ORACLE

Oracle Database In-Memory Implementation Guidelines

September 2025 | Version 2.4

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

TABLE OF CONTENTS

INTRODUCTION

Oracle Database In-Memory (Database In-Memory) was introduced in Oracle Database 12c Release 1 (12.1.0.2) and is available as part of Oracle Database Enterprise Edition on-premises, Oracle Cloud and Cloud at Customer. It adds in-memory columnar 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) processing and a new columnar format for super-fast analytic reporting.

Since Database In-Memory is fully integrated into Oracle Database, the Oracle optimizer is fully aware of the In-Memory column store (IM column store) and transparently takes advantage of the superior analytic capabilities of the columnar formatted data for analytic queries. It accomplishes this based on query cost, just as it does for row-store based queries. This allows application SQL to run unchanged with Database In-Memory. Since the IM column store is a pure, in-memory structure and no on-disk formats have been changed, all existing Oracle Database features continue to be fully supported with Database In-Memory.

This technical brief provides information to help ensure successful implementations of Database In-Memory. Database In-Memory will accelerate analytic workloads without disruption to existing database environments and this technical brief outlines the tasks necessary for a successful implementation with a minimum amount of "tuning" effort, and the best chance of avoiding any performance regressions. This information is based on our experience of participating in many implementations and working directly with customers and should work well in most situations.

This paper assumes that you are familiar with Database In-Memory fundamentals as outlined in the Oracle Database In-Memory Technical Brief. It is also a more detailed companion to the Database In-Memory Quick Start Guide.

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 most of their time scanning and filtering data see the most benefit. Queries that spend most 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 In-Memory Aggregation to perform group by aggregations as part of the in-memory scan, Database In-Memory can, in the best cases, perform most 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.

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 most 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 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.

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
- Database 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. 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 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. A more realistic approach requires 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 can realistically be expected? There are limits to performance improvements based on the technology used.

These are the types of questions that can be used as input to determine 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.

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

If measuring success based on application throughput then that throughput must 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 brief. 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 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 compress segments to a higher level based on usage.

To answer the second question, 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 involving SQL Plan Baselines in 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. The IM column store sizing can be done using the In-Memory Advisor and the Compression Advisor which were described earlier, 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 In-Memory compression unit (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. This will be discussed in more detail in the topic on Auto DOP in the Parallelism section later in this document.

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 (see MOS Note 2070261.1: Is There A Way To Allocate More Than 2Gb For Large Hash Joins Per Process?), 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 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, and 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 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.

Not 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).

INMEMORY_FORCE - this parameter can be used to enable the Base Level feature or enable only Cell Memory and not allocate the IM column store (this is for Exadata only).

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.

Linux Memory Settings

Since most databases utilizing Database In-Memory will have large memory footprints it is likely that they will benefit from the use of HugePages if on Linux. The following are two helpful MOS notes avaialble for HugePages:

- HugePages on Oracle Linux 64-bit (MOS Note: 361468.1)
- HugePages on Linux: What It Is... and What It Is Not... (MOS Note: 361323.1)

For additional database specific information, we recommend that you consult the appropriate installation and/or administration guides for your platform and My Oracle Support (MOS) note(s) for any updates.

Memory Allocation in Multitenant Databases

Oracle Multitenant is a new database consolidation model 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 system global area (SGA) and background processes of a common CDB.

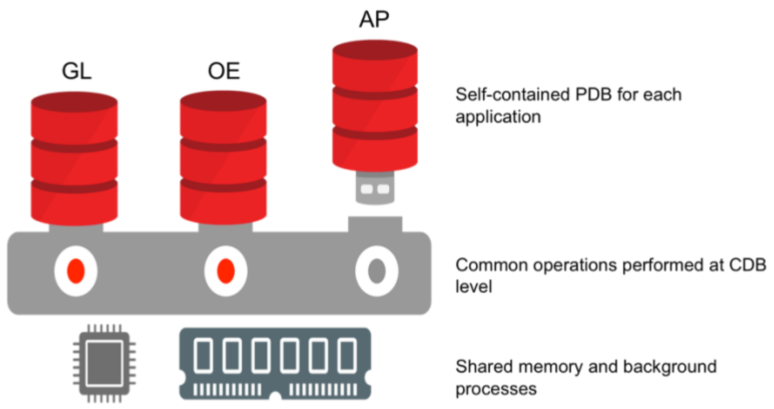

AP
GL
OE
Self-contained PDB for each application
Common operations performed at CDB level
Shared memory and background processes

Figure 2. Multitenant PDBs

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 and has the potential to fully populate it and starve its fellow PDBs. To avoid starving any of the PDBs, you can specify how much of the shared IM column store a PDB can use by setting the INMEMORY_SIZE parameter inside the specific PDB using the following command:

```
ALTER SYSTEM SET INMEMORY_SIZE = 4G CONTAINER = CURRENT SCOPE = spfile
```

Not all PDBs in each CDB 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. The following shows an example with three PDBs:

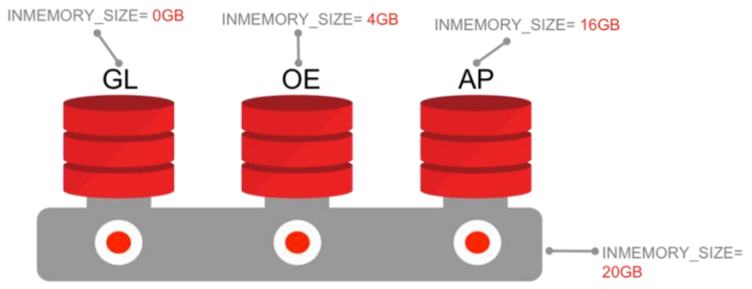


Figure 3. Multitenant In-Memory Allocation

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 possible as this will significantly affect the performance of the application. You also need to consider 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, to gain memory for the IM column store you do not want to "steal" memory from other components of the SGA (i.e., buffer cache, 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 this document. 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 INMEMORY_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 information 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 current data will 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 all partitions will inherit the default attribute of the table and will be populated, including all newly created partitions. However, if 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 created to populate or evict individual partitions/sub-partitions.

APPLICATION ARCHITECTURE

Database In-Memory is highly dependent on the Oracle optimizer's ability to generate optimal execution plans to achieve good performance. It is important to supply the optimizer with the best information possible about the database objects involved in SQL queries accessing the IM column store. We recommend you follow Oracle best practices for understanding optimizer statistics and schema design and review the following topics for the application schema(s).

Review Statistics

It is critical that optimizer statistics be representative for the objects involved in the queries being accessed in conjunction with the IM column store. By default, statistics are collected automatically for all objects that are missing statistics or that have stale statistics. This may need to be addressed if you find large cardinality misestimates for problem SQL.

How and when to gather statistics can be very workload dependent. You should also not re-gather statistics in between testing runs. This can easily prevent an apples-to-apples comparison.

More information about optimizer statistics can be found in Best Practices for Gathering Optimizer Statistics with Oracle Database 19c.

Review Constraint Definitions

The optimizer can use constraint definitions to help determine the optimal execution plan. Even if the system is a data warehouse, constraints are still an important method for the optimizer to determine relationships between the tables being queried. There is the option to use a combination of DISABLE, NOVALIDATE and RELY constraints to help alleviate some of the performance objections that may be issues in data warehousing systems. Of course, the use of these kinds of techniques does shift the responsiblity for data integrity to the ETL process or application. See the Database Data Warehousing Guide for more detailed information.

Review Partitioning

Partitioning can provide a huge performance benefit by allowing the optimizer to perform partition pruning, effectively eliminating the need to access data that is not needed to satisfy the query. Partitioning can also be thought of as a divide and conquer strategy making it much easier to manage very large data sets. It allows the data architect to place only the most performance sensitive data into the IM column store. In many cases, it is only the most recent data that is the subject of most analytic queries since it is the most recent data that drives business decisions.

Review Indexing

Ideally, analytic reporting indexes can be dropped when using Database In-Memory, but it can often be hard to determine which indexes are reporting indexes and which indexes are being used for OLTP type queries. Making indexes invisible is a safer approach than just dropping them. If they're invisible, then the optimizer won't use them even though they are being maintained, and if they are needed again it is simple to alter them to make them visible. If after running through one or two business cycles with the indexes invisible, and if no performance problems occur, then the indexes can be safely dropped.

Review SQL Techniques That Prevent The Use of Database In-Memory

There are SQL techiniques that applications have used in the past to optimize queires for row-store access that can prevent the optimizer from choosing an inmemory plan or can prevent the optimizer from picking the most optimal inmemory plan. An obvious issue is with hints, for example requiring the use of an index. Remember that the IM column store is accessed with a TABLE ACCESS INMEMORY FULL, and that hints in most cases are directives, not suggestions. There are other techniques as well and following sections will list the ones that we have come across. This is by no means an exhasustive list but is meant to provide things to look for.

