Oracle® Database

Database In-Memory Guide

19c
E96137-12
February 2024

Oracle Database Database In-Memory Guide, 19c

E96137-12

Primary Author: Lance Ashdown

Contributing Authors: Frederick Kush , Maria Colgan, Vineet Marwah, Andy Rivenes, Randy Urbano

Contributors: Yasin Baskan, Nigel Bayliss, Eric Belden, Larry Carpenter, Shasank Chavan, William Endress, Michael Gleeson, Allison Holloway, Katsumi Inoue, Jesse Kamp, Chinmayi Krishnappa, Vasudha Krishnaswamy, Hariharan Lakshmanan, Sue Lee, Teck Hua Lee, Huagang Li, Yunrui Li, Yuehua Liu, Roger Macnicol, Aurosish Mishra, Ajit Mylavarapu, Khoa Nguyen, Jay Patel, Kathy Rich, Beth Roeser, Rich Strohm, Dina Thomas, Qiuhong Wang, Bob Zebian

Contents

2 In-Memory Column Store Architecture

Part II Configuring and Populating the IM Column Store

3 Enabling and Sizing the IM Column Store

4 Enabling Objects for In-Memory Population

7 Optimizing Queries with In-Memory Expressions

8 Optimizing Joins with Join Groups

9 Optimizing Aggregation

Preface

This manual explains the architecture and tasks associated with the Oracle Database In-Memory feature set.

This preface contains the following topics:

Audience

This document is intended for database administrators who manage an In-Memory Column Store (IM column store), and developers who optimize analytic queries that use Oracle Database In-Memory features.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Related Documents

This manual assumes that you are familiar with *Oracle Database Concepts*. The following books are frequently referenced:

- *Oracle Database Data Warehousing Guide*
- *Oracle Database VLDB and Partitioning Guide*
- *Oracle Database SQL Tuning Guide*
- *Oracle Database SQL Language Reference*
- *Oracle Database Reference*

Many examples in this book use the sample schemas, which are installed by default when you select the Basic Installation option with an Oracle Database. See *Oracle Database Sample Schemas* for information on how these schemas were created and how you can use them.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

Changes in This Release for Oracle Database In-Memory Guide

This preface summarizes the In-Memory features for Oracle Database 19c, Oracle Database 18c, and Oracle Database 12c.

This section contains the following topics:

Changes in Oracle Database Release 19c, Version 19.20

Oracle Database In-Memory Guide for Oracle Database 19c, version 19.20 includes the new `DBMS_INMEMORY_ADVISE` PL/SQL package.

`DBMS_INMEMORY_ADVISE` contains the `IS_INMEMORY_ELIGIBLE` procedure, which you can run to get a quick assessment about whether or not a database workload can benefit from Database In-Memory.

> **See:**
>
> #unique_18

Changes in Oracle Database Release 19c, Version 19.8

Oracle Database In-Memory Guide for Oracle Database 19c, version 19.8 has the following changes.

New Features

The following major features are new in this release.

- Database In-Memory Base Level

  Starting with Oracle Database release 19c, version 19.8, you can enable the Database In-Memory Base Level by setting the `INMEMORY_FORCE` initialization parameter to `BASE_LEVEL`. The Base Level enables you to experiment with In-Memory features without purchasing the Oracle Database In-Memory option.

  When the Base Level is enabled, the IM column store size is limited to 16 GB for a CDB or non-CDB and for every database instance in an Oracle RAC database. Also, the compression level for all objects and columns is set to `QUERY LOW` automatically and transparently, and Automatic In-Memory is disabled.
- Starting with Oracle Database release 19c, version 19.8, you can use the CellMemory feature without enabling the IM column store by setting

`INMEMORY_FORCE=CELLMEMORY_LEVEL` and `INMEMORY_SIZE=0`. With these settings, the IM column store is not enabled and queries can use CellMemory to scan objects.

In previous releases of Oracle Database, in order to use the CellMemory feature, you were required to enable the IM column store, even if you had no intention of using it. This incurred the overhead of enabling the IM column store without any benefit.

See Also:

- See "Enabling the IM Column Store for a Database."
- *Oracle Database Licensing Information User Manual* for details on which features are supported for different editions and services

Changes in Oracle Database Release 19c, Version 19.1

Oracle Database In-Memory Guide for Oracle Database release 19c, version 19.1 has the following changes.

New Features

The following major features are new in this release.

- Database In-Memory Base Level

  Enable the Database In-Memory Base Level by setting the `INMEMORY_FORCE` initialization parameter to `BASE_LEVEL`. The Base Level enables you to experiment with In-Memory features without purchasing the Oracle Database In-Memory option.

  When the Base Level is enabled, the IM column store size is limited to 16 GB for a CDB or non-CDB and for every database instance in an Oracle RAC database. Also, the compression level for all objects and columns is set to `QUERY LOW` automatically and transparently, and Automatic In-Memory is disabled.

  See "Enabling the IM Column Store for a Database" and *Oracle Database Licensing Information User Manual*.

- Database In-Memory wait on populate

  The `DBMS_INMEMORY_ADMIN.POPULATE_WAIT` function initiates population of all `INMEMORY` objects that have a priority greater than or equal to the specified priority, and returns a status value for the population. A user-specified interval specifies the maximum time that the function waits before returning the value to the caller.

  See "Forcing Initial Population of an In-Memory Object".

- Big Data and performance enhancements for In-Memory external tables

  This release introduces several manageability and performance improvements to the In-Memory external tables feature:

  – The `ORACLE_HIVE` and `ORACLE_BIGDATA` drivers are supported.

  – Parallel query is supported.

  - A full table scan populates an In-Memory external table. In previous releases, population required using the `POPULATE` or `REPOPULATE` procedure of `DBMS_INMEMORY`.
  - The In-Memory background processes, rather than a foreground process, now drop IM segments.

  See "In-Memory External Tables".

- Hybrid partitioned tables

  Partitions can reside in both Oracle Database segments and in external files and sources. This feature significantly enhances partitioning for Big Data SQL, where large portions of a table can reside in external partitions. Only the internal partitions of a hybrid partitioned table inherit the `INMEMORY` attribute.

  See "In-Memory Tables".

- Oracle Database Resource Manager automatically enabled for Database In-Memory

  When `INMEMORY_SIZE` is greater than `0`, the Resource Manager is automatically enabled.

  See "Interface for IM Dynamic Scans".

- Oracle Data Guard Multi-Instance Redo Apply supports the IM column store

  Setting the initialization parameter `ENABLE_IMC_WITH_MIRA` to `TRUE` enables the IM column store and Data Guard Multi-Instance Redo Apply at the same time on an Active Data Guard standby database. By default, `ENABLE_IMC_WITH_MIRA` is `FALSE`.

  See *Oracle Database Reference* to learn more about `ENABLE_IMC_WITH_MIRA`.

> **See Also:**
>
> *Oracle Database Licensing Information User Manual* for details on which features are supported for different editions and services

Changes in Oracle Database Release 18c, Version 18.1

Oracle Database In-Memory Guide for Oracle Database release 18c, version 18.1 has the following changes.

New Features

The following major features are new in this release.

- Automatic In-Memory

  This feature uses segment and column usage statistics to manage the contents of the IM column store automatically. If a populate job fails because the IM column store is full, then Automatic In-Memory evicts inactive segments to make room for active segments.

  See "Configuring Automatic In-Memory".

- In-Memory Dynamic Scans

  IM dynamic scans automatically and transparently parallelize table scans by using lightweight process threads. Oracle Resource Manager allocates these threads when it perceives that CPU resources are idle and can be leveraged to speed up the query.

See "In-Memory Dynamic Scans".

- IM expressions window capture

  You can define an expression capture window of an arbitrary length, which ensures that only the expressions occurring within this window are considered for In-Memory materialization. This mechanism is especially useful when you know of a small interval that is representative of the entire workload. For example, during the trading window, a brokerage firm can gather the set of expressions, and materialize them in the IM column store to speed-up future query processing for the entire workload.

  See "Expression Capture Interval".

- In-Memory support for external tables

  You can populate external tables into the IM column store. This feature is useful for analytical queries that combine internal and external data.

  See "In-Memory External Tables" and "Populating an In-Memory External Table Using DBMS_INMEMORY.POPULATE: Example".

- In-Memory Optimized Arithmetic

  For tables compressed with `QUERY LOW`, `NUMBER` columns are encoded using an optimized format that enables native calculations in hardware. SIMD vector processing of aggregations and arithmetic operations that use this format can achieve significant performance gains. The feature is enabled when `INMEMORY_OPTIMIZED_ARITHMETIC` is set to `ENABLE`.

  See "Optimizing In-Memory Arithmetic".

- Enhanced performance for Large Objects (LOBs)

  In previous releases, although LOBs and LOB pointers were populated in the IM column store, the database satisfied queries by using the buffer cache. In this release, In-Memory queries that apply range predicates to scalar columns or SQL operators to LOB columns benefit from SIMD vector processing.

  The IM column store provides contiguous storage for inline LOBs, which are LOBs less than 4 KB, within the IMCUs. For out-of-line LOBs, the IM column store only stores the LOB locator, which is 40 byes. There is one exception to the preceding rule. An IMEU can allocate up to 32 KB of contiguous storage for JSON columns defined as a LOB data type. The IMEU stores these columns in the OSON (binary JSON) format.

  See "CPU Architecture: SIMD Vector Processing".

- In-Memory join group on one column

  You can use the following syntax to create a join group for a self-join on a single column: `CREATE INMEMORY JOIN GROUP jg_name(table_name(column_name))`.

  See "Optimizing Joins with Join Groups".

Changes in Oracle Database 12c Release 2 (12.2.0.1)

Oracle Database In-Memory Guide for Oracle Database 12c Release 2 (12.2.0.1) has the following changes.

New Features

The following major features are new in this release:

- In-Memory Column Store (IM column store) dynamic resizing

  You can now dynamically increase the size of the In-Memory Area without reopening the database.

  See "Increasing the Size of the IM Column Store Dynamically".

- In-Memory Expressions (IM expressions)

  Oracle Database automatically identifies frequently used (“hot”) expressions that are candidates for population in the IM column store. A candidate expression might be `(monthly_sales*12)/52`. IM expressions can greatly improve the performance of analytic queries that use computationally intensive expressions and access large data sets.

  See "Optimizing Queries with In-Memory Expressions".

- In-Memory virtual columns (IM virtual columns)

  IM virtual columns enable the IM column store to materialize some or all virtual columns in a table.

  See "Enabling and Disabling Columns for In-Memory Tables".

- IM FastStart

  IM FastStart optimizes the population of database objects in the IM column store by storing IMCUs directly on disk.

  See "Managing IM FastStart for the IM Column Store".

- Object-level support for services

  For an individual object, the `INMEMORY ... DISTRIBUTE` clause has a `FOR SERVICE` subclause that limits population to the database instance where this service can run. For example, you can configure an `INMEMORY` object to be populated in the IM column store on instance 1 only, or on instance 2 only, or in both instances.

  See "Object-Level Service Controls".

- IM column store on a standby database

  You can enable an IM column store in an Oracle Active Data Guard standby database. You can populate a completely different set of data in the in-memory column store on the primary and standby databases, effectively doubling the size of the in-memory column store that is available to the application.

  See "Deploying an IM Column Store with Oracle Active Data Guard".

- ADO support for the IM column store

  You can use Automatic Data Optimization (ADO) policies to evict objects such as tables, partitions, or subpartitions from the IM column store based on Heat Map statistics. Successful policy completion results in setting `NO INMEMORY` for the specified object.

  See "Enabling ADO for the IM Column Store".

- Join groups

  A join group is a user-created object that lists two columns that can be meaningfully joined. In certain queries, join groups enable the database to eliminate the performance

overhead of decompressing and hashing column values. Join groups require an IM column store.

See "Optimizing Joins with Join Groups".

Part I

Oracle Database In-Memory Concepts

This part introduces the Oracle Database In-Memory (Database In-Memory) feature set, and explains the basic architecture of the In-Memory Column Store (IM column store).

1

Introduction to Oracle Database In-Memory

6.1 Oracle Database In-Memory (Database In-Memory) is a suite of features that greatly improves performance for real-time analytics and mixed workloads. The In-Memory Column Store (IM column store) is the key feature of Database In-Memory. 6.4

> **Note:**
>
> Database In-Memory features require the Oracle Database In-Memory option. For the Database In-Memory Base Level, the IM column store size is limited to 16 GB at the CDB level. See *Oracle Database Licensing Information User Manual* for details on which features are supported for different editions and services.

1.1 Challenges for Analytic Applications

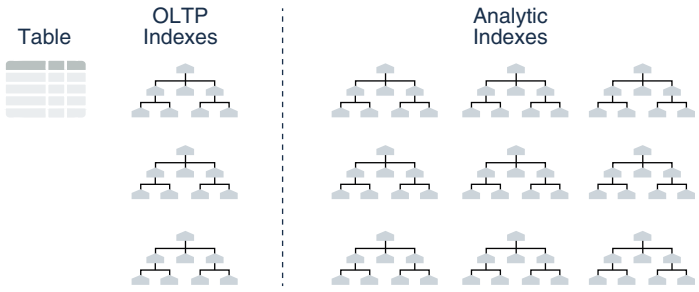
Traditionally, obtaining good performance for analytic queries meant satisfying several requirements.

In a typical data warehouse or mixed-use database, requirements include the following:

- You must understand user access patterns.
- You must provide good performance, which typically requires creating indexes, materialized views, and OLAP cubes.

For example, if you create 1 to 3 indexes for a table (1 primary key and 2 foreign key indexes) to provide good performance for an OLTP application, then you may need to create additional indexes to provide good performance for analytic queries.

Figure 1-1 Multiple Indexes



Meeting the preceding requirements creates manageability and performance problems. Additional access structures cause performance overhead because you must create, manage, and tune them. For example, inserting a single row into a table requires an update to all indexes on this table, which increases response time.

The demand for real-time analytics means that more analytic queries are being executed in a mixed-workload database. The traditional approach is not sustainable.

1.2 The Single-Format Approach

Traditionally, relational databases store data in either row or columnar formats. Memory and disk store data in the same format.

An Oracle database stores rows contiguously in data blocks. For example, in a table with three rows, an Oracle data block stores the first row, and then the second row, and then the third row. Each row contains all column values for the row. Data stored in row format is optimized for transaction processing. For example, updating all columns in a small number of rows may modify only a small number of blocks.

To address the problems relating to analytic queries, some database vendors have introduced a columnar format. A columnar database stores selected columns—not rows—contiguously. For example, in a large sales table, the sales IDs reside in one column, and sales regions reside in a different column.

Analytical workloads access few columns while scanning, but scan the entire data set. For this reason, the columnar format is the most efficient for analytics. Because columns are stored separately, an analytical query can access only required columns, and avoid reading inessential data. For example, a report on sales totals by region can rapidly process many rows while accessing only a few columns.

Database vendors typically force customers to *choose* between a columnar and row-based format. For example, if the data format is columnar, then the database stores data in columnar format both in memory and on disk. Gaining the advantages of one format means losing the advantages of the alternate format. Applications either achieve rapid analytics or rapid transactions, but not both. The performance problems for mixed-use databases are not solved by storing data in a single format.

1.3 The Oracle Database In-Memory Solution

The Oracle Database In-Memory (Database In-Memory) feature set includes the In-Memory Column Store (IM column store), advanced query optimizations, and availability solutions.

The Database In-Memory optimizations enable analytic queries to run orders of magnitude faster on data warehouses and mixed-use databases.

1.3.1 What Is Database In-Memory?

The Database In-Memory feature set includes the IM column store, advanced query optimizations, and availability solutions.

Database In-Memory features combine to accelerate analytic queries by orders of magnitude without sacrificing OLTP performance or availability.

> **See Also:**
>
> *Oracle Database Licensing Information User Manual* to learn about the Database In-Memory option

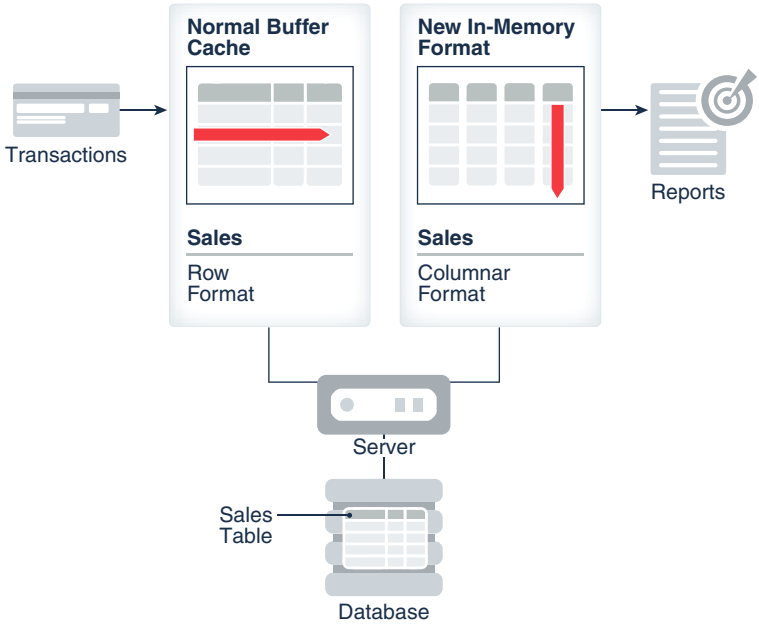
1.3.1.1 IM Column Store

The **IM column store** maintains copies of tables, partitions, and individual columns in a compressed columnar format that is optimized for rapid scans.

> **Video:**
>
> Video

The IM column store resides in the In-Memory Area, which is an optional portion of the system global area (SGA). The IM column store does not *replace* row-based storage or the database buffer cache, but *supplements* it. The database enables data to be in memory in both a row-based and columnar format, providing the best of both worlds. The IM column store provides an additional transaction-consistent copy of table data that is independent of the disk format.

Figure 1-2 Dual-Format Database



> **Note:**
>
> Objects populated in the IM column store do not also need to be loaded into the buffer cache.

Use the `INMEMORY` clause in DDL statements to enable the IM column store at any of the following levels:

- Column (nonvirtual or virtual)
- Table (internal or external), materialized view, or partition

> **Note:**
>
> If you apply the `INMEMORY` attribute to a hybrid partitioned table, then the attribute applies only to the internal partitions.

- Tablespace

If the `INMEMORY` attribute is specified at the tablespace level, then all new tables and materialized views in the tablespace are enabled for the IM column store by default. In the context of Database In-Memory, population is the automatic transformation of row-based data on disk into columnar data in the IM column store. You can configure all or a subset of a database object's columns for population in the IM column store. Similarly, for a partitioned table or materialized view, you can configure all or a subset of the partitions for population.

For example, you might configure three tables from the `sh` schema for population into the IM column store: `customers`, `products`, and `sales`. The IM column store stores the data for each table by column rather than by row, and divides each column into separate row subsets. A special container called an In-Memory Compression Unit (IMCU) stores all columns for a subset of rows in a table segment.

> **See Also:**
>
> - "In-Memory Column Store Architecture"
> - "Enabling the IM Column Store for a Database"
> - *Oracle Database SQL Language Reference* for more information about the `INMEMORY` clause

1.3.1.2 Advanced Query Optimizations

Database In-Memory includes several performance optimizations for analytic queries.

Optimizations include:

- An expression is a combination of one or more values, operators, and SQL functions (`DETERMINISTIC` only) that resolve to a value. By default, the In-Memory Expression (IM expression) optimization enables the `DBMS_INMEMORY_ADMIN.IME_CAPTURE_EXPRESSIONS` procedure to identify and populate “hot” expressions in the IM column store. An IM expression is materialized as a hidden virtual column, but is accessed in the same way as a non-virtual column.
- A join group is a user-defined object that specifies a set of columns frequently used to join a set of tables. In certain queries, join groups enable the database to eliminate the performance overhead of decompressing and hashing column values.

- For aggregation queries that join small dimension tables to a large fact table, In-Memory Aggregation (IM aggregation) uses the `VECTOR GROUP BY` operation to enhance performance. This optimization aggregates data *during* the scan of the fact table rather than afterward.
- In the IM column store, repopulation is the automatic update of IMCUs after the data within them has been significantly modified. If an IMCU has stale entries but does not meet the staleness threshold, then background processes may instigate trickle repopulation, which is the gradual repopulation of the IM column store.

Related Topics

- Optimizing In-Memory Queries
  This Part explains how to optimize queries using In-Memory Expressions, join groups, and In-Memory aggregation. It also explains how the IM column store repopulates modified data.

1.3.1.3 High Availability Support

Availability is the degree to which an application, service, or function is accessible on demand.

Database In-Memory supports the following availability features:

- In-Memory FastStart (IM FastStart) reduces the time to populate data into the IM column store when a database instance restarts. IM FastStart achieves this by periodically saving a copy of the data currently populated in the IM column store on the disk in its compressed columnar format.
- Each node in an Oracle Real Application Clusters (Oracle RAC) environment has its own IM column store. It is possible to have completely different objects populated on every node, or to have larger objects distributed across all IM column stores in the cluster. In Engineered Systems, it is also possible to have the same objects appear in the IM column store on every node.
- Starting in Oracle Database 12c Release 2 (12.2), an IM column store is supported on a standby database in an Active Data Guard environment.

Related Topics

- High Availability and the IM Column Store
  This part explains how to use the IM column store with high availability features such as In-Memory FastStart (IM FastStart), Oracle Data Guard, and Oracle Real Application Clusters (Oracle RAC).

1.3.2 Improved Performance for Analytic Queries

The compressed columnar format enables faster scans, queries, joins, and aggregates.

1.3.2.1 Improved Performance for Data Scans

The columnar format provides fast throughput for scanning large amounts of data.

The IM column store enables you to analyze data in real time, enabling you to explore different possibilities and perform iterations. Specifically, the IM column store can drastically improve performance for queries that do the following:

- Scan many rows and applies filters that use operators such as `<`, `>`, `=`, and `IN`

- Select few columns from a table or a materialized view that has many columns, such as a query that accesses 5 out of 100 columns
- Select LOB columns using SQL operators

Video:

Video

Columnar format uses fixed-width columns for most numeric and short string data types. This optimization enables rapid vector processing, which enables the database to answer queries faster.

Scans of the IM column store are faster than scans of row-based data for the following reasons:

- Elimination of buffer cache overhead

  The IM column store stores data in a pure, in-memory columnar format. The data does not persist in the data files (or generate redo), so the database avoids the overhead of reading data from disk into the buffer cache.

- Data pruning

  The database scans only the columns necessary for the query rather than entire rows of data. Furthermore, the database uses storage indexes and an internal dictionary to read only the necessary IMCUs for a specific query. For example, if a query requests all sales for a store with a store ID less than 8, then the database can use IMCU pruning to eliminate IMCUs that do not contain this value.

- Compression

  Traditionally, the goal of compression is to save space. In the IM column store, the goal of compression is to accelerate scans. The database automatically compresses columnar data using algorithms that allow `WHERE` clause predicates to be applied against the compressed formats. Depending on the type of compression applied, Oracle Database can scan data in its compressed format *without* decompressing it first. Therefore, the volume of data that the database must scan in the IM column store is less than the corresponding volume in the database buffer cache.

- Vector processing

  Each CPU core scans local in-memory columns. To process data as an array (set), the scans use SIMD vector instructions. For example, a query can read a set of values in a single CPU instruction rather than read the values one by one. Vector scans by a CPU core are orders of magnitude faster than row scans.

For example, suppose a user executes the following ad hoc query:

```
SELECT cust_id, time_id, channel_id
FROM   sales
WHERE  prod_id BETWEEN 14 and 29
```

When using the buffer cache, the database would typically scan an index to find the product IDs, use the rowids to fetch the rows from disk into the buffer cache, and then

discard the unwanted column values. Scanning data in row format in the buffer cache requires many CPU instructions, and can result in suboptimal CPU efficiency.

When using the IM column store, the database can scan only the requested `sales` columns, avoiding disk altogether. Scanning data in columnar format pipelines only necessary columns to the CPU, increasing efficiency. Each CPU core scans local in-memory columns using SIMD vector instructions.

Related Topics

- CPU Architecture: SIMD Vector Processing
  For data that is populated in the IM column store, the database uses SIMD (single instruction, multiple data) processing.
- Dual-Format: Column and Row
  When you enable an IM column store, the SGA manages data in separate locations: the In-Memory Area and the database buffer cache.
- Configuring and Populating the IM Column Store
  You can enable and size the In-Memory Column Store (IM column store). You can also configure In-Memory settings for objects, and populate these objects in the IM column store.

1.3.2.2 Improved Performance for Joins

A **Bloom filter** is a low-memory data structure that tests membership in a set. The IM column store takes advantage of Bloom filters to improve the performance of joins.

Bloom filters speed up joins by converting predicates on small dimension tables to filters on large fact tables. This optimization is useful when performing a join of multiple dimensions with one large fact table. The dimension keys on fact tables have many repeat values. The scan performance and repeat value optimization speeds up joins by orders of magnitude.

Related Topics

- About In-Memory Joins
  Joins are an integral part of data warehousing workloads. The IM column store enhances the performance of joins when the tables being joined are stored in memory.

> **See Also:**
>
> "About In-Memory Joins"

1.3.2.3 Improved Performance for Aggregation

An important aspect of analytics is to determine patterns and trends by aggregating data. Aggregations and complex SQL queries run faster when data is stored in the IM column store.

In Oracle Database, aggregation typically involves a `GROUP BY` clause. Traditionally, the database used `SORT` and `HASH` operators. Starting in Oracle Database 12c Release 1 (12.1), the database offered `VECTOR GROUP BY` transformations to enable efficient in-memory, array-based aggregation.

