inmemory or no inmemory. In 12.1 this can be accomplished with a scheduler job or manually. In 12.2 and greater Automatic Data Optimization (ADO) policies can be employed based on heat map heuristics or time to populate or evict individual partitions/sub-partitions.

REAL APPLICATION CLUSTERS (RAC)

RAC databases enable the scale-out of the IM column store by allowing the allocation of an IM coumn store on each RAC database instance. Data is then distributed across IM column stores effectively increasing the IM column store size to the total inmemory size allocated to all of the RAC database instances.

Conceptually it helps to think of Database In-Memory in a RAC environment as a shared-nothing architecture for queries (although it is much more flexible than a true shared-nothing database). IMCUs (i.e. in-memory data) are not shipped between RAC instances so Parallel Query must be used to access the data on other RAC database instances. This makes the distribution of data between IM column stores very important. The goal is to have as even a distribution of data as possible so that all parallel server processes spend approximately the same amount of time scanning data to maximize throughput and minimize response time.

Distribution

How an object is populated into the IM column store in a RAC environment is controlled by the DISTRIBUTE sub-clause. By default, the DISTRIBUTE sub-clause is set to AUTO. This means that Oracle will choose the best way to distribute the object across the available IM column stores using one of the following options, unless the object is so small that it only consists of 1 IMCU, in which case it will reside on just one RAC instance:

- BY ROWID RANGE
- BY PARTITION
- BY SUBPARTITION

The distribution is performed by background processes as a part of the segment population task. The goal of the data distribution is to put an equal amount of data from an object on each RAC instance. If your partition strategy results in a large data skew (one partition is much larger than the others), we recommend you override the default distribution (BY PARTITION) by manually specifying DISTRIBUTE BY ROWID RANGE.

Auto DOP

In a RAC environment when data is distributed across column stores parallel query must be used to access the IMCUs in the column stores on all but the local instance. In order to accomplish this the parallel query coordinator needs to know in which instance's IM column store the IMCUs for each object involved in the query reside. This is what is meant by parallel scans being "affinitized for inmemory" (you may see this in the Notes section of an execution plan).

In order for the parallel query coordinator to allocate parallel query server processes on the other RAC instances the DOP of the query must be greater than or equal to the number of IM column stores involved. There are two ways to ensure the correct DOP. The first is the use of Auto DOP (i.e. the initialization parameter PARALLEL_DEGREE_POLICY set to AUTO) which will ensure that

the cost-based DOP calculation will be greater than or equal to the number of IM column store instances. The second relies on the user application 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 disk/buffer cache.

12.1.0.2 Behavior

In 12.1.0.2 Auto DOP was required in order to guarantee that the degree of parallelism (DOP) chosen would result in at least one parallel server process being allocated for each active instance, and to enable access to the map of the IMCU home locations. There was no workaround, and it was required that Auto DOP be invoked with the PARALLEL_DEGREE_POLICY parameter specified at either the instance or session level. In addition to this restriction, if an IMCU is accessed from an instance in which it doesn't exist then that IMCU's database blocks will be marked as invalid and will not be accessed from the IM column store on any instance.

IMPLEMENTATION

Strategy

As mentioned previously it is imperative that performance baselines be established at critical phases of the implementation. If upgrading from a previous version of Oracle Database then a baseline prior to the upgrade will establish that application performance is at least as good as it was before the upgrade. You do not want to use Database In-Memory to mask a flaw in an upgrade or a poorly performing system. Once the database is at the proper version and patch level then a baseline should again be taken so that it can be compared to the performance of the application after Database In-Memory has been enabled. This will then show just the benefits of Database In-Memory to the application's performance. It is also important to recognize that Database In-Memory is more than just the enhanced columnar format. The Optimizer can take advantage of additional execution plan features that are only enabled with Database In-Memory.

Performance History

Verify that AWR is running and is available for troubleshooting. This can also be useful for verifying initialization parameters and any other anomalies. If this is an implementation from an existing system then AWR information from the existing system can be used as a baseline to compare system workload differences. Be aware that the default retention period for AWR is only 7 days so this should be extended to cover the PoC or implementation period.

Process

In order to implement Database In-Memory, it is our recommendation that the following four-step process outlined below be followed. This process was developed in conjunction with the Optimizer team and will help ensure that no SQL performance regressions occur due to plan changes with the introduction of Database In-Memory or database upgrades, and that plan improvements can be incorporated in a manageable way. This will also provide the lowest risk so that any possibility of surprises is minimized.