Hints

- Hints that force the optimizer to choose a join method (for example: USE_NL)
- Hints that force the optimizer to use an index (for example: INDEX, INDEX_*)
- Hints that result in the creation of temporary tables (for example: MATERIALIZE)
- Hints that disable In-Memory access (for example: NO_INMEMORY)
- Hints that disable features (for example: NO_PARALLEL, NO_REWRITE)

Other Practices

- Subquery factoring (WITH clause) – Can prevent the full use of Database In-Memory techniques. Since the purpose is to materialize a row source, that row source cannot leverage Database In-Memory features like hash joins with Bloom filters or vector group by when accessing other data. A materialzed view, which can be populated into the IM column store, with query rewrite may be a better choice.
- Use of Function-based Indexes – Database In-Memory does not populate function-based indexes, and if the optimizer chooses a function-based index then it must be evaluated after the data scan. This will prevent predicate push down which may be a more efficient choice.
- Common views – Applications often use a set of "common" views to make it easier to access various commonly used attributes. Unfortunately, many times these common views combine and access a lot more data and/or tables than is needed for any single query. This can cause the access of much more data than is needed for the query and will typically be slower.
- Nested Views – Nested views, that is views built on top of other views, can be difficult for the optimizer to efficiently unnest. Nested views often suffer from the same issues that common views have, that is accessing more data and/or tables than needed. You can use `DBMS_UTILITY.EXPAND_SQL_TEXT` to unwind nested views.
- Correlation issues – Attributes that have correlation can cause the optimizer to come up with the wrong cardinalities because it won't know about the logical relationships between column values. Extended statistics can be used define these relationships and help the optimizer calculate more representative cardinalities.

PARALLELISM

Parallel Query

There are multiple forms of parallelism in Oracle Database, for example parallel query, parallel DML and parallel DDL. Database In-Memory can leverage parallel query directly as part of a `TABLE ACCESS INMEMORY FULL` operation for both single instance databases and RAC databases.

Database In-Memory effectively eliminates I/O from the execution plan since queries that access objects that are fully populated in the IM column store access those objects in-memory. The response time of the query is limited only by the number of CPUs and the speed of memory access. This makes Database In-Memory ideally suited to take advantage of parallelism to further increase query performance.

In Data Warehouse environments, the use of parallel execution is common and generally well understood. However, in OLTP and mixed workload environments parallel execution is not as commonly used. From our experience, this is usually rooted in the fear that parallel execution will overwhelm CPU resources and cause performance problems and resource contention. Fortunately, there are strategies that can be put into place to address these concerns and allow mixed workload environments to take advantage of parallel execution. For detailed information about Oracle Database parallel execution see the white paper Parallel Execution with Oracle Database.

Auto DOP with Parallel Query

Auto DOP allows the optimizer to automatically decide the degree of parallelism for all SQL statements. This is Oracle's recommended way to control parallel query. Setting the initialization parameter PARALLEL_DEGREE_POLICY to AUTO enables

Auto DOP and allows the optimizer to determine whether the statement should run in parallel based on the cost of running the statement.

There are two other parameters that affect parallel query as well, PARALLEL_MIN_TIME_THRESHOLD and PARALLEL_DEGREE_LIMIT. These parameters ensure that trivial statements are not parallelized and that database resources are not overused.

By default, the value of PARALLEL_MIN_TIME_THRESHOLD is AUTO, which sets the parameter to 10 seconds, but if all the objects in the query are populated in the IM column store then the parameter is reset to 1 second. If the optimizer estimates that the execution time for the statement will meet or exceed this threshold then the statement is considered for automatic degree of parallelism. This parameter can be set manually at the system or session level and a setting of 0 will enable parallel query for all queries. This will be discussed in more detail in the RAC section.

The actual DOP will be calculated based on the cost of all statement operations. The parameter PARALLEL_DEGREE_LIMIT will limit the DOP and can be set to a system-wide upper limit. If Auto DOP is not used (i.e., PARALLEL_DEGREE_POLICY is set to MANUAL) then the DOP can be set at the object level, with a statement level hint, or with an object level hint. For more detailed information see the white paper Parallel Execution with Oracle Database.

Resource Manager

Oracle Database Resource Manager (Resource Manager) can be used to manage fine-grained control of the maximum DOP in addition to a cap on CPU utilization, prioritize work and restrict access to resources for groups of users. This is the recommended way to manage resource usage when using parallel execution. See the white paper Using Oracle Database Resource Manager for more information.

Resource Manager also works with In-Memory Dynamic Scans (IMDS) to automatically parallelize IMCU scans when there is available CPU capacity, and the CPU count is greater than 24.

Resource Manager can also be used to control resource usage during Database In-Memory population by setting up a 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 following is an example of assigning In-Memory populate servers to a specific resource manager consumer group:

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

Figure 4. Resource Manager Syntax

After invoking this code, the populate servers will be assigned to the BATCH_GROUP consumer group. This will limit the CPU utilization to the value specified by the plan directive for the BATCH_GROUP consumer group as specified in the resource plan.

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 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). Database In-Memory does not use RAC Cache Fusion. IMCUs (i.e., in-memory data) are not shipped between RAC instances so Parallel Query must be used to access in-memory data on other RAC database instances. This makes the distribution of data between IM column stores very important. The goal is to have an even distribution of data 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:

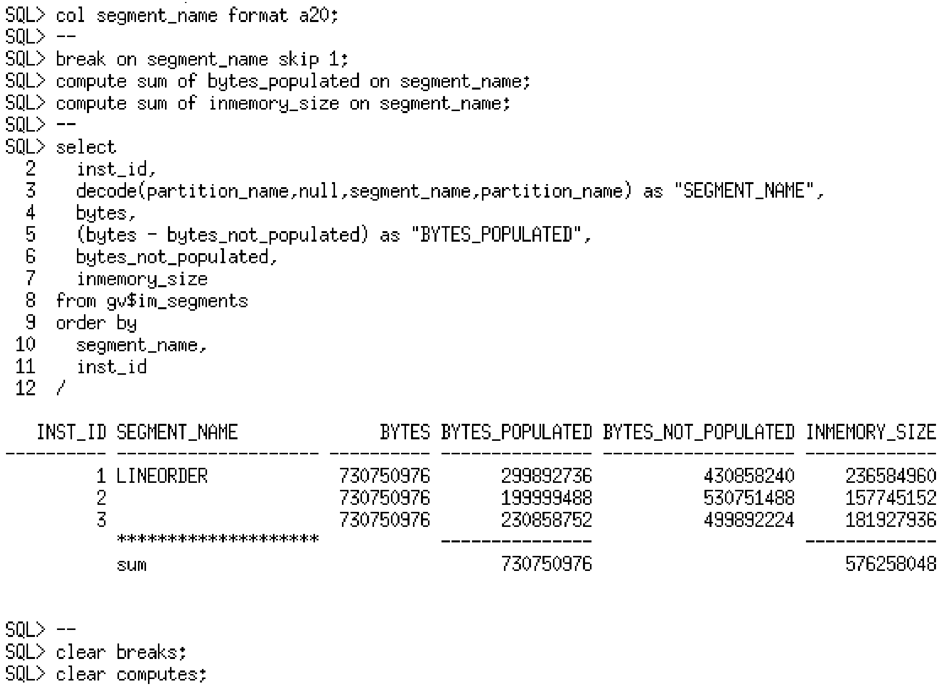
- BY ROWID RANGE
- BY PARTITION
- BY SUBPARTITION

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.

Verifying Population

In a RAC environment the gv$im_segments view can be misleading. In the gv$im_segments view, the BYTES_NOT_POPULATED field represents the bytes not populated on each instance. Therefore, in a RAC database with multiple IM column stores this field will not accurately represent the total bytes not populated. For objects that are distributed across multiple RAC instances the BYTES_NOT_POPULATED field will not be 0 when an object is fully populated, and it can be difficult to determine if objects are indeed fully populated.

The following query shows one way to determine the population of an object across a 3 node RAC database:



```
SQL> col segment_name format a20;
SQL> --
SQL> break on segment_name skip 1;
SQL> compute sum of bytes_populated on segment_name;
SQL> compute sum of inmemory_size on segment_name;
SQL> --
SQL> select
  2    inst_id,
  3    decode(partition_name,null,segment_name,partition_name) as "SEGMENT_NAME",
  4    bytes,
  5    (bytes - bytes_not_populated) as "BYTES_POPULATED",
  6    bytes_not_populated,
  7    inmemory_size
  8  from gv$im_segments
  9  order by
 10    segment_name,
 11    inst_id
 12  /

   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> --
SQL> clear breaks;
SQL> clear computes;
```

Figure 5. RAC GV$IM_SEGMENTS

Note that the `BYTES_NOT_POPULATED` field is non-zero for each instance but the sum of the `BYTES_POPULATED` field equals the total size of the object, so it is fully populated. To further illustrate, the sum of the `BYTES POPULATED` field for instances 2 and 3 totals the size of the `BYTES_NOT_POPULATED` field for instance 1: `199999488 + 230858752 = 430858240`.

