ORACLE
DATABASE IN-MEMORY

Oracle Database In-Memory Implementation and Usage

ORACLE WHITE PAPER | AUGUST 2017

ORACLE

Table of Contents

Introduction 1

Planning 1

- Identify Analytic Workloads 1
- Understanding How Database In-Memory Works 2
- Identify Success Criteria 3
- Database Upgrades 3
  - Preserving Optimizer Statistics 4
  - Preserving Execution Plans 4

Configuration 4

- Apply the Latest Database Release Update 4
- Memory Allocation 4
- Database Parameter Settings 5
- Memory Allocation in Multitenant Databases 6

Preparation 7

- Application Architecture 7
  - Review Statistics 7
  - Review Constraint Definitions 7
  - Review Partitioning 8
  - Review Indexing 8
- Oracle Database In-Memory and Compression Advisors 8

Parallelism 8

Parallel Execution 8
Auto DOP 8
Resource Manager 9
RAC 9
Distribution 10
Verifying Population 10
Auto DOP 11
12.1.0.2 Behavior 11
Services 11
Setting Up Independent In-Memory Column Stores (12.1.0.2) 11
FOR SERVICE Sub-Clause (12.2) 12
Plan Stability 12
Adaptive Query Optimization 12
Adaptive Plans 13
Adaptive Statistics 13
Dynamic Statistics 13
Automatic Reoptimization 13
SQL Plan Directives 14
Optimizer Settings 14
Changes to OPTIMIZER_ADAPTIVE_FEATURES 14
In-Memory Statistics 14
Optimizer Statistics 15

SQL Plan Management 15
Capturing Existing Plans 16
Disable Automatic Plan Evolution 16
Workload and Analysis 17
Performance History 17
Regressions and Problem SQL 17
SQL Monitor Active Reports 17
Optimizer trace 18
Additional In-Memory Views 18
Statistics 18
Optimizer Hints 19
INMEMORY/NO_INMEMORY 19
INMEMORY_PRUNING/NO_INMEMORY_PRUNING 19
VECTOR_TRANSFORM / NO_VECTOR_TRANSFORM 19
PX_JOIN_FILTER / NO_PX_JOIN_FILTER 20
FULL 20
Optimizer Features 20
Identifying In-Memory Usage 20
Scans 20
Predicate Push Downs 21
Joins 22
In-Memory Aggregation 23
Implementation 25

Population Considerations 25

Implementation Strategy 25

Step 1: Run the workload without Database In-Memory 25

Step 2: Populate Tables In-Memory 26

Step 3: Run the Workload with Database In-Memory 27

Step 4: Evolve SQL Plan Baselines 27

Step 5: Final Workload Execution 28

Summary 28

Introduction

Oracle Database In-Memory is available with Oracle Database 12c Release 2 on-premises, Cloud at Customer and on Oracle Cloud. It adds in-memory functionality to Oracle Database for transparently accelerating analytic queries by orders of magnitude, enabling real-time business decisions without needing application code changes. It accomplishes this using a "dual-format" architecture to leverage columnar formatted data for analytic queries while maintaining full compatibility with all of Oracle's existing technologies. This "dual-format" architecture provides the best of both worlds. The traditional row format for incredibly efficient on-line transaction processing (OLTP) processing and the columnar format for super-fast analytic reporting.

Since Oracle 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 Oracle 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 whitepaper provides information to help ensure successful implementations and usage of Oracle Database In-Memory. Oracle Database In-Memory will accelerate analytic workloads without disruption to existing database environments and this white paper 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 Oracle Database In-Memory fundamentals as outlined in the Oracle Database In-Memory whitepaper.

Planning

Identify Analytic Workloads

Oracle Database In-Memory (Database In-Memory) is not a one size fits all solution. It is specifically targeted at analytic 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 that you 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 OLTP workloads.

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 filter criteria to return a small number of values. Queries that spend the majority of their time scanning and filtering data see the most benefit. Queries that spend the majority of their time on complex joins, sorting or returning millions of rows back to the client will see less benefit. A good way to think about it is, the goal 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, use Bloom filters to transform joins into filters that can be applied as part of the scan of the larger table, and use vector group by to perform aggregations as part of the scan, Database In-Memory can complete the majority of a query's processing while scanning the data. Other Database In-Memory features complement these primary attributes and include the use of Single Instruction Multiple Data (SIMD) vector processing and In-Memory storage index pruning to further speed up columnar scans.

Workloads that involves 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. However, the exception to that can be when the source data is in the IM column store. In this case, the time to load may be reduced by the amount of time saved scanning the IM column store rather than having to fetch the data from the row store.

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

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

The answer to both questions is 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 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. A tool called the Oracle Database In-Memory Advisor (In-Memory Advisor) is available to help with determining which objects will benefit from Database In-Memory. Since the In-Memory Advisor is available as a separate tool 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 by searching for note 1965343.1 and the In-Memory Advisor technical white paper.

For the second question, the answer is the Compression Advisor (i.e. the DBMS_COMPRESSION PL/SQL package), which 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 in question based on the target compression level.

Understanding How Database In-Memory Works

It is important to understand how Database In-Memory works since it does not benefit all workload types. As was stated earlier, Database In-Memory benefits queries that spend the majority of their run time scanning and filtering data, performing equality joins and aggregating data. This will be reflected in execution plans showing full table inmemory scans (i.e. TABLE ACCESS INMEMORY FULL), hash joins with Bloom filters on inmemory tables or aggregations using vector group by. In addition, the execution plan may also show predicate push downs and filtering for inmemory scans. All of these operations benefit from the optimizations included in Database In-Memory. All other 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. One of the other big success factors is that none of the SQL statements regress in performance. We will offer methods to help insure that no regressions occur.

An important way to create measurable criteria for success is to establish a baseline for the performance of the application or selected SQL statements. Typical criteria will be transaction or SQL response time and possibly system level workload.

The simplest way to create a baseline is to identify the SQL statements that are analytical in nature and are the target of the implementation. This can be done based on knowledge of the application or with the help of the In-Memory Advisor as mentioned earlier. 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 Oracle Database 12c prior to implementing Database In-Memory. At a minimum, this will involve recording the elapsed times of the SQL statements in question, but it may also include gathering execution plans and measuring other database performance metrics. As part of creating a baseline, it is important to consider the hardware platform and software involved. For example, you cannot expect to see the same performance if migrating to different hardware platforms or CPU generations. As with any benchmark-like project it is also important to ensure that the testing is repeatable. In the case of Oracle Database this means that execution plans need to have stabilized and that the application environment has reached a steady state. This will probably result in the need for many repeated executions and multi-user testing to replicate a representative environment (we will address these issues in more detail later in the paper).