During a fact table scan, the database accumulates aggregate values into in-memory arrays, and uses efficient algorithms to perform aggregation. Joins based on the primary key and foreign key relationships are optimized for both star schemas and snowflake schemas.

> **See Also:**
>
> - "Optimizing In-Memory Aggregation with VECTOR GROUP BY"
> - *Oracle Database Data Warehousing Guide* to learn more about SQL aggregation

1.3.3 Improved Performance for Mixed Workloads

Although OLTP applications do not benefit from accessing data in the IM column store, the dual-memory format can indirectly improve OLTP performance.

When all data is stored in rows, improving analytic query performance requires creating access structures. The standard approach is to create analytic indexes, materialized views, and OLAP cubes. For example, a table might require 3 indexes to improve the performance of the OLTP application (1 primary key and 2 foreign key indexes) and 10-20 additional indexes to improve the performance of the analytic queries. While this technique can improve analytic query performance, it slows down OLTP performance. Inserting a row into the table requires modifying all indexes on the table. As the number of indexes increases, insertion speed decreases.

When you populate data into the IM column store, you can drop analytic access structures. This technique reduces storage space and processing overhead because fewer indexes, materialized views, and OLAP cubes are required. For example, an insert results in modifying 1-3 indexes instead of 11-23 indexes.

While the IM column store can drastically improve performance for analytic queries in business applications, ad hoc analytic queries, and data warehouse workloads, pure OLTP databases that perform short transactions using index lookups benefit less. The IM column store does not improve performance for the following types of queries:

- A query with complex predicates
- A query that selects many columns
- A query that returns many rows

> **See Also:**
>
> *Oracle Database Data Warehousing Guide* to learn more about physical data warehouse design

1.3.4 In-Memory Support for Exadata Flash Cache

Not all objects marked `INMEMORY` may fit in DRAM memory at the same time. If you use Oracle Exadata Storage Server Software, then Exadata Smart Flash Cache can serve as supplemental memory.

When the IM column store is enabled, Exadata Smart Flash Cache reformats data automatically into In-Memory columnar format. In previous Exadata releases, only Hybrid Column Compressed data was eligible for flash storage in IM columnar format. The reformatting occurs for both compressed (including OLTP compression) and uncompressed tables.

> **Note:**
>
> If Database In-Memory Base Level is enabled, then Exadata scans and population for `CELLMEMORY` tables are disabled.

With this format, most Database In-Memory performance enhancements are supported in Smart Scan, including joins and aggregation. Also, reformatting uncompressed and OLTP-compressed data blocks into IM columnar format can significantly reduce the amount of flash memory required.

Exadata Smart Flash Cache transforms the data in the following stages:

1. Oracle Exadata caches data from eligible scans in a legacy columnar format so that the data is available immediately. This format is columnar, but it is not the same format used by the IM column store.
2. In the background, Oracle Exadata reformats data into the pure IM column store format at a lower priority. The background writes prevent interference with the main workload.

If the database is not running an OLTP workload, then a data warehousing workload can consume 100% of the flash cache. However, an OLTP workload limits the data warehouse workload to no more than 50% of the flash cache. This optimization ensures that OLTP workload performance is not sacrificed for analytic scans.

By default, Exadata Smart Flash Cache compresses data using the level `MEMCOMPRESS FOR CAPACITY LOW`. To change the compression level or disable the columnar format altogether, use the `ALTER TABLE ... NO CELLMEMORY` statement.

> **See Also:**
>
> - "Enabling the IM Column Store for a Database"
> - "CPU Architecture: SIMD Vector Processing"
> - *Oracle Exadata Database Machine System Overview* to learn more about the `CELLMEMORY` attribute
> - *Oracle Database Licensing Information User Manual* for details on which features are supported for different editions and services

1.3.5 High Availability Support

The IM column store is fully integrated into Oracle Database. All High Availability features are supported.

The columnar format does not change the Oracle database on-disk storage format. Thus, buffer cache modifications and redo logging function in the same way. Features such as RMAN, Oracle Data Guard, and Oracle ASM are fully supported.

In an Oracle Real Application Clusters (Oracle RAC) environment, each node has its own IM column store by default. Depending on your requirements, you can populate objects in different ways:

- Different tables are populated on every node. For example, the `sales` fact table is on one node, whereas the `products` dimension table is on a different node.
- A single table is distributed among different nodes. For example, different partitions of the same hash-partitioned table are on different nodes, or different rowid ranges of a single nonpartitioned table are on different nodes.
- Some objects appear in the IM column store on every node. For example, you might populate the `products` dimension table in every node, but distribute partitions of the `sales` fact table across different nodes.

> **See Also:**
>
> "High Availability and the IM Column Store"

1.3.6 Ease of Adoption

Database In-Memory is simple to implement, and requires no application changes.

Key aspects of Database In-Memory adoption include:

- Ease of deployment

  No user-managed data migration is required. The database stores data in row format on disk and automatically converts row data into columnar format when populating the IM column store.
- Compatibility with existing applications

  No application changes are required. The optimizer automatically takes advantage of the columnar format. If your application connects to the database and issues SQL, then it can benefit from Database In-Memory features.
- Full SQL compatibility

  Database In-Memory places no restrictions on SQL. Analytic queries can benefit whether they use Oracle analytic functions or customized PL/SQL code.
- Ease of use

  No complex setup is required. The `INMEMORY_SIZE` initialization parameter specifies the amount of memory reserved for use by the IM column store. The `INMEMORY` clause in DDL statements specifies the objects or columns to be

populated into the IM column store. By configuring the IM column store, you can immediately improve the performance of existing analytic workloads and ad hoc queries.

> **See Also:**
>
> - "Enabling and Sizing the IM Column Store" to learn how to enable the IM column store
> - *Oracle Database Reference* to learn about the `INMEMORY_SIZE` and `INMEMORY_FORCE` initialization parameters

1.4 Requirements for Database In-Memory

The Oracle Database In-Memory option is required for all Database In-Memory features. The Database In-Memory Base Level is available for an IM column store that is 16 GB or less.

Requirements include:

- To use the Database In-Memory Base Level, the `INMEMORY_FORCE` initialization parameter must be set to `BASE_LEVEL` in the initialization parameter file at the CDB level. You cannot set this parameter dynamically, or set it at the PDB level. The `BASE_LEVEL` setting has the following consequences:
  - All `INMEMORY` objects and columns automatically and transparently use the compression level of `QUERY LOW`.
  - Automatic In-Memory is disabled.
- For the Base Level, the IM column store size must not exceed 16 GB.
- The IM column store requires a minimum of 100 MB of memory. The store size is included in `MEMORY_TARGET`.
- For Oracle RAC databases, the following additional requirements apply:
  - The `DUPLICATE` and `DUPLICATE ALL` options require Oracle Engineered Systems.
  - If the `INMEMORY_FORCE` initialization parameter is set to `BASE_LEVEL`, then the size of each database instance is limited to 16 GB.

No special hardware is required for an IM column store.

> **See Also:**
>
> - "Estimating the Required Size of the IM Column Store"
> - "Deploying IM Column Stores in Oracle RAC"
> - *Oracle Database Licensing Information User Manual* for all licensing-related information for Database In-Memory

1.5 Principal Tasks for Database In-Memory

For queries to benefit from the IM column store, the only *required* tasks are sizing the IM column store, and specifying objects for population. Query optimization and availability features require additional configuration.

Principal Tasks for Configuring the IM Column Store

The following table lists the principal configuration tasks.

Table 1-1 Configuration Tasks

Task	Notes	To Learn More
Enable the IM column store by specifying its size.	Set `INMEMORY_SIZE` to a minimum of 100 MB. For the In-Memory Free Tier only, the size must be less than or equal to 16 GB for the non-CDB or CDB. In an Oracle RAC database, the size of each instance must be less than or equal to 16 GB. The `COMPATIBLE` initialization parameter must be set to `12.1.0` or higher.	"Enabling the IM Column Store for a Database"
For the Database In-Memory Base Level, perform additional configuration.	For the Database In-Memory Base Level only, the `INMEMORY_FORCE` initialization parameter must be set to `BASE_LEVEL` at the CDB level, and `INMEMORY_SIZE` must be less than or equal to 16 GB.	
For the In-Memory Free Tier only, perform additional configuration.	The `INMEMORY_FORCE` initialization parameter must be set to `FREE`.	"Enabling the IM Column Store for a Database"
Specify tables (internal or external), columns (nonvirtual or virtual), tablespaces, or materialized views for population into the IM column store.	The `INMEMORY` clause enables an object for the IM column store, but does not immediately populate it.	"Enabling Objects for In-Memory Population"
Optionally, create Automatic Data Optimization (ADO) policies to set `INMEMORY` attributes on objects in the IM column store.	For example, a policy can evict the `sales` table from the IM column store after 10 days of no access. In-Memory ADO features require that `HEAT_MAP=ON` is set to `ON` and `INMEMORY_SIZE` is set to a nonzero value.	"Enabling ADO for the IM Column Store"

Table 1-1 (Cont.) Configuration Tasks

Task	Notes	To Learn More
Optionally, configure Automatic In-Memory to evict cold segments to ensure that the working data set is always populated	For example, a populate job only loads half a table partition because the IM column store is almost at maximum capacity, which triggers Automatic In-Memory to evict a cold segment. The next populate job for the new partition completely populates it. In-Memory ADO features require that `AUTOMATIC_INMEMORY_LEVEL` is set to `LOW` or `MEDIUM` and `INMEMORY_SIZE` is set to a nonzero value.	"Configuring Automatic In-Memory"

Principal Tasks for Optimizing In-Memory Queries

In-Memory query optimizations are not required for the IM column store to function. The following optimization tasks are optional.

Table 1-2 Query Optimization Tasks

Task	Notes	To Learn More
Manage automatic detection of IM expressions in the IM column store by using the `DBMS_INMEMORY_ADMIN` package.	For example, invoke the `IME_CAPTURE_EXPRESSIONS` procedure to define the period in which the database can identify “hot” expressions, and then gradually populate them. The `INMEMORY_EXPRESSIONS_USAGE` initialization parameter controls the type of IM expression that the database can populate: static, dynamic, or both.	"INMEMORY_EXPRESSIONS_USAGE"
Define join groups using the `CREATE INMEMORY JOIN GROUP` statement.	Candidates are columns that are frequently paired in a join predicate, for example, a column joining a fact and dimension table.	"Creating Join Groups"
If necessary for a query block, specify the `VECTOR_TRANSFORM` hint to enable in-memory aggregation, or `NO_VECTOR_TRANSFORM` to disable it.	In-memory aggregation is an automatically enabled feature that cannot be controlled with initialization parameters or DDL.	"Controls for IM Aggregation"

Table 1-2 (Cont.) Query Optimization Tasks

Task	Notes	To Learn More
Limit the number of IMCUs updated through trickle repopulation within a two minute interval by setting the initialization parameter `INMEMORY_TRICKLE_REPOPULATE_SERVERS_PERCENT`.	You can disable trickle repopulation by setting this initialization parameter to `0`.	"Threshold-Based and Trickle Repopulation"

Principal Tasks for Managing Availability

The principal tasks are shown in the following table.

Table 1-3 Availability Tasks

Task	Notes	To Learn More
Specify an In-Memory FastStart (IM FastStart) tablespace using the `DBMS_INMEMORY_ADMIN.ENABLE_FASTSTART` procedure.	IM FastStart optimizes the population of database objects in the IM column store when the database is restarted. IM FastStart stores information on disk for faster population of the IM column store.	"Enabling IM FastStart for the IM Column Store"
For an object or tablespace, specify `INMEMORY` in DDL statement with the `DISTRIBUTE` or `DUPLICATE` keywords to control the distribution of data in Oracle RAC.	By default, each In-Memory object is distributed among the Oracle RAC instances, effectively employing a share-nothing architecture for the IM column store.	"Deploying IM Column Stores in Oracle RAC"
In an Oracle Data Guard environment, you can use the same Database In-Memory initialization parameters and statements on a primary or standby database.	For example, you can enable the IM column store on both a primary and standby database by setting `INMEMORY_SIZE`. Optionally, use the `INMEMORY DISTRIBUTE FOR SERVICE` clause in DDL to populate a different set of data in the IM column store on the primary and standby databases.	"About Manually Enabling Objects for In-Memory Population"

1.6 Tools for the IM Column Store

No special tools or utilities are required to manage the IM column store or other Database In-Memory features. Administrative tools such as SQL*Plus, SQL Developer, and Oracle Enterprise Manager (Enterprise Manager) are fully supported.

This section describes tools that have specific Database In-Memory feature support.

1.6.1 In-Memory Eligibility Test

The In-Memory Eligibility Test helps you to determine whether or not a database workload will benefit from use of the Database In-Memory feature.

Many workloads benefit from Database In-Memory, however some may not. The In-Memory Eligibility Test determines if a given workload would benefit or not benefit from Database In-Memory and assesses its eligibility for use of this feature. Eligibility is gauged by the percentage of analytical activity in the workload. If you are planning to implement Database In-Memory, you can use this tool to quickly identify and filter out databases that are ineligible -- those where analytic activity is low and where you would see no substantive gain from the use of Database In-Memory. You can then focus your Database In-Memory deployment on databases whose workload includes more intense analytic activity and could therefore benefit substantially. The higher the percentage of analytical activity in the workload, the more benefit you gain from Database In-Memory.

The In-Memory Eligibility Test is the `IS_INMEMORY_ELIGIBLE` procedure within the PL/SQL package `DBMS_INMEMORY_ADVISE`. This package is built into Oracle Database. You do not need to download and install it.

Running the In-Memory Eligibility Test is a preliminary step you should perform before you run In-Memory Advisor on any database where you are considering enabling Database In-Memory. This can save time, because the In-Memory Eligibility Test does a quick analysis that tells you up front, before you run In-Memory Advisor (which takes longer), whether or not Database In-Memory is appropriate for the given workload. You should skip running In-Memory Advisor on workloads where the In-Memory Eligibility Test tells you that the level of analytic activity is insufficient to warrant the use of Database In-Memory.

If you are considering whether or not to enable Database In-Memory on your databases, do the following:

1. Run the In-Memory Eligibility Test on candidate database workloads to find out which workloads can or cannot effectively use Database In-Memory.
2. Run the In-Memory Advisor on any workload, except those that the In-Memory Eligibility Test has determined are ineligible.

See Also:

- *The Database PL/SQL Packages and Types Reference* documents the DBMS_INMEMORY_ADVISE package, including the `IS_INMEMORY_ELIGIBLE` procedure for testing eligibility for Database In-Memory.
- The Exporting AWR Data in the *Database Performance Tuning Guide* explains how to export AWR snapshot data.
- In-Memory Advisor explains how to download and use the In-Memory Advisor.

1.6.2 In-Memory Advisor

The **In-Memory Advisor** is a downloadable PL/SQL package that analyzes the analytical processing workload in your database.

The In-Memory Advisor differentiates analytics processing from other database activity based on SQL plan cardinality, Active Session History (ASH), parallel query usage, and other statistics. The In-Memory Advisor estimates the size of objects in the IM column store based on statistics and heuristic compression factors.

The advisor estimates analytic processing performance improvement factors based on the following:

- Elimination of wait events such as user I/O waits, cluster transfer waits, and buffer cache latch waits
- Query processing advantages related to specific compression types
- Decompression cost heuristics for specific compression types
- SQL plan cardinality, number of columns in the result set, and so on

The output is a report that recommends a size for the IM column store and a list of objects that would benefit from In-Memory population. The advisor also generates a SQL*Plus script that alters the recommended objects with the `INMEMORY` clause.

The In-Memory Advisor is *not* included in the stored PL/SQL packages. You must download the package from Oracle Support.

> **See Also:**
>
> My Oracle Support note 1965343.1 to learn more about the In-Memory Advisor

1.6.3 Cloud Control Pages for the IM Column Store

Enterprise Manager Cloud Control (Cloud Control) provides the In-Memory Column Store Central Home page. This page gives a dashboard interface to the IM column store.

Use this page to monitor in-memory support for database objects such as tables, indexes, partitions and tablespaces. You can view In-Memory functionality for objects and monitor their In-Memory usage statistics. Unless otherwise stated, this manual describes the command-line interface to Database In-Memory features.

Related Topics

- Using IM Column Store in Cloud Control
  You can configure and manage the IM column store in Oracle Enterprise Manager Cloud Control (Cloud Control).

> **See Also:**
>
> "Using IM Column Store in Cloud Control" explains how to use Cloud Control to manage the IM column store.

1.6.4 Oracle Compression Advisor

Oracle Compression Advisor estimates the compression ratio that you can realize using the `MEMCOMPRESS` clause. The advisor uses the `DBMS_COMPRESSION` interface.

See Also:

- "Oracle Compression Advisor"
- *Oracle Database PL/SQL Packages and Types Reference* to learn more about `DBMS_COMPRESSION`

1.6.5 Oracle Data Pump and the IM Column Store

You can import database objects that are enabled for the IM column store using the `TRANSFORM=INMEMORY:y` option of the `impdp` command.

With this option, Oracle Data Pump keeps the IM column store clause for all objects that have one. When the `TRANSFORM=INMEMORY:n` option is specified, Data Pump drops the IM column store clause from all objects that have one.

You can also use the `TRANSFORM=INMEMORY_CLAUSE:`*`string`* option to override the IM column store clause for a database object in the dump file during import. For example, you can use this option to change the IM column store compression for an imported database object.

Video:

Video

See Also:

Oracle Database Utilities for more information about the `TRANSFORM impdb` parameter

2

In-Memory Column Store Architecture 6.4

The **In-Memory Column Store** (IM column store) stores tables and partitions in memory using a **columnar format** optimized for rapid scans. Oracle Database uses a sophisticated architecture to manage data in columnar and row formats simultaneously.

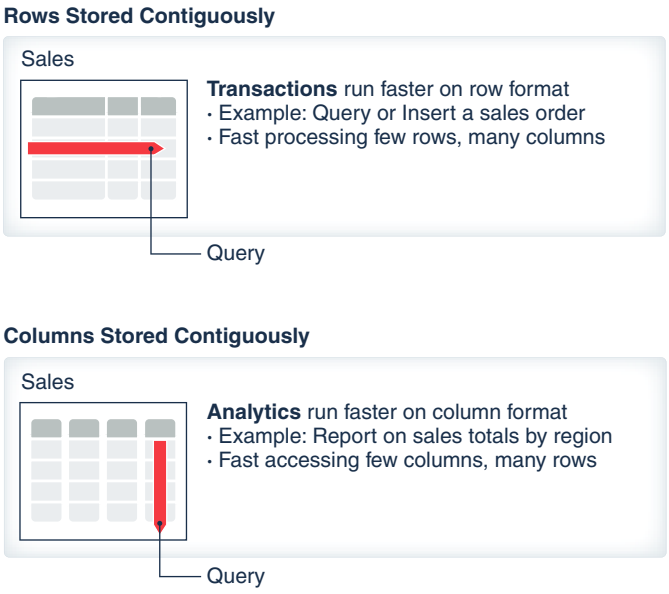
2.1 Dual-Format: Column and Row

When you enable an IM column store, the SGA manages data in separate locations: the In-Memory Area and the database buffer cache.

The IM column store encodes data in a columnar format: each column is a separate structure. The columns are stored contiguously, which optimizes them for analytic queries. The database buffer cache can modify objects that are also populated in the IM column store. However, the buffer cache stores data in the traditional row format. Data blocks store the rows contiguously, optimizing them for transactions.

The following figure illustrates the difference between row-based storage and columnar storage.

Figure 2-1 Columnar and Row-Based Storage



2.1.1 Columnar Data in the In-Memory Area

The **In-Memory Area** is an optional SGA component that contains the IM column store.

2.1.1.1 Size of the In-Memory Area 6.2

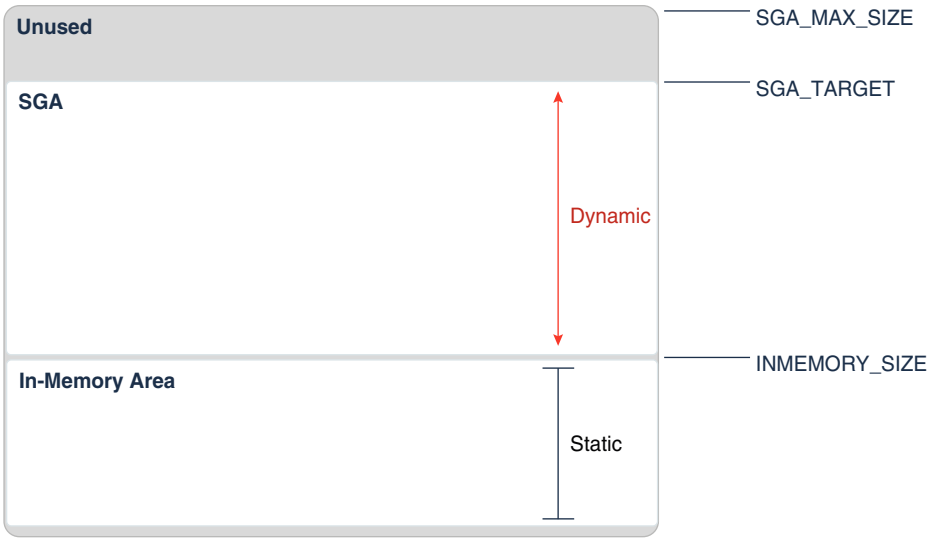
The In-Memory Area is controlled by the `INMEMORY_SIZE` initialization parameter. By default, the size of the In-Memory Area is 0, which means the IM column store is disabled.

To enable the IM column store, set the In-Memory Area to at least 100 MB. The size is shown in `V$SGA`.

The In-Memory Area and SGA_TARGET

The In-Memory Area is subtracted from the `SGA_TARGET` initialization parameter setting. For example, if you set `SGA_TARGET` to 10 GB, and if you set the `INMEMORY_SIZE` to 4 GB, then 40% of the `SGA_TARGET` setting is allocated to the In-Memory Area. The following graphic illustrates the relationship.

Figure 2-2 INMEMORY_SIZE and SGA_TARGET



Unlike the other components of the SGA, including the buffer cache and the shared pool, the In-Memory Area size is not controlled by automatic memory management. The database does not automatically shrink the In-Memory Area when the buffer cache or shared pool requires more memory, or increase the In-Memory Area when it runs out of space.

Dynamic Resizing of the In-Memory Area

Starting in Oracle Database 12c Release 2 (12.2), you can dynamically increase `INMEMORY_SIZE` by using the `ALTER SYSTEM` statement. The database allocates increased memory when the following conditions are met:

- Free memory is available in the SGA.
- The new size for `INMEMORY_SIZE` is at least 128 MB greater than the current setting.

> **Note:**
>
> You cannot use `ALTER SYSTEM` to reduce `INMEMORY_SIZE`.

The `V$INMEMORY_AREA` and `V$SGA` views immediately reflect the change.

The In-Memory Area in a Multitenant Environment

In a CDB, the size of the IM column store is set by the `INMEMORY_SIZE` parameter in the CDB root. By default, the IM column store is shared among the PDBs. Consequently, a PDB can "starve" other PDBs by consuming the available memory.

Within a PDB, you can limit memory consumption by using `ALTER SYSTEM SET INMEMORY_SIZE`. For example, at the CDB level, you might set `INMEMORY_SIZE` to `20G`, and then configure the PDBs as follows:

- In `hrpdb`, set `INMEMORY_SIZE` to `0`
- In `salespdb`, set `INMEMORY_SIZE` to `10G`
- In `oepdb`, set `INMEMORY_SIZE` to `11G`

In the preceding example, the `INMEMORY_SIZE` settings at the PDB level add up to `21G`, even though `INMEMORY_SIZE` at the CDB level is only `20G`. Oversubscription ensures that valuable space in the IM column store is not wasted if a PDB is shut down or unplugged.

> **See Also:**
>
> - "Increasing the Size of the IM Column Store Dynamically"
> - *Oracle Database Administrator's Guide* to learn more about automatic memory management
> - *Oracle Database Reference* to learn about `INMEMORY_SIZE`, `V$INMEMORY_AREA`, and `V$SGA`

2.1.1.2 Memory Pools in the In-Memory Area

The In-Memory Area is divided into subpools for columnar data and metadata.

The In-Memory area is subdivided into the following subpools:

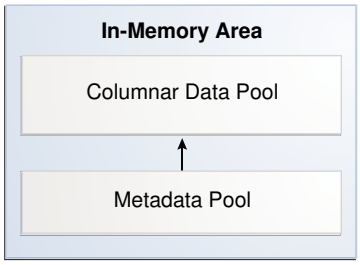
- The columnar data pool

  This subpool stores the IMCUs, which contain the columnar data. The `V$INMEMORY_AREA.POOL` column identifies this subpool as `1MB POOL`, as shown in Example 2-1.

- The metadata pool

  This subpool stores metadata about the objects that reside in the IM column store. The `V$INMEMORY_AREA.POOL` column identifies this subpool as `64KB POOL`, as shown in Example 2-1.

Figure 2-3 Subpools in the In-Memory Area



The database determines the relative size of the two subpools using internal heuristics. The database allocates the majority of space in the In-Memory Area to the columnar data pool (1 MB pool).

> **Note:**
>
> Oracle Database automatically determines the subpool sizes. You cannot change the space allocations.