Step 1: Run the workload without Database In-Memory

This step may be a two-part process if the database is being upgraded. The goal of this step is to establish a baseline for application performance prior to implementing Database In-Memory. If upgrading the database then a baseline should be taken prior to the upgrade to confirm that no performance regressions have occurred due to the upgrade. If there have been regressions, then stop and figure out why. Once performance has been established to be acceptable (i.e. as good or better than prior to the upgrade) then testing of Database In-Memory can begin.

Tasks:

- Optional: Turn on SQL Plan Baseline capture or plan to capture plans manually
- Run the workload
  Capture success criteria measurements. Whatever the success criteria, it is usually not sufficient to run a one user, single execution test. We strongly recommend ensuring that SQL statements have reached a "steady state". That is, plans are not changing and repeated executions have consistent timing.

  Note: If using SQL Plan Baselines then each SQL statement must be run at least twice in order for a plan to be captured automatically.
- Optional: Turn off SQL Plan Baseline capture if enabled

- Optional: Drop unrelated SQL plan baselines

Step 2: Enable Database In-Memory

Enable Database In-Memory by setting the INMEMORY_SIZE initialization parameter and restarting the database. Don't forget to evaluate the overall SGA size and account for the increased size required by the IM column store.

Step 3: Populate Tables In-Memory

Populate the tables identified for the Database In-Memory workload into the IM column store and wait for population to complete. It is important that all tables are fully populated into the IM column store(s). This can be verified by querying the V$IM_SEGMENTS view (GV$IM_SEGMENTS on RAC) and verifying the BYTES_NOT_POPULATED column.

On single instance databases - BYTES_NOT_POPULATED should equal 0.

On RAC databases - BYTES_NOT_POPULATED will not equal 0 for objects that are distributed across IM column stores. You can account for this by calculating the SUM(BYTES - BYTES_NOT_POPULATED) for all instances, the totals should equal the total BYTES for the segment. See Figure 2 for an example.

```
SQL> @racim_sum
   INST_ID SEGMENT_NAME              BYTES BYTES_POPULATED BYTES_NOT_POPULATED INMEMORY_SIZE
---------- -------------------- ---------- --------------- ------------------- -------------
         1 LINEORDER             730750976       299892736           430858240     236584960
         2                       730750976       199999488           530751488     157745152
         3                       730750976       230858752           499892224     181927936
           ********************              ---------------                     -------------
           sum                                     730750976                         576258048
SQL>
```

Figure 2. RAC column store population

Step 4: Run the Workload with Database In-Memory

The same workload that was run in Step 1 should be run again, but now with the IM column store fully populated. We are now expecting that execution plans will change and will reflect the use of in-memory execution paths where the optimizer has determined that the cost is less to access the object(s) in the IM column store. If SQL Plan Baselines are enabled then any new plan changes will be captured but will not be used until they are accepted.

Tasks:

- Optional: Turn on SQL Plan Baseline capture or plan to capture plans manually
- Run the workload
  Capture success criteria measurements. Whatever the success criteria, it is usually not sufficient to run a one user, single execution test. We strongly recommend ensuring that SQL statements have reached a "steady state". That is, plans are not changing and repeated executions have consistent timing.
- Optional: Turn off SQL Plan Baseline capture if enabled
- Optional: Verify any newly captured SQL plan baselines
- Optional: Evolve SQL Plan Baselines
- Optional: Re-run workload with accepted SQL Plan Baselines
- Verify execution times. For any SQL statements that do not improve, or regress, consider analyzing the differences with SQL Monitor active reports or another time based optimization tool. See the next section "Identifying In-Memory Usage".

IDENTIFYING IN-MEMORY USAGE

We know that Database In-Memory helps in three key areas of an execution plan: data access, joins and group-by aggregations. The following will show how to determine if Database In-Memory helped speed up a query. The examples will use SQL Monitor active reports to highlight how to determine whether Database In-Memory affected the SQL execution.