Once a baseline is created, a comparison can be made to determine how much benefit Database In-Memory provided. It is also important to define what the expectations are. 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 improvement acceptable and how much business value can be derived from the improvement? These are the types of questions that can be used as input to determining the success of a Database In-Memory implementation.

Database Upgrades

Since Database In-Memory requires Oracle Database 12.1.0.2 or higher it is very common that a Database In-Memory implementation will also require an upgrade to Oracle Database 12c. If this is the case then the two activities, that is the 12c upgrade and the Database In-Memory implementation, should be performed separately. This is a very important consideration. It can be very difficult to identify the root cause of a performance regression when multiple things change at once. By separating the upgrade activity from the Database In-Memory implementation it will be much easier to troubleshoot any performance regressions.

It is an Oracle best practice to preserve existing optimizer statistics and execution plans prior to upgrading a database. Database upgrades introduce new features that may cause behavioral changes. In order to easily detect changes, and rectify any execution plans that may have regressed, you need to have a good understanding of the execution plans and optimizer statistics that you had before you began the upgrade. Upgrading in-place versus upgrading into a new database, or initially testing against an upgraded test database, will affect how you deal with optimizer statistics and execution plans. It is possible to transport both to a different database as well as ensuring that they don't change during the upgrade.

Preserving Optimizer Statistics

Before performing an upgrade you should capture the current set of optimizer statistics. This will help preserve existing execution plans and will ensure that as few things change as possible. The best way to keep a backup of the optimizer statistics for each application schema is to export them into a statistics table. They can then be restored if necessary or moved to another database. The DBMS_STATS.EXPORT_*_STATS and DBMS_STATS.IMPORT_*_STATS procedures can be used for this purpose.

Preserving Execution Plans

For an existing system, preserving execution plans is the easiest way to guarantee that no plan changes occur as part of the database upgrade. Ensuring that these plans are used is the only practical way to ensure that there will be no SQL execution regressions in the upgraded database. This is commonly referred to as plan stability, and SQL Plan Management is a feature of Oracle Database that allows the optimizer to automatically manage execution plans and ensure that only known or verified plans are used.

The simplest way to preserve existing execution plans is to capture the plans using SQL Tuning Sets (STS). SQL Tuning Sets group SQL statements and related metadata, including existing execution plans (see the attribute_list parameter within the DBMS_SQLTUNE.SELECT procedures). This process is described in detail later in the paper in the Plan Stability section. Note that this does require a license for the Oracle Tuning Pack. If you are not licensed for the Oracle Tuning Pack then you can also capture plans using SQL Plan baselines or stored outlines. See Manual Plan Capture in the Oracle Database 12c Release 2 Database SQL Tuning Guide for more details.

Configuration

Apply the Latest Database Release Update

Before starting any Database In-Memory implementation you should ensure that the latest available Database Release Update (formerly known as the Database Proactive Bundle Patch) has been applied. Database In-Memory fixes are only distributed through the Database Release Updated 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. Each Database Release Update includes the latest Program Service Updates (PSU) and therefore replaces the use of PSU or System Patch Updates (SPUs, formerly CPUs). For more information on patching see the following Oracle support document, Oracle Database - Overview of Database Patch Delivery Methods (MOS Note: 1962125.1)

For more information see the following notes based on your release:

- 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 are described later in this document, but knowledge of the application workload will make this process much easier and more accurate.

When using Database In-Memory in a Real Applications Cluster (RAC) environment additional memory must also be added to the shared pool. In RAC environments, every time a database block is populated in the IM column store as part of an 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 and should be considered minimum recommendations for the additional amount of memory that should be allocated to the System Global Area (SGA). 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	SGA_TARGET	PGA_AGGREGATE_TARGET**
Single-instance Databases	SGA_TARGET + INMEMORY_SIZE	PGA_AGGREGATE_TARGET
RAC Databases	SGA_TARGET + (INMEMORY_SIZE * 1.1)	PGA_AGGREGATE_TARGET

Figure 1. Memory Allocation

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

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

Database Parameter Settings

We strongly recommend that you start with default settings for all initialization parameters. This may mean unsetting existing custom settings, especially any that affect the optimizer. For example, customers on previous database releases often set the following parameters to non-default values:

- OPTIMIZER_DYNAMIC_SAMPLING
- OPTIMIZER_INDEX_CACHING
- OPTIMIZER_INDEX_COST_ADJ
- OPTIMIZER_FEATURES_ENABLE

We also **strongly** recommend unsetting any underscore parameters that have been set unless this has been done at the request of Oracle Support on the release of Oracle Database being used.

The following parameter should be set based on usage:

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

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

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

However, 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.

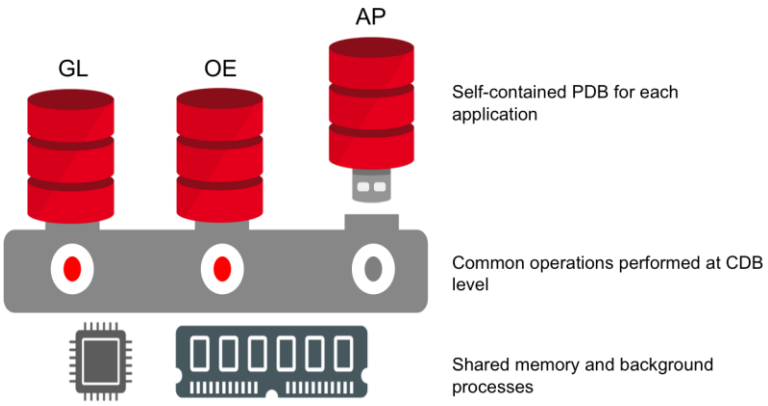

Figure 2. Multitenant PDBs

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. In order 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 a given 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:

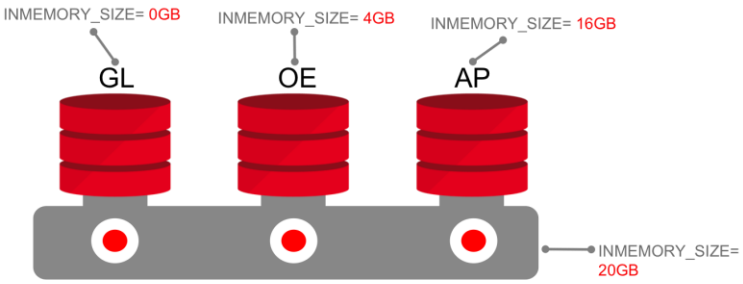

Figure 3. Multitenant In-Memory Allocation

Preparation

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. However, how and when to gather statistics can be very workload dependent and is beyond the scope of this paper. More information can be found in Best Practices for Gathering Optimizer Statistics with Oracle Database 12c.