Also note that the sum of the `INMEMORY_SIZE` field gives us the total size of the object in the IM column store.

Auto DOP with RAC

As stated earlier, 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. 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). This IMCU location awareness is referred to as a home location map of IMCUs and is kept in the shared pool for each instance with a column store allocated (see the Memory Allocation section earlier in this document).

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 manually setting the DOP of the query greater than or equal to the number of IM column stores involved as mentioned in the previous AutoDOP with Parallel Query section. 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 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 that 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.

Services

Services can be used to control node affinity for scheduler jobs as well as enabling application partitioning. They can also be used as a form of connection management. This can allow groups of connections or applications to be directed to a subset of nodes in the cluster.

It is this ability to direct workloads to a subset of nodes that can allow Database In-Memory to be run on just a subset of nodes in a RAC cluster.

Setting Up Independent In-Memory Column Stores

It is possible to set up independent IM column stores on a RAC cluster using services. This might be a good idea if you are trying to enforce application affinity at the node level and don't want to allow inter-instance parallelism.

This technique can be used to allow the running of all the workload for a given application, along with any Database In-Memory workload to support that application, on just a single node in a RAC environment and still make use of parallel query. This technique can also be used to support multiple "independent" IM column stores if, for example, you wanted to partition multiple applications across a subset of nodes in your RAC environment.

The one caveat is that you cannot use this to "duplicate" objects between IM column stores. Duplication is only supported on Engineered Systems (i.e., DUPLICATE sub-clause).

In 12.2 a new FOR SERVICE option was added to the DISTRIBUTE sub-clause to address the ability to support directing objects to specific IM column stores. The primary use of the DISTRIUTE FOR SERVICE subclause is to manage in-memory population when using Active Data Guard. However, a secondary use is to make the process of populating objects in specific IM column stores much easier.

With the DISTRIBUTE FOR SERVICE subclause you can specify a service for each object. This provides much more flexibility and ease of use for directing where objects will get populated. The following example shows the use of the FOR SERVICE clause for the SALES table:

```
ALTER TABLE sales INMEMORY DISTRIBUTE AUTO FOR SERVICE dbim1
```

Figure 6. DISTRIBUTE FOR SERVICE sub-clause

The sales table will now be eligible to be populated in IM column stores that reside on RAC nodes that have the service "dbim1" defined and active.

12.1.0.2 Behavior

Although it was possible to set up independent IM column stores on a RAC cluster running Oracle Database 12.1.0.2 it was not a designed feature and requires quite a bit of setup with services, parallel query settings and application-level connection management to make it work.

In addition, after working with customers trying to implement this approach we have discovered that if you cannot guarantee that objects that are populated will only be accessed by the instances that are part of the defined parallel instance group then you shouldn't attempt it. Based on these experiences we now recommend that if you need to set up independent IM column stores then you should use the DISTRIBUTE FOR SERVICE sub-clause that was introduced starting in Oracle Database 12.2.

PLAN STABILITY

Adaptive Features

Adaptive query optimization enables the optimizer to make run-time adjustments to execution plans. This can require multiple executions of a SQL statement before the execution plan(s) stabilizes. It is therefore strongly recommended that you ensure that all SQL execution plans have stabilized before running timing tests. This will more accurately reflect how the SQL will execute on a running system. The following sections will try to point out how to determine that SQL execution plans are in a “steady state” or no longer in a state of change.

Changes to OPTIMIZER_ADAPTIVE_FEATURES

In Oracle Database Release 12.2 the parameter `OPTIMIZER_ADAPTIVE_FEATURES` was made obsolete and replaced with two new parameters: `OPTIMIZER_ADAPTIVE_PLANS` and `OPTIMIZER_ADAPTIVE_STATISTICS`.

If you are still running on Oracle Database Release 12.1.02 then it is strongly recommended that you apply patches 21171382 and 22652097 and use the new 12.2 parameters. This should solve most "adaptive" problems that were caused by the `OPTIMIZER_ADAPTIVE_FEATURES` parameter.

Adaptive Plans

Adaptive plans allow the optimizer to decide on the optimal join method, parallel distribution method or bitmap index pruning at execution time. Adaptive plans are stored in the child cursor so that subsequent executions of the statement can use it. Unless the current plan ages out of the shared pool or a different optimization feature invalidates the current plan the adaptive plan will be used. In the Notes section of the execution plan there will be an entry indicating that:

```
- this is an adaptive plan
```

If this note exists it is indicative that the plan being used has been adapted, not that it is continuing to change. This sometimes causes confusion. If the plan is adapted again, and recall that this happens only in special cases, it will again only happen once. For this reason, adaptive plans can usually be ignored as they have little to no impact on plan stability.

To identify which SQL statements have adapted the `IS_RESOLVED_ADAPTIVE_PLAN` column in the `V$SQL` view can be queried. If it is `NULL` then the plan was not adaptive and if '`Y`' then it has been adapted and resolved. Note that it is only possible to have a '`N`' value during the initial execution of the query.

OPTIMIZER_ADAPTIVE_PLANS

The `OPTIMIZER_ADAPTIVE_PLANS` parameter controls adaptive plans and should be left at the default value of `TRUE`.

Adaptive Statistics

Adaptive statistics has caused many customers a lot of problems in terms of plan stability. The purpose of adaptive statistics is to supplement base table statistics for query predicates that cause cardinality misestimates. In practice however, they can be a source of instability while the optimizer tries to find a more optimal execution plan.

The default value is FALSE, which for most customers is the right choice. This will prevent the use of SQL plan directives, statistics feedback for joins and adaptive dynamic sampling for parallel execution.

OPTIMIZER_ADAPTIVE_STATISTICS

The parameter `OPTIMIZER_ADAPTIVE_STATISTICS` controls adaptive statistics and should be left at the default value of `FALSE`.

Statistics Feedback

Statistics feedback is part of automatic reoptimization. This is a feature that can significantly affect plan stability. Even with `OPTIMIZER_ADAPTIVE_STATISTICS` set to `FALSE` statistics feedback will still be used for single-table cardinality misestimates. You may also see SQL Plan Directives created but they will not be used.

If statistics feedback occurs then in the Notes section of the execution plan there will be an entry indicating:

```
- statistics feedback used for this statement
```

It is possible to tell whether a SQL statement will be reoptimized by looking at the column `IS_REOPTIMIZABLE` in the `V$SQL` view. If the column value is `'Y'` then the next execution of the child cursor will trigger a reoptimization. Once this column value is set to `'N'` then the child cursor will no longer trigger reoptimizations.

For PoC or implementation benchmarks it is recommended that you verify that the `IS_REOPTIMIZABLE` column in the `V$SQL` view has a value of 'N' for all key SQL statement cursors. This will help ensure that the SQL being run is at a steady state in terms of execution plan(s) and that consistent execution times can be achieved.

```
SQL_ID         CHILD_NUMBER  EXECUTIONS  PLAN_HASH_VALUE IS_REOPTIMIZABLE
------------- ------------- ----------- ---------------- ----------------
czrxzt1nt4vpg             0           1       2827393718 Y
czrxzt1nt4vpg             1           1       1232675861 Y
czrxzt1nt4vpg             2           1       1232675861 Y
czrxzt1nt4vpg             3           4       2827393718 N
```

Figure 7. Statistics Feedback

In the example query from the V$SQL view in Figure 7 above we can see that it took 4 tries before the optimizer marked child cursor number 3 not reoptimizable. All subsequent executions of this SQL statement will use this last child cursor and the plan is now stable from a statistics feedback perspective.

Adaptive Cursor Sharing

Bind Variables and Histograms

For applications that make use of bind variables the optimizer can use different plans based on the data distribution of the different bind values. Although histograms are not required for this to happen in most cases they are used to determine whether a statement will be marked bind sensitive. See Adaptive Cursor Sharing (ACS) and Bind Sensitivity for a more detailed explanation.