Example 2-1 V$INMEMORY_AREA View

This example queries the `V$INMEMORY_AREA` view to determine the amount of available memory in each subpool (sample output included):

```
COL POOL FORMAT a9
COL POPULATE_STATUS FORMAT a15
SSELECT POOL, TRUNC(ALLOC_BYTES/(1024*1024*1024),2) "ALLOC_GB",
        TRUNC(USED_BYTES/(1024*1024*1024),2) "USED_GB",
        POPULATE_STATUS
FROM    V$INMEMORY_AREA;

POOL      ALLOC_GB   USED_GB    POPULATE_STATUS
--------- ---------- ---------- ---------------
1MB POOL  7.99       0          DONE
64KB POOL 1.98       0          DONE
```

The current size of the In-Memory area is visible in `V$SGA`:

```
SELECT NAME, VALUE/(1024*1024*1024) "SIZE_IN_GB"
FROM   V$SGA
WHERE  NAME LIKE '%Mem%';

NAME                SIZE_IN_GB
------------------- ----------
In-Memory Area      10
```

In this example, the memory allocated to the subpools is 9.97 GB, whereas the size of the In-Memory Area is 10 GB. The database uses a small percentage of memory for internal management structures.

> **See Also:**
>
> *Oracle Database Reference* to learn about `V$INMEMORY_AREA`

2.1.2 Row Data in the Database Buffer Cache

The database buffer cache stores and processes data blocks in the same way whether the IM column store is enabled or disabled. Buffer I/O and buffer pools function the same.

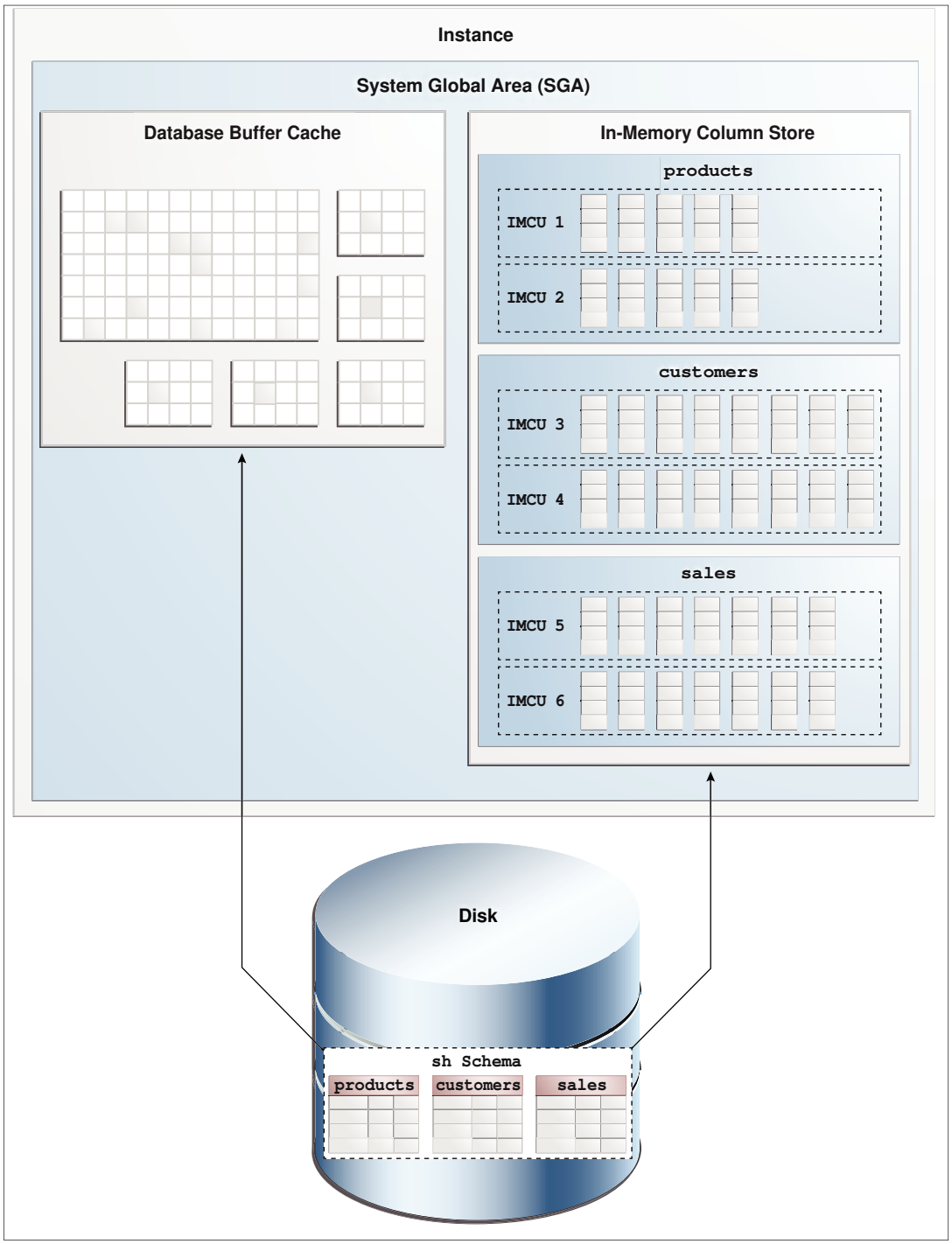
The IM column store enables data to be simultaneously populated in the SGA in both the traditional row format (the buffer cache) and the columnar format. The database transparently sends OLTP queries (such as primary key lookups) to the buffer cache, and analytic and reporting queries to the IM column store. When fetching data, Oracle Database can also read data from both memory areas within the same query.

> **Note:**
>
> In the execution plan, the operation `TABLE ACCESS IN MEMORY FULL` indicates that some or all data is accessed in the IM column store.

The dual-format architecture does not double memory requirements. The buffer cache is optimized to run with a much smaller size than the size of the database.

The following figure shows a sample IM column store. The database stores the `sh.sales` table on disk in traditional row format. The SGA stores the data in columnar format in the IM column store, and in row format in the database buffer cache.

Figure 2-4 IM Column Store



Every on-disk data format for permanent, heap-organized tables is supported by the IM column store. The columnar format does not affect the format of data stored in data files or in the buffer cache, nor does it affect undo data and online redo logging.

The database processes DML modifications in the same way, regardless of whether the IM column store is enabled, by updating the buffer cache, online redo log, and undo tablespace. However, the database uses an internal mechanism to track

changes and ensure that the IM column store is consistent with the rest of the database. For example, if the `sales` table is populated in the IM column store, and if an application updates a row in `sales`, then the database automatically keeps the copy of the `sales` table in the IM column store transactionally consistent. A query that accesses the IM column store always returns the same results for a query that accesses the buffer cache.

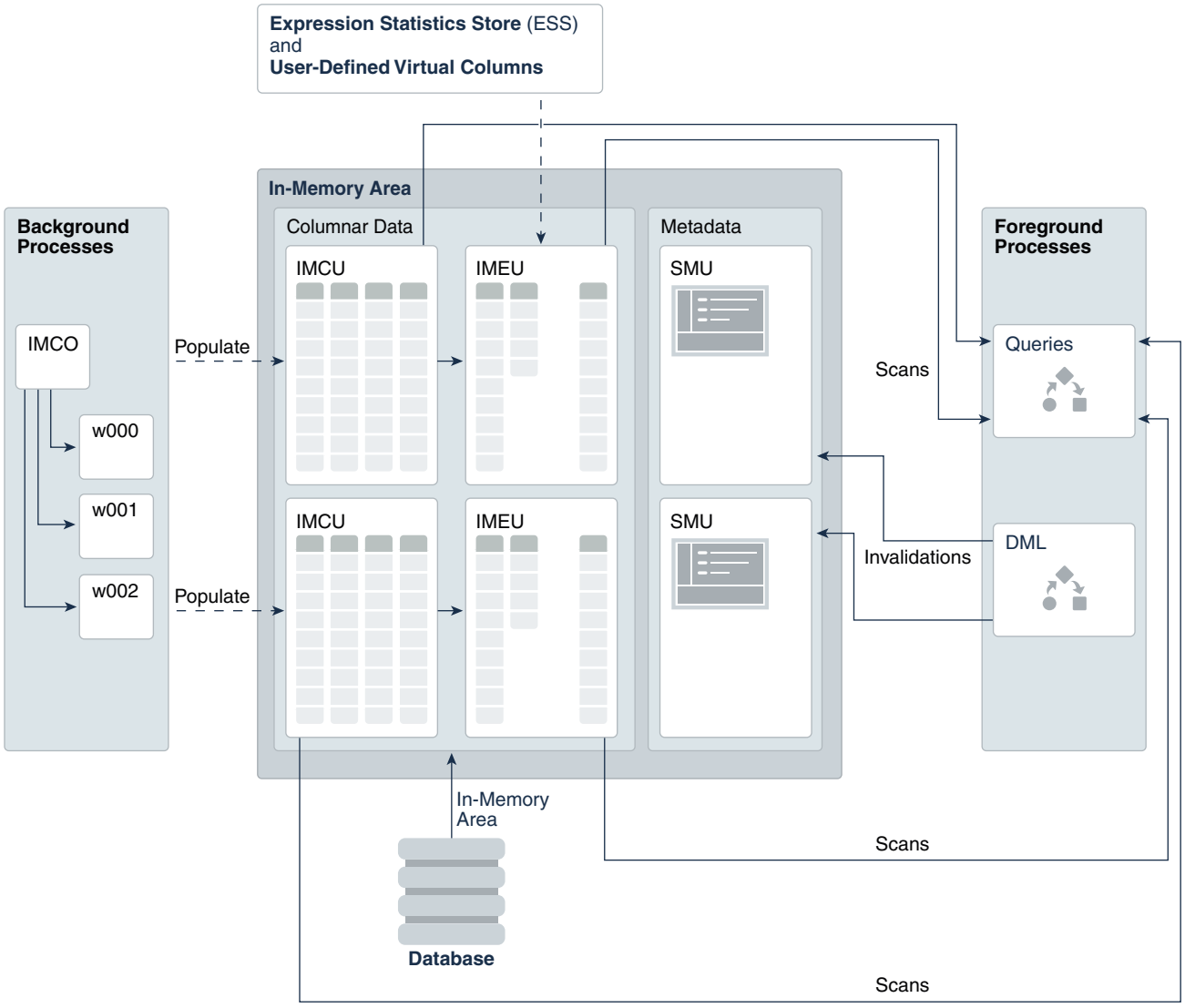
> **See Also:**
>
> *Oracle Database Concepts* to learn more about the database buffer cache

2.2 In-Memory Storage Units

The IM column store manages both data and metadata in optimized storage units, not in traditional Oracle data blocks.

Oracle Database maintains the storage units in the In-Memory Area. The following graphic gives an overview of the In-Memory Area and the database processes that interact with it. The remaining chapter describes the various memory components.

Figure 2-5 IM Column Store: Memory and Process Architecture



2.2.1 In-Memory Compression Units (IMCUs)

An **In-Memory Compression Unit (IMCU)** is a compressed, read-only storage unit that contains data for one or more columns.

An IMCU is analogous to a tablespace extent. An IMCU has two parts: a set of Column Compression Units (CUs), and a header that contains metadata such as the IM storage index.

2.2.1.1 IMCUs and Schema Objects

The IM column store stores data for a single object (table, partition, materialized view) in a set of IMCUs. An IMCU stores columnar data for one and only one object.

2.2.1.1.1 IMCUs and INMEMORY Columns

For an object specified as `INMEMORY`, every column listed in the `INMEMORY` clause is included in every IMCU.

For example, the `sh.sales` table has 7 columns. The following DDL statement specifies the table as `INMEMORY`, which means that every IMCU for `sales` includes columnar data for these 7 columns:

```
ALTER TABLE sh.sales INMEMORY MEMCOMPRESS FOR QUERY LOW;
```

NO INMEMORY Columns in INMEMORY Objects

You can specify that some but not all columns in an `INMEMORY` table have the `INMEMORY` attribute. For example, the `sh.customers` table has 23 columns. The following DDL statement specifies that 15 columns in `sh.customers` have the `NO INMEMORY` attribute, which means that the other 8 columns in the table have the `INMEMORY` attribute:

```
ALTER TABLE sh.customers INMEMORY
  MEMCOMPRESS FOR QUERY LOW
  NO INMEMORY ( cust_gender, cust_year_of_birth, cust_marital_status,
                cust_postal_code, cust_city, cust_state_province,
                cust_main_phone_number, cust_income_level, cust_credit_limit,
                cust_email, cust_total, cust_total_id, cust_eff_from,
                cust_eff_to, cust_valid );
```

The following query shows the compression levels of the columns in `sh.customers`, indicating which columns are `NO INMEMORY`:

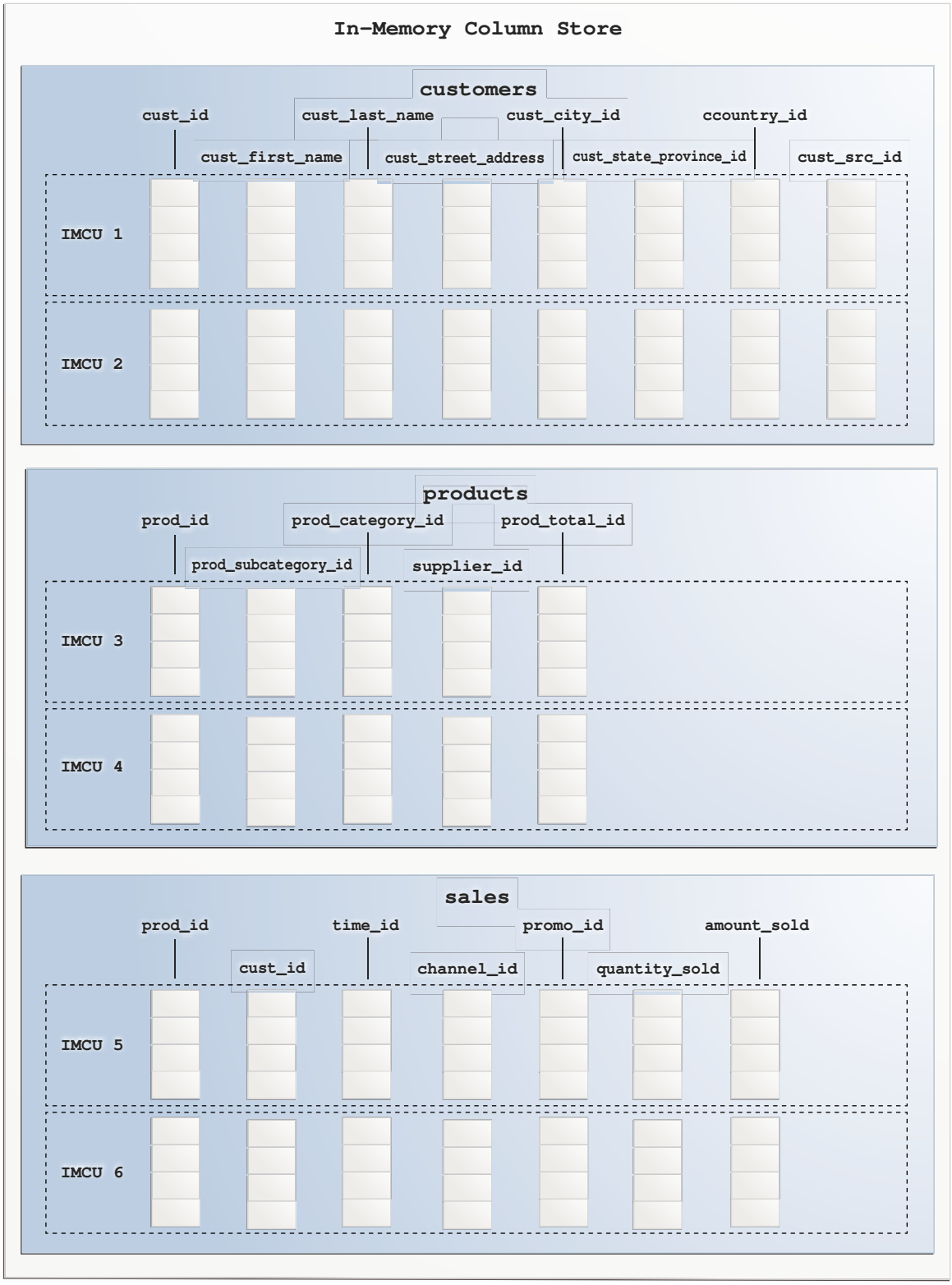
```
SET LINESIZE 200
COL TABLE_NAME FORMAT a25
COL SEG_COL_ID FORMAT 999
COL COLUMN_NAME FORMAT a25
COL INMEMORY_COMPRESSION FORMAT a11
SELECT SEGMENT_COLUMN_ID AS "SEG_COL_ID", COLUMN_NAME, INMEMORY_COMPRESSION
FROM   V$IM_COLUMN_LEVEL WHERE TABLE_NAME = 'CUSTOMERS'
ORDER BY SEG_COL_ID;

SEG_COL_ID COLUMN_NAME               INMEMORY_CO
---------- ------------------------- -----------
         1 CUST_ID                   DEFAULT
         2 CUST_FIRST_NAME           DEFAULT
         3 CUST_LAST_NAME            DEFAULT
         4 CUST_GENDER               NO INMEMORY
         5 CUST_YEAR_OF_BIRTH        NO INMEMORY
         6 CUST_MARITAL_STATUS       NO INMEMORY
         7 CUST_STREET_ADDRESS       DEFAULT
         8 CUST_POSTAL_CODE          NO INMEMORY
         9 CUST_CITY                 NO INMEMORY
        10 CUST_CITY_ID              DEFAULT
        11 CUST_STATE_PROVINCE       NO INMEMORY
        12 CUST_STATE_PROVINCE_ID    DEFAULT
        13 COUNTRY_ID                DEFAULT
```

```
14 CUST_MAIN_PHONE_NUMBER    NO INMEMORY
15 CUST_INCOME_LEVEL         NO INMEMORY
16 CUST_CREDIT_LIMIT         NO INMEMORY
17 CUST_EMAIL                NO INMEMORY
18 CUST_TOTAL                NO INMEMORY
19 CUST_TOTAL_ID             NO INMEMORY
20 CUST_SRC_ID               DEFAULT
21 CUST_EFF_FROM             NO INMEMORY
22 CUST_EFF_TO               NO INMEMORY
23 CUST_VALID                NO INMEMORY
```

The following graphic represents three tables from the `sh` schema populated in the IM column store: `customers`, `products`, and `sales`. In this example, each table has a different number of columns specified `INMEMORY`. The IMCUs for each table include only data for the `INMEMORY` columns.

Figure 2-6 Columns and IMCUs



Queries That Reference NO INMEMORY Columns

When a query references a `NO INMEMORY` column, the table scan retrieves data from the row store rather than the IMCUs in the IM column store. Row store access occurs even if all other columns referenced in the query are populated `INMEMORY` columns.

For example, assume that the `customers` table is populated into the IM column store. The `cust_id` and `cust_last_name` columns are specified `INMEMORY`, but the `cust_postal_code` column is specified as `NO INMEMORY`. You issue the following query:

```
SELECT cust_id, cust_last_name, cust_postal_code
FROM   customers
WHERE  cust_id < 5001
ORDER BY cust_id;
```

In this case, the database accesses the row store, not the IM column store, even though `cust_postal_code` is the only `NO INMEMORY` column referenced in the query. The following query, which has `cust_postal_code` in the predicate but not the `SELECT` list, must also access the row store:

```
SELECT cust_id, cust_last_name
FROM   customers
WHERE  cust_postal_code = 77501
ORDER BY cust_id;
```

See Also:

- "About Enabling INMEMORY Columns"
- https://blogs.oracle.com/in-memory/what-happens-if-a-column-is-not-populated for a blog entry on accessing columns that are not populated in the IM column store
- *Oracle Database SQL Language Reference* to learn about the `ALTER TABLE` statement

2.2.1.1.2 In-Memory Compression

The IM column store uses special compression formats optimized for access speed rather than storage reduction. The columnar format enables queries to execute directly against the compressed columns.

Compression enables scanning and filtering operations to process a much smaller amount of data, which optimizes query performance. Oracle Database only decompresses data when it is required for the result set.

The compression applied in the IM column store is closely related to Hybrid Columnar Compression. Both technologies process column vectors. The primary difference is that the column vectors for the IM column store are optimized for SIMD vector processing, whereas the column vectors for Hybrid Columnar Compression are optimized for disk storage.

When you enable an object for population into the IM column store, you specify the type of compression in the `INMEMORY` clause: `FOR DML`, `FOR QUERY` (`LOW` or `HIGH`), `FOR CAPACITY` (`LOW` or `HIGH`), or `NONE`.

> **See Also:**
>
> - "Controls for In-Memory Objects"
> - *Oracle Database Concepts* to learn more about Hybrid Columnar Compression

2.2.1.1.3 IMCUs and Rows

Each IMCU contains all column values (including nulls) for a subset of rows in a table segment. A subset of rows is called a *granule*.

All IMCUs for a given segment contain approximately the same number of rows. Oracle Database determines the size of a granule automatically depending on data type, data format, and compression type. A higher compression level results in more rows in the IMCU.

A one-to-many mapping exists between an IMCU and a set of database blocks. As illustrated in Example 2-2, each IMCU stores the values for columns for a different set of blocks.

The columns in an IMCU are not sorted. Oracle Database populates them in the order that they are read from disk.

The number of rows in an IMCU dictates the amount of space an IMCU consumes. If the target number of rows causes an IMCU to grow beyond the amount of contiguous 1 MB extents available in the 1 MB pool, then the IMCU creates additional extents (pieces) to hold the remaining column CUs. An IMCU always allocates space in 1 MB increments.

Example 2-2 IMCUs and Row Subsets

In this simplified example, only the following 4 columns of the `customers` table have the `INMEMORY` attribute: `cust_id`, `cust_first_name`, `cust_last_name`, and `cust_gender`. Only 5 rows exist in the table, stored in 2 data blocks. Conceptually, the first data block stores its rows as follows:

```
82,Madeline,Li,F;37004,Abel,Embrey,M;1714,Hardy,Gentle,M
```

The second data block stores rows as follows:

```
100439,Uma,Campbell,F;3047,Lucia,Downey,F
```

Assume IMCU 1 stores the data for the first data block. In this case, the `cust_id` column values for the 3 rows in this data block stores are stored “vertically” within a CU as follows:

```
82
37004
1714
```

IMCU 2 stores the data from the second data block. The `cust_id` column values for these 2 rows are stored within a CU as follows:

```
100439
3047
```

Because the `cust_id` value is the first value for each row in the data block, the `cust_id` column is in the first position within the IMCU. Columns always occupy the same position, so Oracle Database can reconstruct the rows by reading the IMCUs for a segment.

Related Topics

- Controls for In-Memory Objects
  Use the `INMEMORY` clause in DDL statements to specify which objects are eligible for population into the IM column store. You can enable tablespaces, tables (internal and external), partitions, and materialized views.

2.2.1.2 Column Compression Units (CUs)

A **Column Compression Unit (CU)** is contiguous storage for a single column in an IMCU. Every IMCU has one or more CUs.

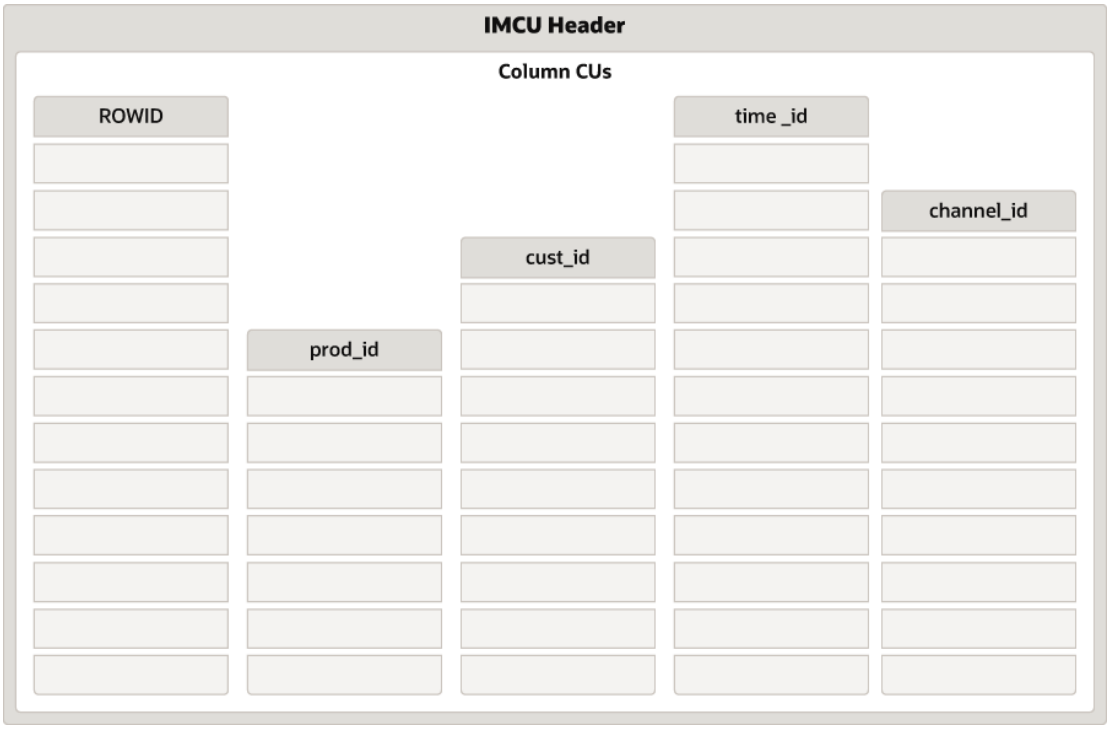
2.2.1.2.1 Structure of a CU

A CU is divided into a body and a header.

The body of every CU stores the column values for the range of rows included in the IMCU. The header contains metadata about the values stored in the CU body, for example, the minimum and maximum value within the CU. It may also contain a local dictionary, which is a sorted list of the distinct values in that column and their corresponding dictionary codes.