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 for very large data sets. Partitioning allows the data architect to place only the most performance sensitive data, for example, into the IM column store. In many cases, it is only the most recent data that is the subject of the majority of analytical 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 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.

Oracle Database In-Memory and Compression Advisors

As mentioned earlier, the Oracle Database In-Memory Advisor (In-Memory Advisor) is available to help identify analytic processing workload in an existing Oracle Database. The In-Memory Advisor analyzes database activity based upon SQL plan cardinality, Active Session History (ASH), use of parallel query, and other statistics. Additional information is also available in two white papers: Examples of Advisor Usage and Best Practices When Using the Advisor. The Compression Advisor is useful to see approximately how much memory objects will consume in the IM column store.

Parallelism

Parallel Execution

Database In-Memory effectively eliminates I/O from the execution plan. For queries that access objects that are fully populated in the IM column store there is no I/O involved in the access of those objects. The response time of the query is limited only by the number of CPUs and the speed of memory access[1]. This makes Database In-Memory ideally suited to parallel execution to 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 inundate system 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 12c Fundamentals.

Auto DOP

Auto DOP allows Oracle Database to automatically decide the degree of parallelism for all SQL statements. This is Oracle's recommended way to control parallel execution. Setting the initialization parameter

[1] This ignores issues with stale rows accessed from the row store, sort spills to TEMP and partially populated objects or objects accessed that are not populated.

PARALLEL_DEGREE_POLICY to AUTO enables Auto DOP and allows the Oracle Database to determine whether the statement should run in parallel based on the cost of running the statement.

There are two other parameters that affect Auto DOP 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. The Optimizer then uses the cost of the statement to determine if the statement will be run in parallel mode based on the setting of the PARALLEL_MIN_TIME_THRESHOLD parameter. This parameter can be set manually at the system or session level and a setting of 0 will force parallel execution for all queries. This will be discussed in more detail in the RAC section.

The actual degree of parallelism (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 parallel execution can be enabled at the object level, with a statement level hint, or with an object level hint. The DOP used can be specified or will default to the DOP specified by a combination of init.ora parameters PARALLEL_THREADS_PER_CPU and CPU_COUNT depending on whether the environment is a single instance database or a RAC environment. For more detailed information see the white paper Parallel Execution with Oracle Database 12c Fundamentals.

Resource Manager

The 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.

The Database 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.

RAC

When running Database In-Memory in a RAC environment it is important to consider that each instance will have an IM column store. By default, objects populated in-memory will be distributed across all of the IM column stores in the cluster. 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). This means that parallel server processes need to be employed to execute queries that access in-memory objects because IMCUs

are not shipped between RAC instances. This makes the distribution of data between IM column stores very important. The goal is to have as even a distribution of data as possible so that all parallel server processes spend approximately the same amount of time scanning data to maximize throughput.

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 [g]v$im_segments view can be misleading. The BYTES_NOT_POPULATED field represents the bytes not populated on the instance in question. In a RAC database with multiple IM column stores this field will represent the bytes not populated on this instance for objects that are distributed. The following query shows how 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

Auto DOP

In a RAC environment when data is distributed between column stores, as opposed to being duplicated across all IM column stores (i.e. DUPLICATE ALL), parallel query must be used to access the IMCUs in the column stores on all but the local instance. In order to accomplish this the parallel query coordinator needs to know in which instance's IM column store the IMCUs for each object involved in the query reside. This is what is meant by parallel scans being affinitized for inmemory. 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.

There are two ways to ensure that the DOP of the query is greater than or equal to the number of IM column stores involved. The first is the use of Auto DOP (i.e. the initialization parameter PARALLEL_DEGREE_POLICY set to AUTO) which will ensure that the cost-based DOP calculation will be greater than or equal to the number of IM column store instances. The second relies on the user application to ensure that the DOP of the query is greater than or equal to the number of IM column stores involved. If this is not the case then the data residing in IM column stores that do not get a parallel server process assigned to them will have to be read from disk/buffer cache.

12.1.0.2 Behavior

In 12.1.0.2 Auto DOP was required in order to guarantee that the degree of parallelism (DOP) chosen would result in at least one parallel server process being allocated for each active instance, and to enable access to the map of the IMCU home locations. There was no workaround and required that Auto DOP be invoked with the PARALLEL_DEGREE_POLICY parameter specified at either the instance or session level. In addition to this restriction, if an IMCU access is attempted from an IM column store 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's 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. Services impact which IM columns stores will have data distributed across them and which will be accessed for a given query because both queries and on-demand population will honor the service that the invoking session connected to.

Setting Up Independent In-Memory Column Stores (12.1.0.2)

It is possible to set up independent IM column stores on a RAC cluster using services and some additional database initialization parameters. 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 of 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 execution. This technique can also be used to support multiple "independent" IM column stores if, for example, you wanted to partition multiple applications across nodes in your RAC environment. However, since the duplication of objects is only supported on Engineered Systems (i.e. DUPLICATE sub-clause) the objects populated in the IM column stores in this example have to be distinct, you cannot intermix objects between multiple IM column stores.

Network and Service Connections

Services and appropriate TNS entries need to be created such that each RAC instances involved can be identified independently.

Parallelism

Next we need to set the initialization parameter PARALLEL_FORCE_LOCAL to TRUE. This has implications for all parallel processing since we will effectively be preventing inter-node parallelism, whether it's for Database In-Memory parallel queries or any other parallel queries. In other words, parallel server processes will be restricted so that they can only run on the node(s) on which the SQL statement was executed.

Priority

To ensure that specific objects are populated into specific IM column stores we will have to set the population priority to NONE for the objects that we want to access in the IM column store, and we will have to access the object(s) (e.g. run a select) to start its population connected to the instance that we want the object(s) populated on.

FOR SERVICE Sub-Clause (12.2)

In 12.2 a new parameter to the DISTRIBUTE sub-clause has been introduced to make directing objects to specific IM column stores much easier. The primary use of the FOR SERVICE sub-clause is to manage in-memory population when using Active Data Guard. However, a secondary use is to make the process of running a subset of IM column stores much easier.