Once a bind-sensitive cursor has been made bind-aware the optimizer chooses plans for future executions based on the bind value and its cardinality benefit. This can result in multiple child cursors. The optimizer will mark the cursor IS_BIND_AWARE and IS_SHAREABLE in the V$SQL view. It is important to take into consideration the different plans that can then be used to execute the SQL statement based on different bind variable values.

```
SQL_ID        CHILD_NUMBER EXECUTIONS PLAN_HASH_VALUE IS_SHAREABLE IS_BIND_AWARE
------------- ------------ ---------- --------------- ------------ -------------
dbc1a7u3jftr2            0          2      1987843869 N            N
dbc1a7u3jftr2            1          1       509473618 Y            Y
dbc1a7u3jftr2            2          3      1987843869 Y            Y
```

Figure 8. Bind Aware

In the example query from the V$SQL view in Figure 8 we can see that this SQL statement can have two possible execution plans using either child cursor number 1 or 2.

Optimizer Statistics

"In order to select an optimal execution plan the optimizer must have representative statistics. Representative statistics are not necessarily up to the minute statistics but a set of statistics that help the optimizer to determine the correct number of rows it should expect from each operation in the execution plan."[2] This statement makes an important point, and in most cases, if the optimizer can make good cardinality estimates then it will create good execution plans.

[2] Best Practices for Gathering Optimizer Statistics with Oracle Database 19c

The following Oracle Technical Briefs provide a wealth of knowledge about optimizer statistics and how they affect SQL execution plans.

- Best Practices for Gathering Optimizer Statistics with Oracle Database 19c
- Understanding Optimizer Statistics with Oracle Database 19c
- The Optimizer in Oracle Database 19c

Gathering Statistics

All the normal optimizer statistic best practices still apply with Database In-Memory. After all, the optimizer is the key to making use of Database In-Memory and getting the best possible execution plan. The optimizer white paper "Understanding Optimizer Statistics With Oracle Database 19c" provides excellent technical information, and the following guidelines haven't changed:

- Use DBMS_STATS.GATHER_*_STATS procedures to gather statistics.
  - The analyze command has been deprecated since Oracle Database 8i.
  - Use default values as much as possible especially AUTO_SAMPLE_SIZE.
- Use histograms to make the optimizer aware of any data skew.
  - New types of histograms in 12c provide more detailed information.
- Use extended statistics to make the optimizer aware of correlation between columns.
  - Column group statistics are used for both single table cardinality estimates, joins and aggregation.
- Use constraints to indicate not-null, primary key and foreign key columns.

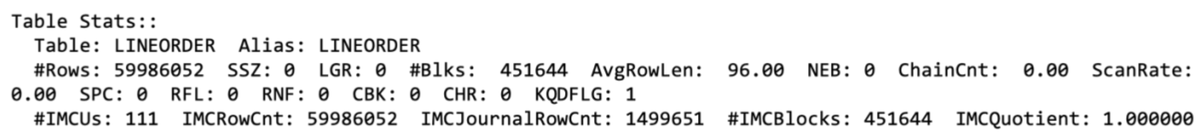
In-Memory Statistics

Unlike segment statistics, which are computed using the `DBMS_STATS` package, Database In-Memory statistics are computed automatically during the hard parse phase for SQL statements that access segments that are enabled for in-memory.

The following statistics are gathered:

- #IMCUs - Number of IMCUs populated
- IMCRowCnt - Number of rows populated
- IMCJournalRowCnt - Value not currently used
- #IMCBlocks - Number of database blocks populated
- IMCQuotient - Fraction of table populated in In-Memory column store, value between 0 and 1

These statistics can be seen in an optimizer trace file:



```
Table Stats::
  Table: LINEORDER  Alias: LINEORDER
  #Rows: 59986052  SSZ: 0  LGR: 0  #Blks:  451644  AvgRowLen:  96.00  NEB: 0  ChainCnt:  0.00  ScanRate:
0.00  SPC: 0  RFL: 0  RNF: 0  CBK: 0  CHR: 0  KQDFLG: 1
  #IMCUs: 111  IMCRowCnt: 59986052  IMCJournalRowCnt: 1499651  #IMCBlocks: 451644  IMCQuotient: 1.000000
```

Figure 9. In-Memory Optimizer Statstics

In-Memory statistics are also RAC-aware (DUPLICATE and DISTRIBUTE) and the optimizer takes into account both I/O cost and CPU cost. Specifically, the following are factored into the cost decision:

- IO cost: Includes the cost of reading:
  - Invalid rows from disk
  - Extent map
- CPU cost:
  - Traversing IMCUs
  - IMCU pruning using storage indexes
  - Decompressing IMCUs
  - Predicate evaluation
  - Stitching rows
  - Scanning transaction journal rows

Starting in Oracle Database Release 12.2 the [ALL|USER|DBA]_TAB_STATISTICS view has three new columns associated with Database In-Memory statistics:

- IM_IMCU_COUNT - Number of IMCUs in the table
- IM_BLOCK_COUNT - Number of In-Memory blocks in the table

- IM_STAT_UPDATE_TIME - The timestamp of the most recent update to the In-Memory statistics

SQL Plan Management

SQL Plan Management is a feature that was introduced in Oracle Database 11g to ensure plan stability by preventing plan changes from occurring unless the new plan is better than the current plan. We strongly recommend that you use SQL Plan Management for your Database In-Memory PoC or implementation to help prevent SQL execution regressions.

SQL Plan Management has three main components:

1. Plan capture:

   Creation of SQL plan baselines that store accepted execution plans for all relevant SQL statements. SQL plan baselines are stored in the SQL Management Base in the SYSAUX tablespace.

2. Plan selection:

   Ensures that only accepted execution plans are used for statements with a SQL plan baseline and records any new execution plans found for a statement as an unaccepted plan in the SQL plan baseline.

3. Plan evolution:

   Evaluate all unaccepted execution plans for a given statement, with only plans that show a performance improvement becoming accepted plans in the SQL plan baseline.

To aid in diagnosing SQL query regressions, we recommend that SQL Plan Baselines be collected for each test run, whether part of an initial PoC, an upgrade to Oracle Database, or a new implementation of Database In-Memory.

For example, if implementing Database In-Memory then a baseline of the test or acceptance workload should be made without Database In-Memory enabled, then with Database In-Memory enabled and finally with all SQL plans fully evolved. The idea is to capture the best possible execution plan(s) and to evolve the best plans into a final run. This should ensure that the best possible performance is obtained with no individual SQL performance regressions.

Disable Automatic Plan Evolution

During upgrades or initial implementations, we recommend turning off automatic plan evolution so that plan acceptance can be run manually at the appropriate time. It can be re-enabled once the upgrade or implementation is complete.

To disable automatic plan evolution:

```
BEGIN
  DBMS_SPM.SET_EVOLVE_TASK_PARAMETER(
    task_name => 'SYS_AUTO_SPM_EVOLVE_TASK',
    parameter => 'ACCEPT_PLANS',
    value     => 'FALSE'
);
END;
```

Figure 10. Evolve Task Syntax

To verify the status:

```
SELECT PARAMETER_NAME, PARAMETER_VALUE AS "VALUE"
FROM   DBA_ADVISOR_PARAMETERS
WHERE  ( (TASK_NAME = 'SYS_AUTO_SPM_EVOLVE_TASK') AND
         ( PARAMETER_NAME = 'ACCEPT_PLANS' ) )
```

Figure 11. Evolve Task Query Syntax

WORKLOAD CAPTURE AND ANALYSIS TOOLS

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. It may also be useful to take additional snapshots before and after each testing period. This can be done using the `DBMS_WORKLOAD_REPOSITORY` package and can be referenced in the MOS Note: How to Create AWR Snapshots Outside the Regular Automatic Intervals? (Doc ID 2100903.1).

Regressions and Problem SQL

For problem SQL statements that result in regressions you will need several additional pieces of information before contacting support.

SQL Monitor Active Reports

SQL Monitor active reports should always be created as a first step for evaluating SQL performance. Ideally a SQL Monitor active report with and without in-memory should be available. One of the key pieces of information in a SQL Monitor active report is the ability to associate how much time was spent executing the various steps of the execution plan. This is not available in any of the other SQL Monitor reports, only in the active report. In addition, the SQL Monitor active report will differentiate CPU time spent accessing objects in-memory versus other CPU time. This is invaluable for evaluating Database In-Memory benefits.

Along with SQL Monitor active reports, an AWR report for the testing time periods, and an optimizer trace, see next section, for each run may be needed as well.

To create a SQL Monitor report, run the following right after the SQL statement:

```
SET TRIMSPOOL on
SET TRIM on
SET PAGESIZE 0
SET LINESIZE 1000
SET LONG 1000000
SET LONGCHUNKSIZE 1000000
SPOOL sqlmon_active.html
SELECT
  DBMS_SQL_MONITOR.REPORT_SQL_MONITOR(
    type=>'active')
FROM dual;
SPOOL off
```

Figure 12. SQL Monitor Report Syntax

There is also an option to specify the sql_id if the statement has already been run:

```
set trimspool on
set trim on
SET TRIMSPOOL on
SET TRIM on
SET PAGESIZE 0
SET LINESIZE 1000
SET LONG 1000000
SET LONGCHUNKSIZE 1000000
SPOOL sqlmon_active.html
SELECT
  DBMS_SQL_MONITOR.REPORT_SQL_MONITOR(
    sql_id=>'1n482vfrxw014',  <== Insert your SQL_ID here
    type=>'active')
FROM dual;
SPOOL off
```

Figure 13. SQL Monitor Report for a single SQL_ID Syntax

Note: If the query runs in less than 5 seconds and does not use parallel query then you may need to add a MONITOR hint to the query to force the creation of the SQL Monitor information needed for the active report.