The following figure shows an IMCU with 4 CUs for the `sales` table: `prod_id`, `cust_id`, `time_id`, and `channel_id`. Each CU stores the column values for the range of rows included in the IMCU.

Figure 2-7 CUs in an IMCU



The CUs store values in rowid order. For this reason, the database can answer queries by "stitching" the rows back together. For example, an application issues the following query:

```
SELECT cust_id, time_id, channel_id
FROM   sales
WHERE  prod_id =5;
```

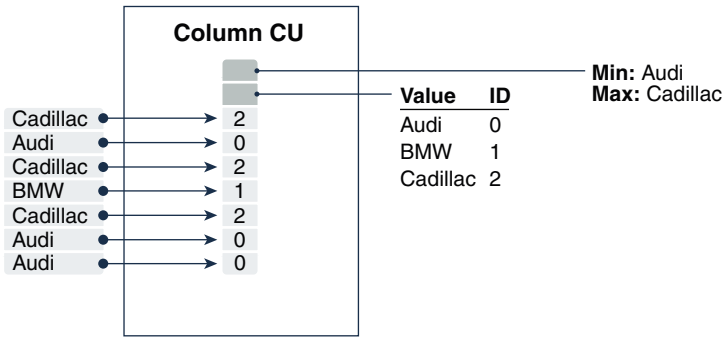
The database begins by scanning the `prod_id` column for entries with the value `5`. Assume that the database finds `5` in position two in the `prod_id` column. The database now must find the corresponding `cust_id`, `time_id`, and `channel_id` for this row. Because the CUs store data in rowid order, the database can find the corresponding `cust_id`, `time_id`, and `channel_id` values in position 2 in those columns. Thus, to answer the query, the database must extract the values from position 2 in the `cust_id`, `time_id`, and `channel_id` columns, and then stitch the row back together to return it to the end user.

2.2.1.2.2 Local Dictionary

In a CU, the local dictionary has a list of distinct values and their corresponding dictionary codes.

The local dictionary stores the symbol contained in the column. The following figure illustrates how a CU stores a `name` column in a `vehicles` table.

Figure 2-8 Local Dictionary



In the preceding figure, the CU contains only 7 rows. Every distinct value in this CU, such as `Cadillac` or `Audi`, is assigned a different dictionary code, such as `2` for `Cadillac` and `0` for `Audi`. The CU stores the dictionary code rather than the original value.

> **Note:**
>
> When the database uses a common dictionary for a join group, the local dictionary contains *references* to the common dictionary rather than the *symbols*. For example, rather than storing the values `Audi`, `BWM`, and `Cadillac` for the `vehicles.name` column, the local dictionary stores dictionary codes such as `101`, `220`, and `66`.

The CU header contains the minimum and maximum values for the column. In this example, the minimum value is `Audi` and the maximum value is `Cadillac`. The local dictionary stores the list of distinct values: `Audi`, `BMW`, and `Cadillac`. Their corresponding dictionary codes (`0`, `1`, and `2`) are implicit. The local dictionary for a CU in each IMCU is independent of the local dictionaries in other IMCUs.

If a query filters on Audi automobiles, then the database scans this IMCU for only `0` codes.

Related Topics

- How a Join Group Uses a Common Dictionary
  A **common dictionary** is a table-level, instance-specific set of dictionary codes.

> **See Also:**
>
> "How a Join Group Uses a Common Dictionary"

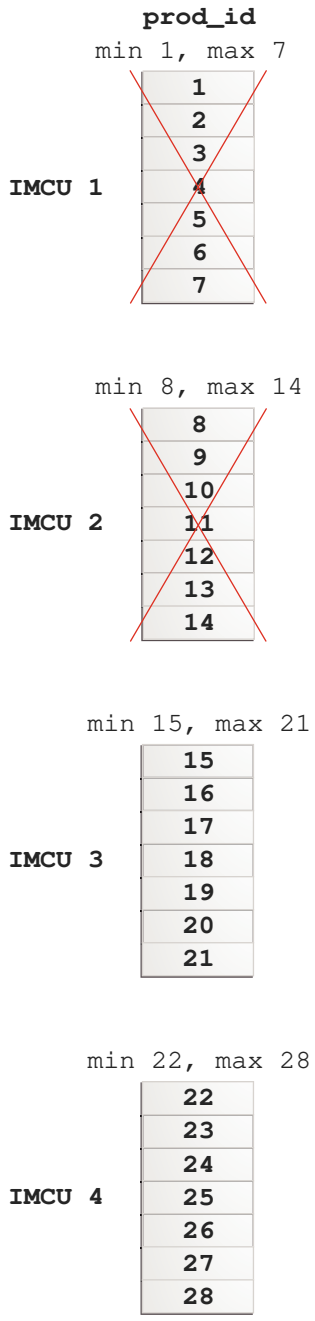
2.2.1.3 In-Memory Storage Indexes

Every IMCU header automatically creates and manages **In-Memory Storage Indexes** (IM storage indexes) for its CUs. An IM storage index stores the minimum and maximum for all columns within the IMCU.

For example, `sales` is populated in the IM column store. Every IMCU for this table has all columns. The `sales.prod_id` column is stored in a separate CU within every IMCU. The IMCU header has the minimum and maximum values of each `prod_id` CU (and every other CU).

To eliminate unnecessary scans, the database can perform IMCU pruning based on SQL filter predicates. The database scans only the IMCUs that satisfy the query predicate, as shown in the `WHERE prod_id > 14 AND prod_id < 29` example in the following graphic.

Figure 2-9 Storage Index for Columnar Data



2.2.2 Snapshot Metadata Units (SMUs)

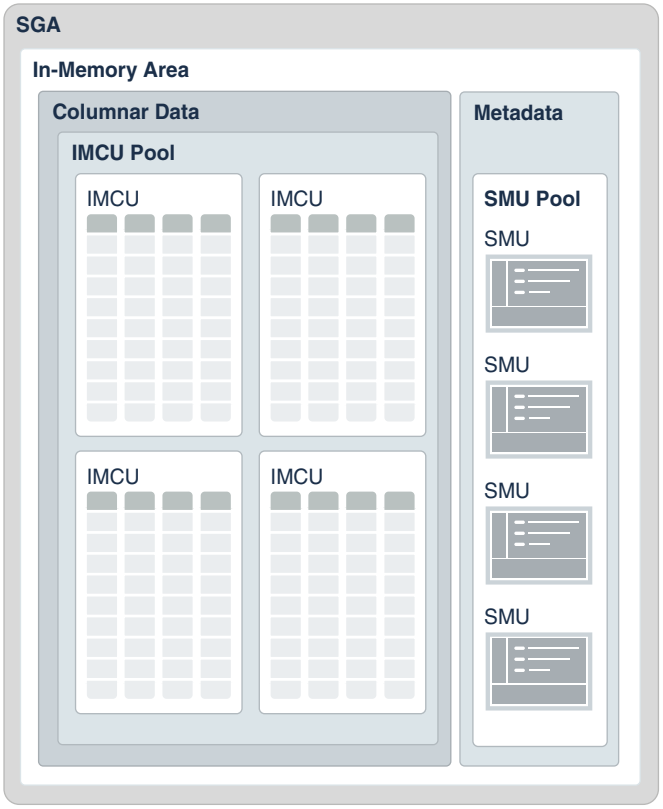
A Snapshot Metadata Unit (SMU) contains metadata and transactional information for an associated IMCU.

2.2.2.1 IMCUs and SMUs

The columnar pool of the In-Memory Area stores the actual data: IMCUs and IMEUs. The metadata pool in the In-Memory Area stores the SMUs.

Figure 2-10 IMCUs and SMUs

This figure shows IMCUs in the data pool, and SMUs in the metadata pool.



Every IMCU maps to a separate SMU. Thus, if the columnar data pool contains 100 IMCUs, then the metadata pool contains 100 SMUs. The SMUs store several types of metadata for their associated IMCUs, including the following:

- Object numbers
- Column numbers
- Mapping information for rows

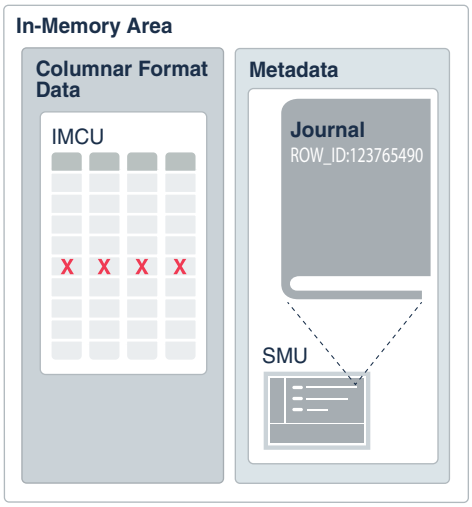
2.2.2.2 Transaction Journal

Every SMU contains a **transaction journal**. The database uses the transaction journal to keep the IMCU transactionally consistent.

The database uses the buffer cache to process DML, just as when the IM column store is not enabled. For example, an `UPDATE` statement might modify a row in an IMCU. In this case, the database adds the rowid for the modified row to the transaction journal and marks it stale as

of the SCN of the DML statement. If a query needs to access the new version of the row, then the database obtains the row from the database buffer cache.

Figure 2-11 Transaction Journal



The database achieves read consistency by merging the contents of the column, transaction journal, and buffer cache. When the IMCU is refreshed during repopulation, queries can access the up-to-date row directly from the IMCU.

> **See Also:**
>
> "Optimizing Repopulation of the IM Column Store" for an in-depth discussion of how the IM column store maintains transactional consistency

2.2.3 In-Memory Expression Units (IMEUs)

An **In-Memory Expression Unit** (IMEU) is a storage container for materialized **In-Memory Expressions** (IM expressions) and user-defined virtual columns.

The database treats materialized expressions just like other columns in the IMCU. Conceptually, an IMEU is a logical extension of its parent IMCU. Just as an IMCU can contain multiple columns, an IMEU can contain multiple virtual columns.

Every IMEU maps to exactly one IMCU, mapping to the same row set. The IMEU contains expression results for the data contained in its associated IMCU. When the IMCU is populated, the associated IMEU is also populated.

A typical IM expression involves one or more columns, possibly with constants, and has a one-to-one mapping with the rows in the table. For example, an IMCU for an `employees` table contains rows 1–1000 for the column `weekly_salary`. For the rows stored in this IMCU, the IMEU calculates the automatically detected IM expression `weekly_salary*52`, and the user-defined virtual column `quarterly_salary` defined as `weekly_salary*12`. The 3rd row down in the IMCU maps to the 3rd row down in the IMEU.

The IMEU is a logical extension of the IMCUs of a particular segment. By default, the IMEU inherits the `INMEMORY` clause properties, including Oracle Real Application Clusters (Oracle RAC) properties such as `DISTRIBUTE` and `DUPLICATE`, from the base segment. You can selectively enable or disable virtual columns for storage in IMEUs. You can also specify compression levels for different columns.

Related Topics

- About Enabling INMEMORY Columns
  For internal tables, both In-Memory virtual columns (IM virtual columns) and nonvirtual columns are eligible for IM population. For external tables, only nonvirtual columns are eligible.
- In-Memory Views
  This topic describes data dictionary and dynamic performance views related to the In-Memory Column Store (IM column store).

2.3 Expression Statistics Store (ESS)

The **Expression Statistics Store (ESS)** is a repository maintained by the optimizer to store statistics about expression evaluation. The ESS resides in the SGA and persists on disk.

When an IM column store is enabled, the database leverages the ESS for its In-Memory Expressions (IM expressions) feature. However, the ESS is independent of the IM column store. The ESS is a permanent component of the database and cannot be disabled.

The database uses the ESS to determine whether an expression is “hot” (frequently accessed), and thus a candidate for an IM expression. During a hard parse of a query, the ESS looks for active expressions in the `SELECT` list, `WHERE` clause, `GROUP BY` clause, and so on.

For each segment, the ESS maintains expression statistics such as the following:

- Frequency of execution
- Cost of evaluation
- Timestamp evaluation

The optimizer assigns each expression a weighted score based on cost and the number of times it was evaluated. The values are approximate rather than exact. More active expressions have higher scores. The ESS maintains an internal list of the most frequently accessed expressions.

Control the behavior of IM expressions using the `DBMS_INMEMORY_ADMIN` package. For example, the `IME_CAPTURE_EXPRESSIONS` procedure prompts the database to identify and gradually populate the hottest expressions in the database. The `IME_POPULATE_EXPRESSIONS` procedure forces the database to populate the expressions immediately.

ESS information is stored in the data dictionary and exposed in the `DBA_EXPRESSION_STATISTICS` view. This view shows the metadata that the optimizer has collected in the ESS. IM expressions are exposed as system-generated virtual columns, prefixed by the string `SYS_IME`, in the `DBA_IM_EXPRESSIONS` view.

See Also:

- "About IM Expressions"
- *Oracle Database SQL Tuning Guide* to learn more about ESS
- *Oracle Database Reference* to learn more about the `DBA_EXPRESSION_STATISTICS` view
- *Oracle Database PL/SQL Packages and Types Reference* to learn more about the `DBMS_INMEMORY_ADMIN` package

2.4 In-Memory Process Architecture

In response to queries and DML, server processes scan columnar data and update SMU metadata. Background processes populate row data from disk into the IM column store.

2.4.1 In-Memory Coordinator Process (IMCO)

The In-Memory Coordinator Process (IMCO) manages many tasks for the IM column store. Its primary task is to initiate background population and repopulation of columnar data.

Population is a streaming mechanism, converting row data into columnar format, and then compressing it. IMCO automatically initiates population of `INMEMORY` objects with any priority other than `NONE`. When objects with priority `NONE` are accessed, IMCO populates them using Space Management Worker Process (W*nnn*) processes.

The IMCO background process also initiates threshold-based repopulation of IM column store objects when they meet a staleness threshold. IMCO may instigate trickle repopulation for any IMCU in the IM column store that has stale entries but does not meet the staleness threshold.

Trickle repopulation occurs automatically in the background. The steps are as follows:

1. IMCO wakes up.
2. IMCO determines whether population tasks need to be performed, including whether any stale entries exist in an IMCU.
3. If IMCO finds stale entries, then it triggers a Space Management Worker Process to repopulate these entries in the IMCU.
4. IMCO sleeps for two minutes, and then returns to Step 1.

See Also:

- "Optimizing Repopulation of the IM Column Store"
- *Oracle Database Reference* to learn more about background processes

2.4.2 Space Management Worker Processes (Wnnn)

Space Management Worker Processes (W*nnn*) populate or repopulate data on behalf of IMCO.

During population, W*nnn* processes are responsible for creating IMCUs, SMUs, and IMEUs. When creating IMEUs, the worker processes perform the following tasks:

- Identify virtual columns for population
- Create virtual column values
- Compute values for each row, transform the data into columnar format, and compress it
- Register the objects with the space layer
- Associate the IMEUs with their corresponding IMCUs

> **Note:**
>
> During IMEU creation, parent IMCUs remain available for queries.

During repopulation, the W*nnn* processes create new versions of the IMCUs based on the existing IMCUs and transactions journals, while temporarily retaining the old versions. This mechanism is called double buffering.

The database can quickly move IM expressions in and out of the IM column store. For example, if an IMCU was created without an IMEU, then the database can add an IMEU later without forcing the IMCU to undergo the full repopulation mechanism.

The `INMEMORY_MAX_POPULATE_SERVERS` initialization parameter controls the maximum number of worker processes that can be started for population. The `INMEMORY_TRICKLE_REPOPULATE_PERCENT` initialization parameter controls the maximum percentage of time that worker processes can perform trickle repopulation.

> **See Also:**
>
> - "About Manually Enabling Objects for In-Memory Population"
> - "About Repopulation of the IM Column Store"
> - "In-Memory Initialization Parameters"
> - *Oracle Database Reference* to learn more about background processes

2.4.3 In-Memory Dynamic Scans

In-Memory Dynamic Scans (IM dynamic scans) use lightweight threads to parallelize In-Memory table scans.

2.4.3.1 Purpose of IM Dynamic Scans

When additional CPU is available, IM dynamic scans accelerate In-Memory table scans that are CPU bound.

IM dynamic scans automatically use idle CPU resources to scan IMCUs in parallel and maximize CPU usage. When CPU resources are available, applications can get even faster analytic query results automatically. Because the scans are dynamic, they enable the use of excess CPU bandwidth without affecting existing workload.

IM dynamic scans are more flexible than traditional Oracle parallel execution, although the two are not mutually exclusive. Dynamic scans use multiple lightweight threads of execution within a process. Typically, the performance overhead for dynamic scans is low.

> **See Also:**
>
> *Oracle Database Administrator's Guide* to learn more about Resource Manager

2.4.3.2 How IM Dynamic Scans Work

IM Dynamic Scans attain optimal performance by reading IMCUs in parallel.

2.4.3.2.1 About Lightweight Threads

A **lightweight thread** is an execution entity that helps to parallelize full table scans. It is "lightweight" because it does not incur the higher memory overhead of Oracle processes.

> **Note:**
>
> A lightweight thread used by IM dynamic scans is not the same as a regular thread in the multithreaded Oracle Database model.

Lightweight threads share the resources of the parent foreground or PQ process, called the table scan process, that coordinates the scan of a set of IMCUs. Threads maintain their own independent flow of execution. The database can parallelize scans by prioritizing threads and executing them asynchronously.

For eligible queries, the process allocates a pool of threads. Resource Manager automatically determines the number of threads in the pool based on the CPU count in the database host and the current load on the system. The pool of threads remains available to the session for subsequent queries unless the idle time reaches an internal threshold, at which point the database terminates the threads.

Communication between threads occurs exclusively within a process. For this reason, contention does not occur at the database instance level.

> **See Also:**
>
> *Oracle Database Concepts* to learn about the multithreaded Oracle Database model

2.4.3.2.2 When the Database Considers IM Dynamic Scans

Lightweight threads are enabled when a CPU resource plan is enabled (for example, `RESOURCE_MANAGER_PLAN=DEFAULT_PLAN`) and CPU utilization of the database is low.

If lightweight threads are enabled, then the database considers an IM dynamic scan when an application queries an object that is currently populated in the IM column store. Typically, a serial or parallel query is a candidate for IM dynamic scans when it has the following characteristics:

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

Oracle Database Resource Manager (the Resource Manager), which is automatically enabled when `INMEMORY_SIZE` is greater than `0`, is required for IM dynamic scans. The Resource Manager decides when and how to use the lightweight threads. Lightweight threads are the lowest priority operation in the database because they are capitalizing on unused resources.

> **Note:**
>
> `CPU_COUNT` must be greater than or equal to `24` in order to perform IM dynamic scans.

2.4.3.2.3 How IM Dynamic Scans Work

Resource Manager allocates lightweight threads to parallelize the scan of IMCUs.

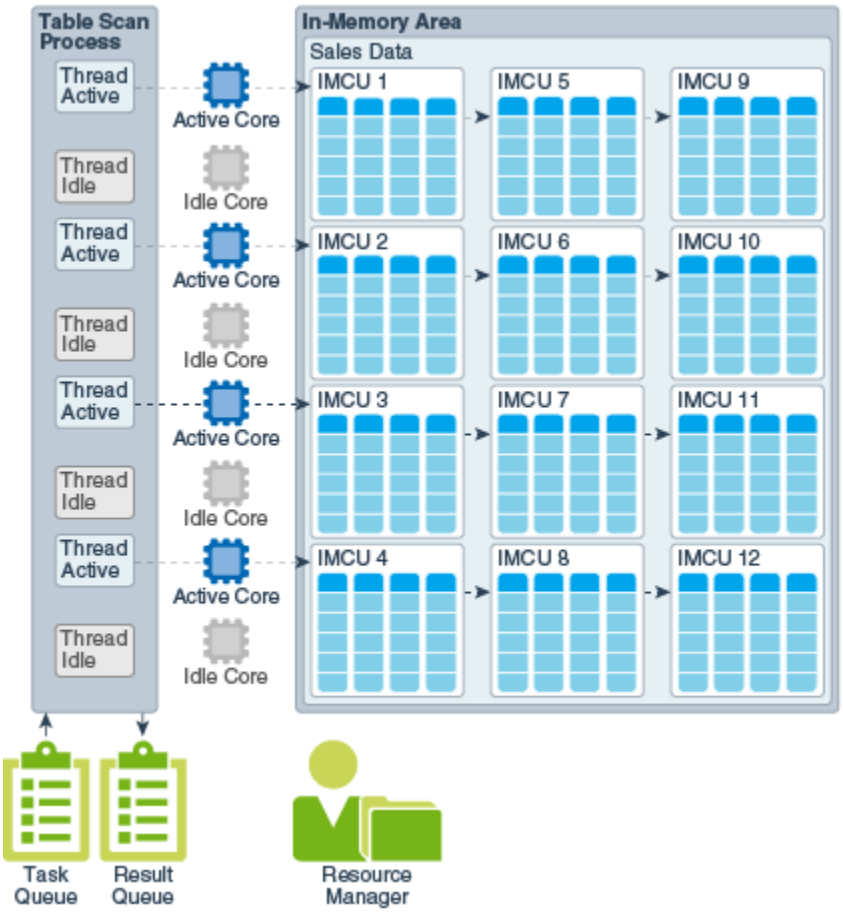
When the database determines that a query can benefit from an IM dynamic scan, it typically proceeds as follows:

1. A table scan process spawns a pool of lightweight threads.
2. The table scan process creates a separate task for every IMCU that must be scanned, and then adds each task to a task queue.
3. Resource Manager determines how many threads can participate in the table scan.
4. Active threads pick up tasks from the task queue, with the table scan process consuming results from completed tasks.

Depending on the database load, Resource Manager continuously adjusts the number of active lightweight threads while the query is running. If CPU resources are not available, then the table scan process performs the scan *without* using lightweight threads.

The following graphic illustrates an IM dynamic scan of 12 IMCUs in the `sales` table.

Figure 2-12 IM Dynamic Scan



In the preceding graphic, the database host has 8 CPU cores. Based on an internal algorithm, Resource Manager assigns 4 threads to assist the table scan process. In this scenario, 4 CPU cores remain idle for other concurrent database operations to use.

2.4.3.3 Interface for IM Dynamic Scans

IM dynamic scans are transparent, which means that they require no application changes and are automatically controlled by the Resource Manager.

IM dynamic scans require the Resource Manager, which is automatically enabled when `INMEMORY_SIZE` is greater than `0`. No specific resource plan is required.

Several new session statistics track the usage of IM dynamic scans. Each thread writes trace data to a separate trace file.

Execution plans are unchanged. The following figure shows a sample execution plan.

```
SQL> SELECT MAX(l_quantity) largest_order FROM lineitem;

LARGEST_ORDER
-------------
           50
```

```
Elapsed: 00:00:03.41

Execution Plan
----------------------------------------------------------
Plan hash value: 1885658499
-------------------------------------------------------------------------------------------------
|Id| Operation                    | Name     |Rows| Bytes |Cost(%CPU)| Time   |Pstart| Pstop|
-------------------------------------------------------------------------------------------------
|0| SELECT STATEMENT             |          |    1 |    3| 116K  (4)| 00:00:05 |      |      |
|1|  SORT AGGREGATE              |          |    1 |    3|          |          |      |      |
|2|   PARTITION RANGE ALL        |          | 600M |1716M| 116K  (4)| 00:00:05 |  1 |  84 |
|3|    TABLE ACCESS INMEMORY FULL| LINEITEM | 600M |1716M| 116K  (4)| 00:00:05 |  1 |  84 |
-------------------------------------------------------------------------------------------------

NAME                                                  VALUE
----------------------------------------------------------
IM scan CUs memcompress for query low                  1147
IM scan bytes in-memory                          5.1790E+10
IM scan bytes uncompressed                       7.6722E+10
IM scan CUs columns accessed                           1147
IM scan rows                                      600037902
IM scan rows projected                                   29
IM scan (dynamic) rows                            600037902
IM scan (dynamic) multi-threaded scans                   1
IM scan (dynamic) tasks processed by thread           1146
```

Consider the characteristics of the plan:

1. The execution plan is unchanged.

   Note that the plan does not mention IM dynamic scans in Step 3. However, clicking the binocular icon in a SQL Monitor report would show “Dynamic Scan Tasks on Thread.”

2. `IM scan (dynamic) multi-threaded scans` is nonzero, which means that the database used an IM dynamic scan.

3. `IM scan CUs memcompress for query low` indicates that 1147 IMCUs exist in the `lineitem` table.

4. `IM scan (dynamic) tasks processed by thread` indicates how many IMCUs were processed in parallel.

   The number is 1146, which is less than the total number of 1147 shown in `IM scan CUs memcompress for query low`. The database analyzed the first IMCU without parallelization to determine whether parallelization was worthwhile. Because the answer was yes, the database proceeded to scan the remaining 1146 IMCUs in parallel.

5. `IM scan (dynamic) rows` and `IM scan rows are equal`, which means that the threads retrieved all rows for the query.