With the FOR SERVICE subclause you can specify a service for each object. This provides much more flexibility and ease of use. The following example shows the use of the FOR SERVICE clause for the sales table:

```
ALTER TABLE sales INMEMORY DISTRIBUTE AUTO FOR SERVICE dbmm1
```

Figure 6. 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 "dbmm1" defined.

Plan Stability

Adaptive Query Optimization

Adaptive Query Optimization is a set of features that were added to the optimizer in Oracle Database 12c. These features include the following:

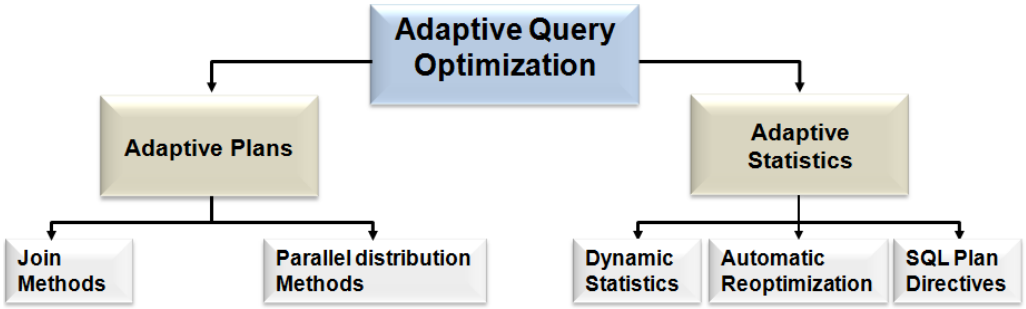

Figure 7. Adaptive Query Optimization components

Each of these features change the optimizer's behavior in Oracle Database 12c from the way it operated in previous releases. This was a major change that was added to Oracle Database 12c and the default behavior is significantly different between Oracle Database 12c Release 1 and Release 2. The following sections will describe the behavior in both Oracle Database 12c Release 1 and Release 2 and will make some recommendations on how to best manage the feature.

A key takeaway is that depending on the settings, Adaptive Query Optimization can take several iterations of a SQL statement before an optimal execution plan is determined. This can have a profound effect on the way a Proof of Concept (PoC) or implementation benchmark is run. It may be necessary to 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 will highlight the features of Adaptive Query Optimization that have specific issues for Database In-Memory.

Adaptive Plans

Adaptive plans allow the optimizer to decide on the optimal join method or parallel distribution method at execution time. Although adaptive plans are only possible during the initial execution of a query it is recommended that for all PoC or implementation benchmarks all adaptive plans should be resolved before any performance metrics are recorded. Executing all SQL statements at least once should achieve this.

To verify which SQL has adapted the IS_RESOLVED_ADAPTIVE_PLAN column in V$SQL should 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.

Adaptive Statistics

Dynamic Statistics

In Oracle Database 12c dynamic statistics allow the optimizer to get more accurate cardinality estimates for single table accesses, joins and group-by predicates. Dynamic statistics can be problematic with Database In-Memory because of the possibility that the execution plan will change. This can be aggravated with parallel queries, since the likelihood of dynamic statistics being invoked for a parallel query is much higher.

In Oracle Database 12c the default level for OPTIMIZER_DYNAMIC_SAMPLING is 2 if OPTIMIZER_FEATURES_ENABLE is set to 10.0.0 or higher. If OPTIMIZER_DYNAMIC_SAMPLING is set to level 11 then the optimizer will be much more aggressive about collecting dynamic statistics and this can cause parse times to take longer and execution plans to change. It is our recommendation that for Oracle Database 12c Release 1 that OPTIMIZER_DYNAMIC_SAMPLING be set to 0 or 4 if any parallel queries will be run. If only serial queries are run then OPTIMIZER_DYNAMIC_SAMPLING can be left unset or set to the default level of 2. If using Oracle Database 12c Release 2 then OPTIMIZER_DYNAMIC_SAMPLING should be left at its default value of 2 and OPTIMIZER_ADAPTIVE_STATISTICS should be set to FALSE, which is the default.

Automatic Reoptimization

Automatic reoptimization will change a SQL execution plan on subsequent executions of a SQL statement. This means that the first execution may not be the optimal execution plan. This is why it is important to ensure that execution plans have stabilized before basing decisions on SQL execution performance. Automatic reoptimization is actually made up of three components:

- Statistics feedback
- Performance feedback

» SQL plan directives

It is possible to tell whether a SQL statement will be reoptimized by looking at the column IS_REOPTIMIZABLE in the V$SQL dynamic performance view. If the column value is Y then the statement will be hard parsed on the next execution. Once this column value is set to N then it will no longer be re-parsed and no additional execution plans will be created for that SQL statement.

SQL Plan Directives

SQL plan directives are used to correct for cardinality misestimates. They will also be used by DBMS_STATS to create statistics for any missing extensions.

Since they are initially stored in memory they must be flushed to disk before they can be used by other processes. You can do this manually using the DBMS_SPD package or wait for the automatic job to kick in (e.g. they are automatically flushed after 15 minutes).

In order to achieve a proper steady state, once all of the SQL statements have been run any SQL Directives should be flushed to disk by executing the DBMS_SPD.FLUSH_SQL_PLAN_DIRECTIVE procedure.

Optimizer Settings

In general, as with our recommendations for initialization parameters, the optimizer specific settings should be left at their defaults. With that said we do recommend that you carefully review the parameter OPTIMIZER_ADAPTIVE_FEATURES in 12.1. It is our recommendation that you follow the guidelines in the next section, "Changes to OPTIMIZER_ADAPTIVE_FEATURES". Prior to the recommendations in this section when customers were experiencing negative impacts on their workload they would simply disable all of the 12c features by setting OPTIMIZER_ADAPTIVE_FEATURES to FALSE. This should be a last resort as this will effectively turn off all adaptive optimizer features and will prevent the Optimizer from using more optimal plans than could be created prior to Database 12c. Fortunately this should no longer be necessary, even for 12.1 customers.

Changes to OPTIMIZER_ADAPTIVE_FEATURES

In 12.2 the parameter OPTIMIZER_ADAPTIVE_FEATURES has been made obsolete and replaced with two new parameters: OPTIMIZER_ADAPTIVE_PLANS and OPTIMIZER_ADAPTIVE_STATISTICS. The default value of OPTIMIZER_ADAPTIVE_PLANS is TRUE and the default value of OPTIMIZER_ADAPTIVE_STATISTICS is FALSE. This should solve most "adaptive" problems and can be retrofitted to 12.1.0.2 with patches 21171382 and 22652097.