Extended SQL Trace

Not all of a SQL statement's time is captured by SQL Monitor. Specifically, parse time and non-idle wait events like "SQL*Net message from client" and "SQL*Net message to client" are not directly captured in a SQL Monitor active report and can only be inferred based on any difference between the duration time and database time captured in the Time & Wait section of the report. It is possible to identify parse time in the bit-vector sampling attributes in the `V$ACTIVE_SESSION_HISTORY` view (i.e. `IN_PARSE`, `IN_HARD_PARSE`). This is similar to how in-memory query time can be identified (i.e. `IN_INMEMORY_QUERY`). However, since active session history is only sampled it is not possible to get exact timing information. It is far more accurate to use an extended SQL trace to identify precisely where time is spent if there are large discrepencies between the duration time and database time within a SQL Monitor active report.

See below for more information on extended SQL trace:

- MOS Note 376442.1, How To Collect 10046 Trace (SQL_TRACE) Diagnostics for Performance Issues
- SQL Tuning Guide: Part VII Monitoring and Tracing SQL

Optimizer trace

In some rare cases it may be useful to look at how the optimizer parsed the SQL statement, what costs were considered, and how the final execution plan was created.

To create an optimizer trace, the following command can be used in the session running the SQL statement. Note that this requires a hard parse to capture the trace data so you may have to flush the statement from the shared pool. Alternatively, you can use EXPLAIN PLAN for the statement(s).

To turn tracing on:

```
ALTER SESSION SET EVENTS 'trace[rdbms.SQL_Optimizer.*]'
```

Figure 14. Set Optimizer trace on

To turn the tracing off:

```
ALTER SESSION SET EVENTS 'trace[rdbms.SQL_Optimizer.*]' OFF
```

Figure 15. Set Optimizer trace off

In addition, you can also specify the SQL_ID of the statement if it has already run and you know it:

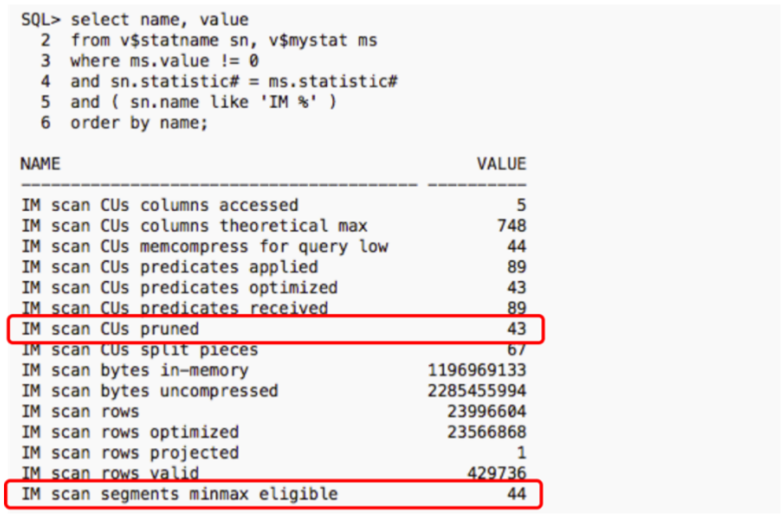
```
ALTER SESSION SET EVENTS 'trace[rdbms.SQL_Optimizer.*][sql:<enter sql_id>]'
```

Figure 16. Set Optimizer trace for a single SQL_ID

More information about using the SQL_ID can be found at the Optimizer Blog.

System and Session Level Statistics

System level and session level statistics can also be a good way to evaluate how work is being performed or time consumed. At the session level (i.e. `V$SESSTAT` or `V$MYSTAT`), statistics can be displayed after the execution of a SQL statement to display what specific Database In-Memory features may have been used or where time has been spent in SQL execution. For example, the following is a query that is used in the Database In-Memory LiveLabs workshops and in many of the Database In-Memory presentations and blog posts:



```
SQL> select name, value
  2  from v$statname sn, v$mystat ms
  3  where ms.value != 0
  4  and sn.statistic# = ms.statistic#
  5  and ( sn.name like 'IM %' )
  6  order by name;

NAME                                            VALUE
---------------------------------------- ----------
IM scan CUs columns accessed                        5
IM scan CUs columns theoretical max               748
IM scan CUs memcompress for query low              44
IM scan CUs predicates applied                     89
IM scan CUs predicates optimized                   43
IM scan CUs predicates received                    89
IM scan CUs pruned                                 43
IM scan CUs split pieces                           67
IM scan bytes in-memory                    1196969133
IM scan bytes uncompressed                 2285455994
IM scan rows                                 23996604
IM scan rows optimized                       23566868
IM scan rows projected                              1
IM scan rows valid                             429736
IM scan segments minmax eligible                   44
```

Figure 17. Session level stats

Here we see that IMCU pruning was performed due to the use of In-Memory storage indexes. In this example 43 of the 44 IMCUs were pruned at run time.

Many of the interesting Database In-Memory statistics have been moved, or only made visible, at the system level (i.e. `V$SYSSTAT`). An example is the many SIMD statistics that are only exposed in `V$SYSSTAT`. Viewing these statistics for individual executions can be challenging and may require exclusive access to the database during the execution time period. For obvious reasons, this may require a dedicated test environment if precise statistic values are required, otherwise an AWR report or a query of the DBA_HIST views is available.

Optimizer Hints

Normally Database In-Memory requires the use of no hints. Your application SQ L can run unchanged, and the optimizer will determine, based on cost, the optimal execution plan. However, the following hints are available to help when analyzing the affect of Database In-Memory on SQL execution plans.

INMEMORY/NO_INMEMORY

The only thing the INMEMORY hint does is enable 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.

The INMEMORY hint will not 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 IM column store and the plan with the lowest cost is a full table scan.

INMEMORY_PRUNING/NO_INMEMORY_PRUNING

Another hint introduced with Oracle Database In-Memory is (NO_)INMEMORY_PRUNING, which controls the use of In-Memory storage indexes. By default every query executed against the IM column store can take advantage of In-Memory storage indexes (IM 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 and should not normally be used.

VECTOR_TRANSFORM / NO_VECTOR_TRANSFORM

These hints force or disable the use of In-Memory Aggregation (IMA), or vector group by, when Database In-Memory is enabled. Note that IMA can be used with segments that are not populated in the IM column store if Database In-Memory is enabled.

PX_JOIN_FILTER / NO_PX_JOIN_FILTER

Enables or disables the use of Bloom filters for hash joins. This is only useful for testing.

FULL

Force a full table scan. Only a full table scan can leverage the IM column store (i.e., TABLES ACCESS INMEMORY FULL).

Optimizer Features

If you want to prevent the optimizer from considering the information it has about the objects populated into the IM column store (in-memory statistics), or in other words, revert the cost model back to what it was before In-Memory came along, you have two options:

- Set OPTIMIZER_FEATURES_ENABLE to 12.1.0.1 or lower. This would force the optimizer to use the cost model from that previous release, effectively removing all knowledge of the IM column store. However, you will also undo all the other changes made to the optimizer in the subsequent releases, which could result in some undesirable side effects.

- Set the new OPTIMIZER_INMEMORY_AWARE parameter to FALSE. This is a less drastic approach, as it will disable only the optimizer cost model enhancements for in-memory. Setting the parameter to FALSE causes the optimizer to ignore the in-memory statistics of tables during the optimization of SQL statements.

Note that even with the optimizer in-memory enhancements disabled, you might still get an In-Memory plan. If a segment is populated in the IM column store and a full table scan is chosen then the scan will be performed in the IM column store.

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.

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.

Process

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
- Optional: Take an AWR snapshot
- 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 for a plan to be captured automatically.
- Optional: Take an AWR snapshot
- 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 *Verifying Population* in the RAC section.