Scans

The following SQL will list the total number of orders and the total value of merchandise shipped by air:

```
SELECT COUNT(*),
```

```
       SUM(l.lo_ordtotalprice)
FROM   lineorder l
WHERE  l.lo_shipmode = 'AIR'
```

A traditional execution plan looks like the following:

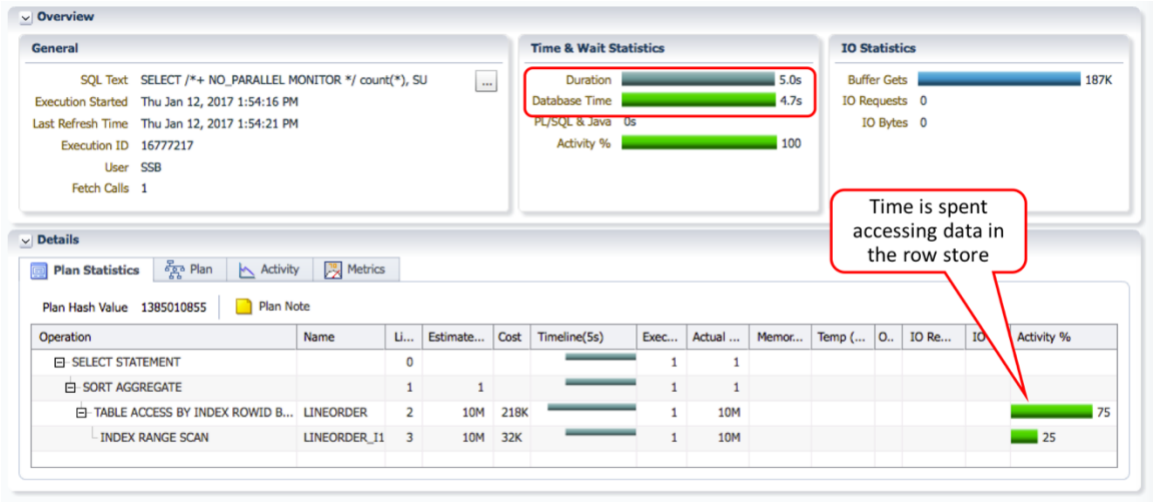

Figure 3. SQL Monitor Report for row store scan

Note that the majority of the execution time is spent accessing data. Specifically, an index scan and table access on the LINEORDER table for a total execution time of 5.0 seconds. Now let's look at what happens when we access the same table in the IM column store:

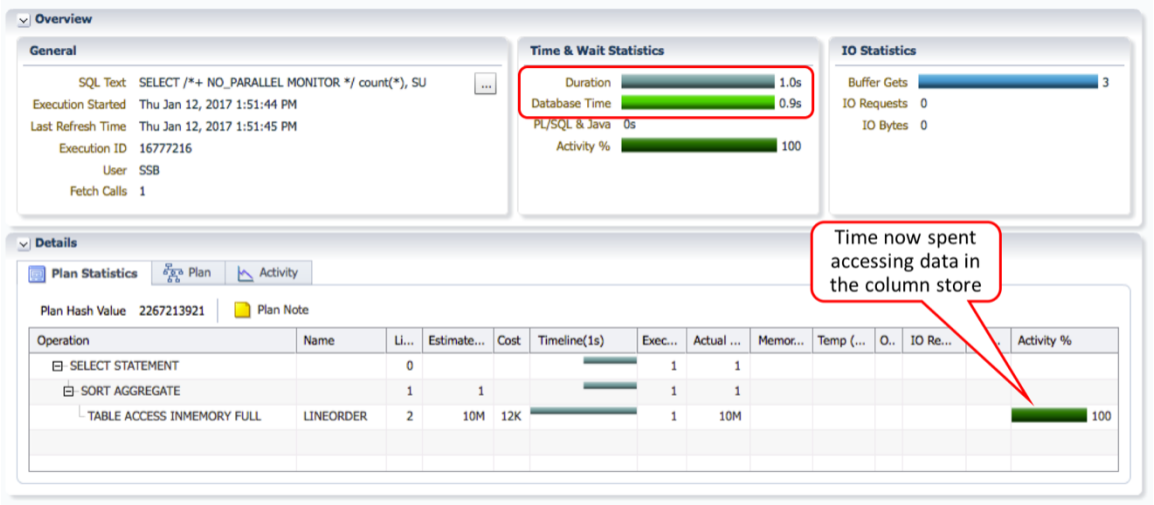

Figure 4. SQL Monitor Report for In-Memory Scan

We see that the query now spends the majority of its time accessing the LINEORDER table in-memory and the execution time has dropped to just 1.0 seconds. Also note that the color of the activity bar has changed to reflect in-memory CPU usage (see SQL Monitor active reports for more details on how CPU time is differentiated).