See Also:

- *Oracle Database Administrator's Guide* to learn more about the Resource Manager
- *Oracle Database Reference* for descriptions of In-Memory statistics

2.5 CPU Architecture: SIMD Vector Processing

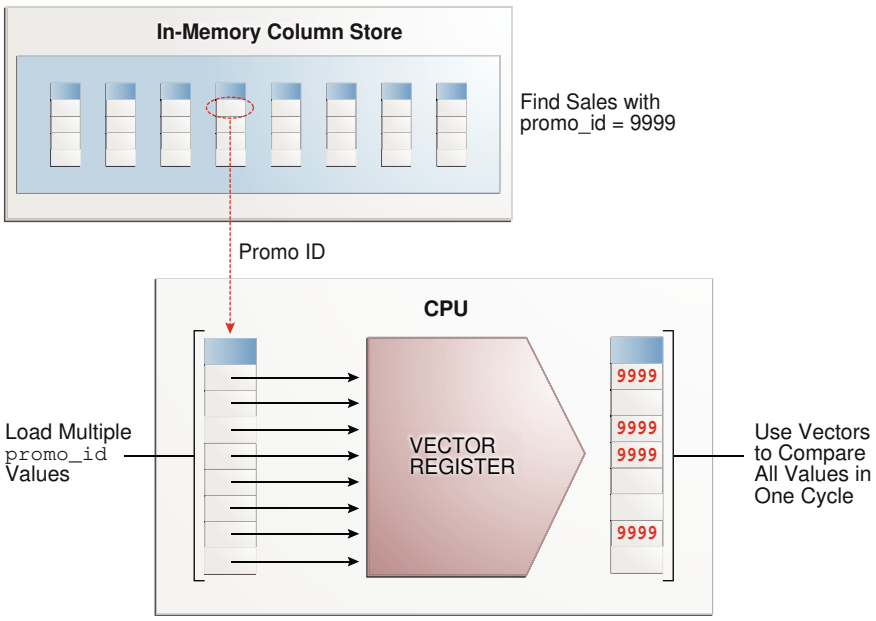
For data that is populated in the IM column store, the database uses SIMD (single instruction, multiple data) processing.

A SIMD unit is a processor that enables a single instruction to process data as a unit, called a **vector**, rather than processing data in separate instructions. For example, instead of using a loop to execute four addition operations, SIMD could load the four sets of numbers into vectors and perform one addition operation. SIMD processing is sometimes called **vectorization**.

The IM column store maximizes the number of column entries that the CPU can load into the vector registers and evaluate. Instead of evaluating each entry in the column one at a time, the database evaluates a set of column values in a single CPU instruction. SIMD vector processing enables the database to scan billions of rows per second.

For example, an application issues a query to find the total number of orders in the `sales` table that use the `promo_id` value of `9999`. The `sales` table resides in the IM column store. The query begins by scanning only the `sales.promo_id` column, as shown in the following diagram:

Figure 2-13 SIMD Vector Processing



The CPU evaluates the data as follows:

1. Loads the first 8 values (the number varies depending on data type and compression mode) from the `promo_id` column into the SIMD register, and then compares them with the value `9999` in a single instruction
2. Discards the entries.
3. Loads another 8 values into the SIMD register, and then continues in this way until it has evaluated all entries.

2.5.1 SIMD and Oracle LOBs

Oracle Database 18c provides SIMD vector support for queries involving SQL operators on specific LOB columns.

The nature of the support depends on the type of LOB:

- Inline LOBs

  The IM column store provides contiguous storage for inline LOBs, which are LOBs less than 4 KB, within the IMCUs. Columnar storage enables faster query processing by removing the overhead of assembling LOB data from the database buffer cache.

- Out-of-line LOBs

  In this case, the IM column store only stores the LOB locator, which is 40 byes. Out-of-line columns do not benefit from columnar optimization.

There is one exception to the preceding rule. An IMEU can allocate up to 32 KB of contiguous storage for JSON columns defined as a LOB data type. The IMEU stores these columns in the OSON format, which can provide faster query performance using SIMD processing.

> **See Also:**
>
> *Oracle Database SecureFiles and Large Objects Developer's Guide* to learn more about LOBs

2.5.2 SIMD and Oracle Numbers

For tables compressed with `QUERY LOW`, `NUMBER` columns are encoded using an optimized format that enables native calculations in hardware.

SIMD vector processing enables simple aggregations, `GROUP BY` aggregations, and arithmetic operations to benefit significantly. The performance improvement depends on the amount of time the aggregation spends on arithmetic computation. Some aggregations may benefit by up to a factor of 9.

> **See Also:**
>
> - "Optimizing In-Memory Arithmetic"
> - *Oracle Database SQL Language Reference*

2.5.3 SIMD and Exadata Smart Flash Cache

Besides storing data in Hybrid Columnar Compression format, Exadata Smart Flash Cache can store data in pure columnar format.

Exadata Smart Scan supports SIMD predicates. The advantage is that In-Memory performance extends from DRAM storage to secondary storage.

By default, Exadata Smart Flash Cache compresses data using the level `MEMCOMPRESS FOR CAPACITY LOW`. To change the compression level or disable the columnar format altogether, use the `ALTER TABLE ... NO CELLMEMORY` statement.

> **See Also:**
>
> - "In-Memory Support for Exadata Flash Cache"
> - *Oracle Exadata Database Machine System Overview*

Part II

Configuring and Populating the IM Column Store

You can enable and size the In-Memory Column Store (IM column store). You can also configure In-Memory settings for objects, and populate these objects in the IM column store.

3
Enabling and Sizing the IM Column Store

To enable or disable the IM column store, specify a value for the `INMEMORY_SIZE` initialization parameter.

3.1 Overview of Enabling the IM Column Store

Enable the IM column store size by setting the `INMEMORY_SIZE` initialization parameter.

By default, `INMEMORY_SIZE` is set to `0`, which means the IM column store is disabled. To enable the IM column store, set the initialization parameter `INMEMORY_SIZE` to a value greater than zero before restarting the database instance. You can dynamically increase the `INMEMORY_SIZE` size setting by using an `ALTER SYSTEM` statement.

For the Database In-Memory Base Level only, you can allocate up to 16 GB on any CDB or any instance of an Oracle RAC database.

By default, you must specify candidates for population in the IM column store using the `INMEMORY` clause of a `CREATE` or `ALTER` statement for a table, tablespace, or materialized view.

See Also:

- "In-Memory Initialization Parameters"
- *Oracle Database Reference* to learn more about the `INMEMORY_SIZE` initialization parameter
- *Oracle Database SQL Language Reference* for more information about the `INMEMORY` clause
- *Oracle Database Licensing Information User Manual* for details on which features are supported for different editions and services

3.2 Estimating the Required Size of the IM Column Store

Estimate the size of the IM column store based on your requirements, and then resize the IM column store to meet those requirements. Applying compression can reduce memory size.

The amount of memory required by the IM column store depends on the database objects stored in it and the compression method applied on each object. When choosing a compression method for the `INMEMORY` objects, balance the performance benefits against the amount of available memory:

- To make the greatest reduction in memory size, choose the `FOR CAPACITY HIGH` or `FOR CAPACITY LOW` compression methods. However, these options require additional CPU during query execution to decompress the data.

- To get the best query performance, choose the `FOR QUERY HIGH` or `FOR QUERY LOW` compression methods. However, these options consume more memory.

When sizing the IM column store, consider the following guidelines:

1. For every object to be populated into the IM column store, estimate the amount of memory it consumes.

   Oracle Compression Advisor estimates the compression ratio that you can realize using the `MEMCOMPRESS` clause. The advisor uses the `DBMS_COMPRESSION` interface.

2. Add the individual amounts to together.

> **Note:**
>
> After population, `V$IM_SEGMENTS` shows the actual size of the objects on disk and their size in the IM column store. You can use this information to calculate the compression ratio for the populated objects. However, if the objects were compressed on disk, then this query does not show the correct compression ratio.

3. If you configured In-Memory Optimized Arithmetic, and if In-Memory tables use `FOR QUERY LOW` compression, then add roughly 15% to account for the dual storage of `NUMBER` columns.
4. Add space to account for the growth of database objects, and to store updated versions of rows after DML operations.

   The minimum amount for dynamic resizing is 128 MB.

> **See Also:**
>
> - "IM Column Store Compression Methods"
> - "Enabling the IM Column Store for a Database"
> - "Increasing the Size of the IM Column Store Dynamically"
> - "About In-Memory Optimized Arithmetic"
> - *Oracle Database Administrator's Guide* to learn how to estimate compression ratio using Compression Advisor
> - *Oracle Database Reference* to learn about `V$IM_SEGMENTS`

3.3 Enabling the IM Column Store for a Database

Before tables or materialized views can be populated into the IM column store, you must enable the IM column store for the database.

In this context, "database" is a non-CDB, CDB, or PDB. In a CDB, the `INMEMORY_SIZE` setting in the CDB root determines the overall size of the IM column store. By default, all PDBs have access to the IM column store.

Within an individual PDB, you can limit access to the shared In-Memory Area by setting `INMEMORY_SIZE` to a different value. For example, in a CDB with 100 PDBs, you could set `INMEMORY_SIZE` to `16G` at the CDB level, and then set `INMEMORY_SIZE` to `10G` in one PDB, to `6G` in a second PDB, and to `0` in the remaining PDBs.

Prerequisites

This task assumes that the following:

- The database is open.
- The `COMPATIBLE` initialization parameter is set to `12.1.0` or higher.
- The `INMEMORY_SIZE` initialization parameter is set to `0` (default).
- You want to use the Database In-Memory Base Level.

To enable the IM column store:

1. In SQL*Plus or SQL Developer, log in to the database as a user with administrative privileges.
2. Set the `INMEMORY_SIZE` initialization parameter to a nonzero value.

   When you set this initialization parameter in a server parameter file (SPFILE) using the `ALTER SYSTEM` statement, you must specify `SCOPE=SPFILE`.

   For example, the following statement sets the In-Memory Area size to 16 GB:

   ```
   ALTER SYSTEM SET INMEMORY_SIZE = 16G SCOPE=SPFILE;
   ```

3. For the Database In-Memory Base Level, set the `INMEMORY_FORCE` initialization parameter to `BASE_LEVEL`.

   For example, the following statement specifies the Base Level:

   ```
   ALTER SYSTEM SET INMEMORY_FORCE=BASE_LEVEL SCOPE=SPFILE;
   ```

   You cannot set `INMEMORY_FORCE=BASE_LEVEL` at the PDB level. Also, you cannot set this parameter dynamically.
4. Shut down the database, and then reopen it.

   You must reopen the database to initialize the IM column store in the SGA.
5. Optionally, check the amount of memory currently allocated for the IM column store:

   ```
   SHOW PARAMETER INMEMORY_SIZE
   ```

> **Note:**
>
> After the IM column store is enabled, you can increase its size dynamically without reopening the database.

Example 3-1 Enabling the IM Column Store

Assume that the `INMEMORY_SIZE` initialization parameter is set to `0`. The following SQL*Plus example sets `INMEMORY_SIZE` to 16 GB, shuts down the database instance, and then reopens the database so that the change can take effect:

```
SQL> SHOW PARAMETER INMEMORY_SIZE

NAME                                 TYPE        VALUE
------------------------------------ ----------- -----
inmemory_size                        big integer 0

SQL> ALTER SYSTEM SET INMEMORY_SIZE=16G SCOPE=SPFILE;

System altered.

SQL> SHUTDOWN IMMEDIATE
Database closed.
Database dismounted.
ORACLE instance shut down.

SQL> STARTUP
ORACLE instance started.

Total System Global Area   11525947392 bytes
Fixed Size                     8213456 bytes
Variable Size                754977840 bytes
Database Buffers              16777216 bytes
Redo Buffers                   8560640 bytes
In-Memory Area             10737418240 bytes
Database mounted.
Database opened.

SQL> SHOW PARAMETER INMEMORY_SIZE

NAME                                 TYPE        VALUE
------------------------------------ ----------- -----
inmemory_size                        big integer 16G
```

See Also:

- "Multiple IM Column Stores"
- *Oracle Database Upgrade Guide* for information about setting the database compatibility level
- *Oracle Database Reference* for more information about the `INMEMORY_SIZE` initialization parameter

3.4 Increasing the Size of the IM Column Store Dynamically

You can increase the size of IM column store at database run time by using SQL to reset `INMEMORY_SIZE`.

> **Note:**
>
> If ASMM is enabled, it automatically decreases or increases the IM column store size dynamically as needed. If you want to set `INMEMORY_SIZE` to a value smaller than its current setting, check the prerequisites listed here, then issue the `ALTER SYSTEM` statement with `SCOPE=SPFILE` as described below. Then restart the database to effect the change.

Prerequisites

To increase the size of the IM column store dynamically using `ALTER SYSTEM`, the instance must meet the following prerequisites:

- The IM column store must be enabled.
- The compatibility level must be `12.2.0` or higher.
- The database instances must be started with an SPFILE.
- The new size of the IM column store must be at least 1 granule.

Steps

1. In SQL*Plus or SQL Developer, log in to the database with administrative privileges.
2. Optionally, check the amount of memory currently allocated for the IM column store:

   ```
   SHOW PARAMETER INMEMORY_SIZE
   ```

3. Set the `INMEMORY_SIZE` initialization parameter to a value greater than the current size of the IM column store with an `ALTER SYSTEM` statement that specifies `SCOPE=BOTH` or `SCOPE=MEMORY`.

   When you set this parameter dynamically, you must set it to a value that is higher than its current value, and there must be enough memory available in the SGA to increase the size of the IM column store dynamically to the new value.

   For example, the following statement sets `INMEMORY_SIZE` to `500M` dynamically:

   ```
   ALTER SYSTEM SET INMEMORY_SIZE = 500M SCOPE=BOTH;
   ```

> **See Also:**
>
> - "Enabling the IM Column Store for a Database"
> - *Oracle Database Reference* for more information about the `INMEMORY_SIZE` initialization parameter

3.5 Disabling the IM Column Store

You can disable the IM column store by setting the `INMEMORY_SIZE` initialization parameter to zero, and then reopening the database.

Assumptions

This task assumes that the IM column store is enabled in an open database.

To disable the IM column store:

1. Set the `INMEMORY_SIZE` initialization parameter to `0` in the server parameter file (SPFILE).
2. Shut down the database.
3. Start a database instance, and then open the database.

> **See Also:**
>
> *Oracle Database Reference* for information about the `INMEMORY_SIZE` initialization parameter

4
Enabling Objects for In-Memory Population

This chapter explains how to enable and disable objects for population in the IM column store, including setting compression and priority options.

This chapter contains the following topics:

4.1 About Manually Enabling Objects for In-Memory Population

Only objects with the `INMEMORY` clause are eligible for population into the IM column store. To apply this clause manually, you must use DDL statements such as `CREATE TABLE` or `ALTER TABLE`.

4.1.1 Purpose of Enabling Objects for In-Memory Population

By default, objects implicitly have the `NO INMEMORY` attribute, which means they are ineligible for population into the IM column store.

Only when you use DDL to specify objects as `INMEMORY` are they eligible for population. Enabling an object as `INMEMORY` is specifying that an object can potentially reside in the IM column store. In-Memory population is a separate step that occurs when the database reads existing row-format data from disk, transforms it into columnar format, and then stores it in the IM column store.

Population, which transforms *existing* data on disk into columnar format, is different from repopulation, which transforms *new* data into columnar format. Because IMCUs are read-only structures, Oracle Database does not populate them when rows change. Rather, the database records the row changes in a transaction journal, and then creates new IMCUs as part of repopulation.

Related Topics

- Optimizing Repopulation of the IM Column Store
  The IM column store periodically refreshes objects that have been modified. You can control this behavior using initialization parameters and the `DBMS_INMEMORY` package.
- In-Memory Compression Units (IMCUs)
  An **In-Memory Compression Unit (IMCU)** is a compressed, read-only storage unit that contains data for one or more columns.

4.1.2 How In-Memory Population Works

You can specify that the database populates objects in the IM column store either at database instance startup or when `INMEMORY` objects are accessed.

The population algorithm also varies depending on whether you use single-instance or Oracle RAC.

This section contains the following topics:

4.1.2.1 Prioritization of In-Memory Population

DDL statements include an `INMEMORY PRIORITY` subclause that provides more control over the population queue.

> **Note:**
>
> The `INMEMORY PRIORITY` subclause controls the priority of population, but not the speed of population.

The priority level setting applies to an entire table, partition, or subpartition, not to different column subsets. Setting the `INMEMORY` attribute on an object means that this object is a *candidate* for population in the IM column store. It does not mean that the database immediately populates the object.

> **Note:**
>
> If a segment on disk is 64 KB or less, then it is not populated in the IM column store. Therefore, some small database objects that were enabled for the IM column store might not be populated.

Oracle Database manages prioritization as follows:

- On-demand population

  By default, the `INMEMORY PRIORITY` parameter is set to `NONE`. In this case, the database *only* populates the object when it is accessed through a full table scan. If the object is never accessed, or if it is accessed only through an index scan or fetch by rowid, then population never occurs.

- Priority-based population

  When `PRIORITY` is set to a value *other than* `NONE`, Oracle database automatically populates the objects using an internally managed priority queue. In this case, a full scan is not a necessary condition for population. The database does the following:

  - Populates columnar data in the IM column store automatically after the database instance restarts
  - Queues population of `INMEMORY` objects based on the specified priority level

    For example, a table altered with `INMEMORY PRIORITY CRITICAL` takes precedence over a table altered with `INMEMORY PRIORITY HIGH`, which in turn takes precedence over a table altered with `INMEMORY PRIORITY LOW`. If the IM column store has insufficient space, then Oracle Database does not populate additional objects until space is available.
  - Waits to return from `ALTER TABLE` or `ALTER MATERIALIZED VIEW` statements until the changes to the object are recorded in the IM column store

After a segment is populated in the IM column store, the database only evicts it when the segment is dropped or moved, or the segment is updated with the `NO INMEMORY` attribute. You can evict a segment manually or by means of an ADO policy.

Example 4-1 Population of an Object in the IM Column Store

Before completing this example, the IM column store must be enabled for the database.

1. Log in to the database as an administrator, and then query the `customers` table as follows:

```
SELECT cust_id, cust_last_name, cust_first_name
FROM   sh.customers
WHERE  cust_city = 'Hyderabad'
AND    cust_income_level LIKE 'C%'
AND    cust_year_of_birth > 1960;
```

2. Display the execution plan for the query:

```
SQL> SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR(FORMAT=>'+ALLSTATS'));
SQL_ID  frgk9dbaftmm9, child number 0
-------------------------------------
SELECT cust_id, cust_last_name, cust_first_name FROM   sh.customers
WHERE  cust_city = 'Hyderabad' AND    cust_income_level LIKE 'C%' AND
 cust_year_of_birth > 1960

Plan hash value: 2008213504

--------------------------------------------------------------------------------
-----
| Id| Operation         | Name      |Starts|E-Rows|A-Rows|   A-Time   |
Buffers|
--------------------------------------------------------------------------------
-----
|  0| SELECT STATEMENT  |           |     1|      |    6 |00:00:00.01 |
1523|
|* 1|  TABLE ACCESS FULL| CUSTOMERS |     1|    6 |    6 |00:00:00.01 |
1523|
--------------------------------------------------------------------------------
-----

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter(("CUST_CITY"='Hyderabad' AND "CUST_YEAR_OF_BIRTH">1960 AND
              "CUST_INCOME_LEVEL" LIKE 'C%'))
```

3. Enable the `sh.customers` table for population in the IM column store:

```
ALTER TABLE sh.customers INMEMORY;
```

The preceding statement uses the default priority of `NONE`. A full scan is required to populate objects with no priority.

4. To determine whether data from the `sh.customers` table has been populated in the IM column store, execute the following query (sample output included):

```
SELECT SEGMENT_NAME, POPULATE_STATUS
FROM   V$IM_SEGMENTS
WHERE  SEGMENT_NAME = 'CUSTOMERS';

no rows selected
```

   In this case, no segments are populated in the IM column store because the `sh.customers` table has not yet been scanned.

5. Query `sh.customers` using the same statement as in Step 1:

```
SELECT cust_id, cust_last_name, cust_first_name
FROM   sh.customers
WHERE  cust_city = 'Hyderabad'
AND    cust_income_level LIKE 'C%'
AND    cust_year_of_birth > 1960;
```

6. Querying the cursor shows that the database performed a full scan and accessed the IM column store:

```
SQL> SELECT * FROM
TABLE(DBMS_XPLAN.DISPLAY_CURSOR(FORMAT=>'+ALLSTATS'));
SQL_ID  frgk9dbaftmm9, child number 0
-------------------------------------
SELECT cust_id, cust_last_name, cust_first_name FROM   sh.customers
WHERE  cust_city = 'Hyderabad' AND    cust_income_level LIKE 'C%'
AND
 cust_year_of_birth > 1960

Plan hash value: 2008213504

-------------------------------------------------------------------------
-------------
| Id| Operation           | Name            |Starts|E-Rows|A-Rows|A-
Time|Buffers|
-------------------------------------------------------------------------
-------------
|  0| SELECT STATEMENT           |           |    1|      | 6 |
00:00:00.02| 1523 |
|* 1|  TABLE ACCESS INMEMORY FULL| CUSTOMERS |    1|    6| 6 |
00:00:00.02| 1523 |
-------------------------------------------------------------------------
-------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - inmemory(("CUST_CITY"='Hyderabad' AND
"CUST_YEAR_OF_BIRTH">1960 AND
              "CUST_INCOME_LEVEL" LIKE 'C%'))
       filter(("CUST_CITY"='Hyderabad' AND
```

```
"CUST_YEAR_OF_BIRTH">1960 AND
              "CUST_INCOME_LEVEL" LIKE 'C%'))
```

7. Query `V$IM_SEGMENTS` again (sample output included):

```
COL SEGMENT_NAME FORMAT a20

SELECT SEGMENT_NAME, POPULATE_STATUS
FROM   V$IM_SEGMENTS
WHERE  SEGMENT_NAME = 'CUSTOMERS';

SEGMENT_NAME         POPULATE_STATUS
-------------------- ---------------
CUSTOMERS            COMPLETED
```

   The value `COMPLETED` in `POPULATE_STATUS` means that the table is populated in the IM column store.

8. The `DBA_FEATURE_USAGE_STATISTICS` view confirms that the database used the IM column store to retrieve the results:

```
COL NAME FORMAT a25
SELECT ul.NAME, ul.DETECTED_USAGES
FROM   DBA_FEATURE_USAGE_STATISTICS ul
WHERE  ul.VERSION= (SELECT MAX(u2.VERSION)
                    FROM   DBA_FEATURE_USAGE_STATISTICS u2
                    WHERE  u2.NAME = ul.NAME
                    AND    ul.NAME LIKE '%Column Store%');

NAME                      DETECTED_USAGES
------------------------- ---------------
In-Memory Column Store    1
```

> **See Also:**
>
> "Priority Options for the Population of In-Memory Objects"
>
> *Oracle Database SQL Language Reference* to learn about the `INMEMORY PRIORITY` clause

4.1.2.2 How Background Processes Populate IMCUs

During population, the database reads data from disk in its row format, pivots the rows to create columns, and then compresses the data into In-Memory Compression Units (IMCUs).

Worker processes (W*nnn*) populate the data in the IM column store. Each worker process operates on a subset of database blocks from the object. Population is a streaming mechanism, simultaneously compressing the data and converting it into columnar format.

The `INMEMORY_MAX_POPULATE_SERVERS` initialization parameter specifies the maximum number of worker processes to use for IM column store population. By default, the setting is one half of `CPU_COUNT`. Set this parameter to an appropriate value for your environment. More worker

processes result in faster population, but they use more CPU resources. Fewer worker processes result in slower population, which reduces CPU overhead.

> **Note:**
>
> If `INMEMORY_MAX_POPULATE_SERVERS` is set to `0`, then population is disabled.

> **See Also:**
>
> *Oracle Database Reference* for more information about the `INMEMORY_MAX_POPULATE_SERVERS` initialization parameter

4.1.3 Controls for In-Memory Objects

Use the `INMEMORY` clause in DDL statements to specify which objects are eligible for population into the IM column store. You can enable tablespaces, tables (internal and external), partitions, and materialized views.

This section contains the following topics:

4.1.3.1 The INMEMORY Subclause

`INMEMORY` is a segment-level attribute, not a column-level attribute. However, you can apply the `INMEMORY` attribute to a subset of columns within a specific object.

To enable or disable an object for the IM column store, specify the `INMEMORY` clause in DDL statements for tablespaces, tables, and materialized views. The `INMEMORY` column in the `DBA_TABLES` view indicates which tables have the `INMEMORY` attribute set (`ENABLED`) or not set (`DISABLED`).

The following objects are not eligible for population in the IM column store:

- Indexes
- Index-organized tables
- Hash clusters
- Objects owned by the `SYS` user and stored in the `SYSTEM` or `SYSAUX` tablespace

4.1.3.1.1 In-Memory Tables

To make heap-organized tables eligible for population, specify `INMEMORY` on the `CREATE TABLE` or `ALTER TABLE` statements.

By default, the IM column store populates all nonvirtual columns in the table. You can specify all or some columns of an internal table. For example, you might exclude the `weight_class` and `catalog_url` columns in `oe.product_information` from eligibility.

For a partitioned table, you can populate all or some partitions in the IM column store. By default, all partitions in a partitioned table inherit the `INMEMORY` attribute. For hybrid

partitioned tables, which contain both internal and external partitions, only internal partitions inherit the `INMEMORY` attribute.

On Oracle Exadata Storage Server, the `CELLMEMORY` keyword (default) enables the flash cache to store data in the In-Memory format. You can use `ALTER TABLE` to choose `FOR QUERY` or `FOR CAPACITY` compression. Specifying `NO CELLMEMORY` disables columnar storage in the flash cache.

If you enable a table for the IM column store and it contains any of the following types of columns, then they will not be populated in the IM column store:

- Out-of-line columns (varrays, nested table columns, and out-of-line LOBs)

  > **Note:**
  >
  > For inline LOB columns, the IM column store allocates up to 4 KB of contiguous buffer storage, and up to 32 KB when the inline LOBs contain OSON (binary JSON) data. For out-of-line LOBs, the IM column store allocates up to 40 bytes for the locator, but does not store the LOB itself.