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
- Optional: Take an AWR snapshot
- 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: Take an AWR snapshot
- 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

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

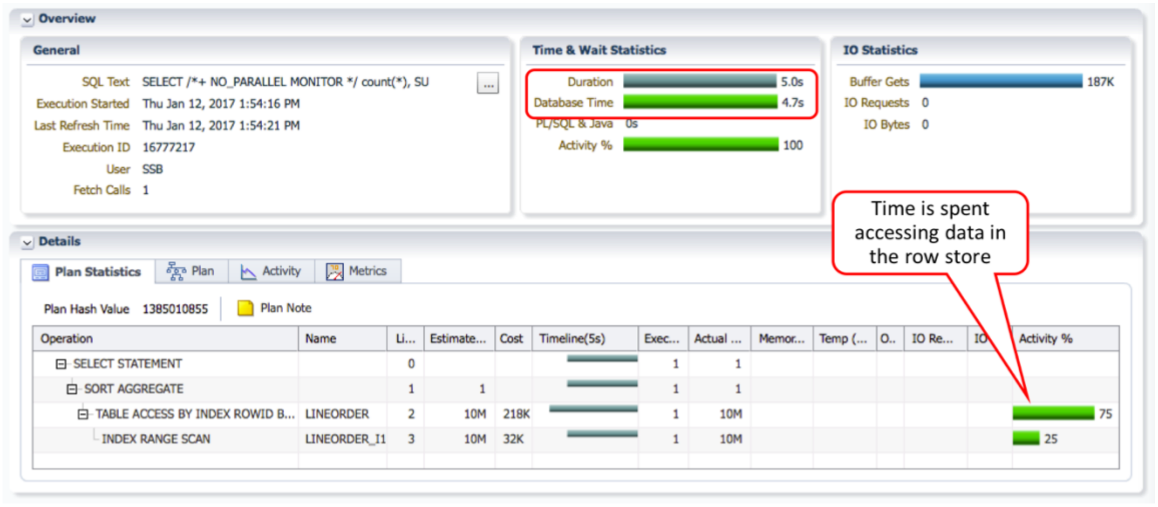
Scans

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'
```

Figure 18. SQL example for In-Memory scans

A traditional execution plan looks like the following:


Overview

General

SQL Text	SELECT /*+ NO_PARALLEL MONITOR */ count(*), SU
Execution Started	Thu Jan 12, 2017 1:54:16 PM
Last Refresh Time	Thu Jan 12, 2017 1:54:21 PM
Execution ID	16777217
User	SSB
Fetch Calls	1

Time & Wait Statistics

Duration	5.0s
Database Time	4.7s
PL/SQL & Java	0s
Activity %	100

IO Statistics

Buffer Gets	187K
IO Requests	0
IO Bytes	0

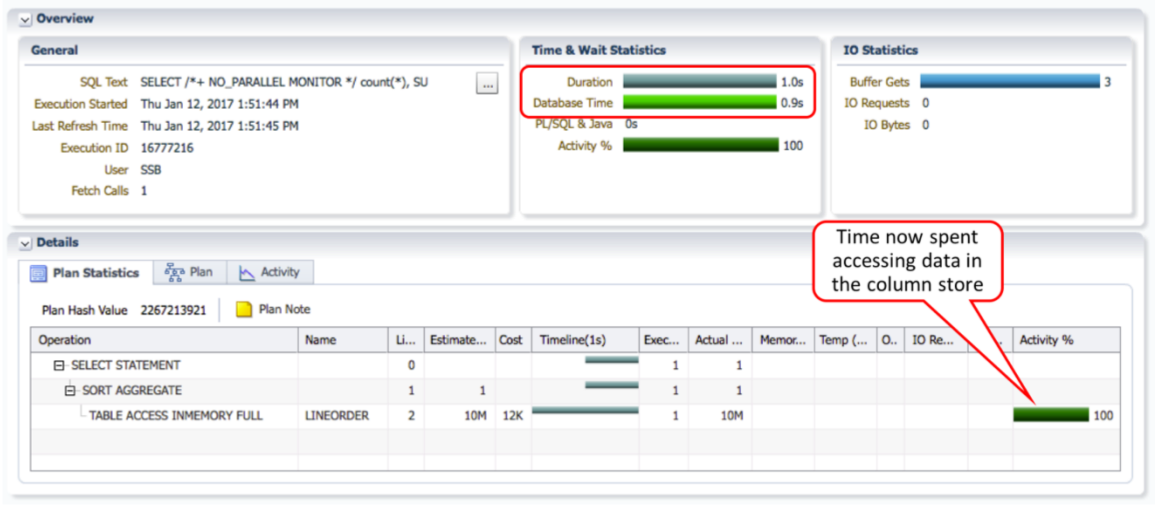
Details

Plan Hash Value 1385010855

Operation	Name	Li...	Estimate...	Cost	Timeline(5s)	Exec...	Actual ...	Memor...	Temp (...	O..	IO Re...	IO	Activity %
SELECT STATEMENT		0				1	1						
SORT AGGREGATE		1	1			1	1						
TABLE ACCESS BY INDEX ROWID B...	LINEORDER	2	10M	218K		1	10M						75
INDEX RANGE SCAN	LINEORDER_I1	3	10M	32K		1	10M						25

Figure 19. SQL Monitor Report for row store scan

Note that most 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:


Overview

General

SQL Text	SELECT /*+ NO_PARALLEL MONITOR */ count(*), SU
Execution Started	Thu Jan 12, 2017 1:51:44 PM
Last Refresh Time	Thu Jan 12, 2017 1:51:45 PM
Execution ID	16777216
User	SSB
Fetch Calls	1

Time & Wait Statistics

Duration	1.0s
Database Time	0.9s
PL/SQL & Java	0s
Activity %	100

IO Statistics

Buffer Gets	3
IO Requests	0
IO Bytes	0

Details

Plan Hash Value 2267213921

Operation	Name	Li...	Estimate...	Cost	Timeline(1s)	Exec...	Actual ...	Memor...	Temp (...	O..	IO Re...		Activity %
SELECT STATEMENT		0				1	1						
SORT AGGREGATE		1	1			1	1						
TABLE ACCESS INMEMORY FULL	LINEORDER	2	10M	12K		1	10M						100

Figure 20. SQL Monitor Report for In-Memory Scan

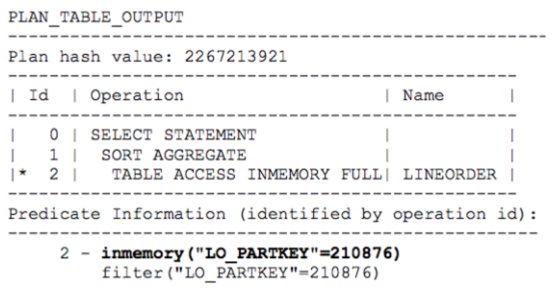
We see that the query now spends most of its time accessing the LINEORDER table in-memory and the execution time has dropped to just 1.0 seconds.

Predicate Push Downs

In addition to being able to scan segments in the IM column store, Database In-Memory can also aggregate and filter data as part of the scan. If we execute the following SQL statement with a where clause of LO_PARTKEY=210876, we will see that the predicate can be pushed into the scan of the LO_PARTKEY column. You can tell if a predicate has been pushed by looking for the keyword inmemory in the predicate information under the plan. The keyword inmemory replaces the traditional ACCESS keyword, when the predicate is applied as part of a scan in the IM column store.

```
SELECT
  COUNT(*)
FROM lineorder
WHERE lo_partkey=210876
```

Figure 21. SQL Query for Predicate Pushdown



```
PLAN_TABLE_OUTPUT
---------------------------------------------------
Plan hash value: 2267213921
------------------------------------------------
| Id  | Operation                   | Name      |
------------------------------------------------
|   0 | SELECT STATEMENT            |           |
|   1 |  SORT AGGREGATE             |           |
|*  2 |   TABLE ACCESS INMEMORY FULL| LINEORDER |
------------------------------------------------
Predicate Information (identified by operation id):
--------------------------------------------------
     2 - inmemory("LO_PARTKEY"=210876)
         filter("LO_PARTKEY"=210876)
```

Figure 22. SQL Execution Plan for Predicate Pushdown

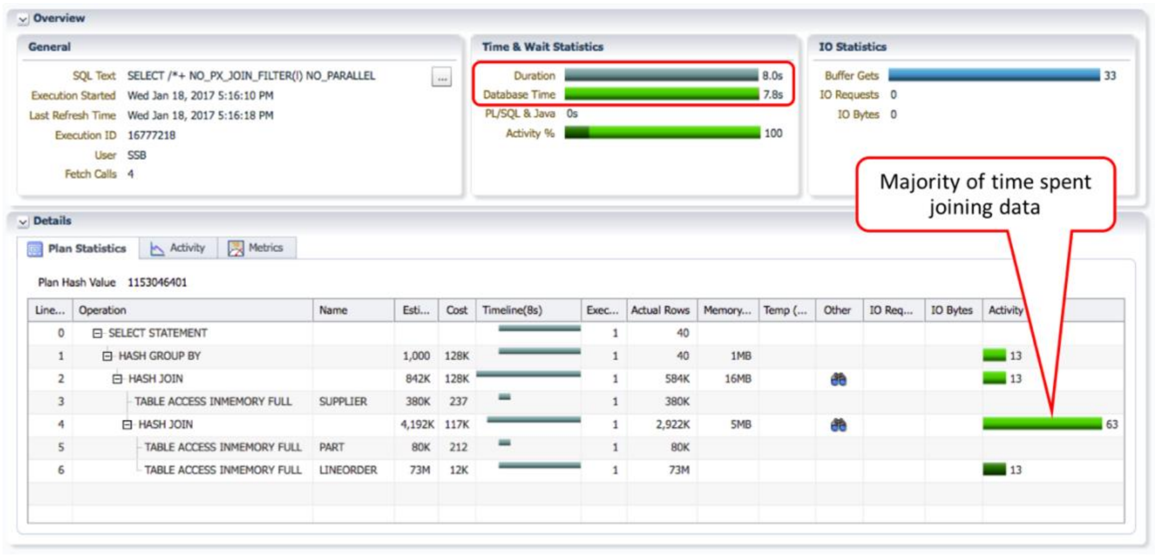
Joins

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
```