Joins

Now let's look at how Database In-Memory can optimize joins. The following SQL will show total revenue by brand:

```
SELECT   p.p_brand1,
         (lo_revenue) rev
FROM     lineorder l,
         part p,
         supplier s
WHERE    l.lo_partkey = p.p_partkey
AND      l.lo_suppkey = s.s_suppkey
```

```
AND       p.p_category = 'MFGR#12'
AND       s.s_region   = 'AMERICA'
GROUP BY p.p_brand1
```

The query will access the tables in-memory, but will perform a traditional hash join. Note that the majority of the time spent for this query, 8.0 seconds, is spent in the hash join at line 4:

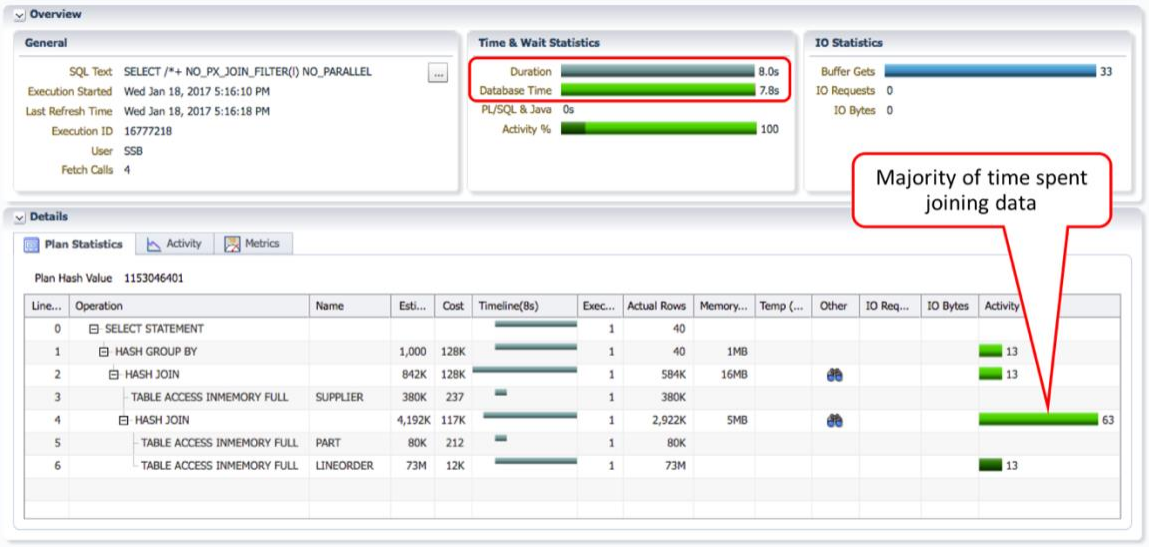

Figure 5. SQL Monitor Report for Join - No Bloom Filter

Now let's take a look at the same query when we let Database In-Memory use a Bloom filter to effectively turn a hash join into a scan and filter operation:

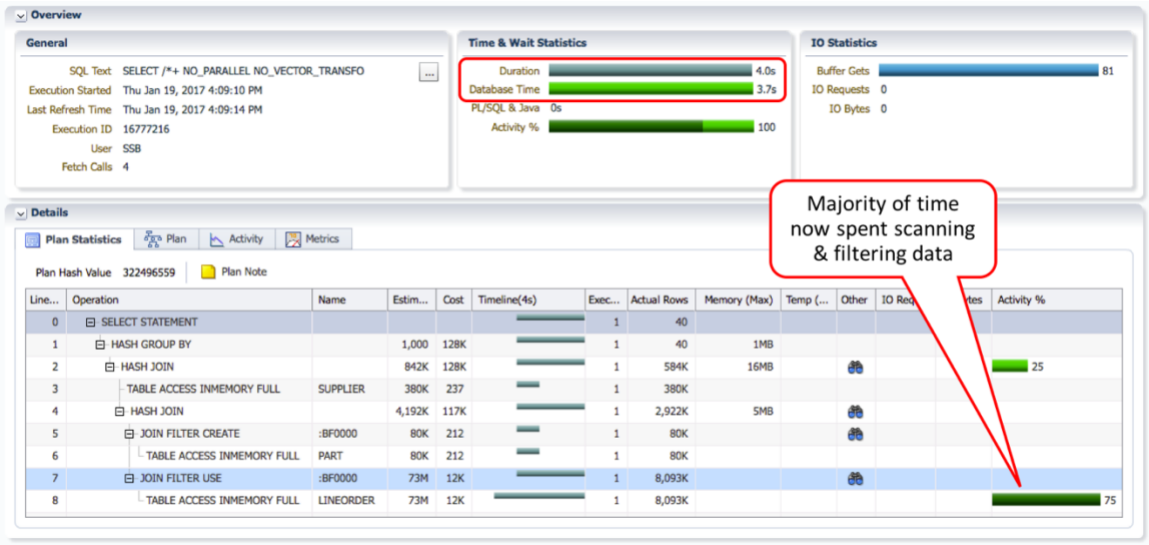

Figure 6. SQL Monitor Report for Join - With Bloom Filter

We see that now the majority of the query time is spent accessing the LINEORDER table in-memory and uses a Bloom filter. The Bloom filter (:BF0000) is created immediately after the scan of the PART table completes (line 5). The Bloom filter is then applied as part of the in-memory scan of the LINEORDER table (lines 7 & 8). The query now runs in just 4.0 seconds. More information about how Bloom filters are used by Database In-Memory can be found in the Oracle Database In-Memory whitepaper.

In-Memory Aggregation

The following query is a little more complex and will show the total profit by year and nation:

```
SELECT   d.d_year, c.c_nation, SUM(lo_revenue - lo_supplycost)
FROM     LINEORDER l, DATE_DIM d, PART p, SUPPLIER s, CUSTOMER c
```

```
WHERE     l.lo_orderdate = d.d_datekey
AND       l.lo_partkey  = p.p_partkey
AND       l.lo_suppkey  = s.s_suppkey
AND       l.lo_custkey  = c.c_custkey
AND       s.s_region    = 'AMERICA'
AND       c.c_region    = 'AMERICA'
GROUP BY d.d_year, c.c_nation
ORDER BY d.d_year, c.c_nation
```

The following shows a traditional group by even though all but one of the tables being queried are populated in the IM column store:

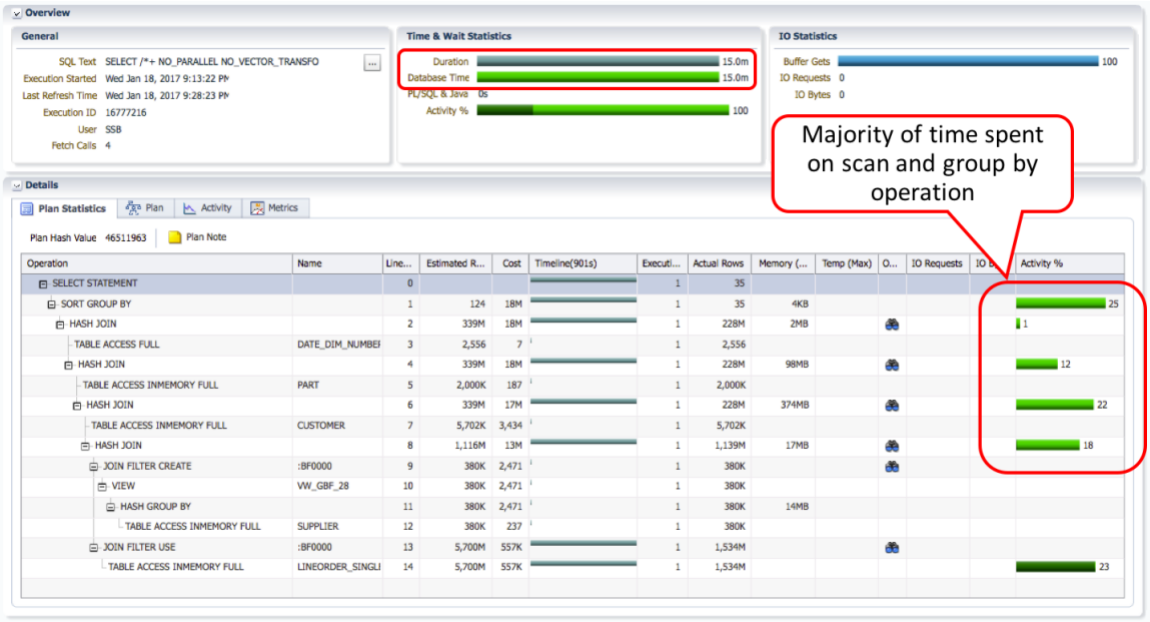

Figure 7. SQL Monitor Report with no In-Memory Aggregation

Note that the majority of the time is spent in the scan and group by operations and the total run time is 15.0 minutes. Note that the run time is much longer than in the previous examples because we have switched to a much larger data set.