- Columns that use the `LONG` or `LONG RAW` data types
- Extended data type columns

Example 4-2 Specifying a Table as INMEMORY

Assume that you are connected to the database as user `sh`. You enable the `customers` table for population in the IM column store, using the default compression level of `FOR QUERY LOW`:

```
SQL> SELECT TABLE_NAME, INMEMORY FROM USER_TABLES WHERE TABLE_NAME =
'CUSTOMERS';

TABLE_NAME INMEMORY
---------- --------
CUSTOMERS  DISABLED

SQL> ALTER TABLE customers INMEMORY;

Table altered.

SQL> SELECT TABLE_NAME, INMEMORY, INMEMORY_COMPRESSION FROM USER_TABLES
WHERE TABLE_NAME='CUSTOMERS';

TABLE_NAME INMEMORY INMEMORY_COMPRESS
---------- -------- -----------------
CUSTOMERS  ENABLED  FOR QUERY LOW
```

See Also:

- "In-Memory Compression"
- "Enabling the IM Column Store for a Database"
- *Oracle Exadata System Software User's Guide* to learn more about `ALTER TABLE ... CELLMEMORY`
- *Oracle Database SQL Language Reference* for information about `INMEMORY` clause of the `CREATE TABLE` statement

4.1.3.1.2 In-Memory External Tables

To make external tables eligible for population, specify the `EXTERNAL ... INMEMORY` clause in `CREATE TABLE` or `ALTER TABLE`.

Purpose of In-Memory External Tables

In-Memory external tables are useful in the following cases:

- Short-term data that must be scanned repeatedly in a short time span and does not require retention in Oracle Database
- External data that must be joined to relational data for fast analytic processing
- Data that is accessed by analytic queries in both Oracle Database and external tools, and which does not need to be materialized in database storage

Restrictions for In-Memory External Tables

The IM column store manages the data for external tables in the same way as for heap-organized tables. For example, a full table scan populates both internal tables and external tables into the IM column store. The same drivers supported for external tables are supported for In-Memory external tables. However, note the following differences:

- Some `INMEMORY` subclauses for external tables are not valid, including the column clause, distribute clause, and priority clause.
- In-Memory Optimized Arithmetic does not support external tables.
- Partitioning is not supported for In-Memory external tables. For hybrid partitioned tables, which contain both internal and external partitions, only *internal* partitions—not external partitions—support the `INMEMORY` attribute.
- Join groups are not supported for In-Memory external tables.
- IM expressions are not supported for In-Memory external tables.
- In-Memory external tables do not support the `DISTRIBUTE ... FOR SERVICE` clause for Oracle Active Data Guard instances.

> **Note:**
>
> Sessions that query In-Memory external tables must have the initialization parameter `QUERY_REWRITE_INTEGRITY` set to `stale_tolerated`.
>
> It is important to keep in mind that if an external table is modified, then the results from the IM column store are undefined. Results are also undefined if a partition is altered (by dropping or adding values). This may lead to differences in results between IM and non-IM based scans. You can run `DBMS_INMEMORY.REPOPULATE` to refresh the IM store so that it is resynchronized with the table data.

> **See Also:**
>
> - "Populating an In-Memory External Table Using DBMS_INMEMORY.POPULATE: Example"
> - *Oracle Database SQL Language Reference* for information about `INMEMORY` clause of the `CREATE TABLE ... EXTERNAL` statement
> - *Oracle Database Licensing Information User Manual* for details on which features are supported for different editions and services

4.1.3.1.3 In-Memory Materialized Views

You can make materialized views eligible for population by specifying `INMEMORY` on the `CREATE MATERIALIZED VIEW` or `ALTER MATERIALIZED VIEW` statements.

For a partitioned materialized view, you can populate all or a subset of the partitions in the IM column store.

> **See Also:**
>
> *Oracle Database SQL Language Reference* for `ALTER MATERIALIZED VIEW` syntax and semantics

4.1.3.1.4 In-Memory Tablespaces

You can make permanent tablespaces eligible for population by specifying `INMEMORY` on the `CREATE TABLESPACE` or `ALTER TABLESPACE` statements.

By default, all tables and materialized views in the tablespace are enabled for the IM column store. Individual tables and materialized views in the tablespace may have different `INMEMORY` attributes. The attributes for individual database objects override the attributes for the tablespace.

> **Note:**
>
> Temporary tablespaces are not eligible for In-Memory population.

> **See Also:**
>
> *Oracle Database SQL Language Reference* for `ALTER TABLESPACE` syntax and semantics

4.1.3.2 Priority Options for the Population of In-Memory Objects

When you enable an object for the IM column store, you can either let Oracle Database control when the object is populated (default), or you can specify a level that determines the priority of the object in the population queue.

Oracle SQL includes an `INMEMORY PRIORITY` clause that provides more control over the queue for population. For example, it might be more important or less important to populate a database object's data before populating the data for other database objects.

> **Video:**
>
> Video

The following table describes the supported priority levels.

Table 4-1 Priority Levels for Populating a Database Object in the IM Column Store

CREATE/ALTER Syntax	Description
`PRIORITY NONE`	The database populates the object on demand only. A full scan of the database object triggers the population of the object into the IM column store. This is the default level when `PRIORITY` is not included in the `INMEMORY` clause.
`PRIORITY LOW`	The database assigns the object a low priority and populates it after startup based on its position in the queue. Population does not depend on whether the object is accessed. The object is populated in the IM column store before database objects with the following priority level: `NONE`. The database object's data is populated in the IM column store after database objects with the following priority levels: `MEDIUM`, `HIGH`, or `CRITICAL`.

Table 4-1 (Cont.) Priority Levels for Populating a Database Object in the IM Column Store

CREATE/ALTER Syntax	Description
`PRIORITY MEDIUM`	The database assigns the object a medium priority and populates it after startup based on its position in the queue. Population does not depend on whether the object is accessed. The database object is populated in the IM column store before database objects with the following priority levels: `NONE` or `LOW`. The database object's data is populated in the IM column store after database objects with the following priority levels: `HIGH` or `CRITICAL`.
`PRIORITY HIGH`	The database assigns the object a high priority and populates it after startup based on its position in the queue. Population does not depend on whether the object is accessed. The database object's data is populated in the IM column store before database objects with the following priority levels: `NONE`, `LOW`, or `MEDIUM`. The database object's data is populated in the IM column store after database objects with the following priority level: `CRITICAL`.
`PRIORITY CRITICAL`	The database assigns the object a low priority and populates it after startup based on its position in the queue. Population does not depend on whether the object is accessed. The database object's data is populated in the IM column store before database objects with the following priority levels: `NONE`, `LOW`, `MEDIUM`, or `HIGH`.

When more than one database object has a priority level other than `NONE`, Oracle Database queues the data for objects to be populated based on priority level. Database objects with the `CRITICAL` priority level are populated first; database objects with the `HIGH` priority level are populated next, and so on. If no space remains in the IM column store, then no additional objects are populated in it until space becomes available.

> **Note:**
>
> If you specify all objects as `CRITICAL`, then the database does not consider any object as more critical than any other.

When a database is restarted, all of the data for database objects with a priority level other than `NONE` are populated in the IM column store during startup. For a database object with a priority level other than `NONE`, an `ALTER TABLE` or `ALTER MATERIALIZED VIEW` DDL statement involving the database object does not return until the DDL changes are recorded in the IM column store.

Note:

- The priority level setting must apply to an entire table or to a table partition. Specifying different IM column store priority levels for different subsets of columns in a table is not permitted.
- If a segment on disk is 64 KB or less, then it is not populated in the IM column store. Therefore, some small database objects that were enabled for the IM column store might not be populated.

See Also:

- "Prioritization of In-Memory Population"
- *Oracle Database SQL Language Reference* for `CREATE TABLE ... INMEMORY PRIORITY` syntax and semantics

4.1.3.3 IM Column Store Compression Methods

Depending on your requirement, you can compress In-Memory objects at different levels.

Typically, compression is a space-saving mechanism. However, the IM column store can compress data using a new set of algorithms that also improve query performance. If the columnar data is compressed using the `FOR DML` or `FOR QUERY` options, then SQL queries execute directly on the compressed data. Thus, scanning and filtering operations execute on a much smaller amount of data. The database only decompresses data when it is required for the result set.

Video:

Video

The `V$IM_SEGMENTS` and `V$IM_COLUMN_LEVEL` views indicate the current compression level. You can change compression levels by using the appropriate `ALTER` command. If a table is currently populated in the IM column store, and if you change any `INMEMORY` attribute of the table *other than* `PRIORITY`, then the database evicts the table from the IM column store. The repopulation behavior depends on the `PRIORITY` setting.

The following table summarizes the data compression methods supported in the IM column store.

Table 4-2 IM Column Store Compression Methods

CREATE/ALTER Syntax	Description
`NO MEMCOMPRESS`	The data is not compressed.

Table 4-2 (Cont.) IM Column Store Compression Methods

CREATE/ALTER Syntax	Description
`MEMCOMPRESS FOR DML`	This method results in the best DML performance. This method compresses IM column store data the least, with the exception of `NO MEMCOMPRESS`. **Note:** This compression method is not supported for `CELLMEMORY` storage on Exadata flash cache.
`MEMCOMPRESS FOR QUERY LOW`	This method results in the best query performance. This method compresses IM column store data more than `MEMCOMPRESS FOR DML` but less than `MEMCOMPRESS FOR QUERY HIGH`. This method is the default when the `INMEMORY` clause is specified without a compression method in a `CREATE` or `ALTER` SQL statement or when `MEMCOMPRESS FOR QUERY` is specified without including either `LOW` or `HIGH`. **Note:** If the `INMEMORY_FORCE` initialization parameter is set to `BASE_LEVEL`, then `INMEMORY` objects and columns automatically use `QUERY LOW` compression. The data dictionary views may continue to show pre-existing compression settings, but the Base Level always transparently compresses objects and columns at the `QUERY LOW` level.
`MEMCOMPRESS FOR QUERY HIGH`	This method results in good query performance, and saves space. This method compresses IM column store data more than `MEMCOMPRESS FOR QUERY LOW` but less than `MEMCOMPRESS FOR CAPACITY LOW`.
`MEMCOMPRESS FOR CAPACITY LOW`	This method balances space saving and query performance, with a bias toward space saving. This method compresses IM column store data more than `MEMCOMPRESS FOR QUERY HIGH` but less than `MEMCOMPRESS FOR CAPACITY HIGH`. This method applies a proprietary compression technique called Oracle Zip (OZIP) that offers extremely fast decompression that is tuned specifically for Oracle Database. That data must be decompressed before it can be scanned. This method is the default when `MEMCOMPRESS FOR CAPACITY` is specified without including either `LOW` or `HIGH`.
`MEMCOMPRESS FOR CAPACITY HIGH`	This method results in the best space saving. This method compresses IM column store data the most.

In a SQL statement, the `MEMCOMPRESS` keyword must be preceded by the `INMEMORY` keyword.

> **See Also:**
>
> - "About Repopulation of the IM Column Store"
> - *Oracle Exadata System Software User's Guide* to learn more about `ALTER TABLE ... CELLMEMORY`
> - *Oracle Database SQL Language Reference* for `CREATE TABLE ... INMEMORY PRIORITY` syntax and semantics

4.1.3.4 Oracle Compression Advisor

Oracle Compression Advisor estimates the compression ratio that you can realize using the `MEMCOMPRESS` clause. The advisor uses the `DBMS_COMPRESSION` interface.

When you run `DBMS_COMPRESSION.GET_COMPRESSION_RATIO` for a table, Oracle Database analyzes a sample of the rows. For this reason, Oracle Compression Advisor provides a good estimate of the compression results that a table achieves after it is populated into the IM column store.

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* to learn about `DBMS_COMPRESSION.GET_COMPRESSION_RATIO`

4.2 Enabling and Disabling Tables for the IM Column Store

Enable a table for the IM column store by including an `INMEMORY` clause in a `CREATE TABLE` or `ALTER TABLE` statement. Disable a table for the IM column store by including a `NO INMEMORY` clause in a `CREATE TABLE` or `ALTER TABLE` statement.

4.2.1 Enabling New Tables for the In-Memory Column Store

You enable a new table for the IM column store by including an `INMEMORY` clause in a `CREATE TABLE` statement.

You can enable either internal or external tables for the IM column store. Some `INMEMORY` subclauses, including the columns and priority subclauses, are not valid for external tables.

Prerequisites

Ensure that the IM column store is enabled for the database. See "Enabling the IM Column Store for a Database".

To enable a new table for the IM column store:

1. Log in to the database as a user with the necessary privileges to create the table.
2. Run a `CREATE TABLE` statement with an `INMEMORY` clause.

> **See Also:**
>
> - "Enabling and Disabling Tables for the IM Column Store"
> - "Enabling a Subset of Columns for the IM Column Store: Example"
> - *Oracle Database SQL Language Reference* for information about `INMEMORY` clause of the `CREATE TABLE` statement

4.2.2 Enabling and Disabling Existing Tables for the IM Column Store

Enable or disable an existing table for the IM column store by including an `INMEMORY` or `NO INMEMORY` clause in an `ALTER TABLE` statement.

Prerequisites

Ensure that the IM column store is enabled for the database. See "Enabling the IM Column Store for a Database".

To enable or disable an existing table for the IM column store:

1. Log in to the database as a user with `ALTER TABLE` privileges.
2. Run an `ALTER TABLE` statement with an `INMEMORY` clause or a `NO INMEMORY` clause.
3. Optionally, to view metadata (size, priority, compression level) about the In-Memory segment, query `V$IM_SEGMENTS`.

> **See Also:**
>
> - "Enabling and Disabling Tables for the IM Column Store"
> - "Enabling a Subset of Columns for the IM Column Store: Example"
> - *Oracle Database SQL Language Reference* for information about the `ALTER TABLE` statement
> - *Oracle Database Reference* for information about the `V$IM_SEGMENTS` view

4.2.3 Enabling and Disabling Tables for the IM Column Store

The following examples illustrate how to enable or disable tables for the IM column store.

4.2.3.1 Creating an In-Memory Table: Example

This example creates the `test_inmem` table and enables it for the IM column store.

This example creates the test_inmem table and enables it for the IM column store. In SQL*Plus, log in to the database as the user who will own the table, and then execute the following SQL statement:

```
CREATE TABLE test_inmem ( id NUMBER(5) PRIMARY KEY, test_col
VARCHAR2(15)) INMEMORY;
```

The preceding statement uses the defaults for the `INMEMORY` clause: `MEMCOMPRESS FOR QUERY` and `PRIORITY NONE`. Because `PRIORITY` is `NONE`, the database will not automatically populate the table.

You can prepopulate a new table by creating it with a CTAS (`CREATE TABLE AS SELECT`) statement and at the same time enable In-Memory on the new table with this statement, which pulls data from the sh sample schema provided by Oracle:

```
CREATE TABLE test_inmem INMEMORY AS SELECT * FROM sh.sales;
```

You can also include `INMEMORY` subclauses:

```
CREATE TABLE test_inmem INMEMORY MEMCOMPRESS for capacity high AS
SELECT * FROM sh.sales;
```

Note that the following statement format is **not** correct. The `INMEMORY` clause and related subclauses must precede the `SELECT`. Here it it added after the `SELECT`:

```
CREATE TABLE test_inmem AS SELECT * from sh.sales INMEMORY MEMCOMPRESS
for capacity high;
```

In this incorrect usage, INMEMORY is interpreted as keyword, not a reserved word. It is silently ignored and INMEMORY remains disabled. Although `MEMCOMPRESS`, which is also interpreted as a keyword, raises an error.

4.2.3.2 Creating a Table with In-Memory Partitions: Example

This example creates a partitioned table named `range_sales`, specifying a subset of the partitions as `INMEMORY`.

Log in to SQL*Plus as the user that will own the new table, and then execute the following DDL statement:

```
CREATE TABLE range_sales
    ( prod_id        NUMBER(6)
    , cust_id        NUMBER
    , time_id        DATE
    , channel_id     CHAR(1)
    , promo_id       NUMBER(6)
    , quantity_sold  NUMBER(3)
    , amount_sold    NUMBER(10,2)
    )
PARTITION BY RANGE (time_id)
  (PARTITION SALES_Q4_1999
      VALUES LESS THAN (TO_DATE('01-JAN-2015','DD-MON-YYYY'))
```

```
   INMEMORY MEMCOMPRESS FOR DML,
PARTITION SALES_Q1_2000
   VALUES LESS THAN (TO_DATE('01-APR-2015','DD-MON-YYYY'))
   INMEMORY MEMCOMPRESS FOR QUERY,
PARTITION SALES_Q2_2000
   VALUES LESS THAN (TO_DATE('01-JUL-2015','DD-MON-YYYY'))
   INMEMORY MEMCOMPRESS FOR CAPACITY,
PARTITION SALES_Q3_2000
   VALUES LESS THAN (TO_DATE('01-OCT-2015','DD-MON-YYYY'))
   NO INMEMORY,
PARTITION SALES_Q4_2000
   VALUES LESS THAN (MAXVALUE));
```

The preceding SQL specifies a different compression level for the first three partitions in the IM column store. The last two partitions are not eligible for population in the IM column store.

4.2.3.3 Creating an In-Memory External Table: Example

This example creates an external table with the `INMEMORY` option.

This example assumes that the host has the directories `/tmp/data/`, `/tmp/log/`, and `/tmp/bad/`.

The following SQL script creates the comma-delimited flat file `/tmp/data/sh_sales.csv` from the `sh.sales` table. Execute the script as user `sh`.

```
SET HEAD OFF
SET TRIMSPOOL ON
SET PAGES 0
SET FEEDBACK OFF
SET TERMOUT OFF
SPOOL /tmp/data/sh_sales.csv
SELECT prod_id       || ',' || cust_id  || ',' || time_id || ',' ||
       channel_id    || ',' || promo_id || ',' ||
       quantity_sold || ',' || amount_sold
FROM   sales;
SPOOL OFF
```

Using the `sh_sales.csv` file, the following SQL script creates the external table `sh.admin_ext_sales` with the `INMEMORY` option:

```
CONNECT  /  AS SYSDBA;
-- Set up directories and grant access to sh
CREATE OR REPLACE DIRECTORY admin_dat_dir
    AS '/tmp/data';
CREATE OR REPLACE DIRECTORY admin_log_dir
    AS '/tmp/log';
CREATE OR REPLACE DIRECTORY admin_bad_dir
    AS '/tmp/bad';
GRANT READ ON DIRECTORY admin_dat_dir TO sh;
GRANT WRITE ON DIRECTORY admin_log_dir TO sh;
GRANT WRITE ON DIRECTORY admin_bad_dir TO sh;
-- sh connects. Provide the user password (sh) when prompted.
CONNECT sh
```

```
-- create the external table
DROP TABLE admin_ext_sales;
CREATE TABLE admin_ext_sales
     (  prod_id           NUMBER,
        cust_id           NUMBER,
        time_id           DATE,
        channel_id        NUMBER,
        promo_id          NUMBER,
        quantity_sold     NUMBER(10,2),
        amount_sold       NUMBER(10,2)
     )
     ORGANIZATION EXTERNAL
     ( TYPE ORACLE_LOADER
       DEFAULT DIRECTORY admin_dat_dir
       ACCESS PARAMETERS
       ( records delimited by newline
         badfile admin_bad_dir:'empxt%a_%p.bad'
         logfile admin_log_dir:'empxt%a_%p.log'
         fields terminated by ','
         missing field values are null
         ( prod_id, cust_id,
           time_id char date_format date mask "dd-mon-yy",
           channel_id, promo_id, quantity_sold, amount_sold
         )
       )
       LOCATION ('sh_sales.csv')
     )
     REJECT LIMIT UNLIMITED
     INMEMORY;
```

The following query of `ALL_EXTERNAL_TABLES` shows that the `admin_ext_sales` table is enabled for `INMEMORY`:

```
COL OWNER FORMAT A10
COL TABLE_NAME FORMAT A15

SELECT OWNER, TABLE_NAME,
       INMEMORY, INMEMORY_COMPRESSION
FROM   ALL_EXTERNAL_TABLES
WHERE  TABLE_NAME = 'ADMIN_EXT_SALES';

OWNER      TABLE_NAME      INMEMORY INMEMORY_COMPRESS
---------- --------------- -------- -----------------
SH         ADMIN_EXT_SALES ENABLED  FOR QUERY LOW
```

Related views include `ALL_XTERNAL_PART_TABLES`, `ALL_XTERNAL_TAB_PARTITIONS`, and `ALL_XTERNAL_TAB_SUBPARTITIONS`.

> **See Also:**
>
> - "Populating an In-Memory External Table Using DBMS_INMEMORY.POPULATE: Example"
> - *Oracle Database Reference* to learn about `ALL_EXTERNAL_TABLES` and the related external table views

4.2.3.4 Creating and Populating a Hybrid External Table: Example

This example creates a hybrid external table with the `INMEMORY` option, and then populates both the internal and external partitions.

This example assumes the existence of the `sh.sales` table. The goal is to create a hybrid partitioned table `sales_hpt` with two internal partitions, one of which uses the data from `sh.sales`, and then add one external partition. When you apply the `INMEMORY` attribute to `sales_hpt`, this attribute applies to all partitions.

1. In Linux, create a temporary directory, and then create a text file with one row of sales data.

```
rm -rf /tmp/sales_data
mkdir /tmp/sales_data
echo "1002,110,19-MAR-2016,12,18,150,4800" > /tmp/sales_data/
sales2016_data.txt
```

2. In SQL*Plus, log is with administrator privileges, and then create a directory object for the sales data:

```
CONNECT / AS SYSDBA
CREATE DIRECTORY sales_data AS '/tmp/sales_data';
GRANT READ,WRITE ON DIRECTORY sales_data TO sh;
```

3. Log in as user `sh`, and then create the `sales_hpt` table:

```
CONNECT sh

DROP TABLE sales_hpt;
CREATE TABLE sales_hpt
  ( prod_id       NUMBER        NOT NULL,
    cust_id       NUMBER        NOT NULL,
    time_id       DATE          NOT NULL,
    channel_id    NUMBER        NOT NULL,
    promo_id      NUMBER        NOT NULL,
    quantity_sold NUMBER(10,2)  NOT NULL,
    amount_sold   NUMBER(10,2)  NOT NULL
  )
    EXTERNAL PARTITION ATTRIBUTES (
      TYPE ORACLE_LOADER
      DEFAULT DIRECTORY sales_data
       ACCESS PARAMETERS(
         FIELDS TERMINATED BY ','
```

```
         (prod_id,cust_id,time_id DATE 'dd-mm-
yyyy',channel_id,promo_id,quantity_sold,amount_sold)
       )
      REJECT LIMIT UNLIMITED
     )
    PARTITION BY RANGE (time_id)
    (
     PARTITION sales_2014 VALUES LESS THAN
(TO_DATE('01-01-2015','dd-mm-yyyy')),
     PARTITION sales_2015 VALUES LESS THAN
(TO_DATE('01-01-2016','dd-mm-yyyy')),
     PARTITION sales_2016 VALUES LESS THAN
(TO_DATE('01-01-2017','dd-mm-yyyy'))
       EXTERNAL LOCATION ('sales2016_data.txt')
);
```

The preceding statement shows that the table has three partitions: `sales_2014`, `sales_2015`, and `sales_2016`. Only `sales_2016` is designated as external.

4. Query the data dictionary to confirm that the table is a hybrid (sample output included):

```
COL TABLE_NAME FORMAT a25
SELECT TABLE_NAME, HYBRID FROM USER_TABLES WHERE HYBRID = 'YES';

TABLE_NAME                HYB
------------------------- ---
SALES_HPT                 YES
```

5. Insert rows into the internal partitions `sales_2014` and `sales_2015`:

```
INSERT INTO sh.sales_hpt (SELECT * FROM sales);
INSERT INTO sh.sales_hpt
  VALUES (30, 21086, TO_DATE('2015-12-30','SYYYY-MM-DD'), 2, 999,
1, 10.19);
COMMIT;
```

The first of the preceding statements inserts all rows from the `sales` table. All dates in `sales` are before 2002, so all rows from `sales` are inserted into the `sales_2014` partition. The second statement inserts a single row into the `sales_2015` partition.

6. Query the partitions to confirm that the correct data exists:

```
SQL> SELECT COUNT(*) FROM sales_hpt PARTITION(sales_2014);

  COUNT(*)
----------
    918843

SQL> SELECT COUNT(*) FROM sales_hpt PARTITION(sales_2015);

  COUNT(*)
----------
```

```
         1

SQL> SELECT COUNT(*) FROM sales_hpt PARTITION(sales_2016);

  COUNT(*)
----------
         1
```

7. Apply the `INMEMORY` attribute at the table level, and then force the database to populate the table into the IM column store:

```
ALTER TABLE sales_hpt INMEMORY;
EXEC DBMS_INMEMORY.POPULATE('SH', 'SALES_HPT');
```

8. Query the population status of the `sales_hpt` partitions:

```
COL OWNER FORMAT a3
COL SEGMENT FORMAT a18
COL PARTITION FORMAT a13
COL STATUS FORMAT a9
COL BNP FORMAT 99999

SELECT OWNER, SEGMENT_NAME SEGMENT, PARTITION_NAME PARTITION,
       IS_EXTERNAL AS EXT, POPULATE_STATUS STATUS,
       BYTES_NOT_POPULATED AS "BNP"
FROM   V$IM_SEGMENTS
WHERE  SEGMENT_NAME = 'SALES_HPT'
ORDER BY PARTITION;

OWN SEGMENT            PARTITION     EXT   STATUS       BNP
--- ------------------ ------------- ----- --------- ------
SH  SALES_HPT          SALES_2014    FALSE COMPLETED      0
SH  SALES_HPT          SALES_2015    FALSE COMPLETED      0
SH  SALES_HPT          SALES_2016    TRUE  COMPLETED      0
```

The query shows that all partitions, both internal and external, were populated.