This is good news for 12.1.0.2 customers who want to avoid issues with adaptive statistics and were considering disabling OPTIMIZER_ADAPTIVE_FEATURES altogether by setting it to FALSE.

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 each SQL statement that accesses 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: 11997996  SSZ: 0  LGR: 0  #Blks:  167239  AvgRowLen:  96.00  NEB: 0  ChainCnt:  0.00  ScanRate:
SPC: 0  RFL: 0  RNF: 0  CBK: 0  CHR: 0  KQDFLG: 1
  #IMCUs: 24  IMCRowCnt: 11997996  IMCJournalRowCnt: 299950  #IMCBlocks: 167239  IMCQuotient: 1.000000
try to generate single-table filter predicates from ORs for query block SEL$1 (#0)
finally: "LINEORDER"."LO_CUSTKEY"=5641 AND "LINEORDER"."LO_SHIPMODE"='REG AIR' AND
```

Figure 8. 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

In 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

Optimizer Statistics

All of 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 12c Release 2" 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 skews
  - New types of histograms in 12c provide more detailed information
- Use extended statistics to make the Optimizer aware of correlation
  - 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

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.

In order to aid in diagnosing SQL query regressions, we strongly recommend that SQL Plan Baselines be collected for each test run, whether part of an initial PoC, an upgrade to Database 12c, 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.

Capturing Existing Plans

An STS can be used to capture execution plans for SQL statements in an existing system and move them to a new environment. SQL statements can be captured from various sources (i.e. AWR, library cache, sql trace file or another STS). The STS can then be packed into a staging table, transferred to the new system and unpacked into active baselines. The following will outline these steps:

Note: If you are just capturing execution plans to preserve them then you only need to perform steps one and two.

1. Create a new STS. (DBMS_SQLTUNE.CREATE_SQLSET)
2. Populate the STS with existing execution plans. There are multiple options available (i.e. loading from AWR, cursor cache, trace files, etc.) using the various DBMS_SQLTUNE.SELECT_... funcions and DBMS_SQLSET.LOAD_SQLSET.
3. Export the STS from the existing database. (DBMS_SQLTUNE.CREATE_STGTAB_SQLSET, DBMS_SQLTUNE.PACK_STGTAB_SQLSET, export/data pump)
4. Import the STS into the new database. (import/data pump)
5. Load the plans into SQL plan baselines. (DBMS_SPM.UNPACK_STGTAB_BASELINE)

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 9. 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 10. Evolve Task Query Syntax

Workload and Analysis

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.

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. 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 with and without in-memory (i.e. the NO_INMEMORY hint can be used to disable in-memory for an individual SQL statement), an AWR report for the testing time periods, and an optimizer trace (e.g. 10053) 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 11. SQL Monitor Report Syntax

There is also an option to specify the sql_id:

```
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',
    type=>'active')
FROM dual;
SPOOL off
```

Figure 12. SQL Monitor Report for a single SQL_ID Syntax

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.

Optimizer trace

To create an optimizer trace, use the following command 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 shared pool. Alternatively, you can use EXPLAIN PLAN for the statement(s):

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

Figure 13. Set Optimizer trace on

To turn the tracing off:

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

Figure 14. 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 15. Set Optimizer trace for a single SQL_ID

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

Additional In-Memory Views

- V$INMEMORY_AREA - is there enough memory
- V$IM_SEGMENTS - are all objects populated
- V$IM_HEADER - all rows populated per object
- V$IM_SMU_HEAD - are changes affecting journal space

Statistics

Both system level and session level statistics are available as well to help evaluate what optimizations may have been used during the execution of the SQL statement. More information is available in the Oracle Database Reference. For example, statistics can help determine IMCU pruning due to the use of In-Memory storage indexes. See the following example where 43 of the 44 IMCUs were pruned at run time:

```
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 16. Session level stats

Optimizer Hints

The following hints have a direct affect on in-memory execution plans and can be useful tools when analyzing in-memory queries.

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 as long as 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 of 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 definitely the less dramatic 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.

Identifying In-Memory Usage

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

Scans

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

```
SELECT
  COUNT(*),
  SUM(l.lo_ordtotalprice)
FROM  lineorder l
WHERE l.lo_shipmode = 'AIR'
```

Figure 17. SQL example for In-Memory scans

A traditional execution plan looks like the following:

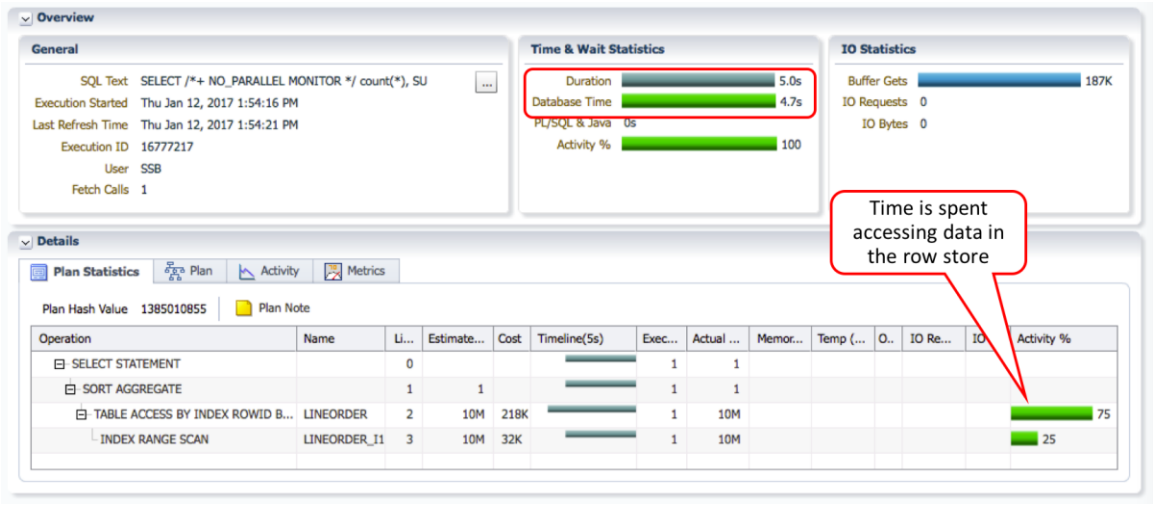

Figure 18. SQL Monitor Report for row store scan

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

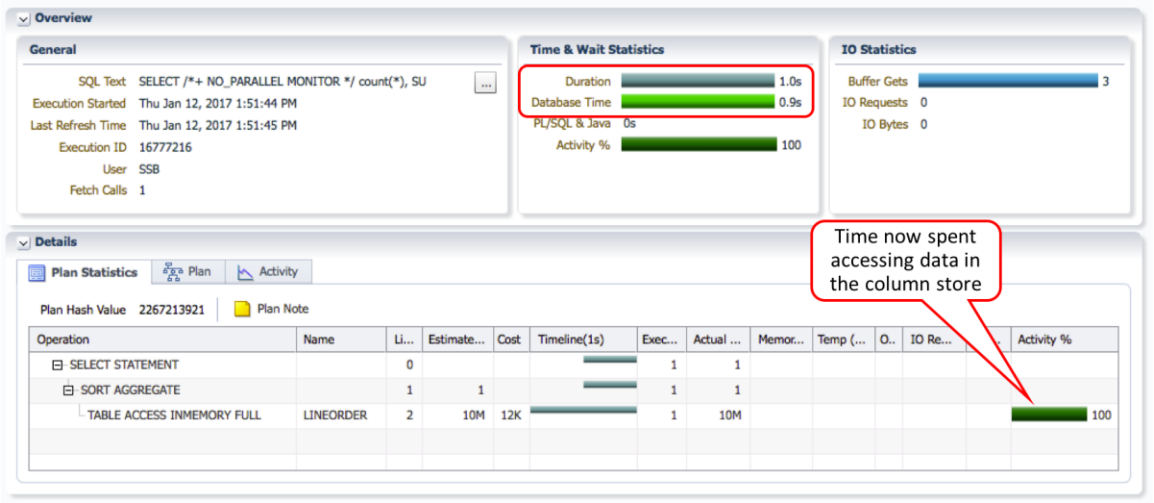

Figure 19. SQL Monitor Report for In-Memory Scan

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

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 20. 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 21. 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,
        SUM(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 22. SQL Query to Show Joins

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

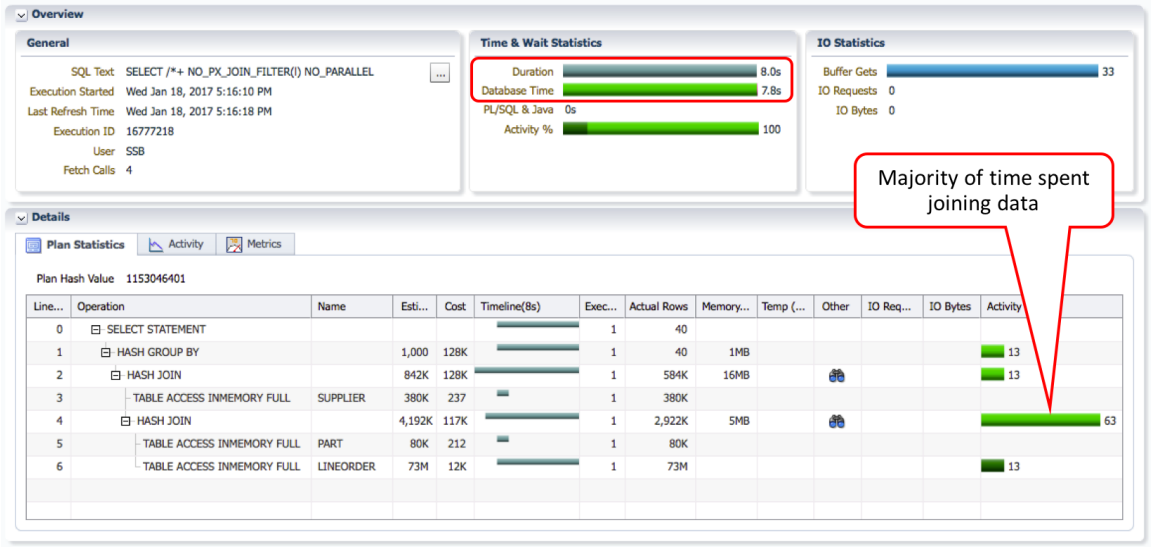

Figure 23. SQL Monitor Report for Join - No Bloom Filter

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

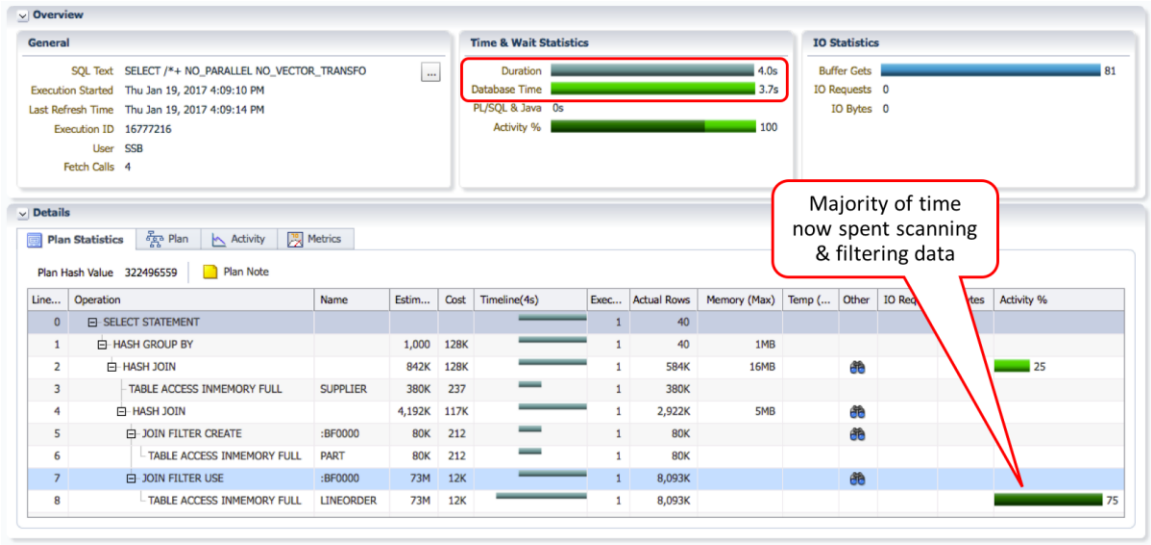

Figure 24. SQL Monitor Report for Join - With Bloom Filter

We see that now the majority of the query time is spent accessing the LINEORDER table in-memory and uses a Bloom filter. The Bloom filter (:BF0000) is created immediately after the scan of the PART table completes (line 5). The Bloom filter is then applied as part of the in-memory 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 whitepaper.