Next let's look at how the query runs when we enable In-Memory Aggregation:

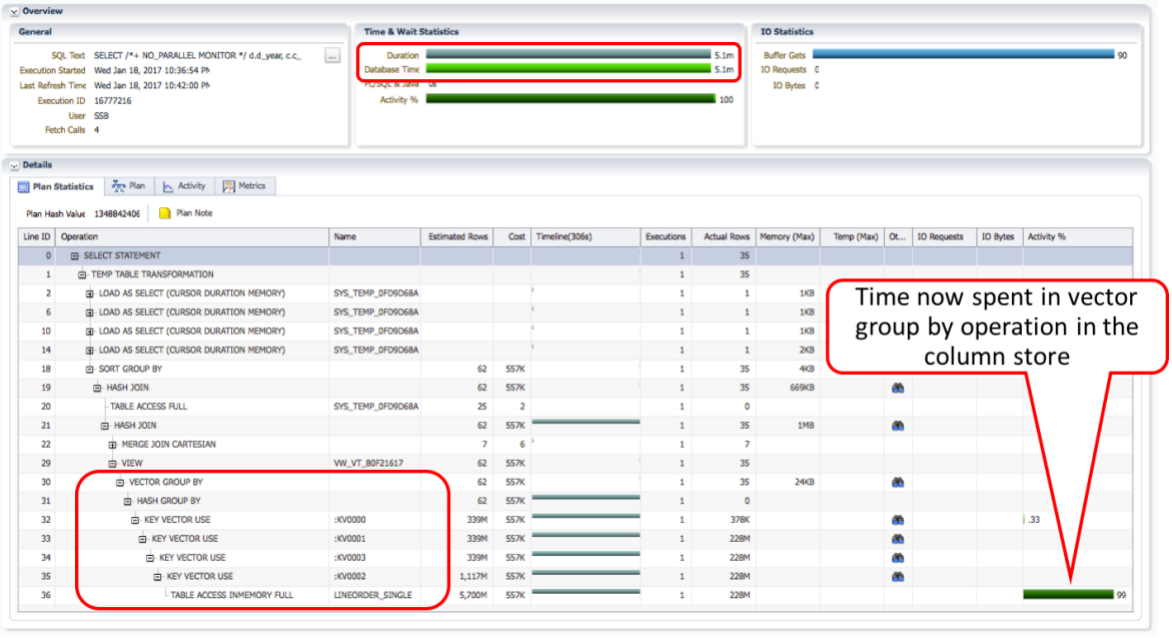

Figure 8. SQL Monitor Report with In-Memory Aggregation

Note that now the majority of the time is spent accessing the LINEORDER_SINGLE table and that four key vectors are applied as part of the scan of the LINEORDER_SINGLE table within a VECTOR GROUP BY operation that starts at line 30. We also see that the total run time is now only 5.1 minutes rather than 15.0 minutes. To investigate more about In-Memory Aggregation see the In-Memory Aggregation white paper.

SUMMARY

Oracle Database In-Memory can typically provide a 10x or better performance improvement for analytical style queries, but it is not a one size fits all option. The application workload must be analytical in nature. By analytical we mean that it asks "What if ..." type questions that can be answered by using aggregation to scan and filter data. Optimally Database In-Memory will perform aggregations, scans and filtering in one pass over the data.

Database In-Memory is extremely simple to implement. No application code has to change, and other than allocating additional memory and identifying objects to populate into the IM column store, no other changes have to be made to the database or the application SQL. However, Oracle Database is complex with lots of features and options to handle all sorts of application tasks. Database In-Memory leverages many of these features to provide outstanding performance improvements. Just because application SQL will run unchanged does not mean that it will leverage all of the features of Database In-Memory. There are many things that can be done at the application level that will not allow Oracle Database to fully leverage the benefits of Database In-Memory. The most obvious example would be hints that disable the ability to access the IM column store (e.g. index hints).

This paper has provided some basic quick start guidelines to get you started with a minimal amount of effort and the best chance for success. It has also included examples to show how you can determine whether you benefitted from the enhancements that Database In-Memory provides. However, this is a quick start guide and not a comprehensive document on how to implement Database In-Memory.

CONNECT WITH US

Call +1.800.ORACLE1 or visit oracle.com.
Outside North America, find your local office at oracle.com/contact.

blogs.oracle.com facebook.com/oracle twitter.com/oracle

Oracle Database In-Memory Quick Start Guide
May, 2021
Author: Andy Rivenes