Figure 23. SQL Query to Show Joins

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



SQL Text	SELECT /*+ NO_PX_JOIN_FILTER(l) NO_PARALLEL
Execution Started	Wed Jan 18, 2017 5:16:10 PM
Last Refresh Time	Wed Jan 18, 2017 5:16:18 PM
Execution ID	16777218
User	SSB
Fetch Calls	4

Duration	8.0s
Database Time	7.8s
PL/SQL & Java	0s
Activity %	100

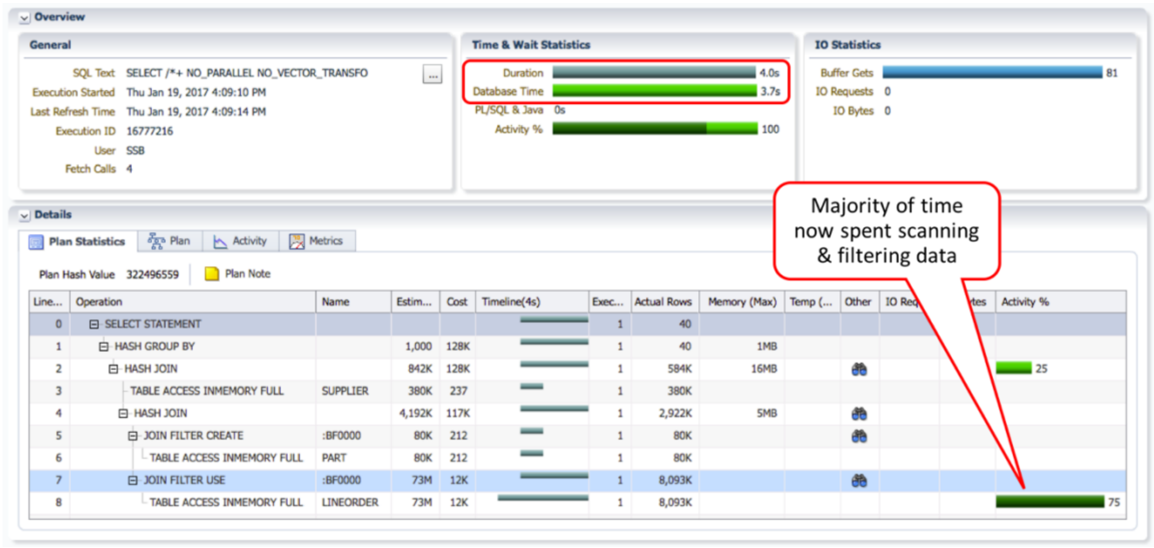
Buffer Gets	33
IO Requests	0
IO Bytes	0

Plan Hash Value 1153046401

Line...	Operation	Name	Esti...	Cost	Timeline(8s)	Exec...	Actual Rows	Memory...	Temp (...	Other	IO Req...	IO Bytes	Activity
0	SELECT STATEMENT					1	40						
1	HASH GROUP BY		1,000	128K		1	40	1MB					13
2	HASH JOIN		842K	128K		1	584K	16MB					13
3	TABLE ACCESS INMEMORY FULL	SUPPLIER	380K	237		1	380K						
4	HASH JOIN		4,192K	117K		1	2,922K	5MB					63
5	TABLE ACCESS INMEMORY FULL	PART	80K	212		1	80K						
6	TABLE ACCESS INMEMORY FULL	LINEORDER	73M	12K		1	73M						13

Figure 24. 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:



General

SQL Text SELECT /*+ NO_PARALLEL NO_VECTOR_TRANSFO
Execution Started Thu Jan 19, 2017 4:09:10 PM
Last Refresh Time Thu Jan 19, 2017 4:09:14 PM
Execution ID 16777216
User SSB
Fetch Calls 4

Time & Wait Statistics

Duration 4.0s
Database Time 3.7s
PL/SQL & Java 0s
Activity % 100

IO Statistics

Buffer Gets 81
IO Requests 0
IO Bytes 0

Plan Hash Value 322496559

Line...	Operation	Name	Estim...	Cost	Timeline(4s)	Exec...	Actual Rows	Memory (Max)	Temp (...	Other	IO Req	tes	Activity %
0	SELECT STATEMENT					1	40						
1	HASH GROUP BY		1,000	128K		1	40	1MB					
2	HASH JOIN		842K	128K		1	584K	16MB					25
3	TABLE ACCESS INMEMORY FULL	SUPPLIER	380K	237		1	380K						
4	HASH JOIN		4,192K	117K		1	2,922K	5MB					
5	JOIN FILTER CREATE	:BF0000	80K	212		1	80K						
6	TABLE ACCESS INMEMORY FULL	PART	80K	212		1	80K						
7	JOIN FILTER USE	:BF0000	73M	12K		1	8,093K						
8	TABLE ACCESS INMEMORY FULL	LINEORDER	73M	12K		1	8,093K						75

Figure 25. SQL Monitor Report for Join - With Bloom Filter

We see that now most 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 full table scan of the LINEORDER table (line 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 Technical Brief.