In-Memory Aggregation

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

```
SELECT   d.d_year, c.c_nation, SUM(lo_revenue - lo_supplycost)
FROM     LINEORDER l, DATE_DIM d, PART p, SUPPLIER s, CUSTOMER c
WHERE    l.lo_orderdate = d.d_datekey
AND      l.lo_partkey  = p.p_partkey
AND      l.lo_suppkey  = s.s_suppkey
AND      l.lo_custkey  = c.c_custkey
AND      s.s_region    = 'AMERICA'
AND      c.c_region    = 'AMERICA'
GROUP BY d.d_year, c.c_nation
ORDER BY d.d_year, c.c_nation
```

Figure 25. 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:

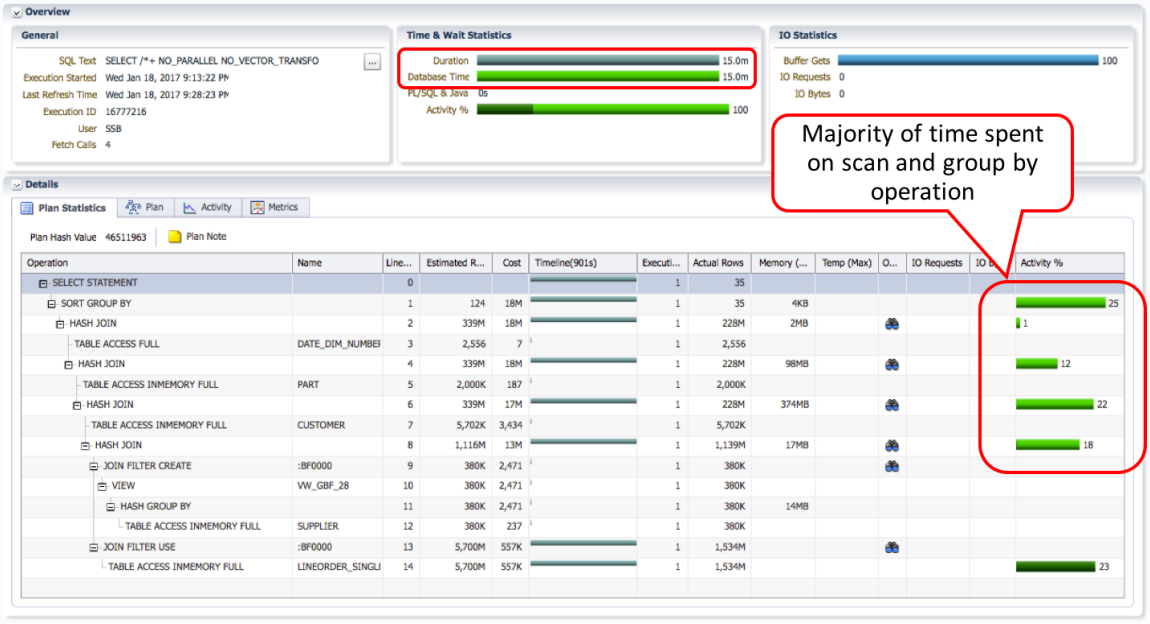

Figure 26. SQL Monitor Report with no In-Memory Aggregation

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

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

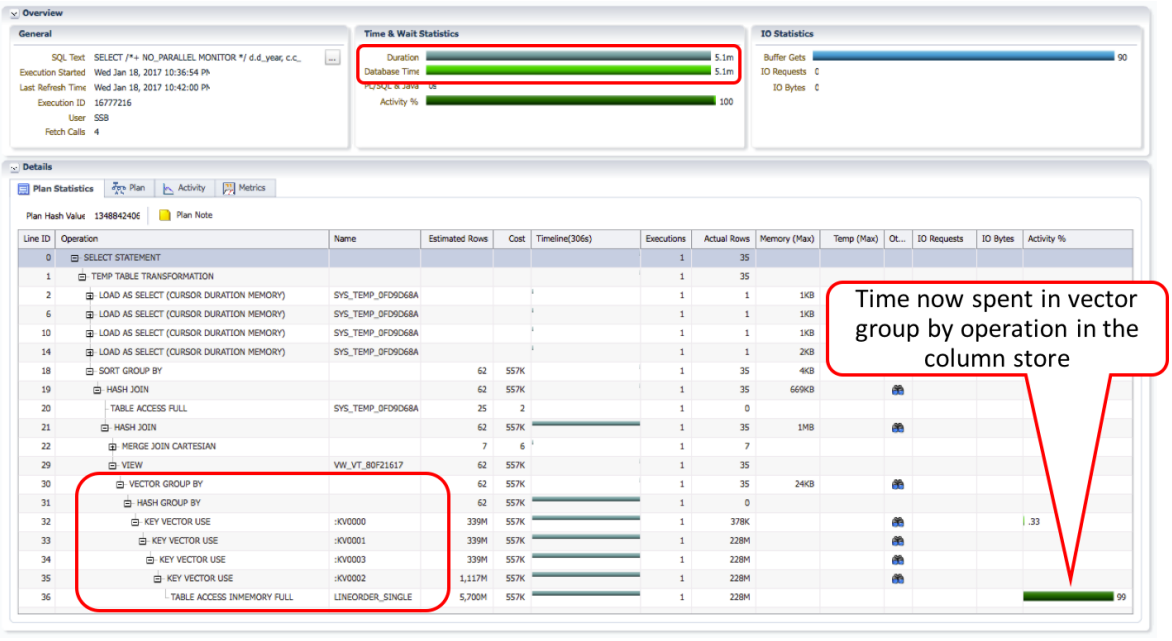

Figure 27. SQL Monitor Report with In-Memory Aggregation

Note that now the majority of the time is spent accessing the LINEORDER_SINGLE table and that four key vectors are applied as part of the scan of the LINEORDER_SINGLE table within a vector group by operation that starts at

line 30. We also see that the total run time is now only 5.1 minutes rather than 15.0 minutes. To investigate more about In-Memory Aggregation see the In-Memory Aggregation white paper.

Implementation

Population Considerations

There are several things to consider for population. Obviously having enough memory available in the IM column store is crucial. You do not want to have partially populated objects if at all possible as this will significantly affect the performance of the application. You also do not want to "steal" memory from other parts of the database (i.e. buffer pool, shared pool, etc.) because that can affect the existing application performance. This is especially important in mixed workload environments. Very often the best strategy is to populate the entire schema in the IM column store. The ability to achieve this will be dependent on the amount of memory that the customer is willing to allocate to the IM column store though.

Population is largely a CPU dominated process, 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 though, unless done on purpose, 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, the MAX_POPULATE_SERVERS parameter is dynamic, so it can be adjusted. In addition, services can be used to effectively block connection from the application until the population is complete.