4.2.3.5 Enabling an Existing Table for the IM Column Store: Example

This example enables the existing `sh.sales` table for the IM column store.

In SQL*Plus, log in to the database as the `sh` user, and then execute the following DDL statement:

```
ALTER TABLE sales INMEMORY;
```

The preceding statement uses the defaults for the `INMEMORY` clause: `MEMCOMPRESS FOR QUERY` and `PRIORITY NONE`.

4.2.3.6 Setting In-Memory Compression to FOR CAPACITY LOW: Example

This example enables the existing `oe.product_information` table for the IM column store and specifies the compression method `FOR CAPACITY LOW`.

In SQL*Plus, log in to the database as the `oe` user, and then execute the following DDL statement:

```
ALTER TABLE product_information
  INMEMORY
  MEMCOMPRESS FOR CAPACITY LOW;
```

The preceding statement uses the default for the `PRIORITY` clause of `NONE`. Populate the table by forcing a full table scan as follows (sample output included):

```
SELECT /*+ FULL(p) NO_PARALLEL(p) */ COUNT(*)
FROM   product_information p;

  COUNT(*)
----------
       288
```

In a separate session, log in as a user with administrative privileges, and then calculate the compression ratio by executing the following query (sample output included):

```
COL OWNER FORMAT a5
COL SEGMENT_NAME FORMAT a19
SET PAGESIZE 50000

SELECT OWNER, SEGMENT_NAME, BYTES ORIG_SIZE,
       INMEMORY_SIZE IN_MEM_SIZE,
       ROUND (BYTES / INMEMORY_SIZE, 2) COMP_RATIO
FROM   V$IM_SEGMENTS
WHERE  SEGMENT_NAME LIKE 'P%'
ORDER BY 4;

OWNER SEGMENT_NAME         ORIG_SIZE IN_MEM_SIZE COMP_RATIO
----- ------------------- ---------- ----------- ----------
OE    PRODUCT_INFORMATION      98304     1310720        .08
```

4.2.3.7 Setting In-Memory Priority to HIGH: Example

This example enables the `oe.product_information` table for the IM column store and specifies `PRIORITY HIGH` for populating the table data in the IM column store.

In SQL*Plus, log in to the database as the `oe` user, and then execute the following DDL statement:

```
ALTER TABLE
  product_information
  INMEMORY
  PRIORITY HIGH;
```

4.2.3.8 Changing the Compression and Priority Settings for an In-Memory Table: Example

This example alters the `oe.product_information` table to use `FOR CAPACITY HIGH` table compression and a `LOW` priority setting.

In SQL*Plus, log in to the database as an administrative user, and then execute the following query to show the current priority and compression setting for the `oe.product_information` table:

```
COL OWNER FORMAT a5
COL SEGMENT_NAME FORMAT a19
SET PAGESIZE 50000

SELECT v.OWNER, v.SEGMENT_NAME, v.INMEMORY_PRIORITY,
       v.INMEMORY_COMPRESSION
FROM   V$IM_SEGMENTS v
WHERE  SEGMENT_NAME LIKE 'P%';

OWNER SEGMENT_NAME        INMEMORY INMEMORY_COMPRESS
----- ------------------- -------- -----------------
OE    PRODUCT_INFORMATION HIGH     FOR CAPACITY LOW
```

The following DDL statement alters `oe.product_information` to use `FOR CAPACITY HIGH` table compression and `PRIORITY LOW`:

```
ALTER TABLE oe.product_information
  INMEMORY
  MEMCOMPRESS FOR CAPACITY HIGH
  PRIORITY LOW;
```

4.2.3.9 Disabling a Table for the IM Column Store: Example

To disable a table for the IM column store, specify the `NO INMEMORY` clause.

Log in to the database as the user `oe`, and then execute the following statement to disable the `product_information` table for the IM column store:

```
ALTER TABLE oe.product_information NO INMEMORY;
```

The `V$IM_SEGMENTS` view lists the database objects that are populated in the IM column store.

4.2.3.10 Disabling Columnar Format on Exadata Smart Flash Cache: Example

This example disables the columnar format for `oe.product_information` on Exadata Smart Flash Cache storage.

By default, Exadata Smart Flash Cache compresses data using the level `MEMCOMPRESS FOR CAPACITY LOW`. To change the compression level or disable the columnar format altogether, use the `ALTER TABLE ... NO CELLMEMORY` statement.

Log in to the database as user `oe`, and execute the following DDL statement:

```
ALTER TABLE product_information NO CELLMEMORY;
```

4.3 Enabling and Disabling Columns for In-Memory Tables

You can specify the `INMEMORY` clause for individual columns in an internal table. External tables do not support specifying `INMEMORY` at the column level.

4.3.1 About Enabling INMEMORY Columns

For internal tables, both In-Memory virtual columns (IM virtual columns) and nonvirtual columns are eligible for IM population. For external tables, only nonvirtual columns are eligible.

NO INMEMORY Columns in INMEMORY Objects

> **Note:**
>
> Excluded columns can be specified at the table level only. You cannot specify them for partitions or sub-partitions

By default, if an internal object has the `INMEMORY` attribute, then all columns in this object are eligible for population into the IM column store. However, you can specify that some columns in an `INMEMORY` table should have the `NO INMEMORY` attribute, in which case only the `INMEMORY` columns are eligible for population.

To apply the `INMEMORY` attribute to a subset of columns, specify `ALTER TABLE` *`table_name`* `INMEMORY ... NO INMEMORY` *`excluded_columns`*, where *excluded_columns* lists the columns to be specified as `NO INMEMORY`. Only the columns that are not specified as `NO INMEMORY`, that is, the columns that are not in the excluded columns list, inherit the segment-level `INMEMORY` attribute.

Specifying `NO INMEMORY` columns in an `INMEMORY` table has an important consequence. A query that references a `NO INMEMORY` column uses the row store *exclusively*. For example, if `sales` is an `INMEMORY` table with 7 columns, and if only the `promo_id` column has the `NO INMEMORY` attribute, then any query whose `SELECT` list includes `promo_id` retrieves data from the row store rather than the IM column store. Likewise, any query whose predicate references `promo_id`, regardless of whether `promo_id` is in the `SELECT` list, retrieves data exclusively from the row store.

Virtual Columns

An IM virtual column is like any other column, except that its value is derived by evaluating an expression. Storing the precalculated IM virtual column values in the IM column store can improve query performance. The expression can include columns from the same table, constants, SQL functions, and user-defined PL/SQL functions (`DETERMINISTIC` only). You cannot explicitly write to an IM virtual column.

> **Note:**
>
> A virtual column or IM expression counts toward the limit of 1000 columns per populated object.

To populate IM virtual columns in the IM column store, set the `INMEMORY_VIRTUAL_COLUMNS` initialization parameter to one of the following values:

- `MANUAL` (default): If a table is enabled for the IM column store, then no IM virtual columns defined on this table are eligible for population, unless they are explicitly set as `INMEMORY`.
- `ENABLE`: If a table is enabled for the IM column store, then all IM virtual columns defined on this table are eligible for population, unless they are explicitly set as `NO INMEMORY`.

  By default, the compression level of the column in the IM column store is the same as the table or partition in which it is stored. However, when a different compression level is specified for the IM virtual column, it is populated at the specified compression level.

To specify that no IM virtual columns are populated in the IM column store, set this initialization parameter to `DISABLE`.

The underlying storage structures for IM virtual columns and IM expressions are the same. However, different mechanisms control IM expressions and IM virtual columns.

> **Note:**
>
> - The IM column store only populates virtual columns for tables marked `INMEMORY`.
> - To populate IM virtual columns in the IM column store, the value for the initialization parameter `COMPATIBLE` must be set to 12.1.0 or higher.

> **See Also:**
>
> - "Enabling and Disabling Tables for the IM Column Store"
> - "IM Expressions Infrastructure"
> - "In-Memory Initialization Parameters"
> - *Oracle Database SQL Language Reference* for the syntax and semantics of the `INMEMORY` clause
> - https://blogs.oracle.com/in-memory/what-happens-if-a-column-is-not-populated for a blog entry on accessing columns that are not populated

4.3.2 Enabling IM Virtual Columns

IM virtual columns improve query performance by avoiding repeated calculations. Also, the database can scan and filter IM virtual columns using techniques such as SIMD vector processing.

Prerequisites

To enable IM virtual columns, the following conditions must be true:

- The IM column store is enabled for the database.

  See "Enabling the IM Column Store for a Database".
- The table that contains the virtual columns is internal and has the `INMEMORY` attribute.

  See "Enabling and Disabling Tables for the IM Column Store".
- The `INMEMORY_VIRTUAL_COLUMNS` initialization parameter is *not* set to `DISABLE`.
- The value for the initialization parameter `COMPATIBLE` is set to `12.1.0` or higher.

To enable IM virtual columns:

1. In SQL*Plus or SQL Developer, log in to the database as a user with the necessary privileges.
2. Either set the `INMEMORY_VIRTUAL_COLUMNS` initialization parameter to `ENABLE`, or enable specific virtual columns for the IM column store.

Example 4-3 Enabling Virtual Columns for the IM Column Store

In this example, you are logged in to the database as `SYSTEM`. The IM column store is enabled, but population of virtual columns is currently disabled:

```
SQL> SHOW PARAMETER INMEMORY_SIZE

NAME                                 TYPE        VALUE
------------------------------------ ----------- -----
inmemory_size                        big integer 200M

SQL> SHOW PARAMETER INMEMORY_VIRTUAL_COLUMNS

NAME                                 TYPE        VALUE
------------------------------------ ----------- -------
inmemory_virtual_columns             string      DISABLE
```

You add a virtual column to the `hr.employees` table, and then specify that the table is `INMEMORY`:

```
SQL> ALTER TABLE hr.employees ADD (weekly_sal AS
(ROUND(salary*12/52,2)));

Table altered.

SQL> ALTER TABLE hr.employees INMEMORY;
```

```
Table altered.
```

At this stage, `weekly_sal` is not eligible for population, although the non-virtual columns in `hr.employees` are eligible for population. The following statement enables `weekly_sal`, and any other virtual columns in `hr.employees`, to be populated:

```
SQL> ALTER SYSTEM SET INMEMORY_VIRTUAL_COLUMNS=ENABLE SCOPE=BOTH;

System altered.
```

Note that you could also use `SCOPE=SPFILE`, but in that case the change will not take effect until the next database restart. When `SCOPE=BOTH` is used, the alteration takes place immediately and a restart is not required.

Example 4-4 Enabling a Specific IM Virtual Column for the IM Column Store

This example assumes that the `INMEMORY_VIRTUAL_COLUMNS` initialization parameter is set to `MANUAL`, which means that IM virtual columns must be added to the IM column store explicitly. This example first creates the `hr.admin_emp` table:

```
CREATE TABLE hr.admin_emp (
      empno      NUMBER(5) PRIMARY KEY,
      ename      VARCHAR2(15) NOT NULL,
      job        VARCHAR2(10),
      sal        NUMBER(7,2),
      hrly_rate  NUMBER(7,2) GENERATED ALWAYS AS (sal/2080),
      deptno     NUMBER(3) NOT NULL)
   INMEMORY;
```

At this stage, the `hrly_rate` virtual column is not eligible for population. The following statement explicitly specifies the virtual column as `INMEMORY`:

```
ALTER TABLE hr.admin_emp INMEMORY(hrly_rate);
```

4.3.3 Enabling a Subset of Columns for the IM Column Store: Example

This example enables all columns in the `oe.product_information` table for the IM column store except `weight_class` and `catalog_url`.

The following statement also specifies different IM column store compression methods for the columns enabled for the IM column store:

```
ALTER TABLE oe.product_information
   INMEMORY MEMCOMPRESS FOR QUERY (
      product_id, product_name, category_id, supplier_id, min_price)
   INMEMORY MEMCOMPRESS FOR CAPACITY HIGH (
      product_description, warranty_period, product_status, list_price)
   NO INMEMORY (
      weight_class, catalog_url);
```

Note the following:

- The columns `product_id`, `product_name`, `category_id`, `supplier_id`, and `min_price` are enabled for the IM column store with the `MEMCOMPRESS FOR QUERY` compression method.
- The columns `product_description`, `warranty_period`, `product_status`, and `list_price` are enabled for the IM column store with the `MEMCOMPRESS FOR CAPACITY HIGH` compression method.
- The `weight_class` and `catalog_url` columns are not enabled for the IM column store. Consequently, any query that references these two columns, either in the `SELECT` list or in the predicate, must use the row store rather than the IM column store.
- The table uses the default for the `PRIORITY` clause, which is `PRIORITY NONE`.

> **Note:**
>
> The priority level setting must apply to an entire table or partition. Specifying different IM column store priority levels for different subsets of columns in a table is not allowed.

To determine the selective column compression levels defined for a database object, query the `V$IM_COLUMN_LEVEL` view, as shown in the following example:

```
COL TABLE_NAME FORMAT a20
COL COLUMN_NAME FORMAT a20

SELECT TABLE_NAME, COLUMN_NAME, INMEMORY_COMPRESSION
FROM   V$IM_COLUMN_LEVEL
WHERE  TABLE_NAME = 'PRODUCT_INFORMATION'
ORDER BY COLUMN_NAME;

TABLE_NAME           COLUMN_NAME          INMEMORY_COMPRESSION
-------------------- -------------------- --------------------------
PRODUCT_INFORMATION  CATALOG_URL          NO INMEMORY
PRODUCT_INFORMATION  CATEGORY_ID          FOR QUERY LOW
PRODUCT_INFORMATION  LIST_PRICE           FOR CAPACITY HIGH
PRODUCT_INFORMATION  MIN_PRICE            FOR QUERY LOW
PRODUCT_INFORMATION  PRODUCT_DESCRIPTION  FOR CAPACITY HIGH
PRODUCT_INFORMATION  PRODUCT_ID           FOR QUERY LOW
PRODUCT_INFORMATION  PRODUCT_NAME         FOR QUERY LOW
PRODUCT_INFORMATION  PRODUCT_STATUS       FOR CAPACITY HIGH
PRODUCT_INFORMATION  SUPPLIER_ID          FOR QUERY LOW
PRODUCT_INFORMATION  WARRANTY_PERIOD      FOR CAPACITY HIGH
PRODUCT_INFORMATION  WEIGHT_CLASS         NO INMEMORY
```

> **See Also:**
>
> - "In-Memory Compression"
> - *Oracle Database Reference* for more information about the `V$IM_COLUMN_LEVEL` view

4.3.4 Specifying INMEMORY Column Attributes on a NO INMEMORY Table: Example

Starting in Oracle Database 12c Release 2 (12.2), you can specify the `INMEMORY` clause at the column level on an object that is not yet specified as `INMEMORY`.

In previous releases, the column-level `INMEMORY` clause was only valid when specified on an `INMEMORY` table or partition. This restriction meant that a column could not be associated with an `INMEMORY` clause before the table or partition was associated with an `INMEMORY` clause.

Starting in Oracle Database 12c Release 2 (12.2), if you specify the `INMEMORY` clause at the column level, then the database records the attributes of the specified column. If the table is `NO INMEMORY` (default), then the column-level attributes do not affect how the table is queried until the table or partition is specified as `INMEMORY`. If you mark the table itself as `NO INMEMORY`, then the database drops any existing column-level attributes.

In this example, your goal is to ensure that column `c3` in a partitioned table is never populated in the IM column store. You perform the following steps:

1. Create a partitioned table `t` as follows:

```
CREATE TABLE t (c1 NUMBER, c2 NUMBER, c3 NUMBER)
  NO INMEMORY -- this clause specifies the table itself as NO INMEMORY
  PARTITION BY LIST (c1)
    ( PARTITION p1 VALUES (0),
      PARTITION p2 VALUES (1),
      PARTITION p3 VALUES (2) );
```

   Table `t` is `NO INMEMORY`. The table is partitioned by list on column `c1`, and has three partitions: `p1`, `p2`, and `p3`.

2. Query the compression of the columns in the table (sample output included):

```
COL TABLE_NAME FORMAT a20
COL COLUMN_NAME FORMAT a20

SELECT TABLE_NAME, COLUMN_NAME, INMEMORY_COMPRESSION
FROM   V$IM_COLUMN_LEVEL
WHERE  TABLE_NAME = 'T'
ORDER BY COLUMN_NAME;

no rows selected
```

   As shown by the output, no column-level `INMEMORY` attributes are set.

3. To ensure that column `c3` is never populated, apply the `NO INMEMORY` attribute to column `c3`:

```
ALTER TABLE t NO INMEMORY (c3);
```

4. Query the compression of the columns in the table (sample output included):

```
SELECT TABLE_NAME, COLUMN_NAME, INMEMORY_COMPRESSION
FROM   V$IM_COLUMN_LEVEL
WHERE  TABLE_NAME = 'T'
ORDER BY COLUMN_NAME;

TABLE_NAME           COLUMN_NAME          INMEMORY_COMPRESSION
-------------------- -------------------- --------------------
T                    C1                   DEFAULT
T                    C2                   DEFAULT
T                    C3                   NO INMEMORY
```

The database has recorded the `NO INMEMORY` attribute for `c3`. The other columns use the default compression.

5. Specify partition `p3` as `INMEMORY`:

```
ALTER TABLE t
  MODIFY PARTITION p3
    INMEMORY PRIORITY CRITICAL;
```

Because column `c3` was previously specified as `NO INMEMORY`, initial population of partition `p3` will not include column `c3`.

6. Specify the entire table as `INMEMORY`:

```
ALTER TABLE t INMEMORY;
```

7. Query the compression of the columns in the table (sample output included):

```
SELECT TABLE_NAME, COLUMN_NAME, INMEMORY_COMPRESSION
FROM   V$IM_COLUMN_LEVEL
WHERE  TABLE_NAME = 'T'
ORDER BY COLUMN_NAME;

TABLE_NAME           COLUMN_NAME          INMEMORY_COMPRESSION
-------------------- -------------------- -------------------------
T                    C1                   DEFAULT
T                    C2                   DEFAULT
T                    C3                   NO INMEMORY
```

The database has retained the `NO INMEMORY` setting for column `c3`. The other columns use the default compression.

8. Apply different compression levels to columns `c1` and `c2`:

```
ALTER TABLE t
  INMEMORY MEMCOMPRESS FOR CAPACITY HIGH (c1)
  INMEMORY MEMCOMPRESS FOR CAPACITY LOW (c2);
```

9. Query the compression of the columns in the table (sample output included):

```
SELECT TABLE_NAME, COLUMN_NAME, INMEMORY_COMPRESSION
FROM   V$IM_COLUMN_LEVEL
WHERE  TABLE_NAME = 'T'
ORDER BY COLUMN_NAME;

TABLE_NAME           COLUMN_NAME          INMEMORY_COMPRESSION
-------------------- -------------------- -------------------------
T                    C1                   FOR CAPACITY HIGH
T                    C2                   FOR CAPACITY LOW
T                    C3                   NO INMEMORY
```

Each column now has a different compression level.

10. Specify the entire table as `NO INMEMORY`:

```
ALTER TABLE t NO INMEMORY;
```

11. Query the compression of the columns in the table (sample output included):

```
SELECT TABLE_NAME, COLUMN_NAME, INMEMORY_COMPRESSION
FROM   V$IM_COLUMN_LEVEL
WHERE  TABLE_NAME = 'T'
ORDER BY COLUMN_NAME;

no rows selected
```

Because the entire table was specified as `NO INMEMORY`, the database dropped all column-level `INMEMORY` attributes.

> **See Also:**
>
> *Oracle Database SQL Language Reference* for `ALTER TABLE` syntax and semantics

4.4 Enabling and Disabling Tablespaces for the IM Column Store

You can enable or disable tablespaces for the IM column store.

Enable a tablespace for the IM column store during tablespace creation with a `CREATE TABLESPACE` statement that includes the `INMEMORY` clause. You can also alter a tablespace to

enable it for the IM column store with an `ALTER TABLESPACE` statement that includes the `INMEMORY` clause.

Disable a tablespace for the IM column store by including a `NO INMEMORY` clause in a `CREATE TABLESPACE` or `ALTER TABLESPACE` statement.

When a tablespace is enabled for the IM column store, all tables and materialized views in the tablespace are enabled for the IM column store by default. The `INMEMORY` clause is the same for tables, materialized views, and tablespaces. The `DEFAULT` storage clause is required before the `INMEMORY` clause when enabling a tablespace for the IM column store and before the `NO INMEMORY` clause when disabling a tablespace for the IM column store.

When a tablespace is enabled for the IM column store, individual tables and materialized views in the tablespace can have different in-memory settings, and the settings for individual database objects override the settings for the tablespace. For example, if the tablespace is set to `PRIORITY LOW` for populating data in memory, and if a table in the tablespace is set to `PRIORITY HIGH`, then the table uses `PRIORITY HIGH`.

Prerequisites

Ensure that the IM column store is enabled for the database. See "Enabling the IM Column Store for a Database".

To enable or disable tablespaces for the IM column store:

1. In SQL*Plus or SQL Developer, log in to the database as a user with the necessary privileges.
2. Run a `CREATE TABLESPACE` or `ALTER TABLESPACE` statement with an `INMEMORY` clause or a `NO INMEMORY` clause.

Example 4-5 Creating a Tablespace and Enabling It for the IM Column Store

The following example creates the `users01` tablespace and enables it for the IM column store:

```
CREATE TABLESPACE users01
   DATAFILE 'users01.dbf' SIZE 40M
   ONLINE
   DEFAULT INMEMORY;
```

This example uses the defaults for the `INMEMORY` clause. Therefore, `MEMCOMPRESS FOR QUERY` is used, and `PRIORITY NONE` is used.

Example 4-6 Altering a Tablespace to Enable It for the IM Column Store

The following example alters the `users01` tablespace to enable it for the IM column store and specifies `FOR CAPACITY HIGH` compression for the database objects in the tablespace and `PRIORITY LOW` for populating data in memory:

```
ALTER TABLESPACE users01 DEFAULT INMEMORY
   MEMCOMPRESS FOR CAPACITY HIGH
   PRIORITY LOW;
```

4.5 Enabling and Disabling Materialized Views for the IM Column Store

You can enable and disable materialized views for the IM column store.

Enable a materialized view for the IM column store by including an `INMEMORY` clause in a `CREATE MATERIALIZED VIEW` or `ALTER MATERIALIZED VIEW` statement. Disable a materialized view for the IM column store by including a `NO INMEMORY` clause in a `CREATE MATERIALIZED VIEW` or `ALTER MATERIALIZED VIEW` statement.

Prerequisites

Ensure that the IM column store is enabled for the database. See "Enabling the IM Column Store for a Database".

To enable or disable a materialized view for the IM column store:

1. In SQL*Plus or SQL Developer, log in to the database as a user with the necessary privileges.
2. Run a `CREATE MATERIALIZED VIEW` or `ALTER MATERIALIZED VIEW` statement with either an `INMEMORY` clause or a `NO INMEMORY` clause.

Example 4-7 Creating a Materialized View and Enabling It for the IM Column Store

The following statement creates the `oe.prod_info_mv` materialized view and enables it for the IM column store:

```
CREATE MATERIALIZED VIEW oe.prod_info_mv INMEMORY
  AS SELECT * FROM oe.product_information;
```

This example uses the defaults for the `INMEMORY` clause: `MEMCOMPRESS FOR QUERY LOW` and `PRIORITY NONE`.

Example 4-8 Enabling a Materialized View for the IM Column Store with HIGH Data Population Priority

The following statement enables the `oe.prod_info_mv` materialized view for the IM column store:

```
ALTER MATERIALIZED VIEW oe.prod_info_mv INMEMORY PRIORITY HIGH;
```

This example uses the default compression: `MEMCOMPRESS FOR QUERY LOW`.

> **See Also:**
>
> *Oracle Database SQL Language Reference* to learn more about the `CREATE` or `ALTER MATERIALIZED VIEW` statements

5
Populating the IM Column Store Manually

The database does not automatically populate In-Memory objects whose `PRIORITY` setting is `NONE`. To populate these objects, you must execute SQL or PL/SQL.

This chapter contains the following topics:

5.1 About Manual Population of In-Memory Objects

Populate objects manually using either a full table scan, `DBMS_INMEMORY` program units, or `DBMS_INMEMORY_ADMIN.POPULATE_WAIT`.

If you enabled an object with `PRIORITY` set to `NONE`, and if you want to populate it immediately, then you have the options described in the following sections:

5.1.1 Population Using SELECT

You can initiate population by issuing a `SELECT` statement that forces a full table scan.

In this case, the database reads each row in the object and converts it to columnar format. Note that the following statement does not *guarantee* a full table scan:

```
SELECT COUNT(*) FROM object
```

The reason is that the optimizer may choose to scan an index. Therefore, Oracle recommends forcing a full table scan by using the `FULL` hint for `SELECT COUNT(*)` queries, as in the following example:

```
SELECT /*+ FULL(customers) NO_PARALLEL(customers) */ COUNT(*) FROM customers;
```

> **See Also:**
>
> - "Populating an In-Memory Table Using a Full Table Scan: Example"
> - *Oracle Database SQL Language Reference* to learn more about the `FULL` hint

5.1.2 Population Using DBMS_INMEMORY.POPULATE

The `DBMS_INMEMORY.POPULATE` procedure achieves the same goal as a full scan.