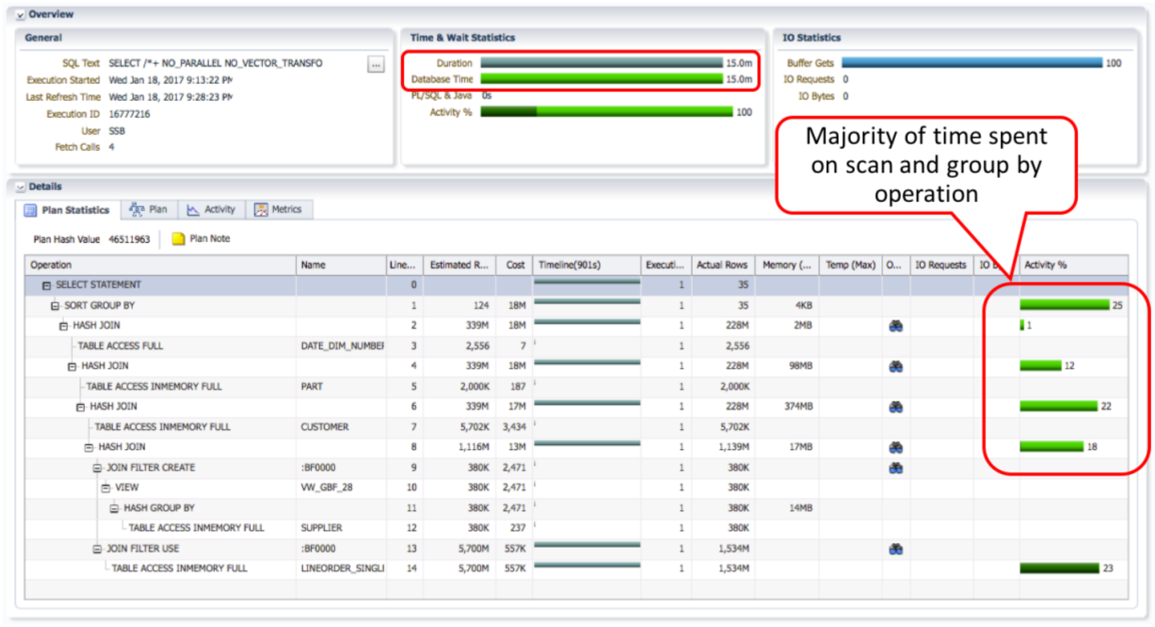
In-Memory Aggregation

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
```

Figure 26. SQL Query to show In-Memory Aggregation

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



Overview

General

SQL Text	SELECT /*+ NO_PARALLEL NO_VECTOR_TRANSFO
Execution Started	Wed Jan 18, 2017 9:13:22 PM
Last Refresh Time	Wed Jan 18, 2017 9:28:23 PM
Execution ID	16777216
User	SSB
Fetch Calls	4

Time & Wait Statistics

Duration	15.0m
Database Time	15.0m
PL/SQL & Java	0s
Activity %	100

IO Statistics

Buffer Gets	100
IO Requests	0
IO Bytes	0

Details

Plan Statistics | Plan | Activity | Metrics

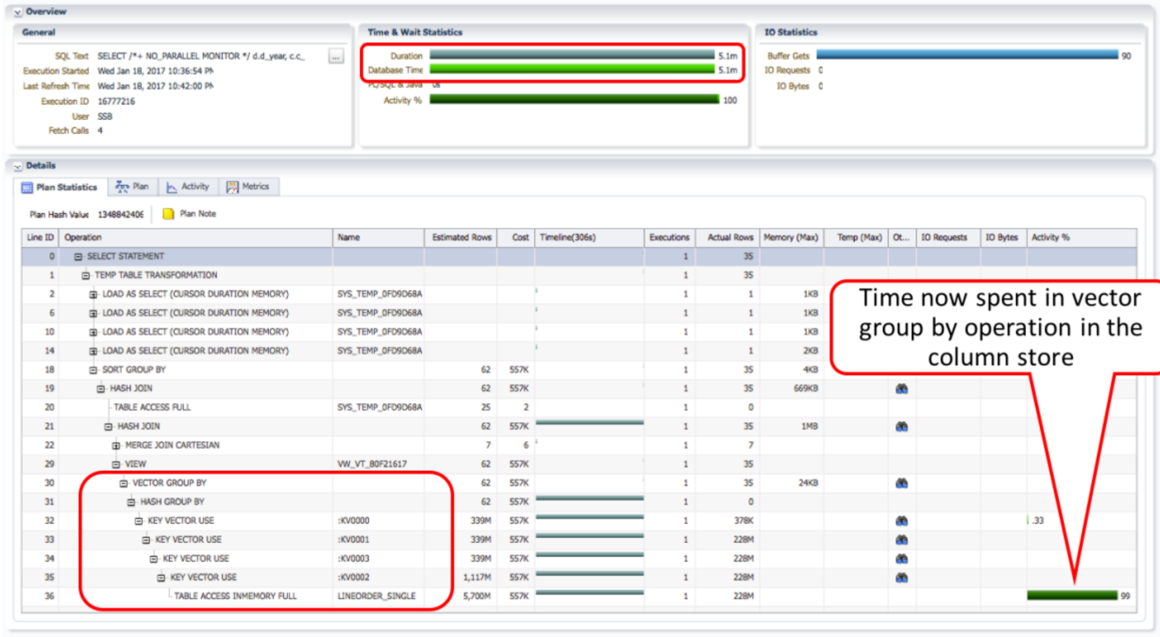
Plan Hash Value 46511963 Plan Note

Operation	Name	Line...	Estimated R...	Cost	Timeline(901s)	Executi...	Actual Rows	Memory (...	Temp (Max)	O...	IO Requests	IO B...	Activity %
SELECT STATEMENT		0				1	35						
SORT GROUP BY		1	124	18M		1	35	4KB					25
HASH JOIN		2	339M	18M		1	228M	2MB					1
TABLE ACCESS FULL	DATE_DIM_NUMBEI	3	2,556	7		1	2,556						
HASH JOIN		4	339M	18M		1	228M	98MB					12
TABLE ACCESS INMEMORY FULL	PART	5	2,000K	187		1	2,000K						
HASH JOIN		6	339M	17M		1	228M	374MB					22
TABLE ACCESS INMEMORY FULL	CUSTOMER	7	5,702K	3,434		1	5,702K						
HASH JOIN		8	1,116M	13M		1	1,139M	17MB					18
JOIN FILTER CREATE	:BF0000	9	380K	2,471		1	380K						
VIEW	VW_GBF_28	10	380K	2,471		1	380K						
HASH GROUP BY		11	380K	2,471		1	380K	14MB					
TABLE ACCESS INMEMORY FULL	SUPPLIER	12	380K	237		1	380K						
JOIN FILTER USE	:BF0000	13	5,700M	557K		1	1,534M						
TABLE ACCESS INMEMORY FULL	LINEORDER_SINGLI	14	5,700M	557K		1	1,534M						23

Figure 27. SQL Monitor Report with no In-Memory Aggregation

Note that most 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:


Time now spent in vector group by operation in the column store

Overview

General

SQL Text	SELECT /*+ NO_PARALLEL MONITOR */ d.d_year, c.c_
Execution Started	Wed Jan 18, 2017 10:36:54 PM
Last Refresh Time	Wed Jan 18, 2017 10:42:00 PM
Execution ID	16777216
User	SSB
Fetch Calls	4

Time & Wait Statistics

Duration	5.1m
Database Time	5.1m
PL/SQL & Java	0s
Activity %	100

IO Statistics

Buffer Gets	90
IO Requests	0
IO Bytes	0

Details

Plan Statistics | Plan | Activity | Metrics

Plan Hash Value 1348842406 Plan Note

Line ID	Operation	Name	Estimated Rows	Cost	Timeline(306s)	Executions	Actual Rows	Memory (Max)	Temp (Max)	Ot...	IO Requests	IO Bytes	Activity %
0	SELECT STATEMENT					1	35						
1	TEMP TABLE TRANSFORMATION					1	35						
2	LOAD AS SELECT (CURSOR DURATION MEMORY)	SYS_TEMP_0FD9D68A				1	1	1KB					
6	LOAD AS SELECT (CURSOR DURATION MEMORY)	SYS_TEMP_0FD9D68A				1	1	1KB					
10	LOAD AS SELECT (CURSOR DURATION MEMORY)	SYS_TEMP_0FD9D68A				1	1	1KB					
14	LOAD AS SELECT (CURSOR DURATION MEMORY)	SYS_TEMP_0FD9D68A				1	1	2KB					
18	SORT GROUP BY		62	557K		1	35	4KB					
19	HASH JOIN		62	557K		1	35	669KB					
20	TABLE ACCESS FULL	SYS_TEMP_0FD9D68A	25	2		1	0						
21	HASH JOIN		62	557K		1	35	1MB					
22	MERGE JOIN CARTESIAN		7	6		1	7						
29	VIEW	VW_VT_80F21617	62	557K		1	35						
30	VECTOR GROUP BY		62	557K		1	35	24KB					
31	HASH GROUP BY		62	557K		1	0						
32	KEY VECTOR USE	:KV0000	339M	557K		1	378K						.33
33	KEY VECTOR USE	:KV0001	339M	557K		1	228M						
34	KEY VECTOR USE	:KV0003	339M	557K		1	228M						
35	KEY VECTOR USE	:KV0002	1,117M	557K		1	228M						
36	TABLE ACCESS INMEMORY FULL	LINEORDER_SINGLE	5,700M	557K		1	228M						99

Figure 28. SQL Monitor Report with In-Memory Aggregation

Note that now most 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 provide a 10x or better performance improvement for analytical style queries. These can be traditional reporting type queries, Business Intelligence type queries, or any type of query where "What if ..." questions are being asked of the data. Database In-Memory excels at aggregating large amounts of data to provide information that can be used to answer complex business questions. However, Database In-Memory is not a one size fits all option. Database In-Memory uses columnar formatted data to enable fast scans, joins and group by aggregations. It does not speed up traditional DML operations like insert, update and deletes other than by eliminating unnecessary analytic indexes. Those DML operations are still performed on row formatted data in Oracle Database. Database In-Memory columnar formatted data is always kept transactionally consistent and in-sync with the row formatted data to preserve Oracle Database's transaction architecture.

This paper provides guidelines that, if followed, will help ensure the successful implementation of Database In-Memory with no SQL performance regressions. Database In-Memory is part of Oracle Database Enterprise Edition and Oracle Database is complex with lots of features and options to handle all sorts of application tasks. Database In-Memory leverages many of Oracle Database's features to provide outstanding performance improvements.

Database In-Memory is also tightly integrated with the Oracle optimizer, and the optimizer is more adaptive than it was in previous releases. This can require some additional effort to ensure a successful Database In-Memory implementation. Although no application code has to change to take advantage of Database In-Memory, that does not necessarily mean that unchanged application SQL will leverage all the features of Database In-Memory. One of the other purposes of this paper has been to illustrate how Database In-Memory works within Oracle Database and your application architecture.

With proper planning and the knowledge of how Database In-Memory works you should be able to successfully implement Database In-Memory with your applications and get outstanding analytic query performance.

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 Implementation Guidelines
September, 2025
Author: Andy Rivenes