Partitions

Population of partitions can be handled in a couple of ways. The table can be enabled for in-memory and then all partitions will be populated, including newly created partitions since they will inherit the default attribute of inmemory enabled. If however, a rolling window partitioning strategy is desired for population in the IM column store then individual partition/sub-partitions can be altered to be inmemory or no inmemory. In 12.1 this can be accomplished with a scheduler job or manually. In 12.2 Automatic Data Optimization (ADO) policies can be employed based on heat map heuristics or time based policies to populate or evict individual partitions/sub-partitions.

Implementation Strategy

As we mentioned in the Identify Success Criteria section at the beginning of this document 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 the upgrade or a poorly performing system. Once the database is at the proper version and patch level then a baseline should 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.

In order to implement this strategy, it is our recommendation that the following five-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 to the customer so that any possibility of surprises is minimized.

Step 1: Run the workload without Database In-Memory

This step may be a two-part process if the database is being upgraded. The goal of this step is to establish a baseline for application performance prior to implementing Database In-Memory. If upgrading the database to Oracle Database 12c 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:

- Turn on SQL Plan Baseline capture

```
ALTER SYSTEM SET OPTIMIZER_CAPTURE_SQL_PLAN_BASELINES = true
```

Figure 28. Set Baseline Capture on

Note: An alter session command can be used if workload/application allows.

- Run the workload
  Capturing elapsed times is important.

  Note: Each SQL statement must be run at least twice in order for a baseline to be captured.

- Turn off SQL Plan Baseline capture

```
ALTER SYSTEM SET OPTIMIZER_CAPTURE_SQL_PLAN_BASELINES = false
```

Figure 29. Set Baseline Capture off

Note: An ALTER SESSION command if used in task a.

- Drop unrelated SQL plan baselines

```
DECLARE
 tot number := 0;
BEGIN
 FOR rt IN (
   SELECT DISTINCT sql_handle
   FROM dba_sql_plan_baselines
   WHERE
     sql_text LIKE '%dba_sql_plan_baselines%'
     OR parsing_schema_name IN ('SYSTEM', 'SYS', 'DBSNMP', 'ORACLE_OCM'))
LOOP
   tot := tot + DBMS_SPM.DROP_SQL_PLAN_BASELINE(rt.sql_handle);
END LOOP;
 DBMS_OUTPUT.PUT_LINE(tot || ' baselines dropped.');
END;
```

Figure 30. SQL to drop unneeded Baselines

Note: This step is optional if session level capture was used.

- Confirm captured SQL plan baselines

```
SELECT
  sql_handle, plan_name, enabled, accepted,
  parsing_schema_name schema, sql_text
FROM dba_sql_plan_baselines
ORDER BY 1,2
```

Figure 31. SQL to list captured Baselines

Step 2: Populate Tables In-Memory

Populate the tables identified for the Database In-Memory workload into the IM column store and wait for completion of the population. It is important that all tables are fully populated into the IM column store. 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 should equal 0 or the SUM(BYTES - BYTES_NOT_POPULATED) for all instances should equal the total BYTES for the segment. See Figure 5 for an example.

Step 3: Run the Workload with Database In-Memory

The exact 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. Any new plan changes will be captured but will not be used yet.

- Ensure that OPTIMIZER_INMEMORY_AWARE is enabled (this is the default if OFE is 12.1.0.2 or higher), or more preferably that our recommendations for using OPTIMIZER_ADAPTIVE_PLANS and OPTIMIZER_ADAPTIVE_STATISTICS in 12.1 have been followed from the Optimizer Settings section earlier in this document.
- Run the workload
- Show SQL Plan baselines

```
SELECT
  sql_handle, plan_name, enabled, accepted,
  parsing_schema_name schema, sql_text
FROM dba_sql_plan_baselines
ORDER BY 1,2
```

Figure 32. SQL list Baselines

Step 4: Evolve SQL Plan Baselines

In this step we will evolve the SQL plan baselines to accept the best execution plans. This process will be accomplished using the following steps:

1. Create an evolve task:

```
VARIABLE cnt NUMBER
VARIABLE tk_name VARCHAR2(50)
VARIABLE exe_name VARCHAR2(50)
VARIABLE evol_out CLOB

EXECUTE :tk_name := DBMS_SPM.CREATE_EVOLVE_TASK;

SELECT :tk_name FROM DUAL;
```

Figure 33. SQL to Create an Evolve Baseline Task

2. Execute the evolve task:

```
EXECUTE :exe_name := DBMS_SPM.EXECUTE_EVOLVE_TASK(task_name=>:tk_name);

SELECT :exe_name FROM DUAL;
```

Figure 34. SQL to run an evolve task

3. View the evolve task report:

```
EXECUTE :evol_out := DBMS_SPM.REPORT_EVOLVE_TASK( task_name=>:tk_name,
execution_name=>:exe_name );

SELECT :evol_out FROM DUAL;
```

Figure 35. SQL to run an evolve task report

4. Implement the evolve task recommendations:

```
EXECUTE :cnt := DBMS_SPM.IMPLEMENT_EVOLVE_TASK( task_name=>:tk_name,
execution_name=>:exe_name );
```

Figure 36. SQL to implement evolve task recommendations

Step 5: Final Workload Execution

This is the final step and should provide the best possible run of the baseline workload. If there are any regressions or questions about the SQL statements run as part of the workload there will be a record of the execution plans created by the optimizer available.

Summary

Oracle Database In-Memory can provide up to a 100x performance improvement for analytical style queries, but it is not a one size fits all option. The application workload must be analytical in nature. By analytical we mean that it asks "What if ..." type questions that can be answered by using aggregation to scan and filter data. Optimally Database In-Memory will perform aggregations, scans and filtering in one pass over the data.

Database In-Memory is extremely simple to implement. No application code has to change, and other than allocating additional memory and identifying objects to populate into the IM column store, no other changes have to be made to the Oracle database.

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 12c and is tightly integrated with the Oracle optimizer. The 12c optimizer is more adaptive than it was in previous releases and can be the biggest challenge to a successful Database In-Memory implementation. Hopefully this paper has highlighted these changes and has created a repeatable process to enable successful Database In-Memory implementations.

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

Worldwide Inquiries
Phone: +1.650.506.7000
Fax: +1.650.506.7200

ORACLE

CONNECT WITH US

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

Integrated Cloud Applications & Platform Services

Oracle Database In-Memory Implementation Guidelines
August 2017, Revision 1.1
Author: Andy Rivenes