The database reads every row in the specified object, converts it from row format to columnar format, and then populates it in the IM column store. The following PL/SQL block initiates population of the `customer` table:

```
BEGIN
  DBMS_INMEMORY.POPULATE( schema_name => 'SH', table_name =>
'CUSTOMERS');
END;
/
```

See Also:

- "Populating a Table Using the POPULATE Procedure: Example"
- *Oracle Database PL/SQL Packages and Types Reference* to learn about the `DBMS_INMEMORY` package

5.1.3 Population Using DBMS_INMEMORY_ADMIN.POPULATE_WAIT

The `DBMS_INMEMORY_ADMIN.POPULATE_WAIT` function initiates population of all `INMEMORY` objects that have a priority greater than or equal to the specified priority, and returns a status value for the population. A user-specified interval specifies the maximum time that the function waits before returning the value to the caller.

Sample use cases for ensuring that objects are populated include:

- When the database is closed, open the database with `STARTUP RESTRICT` so that only administrators can access the database, and then execute `POPULATE_WAIT` with the desired timeout setting. If `POPULATE_WAIT` returns `-1`, indicating a timeout, then reexecute `POPULATE_WAIT`. When the function returns `0`, disable the restricted session so that non-administrative users can query the database.
- Block database connections by using services or an application tier technique. When no analytic indexes exists, and when the application depends on the IM column store to provide reasonable performance, these techniques prevent runaway queries.

The `POPULATE_WAIT` function does not accept a table name as input. Rather, the function submits population tasks for *all* `INMEMORY` objects with a `PRIORITY` setting greater than or equal to the `priority` specified (the default is `LOW`). If priority is `NONE`, then the function initiates population for all `INMEMORY` objects. `POPULATE_WAIT` does not apply to external tables, which have no priority setting.

The function accepts a population percentage as input, which defaults to `100`, and a timeout interval, which defaults to `99999999` seconds (115.74 days). When you execute the function, the database attempts to populate the objects that meet the specified `PRIORITY` criteria within the timeout interval, and then returns a value indicating the population status.

The following table describes the possible return values for `POPULATE_WAIT`. The function returns the values `0`, `1`, `2`, and `3` only if the condition is met before the end of

the interval specified by `timeout`. For example, if `timeout` is `600`, then the function returns `1` only if an out-of-memory error occurs before 600 seconds pass. The function returns `-1` only if the end of the timeout interval occurs *before* the database completes the requested operation.

Table 5-1 Return Values for POPULATE_WAIT

Constant	Value	Description
POPULATE_TIMEOUT	-1	The function timed out while waiting for population to complete. Existing population jobs continue running in the background after `-1` is returned. Reissuing `POPULATE_TIMEOUT` after `-1` is returned reinitiates population; segments that are already populated are not dropped.
POPULATE_SUCCESS	0	All objects that met the `priority` criteria were populated to the specified `percentage` of completion.
POPULATE_OUT_OF_MEMORY	1	The In-Memory pool had insufficient memory to populate the objects that met the `priority` criteria to the specified `percentage` of completion.
POPULATE_NO_INMEMORY_OBJECTS	2	No `INMEMORY` objects met the specified `priority` criteria.
POPULATE_INMEMORY_SIZE_ZERO	3	The In-Memory column store is not enabled.

See Also:

- "Setting a Timeout Using the POPULATE_WAIT Function: Example"
- *Oracle Database PL/SQL Packages and Types Reference* to learn about the `DBMS_INMEMORY_ADMIN` package

5.1.4 Population Using DBMS_INMEMORY.REPOPULATE

`DBMS_INMEMORY.REPOPULATE` forces repopulation of a table, partition, or subpartition that is currently populated in the IM column store.

If you use this procedure on an In-Memory object that is *not* currently populated, then this procedure is functionally equivalent to `DBMS_INMEMORY.POPULATE`.

See Also:

- "Refreshing an In-Memory External Table Using the REPOPULATE Procedure: Example"
- *Oracle Database PL/SQL Packages and Types Reference* to learn about the `DBMS_INMEMORY` package
- *Oracle Database SQL Language Reference* to learn more about the `FULL` hint

5.2 Forcing Initial Population of an In-Memory Object

You can force population of an object using a full table scan, the `POPULATE` procedure, the `POPULATE_WAIT` function, or the `REPOPULATE` procedure.

Assumptions

This task assumes the following:

- The IM column store is enabled.
- You want to enable a table for In-Memory population.
- You want to force the immediate population of the table into the IM column store.

To force population of an INMEMORY table:

1. In SQL*Plus or SQL Developer, log in to the database as a user with administrative privileges.
2. Apply the `INMEMORY` attribute to the table.

   For example, enable `sh.customers` for IM population as follows:

   ```
   ALTER TABLE sh.customers INMEMORY;
   ```

   In the preceding example, the default priority is `NONE`.
3. Optionally, check the population status by querying `V$IM_SEGMENTS`.

   For example, use the following statement (sample output included):

   ```
   COL OWNER FORMAT a10;
   COL NAME FORMAT a25;
   COL STATUS FORMAT a10;

   SELECT OWNER, SEGMENT_NAME NAME,
          POPULATE_STATUS STATUS
   FROM   V$IM_SEGMENTS
   WHERE  SEGMENT_NAME = 'CUSTOMERS';

   no rows selected
   ```

The preceding output shows that the object is not yet populated in the IM column store.

4. Initiate population using one of the following techniques:

   - Query all rows in the table using `SELECT` with the `FULL` hint.

     For example, issue the following statement:

     ```
     SELECT /*+ FULL(customers) NO_PARALLEL(customers) */ COUNT(*) FROM
     sh.customers;
     ```

   - Execute the `DBMS_INMEMORY.POPULATE` procedure.

     For example, execute this procedure for `sh.customers` as follows:

     ```
     EXEC DBMS_INMEMORY.POPULATE('SH', 'CUSTOMERS');
     ```

   - Execute the `DBMS_INMEMORY.REPOPULATE` procedure.

     For unpopulated tables, this procedure is functionally equivalent to `POPULATE`. For example, execute this procedure for `sh.customers` as follows:

     ```
     EXEC DBMS_INMEMORY.REPOPULATE('SH', 'CUSTOMERS');
     ```

   - Execute the `DBMS_INMEMORY_ADMIN.POPULATE_WAIT` function.

     The following code example populates all `INMEMORY` objects, regardless of `PRIORITY` setting. The example specifies that the function should wait until all objects are 100% populated, and it should time out with an error if success is not achieved within 1800 seconds (30 minutes).

     ```
     VARIABLE b_pop_status NUMBER
     BEGIN
        SELECT DBMS_INMEMORY_ADMIN.POPULATE_WAIT(
                  priority   => 'NONE' ,
                  percentage => 100    ,
                  timeout    => 1800   ,
                  force      => FALSE  )
          INTO :b_pop_status
        FROM   dual;
     END;
     /
     PRINT b_pop_status
     ```

5. Optionally, to check the population status, query `V$IM_SEGMENTS`.

   For example, use the following statement (sample output included):

   ```
   SELECT OWNER, SEGMENT_NAME NAME,
          POPULATE_STATUS STATUS
   FROM   V$IM_SEGMENTS
   WHERE  SEGMENT_NAME = 'CUSTOMERS';

   OWN NAME       STATUS
   --- ---------- --------------------
   SH  CUSTOMERS  COMPLETED
   ```

The table is now populated in the IM column store.

> **See Also:**
>
> - *Oracle Database Reference* to learn about `V$IM_SEGMENTS`
> - *Oracle Database PL/SQL Packages and Types Reference* to learn about `DBMS_INMEMORY` subprograms

5.3 Populating In-Memory Tables Manually: Examples

The following examples illustrate how to populate In-Memory tables manually.

> **See Also:**
>
> "Enabling and Disabling Tables for the IM Column Store"

5.3.1 Populating an In-Memory Table Using a Full Table Scan: Example

This example using a full table scan to populate the `sh.sales` table into the IM column store.

Assume that you are logged in to the database as an administrator, and that you have issued the following DDL statement to add the `INMEMORY` clause to the `sh.sales` table:

```
ALTER TABLE sh.sales INMEMORY;
```

The preceding statement uses the defaults for the `INMEMORY` clause: `MEMCOMPRESS FOR QUERY` and `PRIORITY NONE`. Because `PRIORITY` is `NONE`, the database will not automatically populate the table into the IM column store. The following query confirms that the `sh.sales` table is not currently populated:

```
COL OWNER FORMAT a10;
COL NAME FORMAT a25;
COL STATUS FORMAT a10;

SELECT OWNER, SEGMENT_NAME NAME,
       POPULATE_STATUS STATUS
FROM   V$IM_SEGMENTS
WHERE  SEGMENT_NAME = 'SALES';

no rows selected
```

The following query uses the `FULL` hint to force a full table scan of `sales`, thereby initiating population (sample output included):

```
SELECT /*+ FULL(sales) NO_PARALLEL(sales) */ COUNT(*) FROM sh.sales;

  COUNT(*)
----------
    918843
```

The following query shows the population status of `sales` (sample output included):

```
SET PAGESIZE 50000
COL OWNER FORMAT a3
COL NAME FORMAT a10
COL STATUS FORMAT a20

SELECT OWNER, SEGMENT_NAME NAME,
       POPULATE_STATUS STATUS
FROM   V$IM_SEGMENTS
WHERE  SEGMENT_NAME = 'SALES';

OWN NAME       STATUS
--- ---------- --------------------
SH  SALES      COMPLETED
SH  SALES      COMPLETED
SH  SALES      COMPLETED
SH  SALES      COMPLETED
SH  SALES      COMPLETED
SH  SALES      COMPLETED
SH  SALES      COMPLETED
SH  SALES      COMPLETED
SH  SALES      COMPLETED
SH  SALES      COMPLETED
SH  SALES      COMPLETED
SH  SALES      COMPLETED
SH  SALES      COMPLETED
SH  SALES      COMPLETED
SH  SALES      COMPLETED
SH  SALES      COMPLETED

16 rows selected.
```

The following query calculates the compression ratio of the table. The query assumes that the tables are not further compressed on disk.

```
COL OWNER FORMAT a5
COL SEGMENT_NAME FORMAT a5
SET PAGESIZE 50000

SELECT v.OWNER, v.SEGMENT_NAME, v.BYTES ORIG_SIZE,
       v.INMEMORY_SIZE IN_MEM_SIZE,
       ROUND(v.BYTES / v.INMEMORY_SIZE, 2) COMP_RATIO
FROM   V$IM_SEGMENTS v
```

```
ORDER BY 4;

OWNER SEGME  ORIG_SIZE IN_MEM_SIZE COMP_RATIO
----- ----- ---------- ----------- ----------
SH    SALES     851968     1310720        .65
SH    SALES     835584     1310720        .64
SH    SALES     925696     1310720        .71
SH    SALES     958464     1310720        .73
SH    SALES     950272     1310720        .73
SH    SALES     786432     1310720         .6
SH    SALES     876544     1310720        .67
SH    SALES     753664     1310720        .58
SH    SALES    1081344     1310720        .83
SH    SALES     901120     1310720        .69
SH    SALES     925696     1310720        .71
SH    SALES     933888     1310720        .71
SH    SALES     843776     1310720        .64
SH    SALES     999424     1310720        .76
SH    SALES     581632     1507328        .39
SH    SALES     696320     1507328        .46

16 rows selected.
```

5.3.2 Populating a Table Using the POPULATE Procedure: Example

This example uses `DBMS_INMEMORY.POPULATE` to initiate population of the `sh.customers` table into the IM column store.

Assume that you are logged in to the database as an administrator, and that you have issued the following DDL statement to add the `INMEMORY` clause to the `sh.customers` table:

```
ALTER TABLE sh.customers INMEMORY;
```

The preceding statement uses the defaults for the `INMEMORY` clause: `MEMCOMPRESS FOR QUERY` and `PRIORITY NONE`. Because `PRIORITY` is `NONE`, the database will not automatically populate the table into the IM column store. The following query confirms that the `sh.customers` table is not currently populated:

```
COL OWNER FORMAT a10;
COL NAME FORMAT a25;
COL STATUS FORMAT a10;

SELECT OWNER, SEGMENT_NAME NAME,
       POPULATE_STATUS STATUS
FROM   V$IM_SEGMENTS
WHERE  SEGMENT_NAME = 'CUSTOMERS';

no rows selected
```

The following PL/SQL code uses the `POPULATE` procedure to initiative population:

```
EXEC DBMS_INMEMORY.POPULATE('SH', 'CUSTOMERS');
```

The following query shows the population status of `customers` (sample output included):

```
SELECT OWNER, SEGMENT_NAME NAME,
       POPULATE_STATUS STATUS
FROM   V$IM_SEGMENTS
WHERE  SEGMENT_NAME = 'CUSTOMERS';

OWN NAME       STATUS
--- ---------- --------------------
SH  CUSTOMERS  COMPLETED
```

5.3.3 Setting a Timeout Using the POPULATE_WAIT Function: Example

This example uses `DBMS_INMEMORY_ADMIN.POPULATE_WAIT` to populate all In-Memory tables, regardless of priority setting.

Example 5-1 Specifying a Timeout Interval for In-Memory Population

In this example, the database contains a number of In-Memory tables with a variety of priority settings. Your goal is to populate *every* In-Memory table to 100% completion in a restricted database session, and then disable the restricted session so that the application can be guaranteed of querying only the In-Memory representations.

Assume that the database is shut down. In SQL*Plus, you connect to an idle instance as `SYSDBA`, and then execute the following command (sample output included):

```
SQL> STARTUP RESTRICT
ORACLE instance started.

Total System Global Area 1157624280 bytes
Fixed Size                  8839640 bytes
Variable Size             754974720 bytes
Database Buffers           16777216 bytes
Redo Buffers                7933952 bytes
In-Memory Area            369098752 bytes
Database mounted.
Database opened.
```

The database is open, but is accessible only to administrative users. You execute the following statements in SQL*Plus (sample output shown in bold):

```
VARIABLE b_pop_status NUMBER

SELECT DBMS_INMEMORY_ADMIN.POPULATE_WAIT(
         priority   => 'NONE' ,
         percentage => 100    ,
         timeout    => 300    )
  INTO b_pop_status
FROM   DUAL;
```

```
PRINT b_pop_status
-1
```

After 5 minutes, the function returns the number `-1`. This code indicates that the function timed out while waiting for population to complete. 5 minutes is not long enough to populate all `INMEMORY` tables. You re-execute the `SELECT` statement, specifying a 30-minute timeout:

```
SELECT DBMS_INMEMORY_ADMIN.POPULATE_WAIT(
         priority   => 'NONE' ,
         percentage => 100    ,
         timeout    => 1800   )
  INTO b_pop_status
FROM   DUAL;

PRINT b_pop_status
0
```

After 8 minutes, the function returns the number `0`. This code indicates that all tables are completely populated. You now disable the restricted session so that the application can start query In-Memory objects with full confidence that only In-Memory representations will be accessed:

```
ALTER SYSTEM DISABLE RESTRICTED SESSION;
```

5.3.4 Populating an In-Memory External Table Using DBMS_INMEMORY.POPULATE: Example

This example populates an external table that has the `INMEMORY` option.

This example assumes that you created the external table `sh.admin_ext_sales` with the `INMEMORY` option using the `sh_sales.csv` file.

The `admin_ext_sales` table is not yet populated. Starting in Oracle Database 19c, a full table scan populates an external table just as it populates a standard table. However, in this scenario, you choose to initiate population by using the `DBMS_INMEMORY` package.

> **Note:**
>
> Sessions that query In-Memory external tables must have the initialization parameter `QUERY_REWRITE_INTEGRITY` set to `stale_tolerated`.
>
> It is important to keep in mind that if an external table is modified, then the results from the IM column store are undefined. Results are also undefined if a partition is altered (by dropping or adding values). This may lead to differences in results between IM and non-IM based scans. You can run `DBMS_INMEMORY.REPOPULATE` to refresh the IM store so that it is resynchronized with the table data.

To populate the table, you perform the following steps:

1. Log in as user `sh`.
2. Set `QUERY_REWRITE_INTEGRITY` in the database session:

```
ALTER SESSION SET QUERY_REWRITE_INTEGRITY=stale_tolerated;
```

3. Execute the following PL/SQL program:

```
EXEC DBMS_INMEMORY.POPULATE('SH', 'ADMIN_EXT_SALES');
```

4. Check the population status by querying `V$IM_SEGMENTS`:

```
COL OWNER FORMAT a3
COL NAME FORMAT a15
COL STATUS FORMAT a9

SELECT OWNER, SEGMENT_NAME NAME,
       POPULATE_STATUS STATUS
FROM   V$IM_SEGMENTS;

OWN NAME            STATUS
--- --------------- ---------
SH  ADMIN_EXT_SALES COMPLETED
```

The preceding output shows that the `admin_ext_sales` table has been populated

See Also:

- "In-Memory Views"
- *Oracle Database PL/SQL Packages and Types Reference* to learn more about `DBMS_INMEMORY.POPULATE`
- *Oracle Database Reference* to learn about the `QUERY_REWRITE_INTEGRITY` initialization parameter

5.3.5 Refreshing an In-Memory External Table Using the REPOPULATE Procedure: Example

This example repopulates a currently populated In-Memory external table.

This example assumes that you used the comma-delimited flat file `/tmp/data/sh_sales.csv` to create the `sh.admin_ext_sales` table with the `INMEMORY` option, and that you populated this table into the IM column store.

Assume you add a record to `/tmp/data/sh_sales.csv` as follows:

```
echo "148,8787,23-NOV-01,2,999,1,23.43" >> /tmp/data/sh_sales.csv
```

Unlike internal tables, external tables do not use the automatic repopulation mechanism. To refresh external tables, you must use one of the following techniques:

- Call the `DBMS_INMEMORY.REPOPULATE` procedure
- Specify the table as `NO INMEMORY`, specify it as `INMEMORY`, and then perform a full table scan

The following example uses the `REPOPULATE` procedure to force the IM column store to refresh `admin_ext_sales`:

```
EXEC DBMS_INMEMORY.REPOPULATE('SH', 'ADMIN_EXT_SALES');
```

See Also:

- "Creating an In-Memory External Table: Example"
- "Controls for Repopulation of the IM Column Store"
- *Oracle Database PL/SQL Packages and Types Reference* to learn more about `DBMS_INMEMORY.REPOPULATE`

6

Automating Management of In-Memory Objects

You can use Automatic Data Optimization (ADO) and Automatic In-Memory to manage objects in the IM column store dynamically.

ADO uses Heat Map, which tracks data access patterns for blocks and segments. ADO and Heat Map are a part of Information Lifecycle Management (ILM), which is a set of processes and policies for managing data from creation to archival or deletion. This chapter assumes that you are familiar with ILM, ADO, and Heat Map.

This chapter contains the following topics:

> **See Also:**
>
> *Oracle Database VLDB and Partitioning Guide* for background about ILM, ADO, and Heat Map

6.1 Enabling ADO for the IM Column Store

Automatic Data Optimization (ADO) creates policies, and automates actions based on those policies, to implement your ILM strategy.

This section contains the following topics:

6.1.1 About ADO Policies and the IM Column Store

ADO manages the IM column store through **ADO policies**. You can only create an ADO policy with an `INMEMORY` clause at the segment level.

The database treats an ADO policy like an attribute of an object. ADO policies are at the database level, not the instance level. Oracle Database supports the following types of ADO policies for Database In-Memory:

- `INMEMORY` policy

  This policy marks objects with the `INMEMORY` attribute, enabling them for population in the IM column store. When set at the table level, the `INMEMORY` attribute applies only to internal partitions; therefore, external partitions of a hybrid partitioned table are not managed by the policy.
- Recompression policy

  This policy changes the compression level on an `INMEMORY` object.
- `NO INMEMORY` policy

This policy removes an object from the IM column store and removes its `INMEMORY` attribute.

Oracle Database supports the following criteria to determine when policies apply:

- A specified number of days since the object was modified

  Obtain this value from the column `SEGMENT_WRITE_TIME` in the `DBA_HEAT_MAP_SEGMENT` view.

- A specified number of days since the object was accessed

  This value is the greater value in the columns `SEGMENT_WRITE_TIME`, `FULL_SCAN`, and `LOOKUP_SCAN` in the `DBA_HEAT_MAP_SEGMENT` view.

- A specified number of days since the object was created

  Obtain this value from the `CREATED` column in `DBA_OBJECTS`.

- A user-defined function returns a Boolean value

> **See Also:**
>
> - *Oracle Database Reference* to learn about the `DBA_HEAT_MAP_SEGMENT` view
> - *Oracle Database SQL Language Reference* to learn about the `INMEMORY` clause

6.1.2 Purpose of ADO and the IM Column Store

ADO manages the IM column store as a new data tier.

You can create policies to evict objects from the IM column store when they are being accessed less often, and populate objects when they are being accessed more often and would improve query performance. ADO manages the IM column store using Heat Map statistics.

Purpose of INMEMORY Policies

In many databases, segments undergo heavy modification after creation. To maximize performance, ADO can populate these segments in the IM column store when write activity subsides. For example, if you add a partition to a table every day, then you can create a policy that populates the `sales_2016_d100` partition one day after creation:

```
ALTER TABLE sales MODIFY PARTITION sales_2016_d100
  ILM ADD POLICY SET INMEMORY MEMCOMPRESS FOR QUERY
    PRIORITY HIGH
  AFTER 1 DAYS OF CREATION
```

Similarly, you may know that write activity on a table subsides two months after creation, and want to populate this object when this time condition is met:

```
ALTER TABLE 2016_ski_sales
  ILM ADD POLICY SET INMEMORY MEMCOMPRESS FOR QUERY
```

```
    PRIORITY CRITICAL
  AFTER 60 DAYS OF CREATION
```

The preceding policy causes all existing and new partitions of the `2016_ski_sales` table to inherit the policy. When the segment qualifies for the policy, the database marks every partition independently with the specified `INMEMORY` clause. If the segment already has an `INMEMORY` policy, then the database ignores the new policy.

Purpose of Recompression Policies

You may want to compress data in the IM column store based on access patterns. For example, you may want to change a segment from DML compression to query compression 2 days after DML activity on the segment has ceased:

```
ALTER TABLE lineorders
  ILM ADD POLICY MODIFY INMEMORY MEMCOMPRESS FOR QUERY HIGH
  AFTER 2 DAYS OF NO MODIFICATION
```

If the object is not populated in IM column store, then this policy only changes the compression attribute. If the object is populated in the IM column store, then ADO repopulates the object using the new compression level. The database ignores the policy if the segment does not already have the `INMEMORY` attribute.

Purpose of NO INMEMORY Policies

To optimize space in the IM column store, you may want to evict inactive segments using a `NO INMEMORY` policy. This policy is also useful for preventing population of inactive segments by infrequent queries. For example, if reports on a specific sales partition run frequently during the year, but typically not every week, then you may want to may want to evict this partition after a week of no access:

```
ALTER TABLE sales MODIFY PARTITION sales_2015_q1
  ILM ADD POLICY NO INMEMORY AFTER 7 DAYS OF NO ACCESS;
```

If the sales table for 1998 is rarely queried, then you may want to evict after 1 day of no access:

```
ALTER TABLE sales_1998
  ILM ADD POLICY NO INMEMORY AFTER 1 DAYS OF NO ACCESS;
```

Queries of an evicted segment are never blocked. The database can always access the data through the traditional buffer cache mechanism.

> **See Also:**
>
> - "Row Data in the Database Buffer Cache"
> - *Oracle Database VLDB and Partitioning Guide*

6.1.3 How ADO Works with Columnar Data

From the ADO perspective, the IM column store is another storage tier.

6.1.3.1 How Heat Map Works

When enabled, Heat Map automatically discovers data access patterns. ADO uses the Heat Map data to implement user-defined policies at the database level.

Heat Map automatically tracks usage information at the row and segment levels. At the row level, Heat Map tracks data modification times, and then aggregates these times to the block level. At the segment level, Heat Map tracks times for modifications, full table scans, and index lookups.

When an IM column store is enabled, Heat Map tracks access patterns for columnar data. For example, the `sales` table may be "hot," whereas the `locations` table may be "cold." The ADO algorithms work the same way for columnar data as for row-based data.

The database periodically writes Heat Map data to the data dictionary. The database exposes Heat Map data in data dictionary views. For example, to obtain the read and write time for In-Memory objects, query the `ALL_HEAT_MAP_SEGMENT` view.

> **See Also:**
>
> - *Oracle Database VLDB and Partitioning Guide* to learn more about Heat Map
> - *Oracle Database Reference* to learn about the `ALL_HEAT_MAP_SEGMENT` view

6.1.3.2 How Policy Evaluation Works

The policy evaluation for IM column store policies uses the same infrastructure as the evaluation of other ADO policies. The database evaluates and executes policies automatically during the maintenance window.

The database evaluates policies using Heat Map statistics, which are stored in the data dictionary. Setting `INMEMORY` attributes is mostly a metadata operation, and thus minimally affects performance.

ADO uses the Job Scheduler to perform population. The In-Memory Coordinator Process (IMCO) performs the population.

Related Topics

- In-Memory Coordinator Process (IMCO)
  The In-Memory Coordinator Process (IMCO) manages many tasks for the IM column store. Its primary task is to initiate background population and repopulation of columnar data.

6.1.4 Controls for ADO and the IM Column Store

Enable Heat Map using the `HEAT_MAP` initialization parameter. Control ADO through a SQL and PL/SQL interface.

ILM Clause in DDL Statements

No new SQL statements are required to create In-Memory policies, but the ILM clause has new options. The following table describes SQL options for ADO and the IM column store.

> **Note:**
>
> The `INMEMORY` attribute only applies to internal partitions of a hybrid partitioned table.

Table 6-1 ILM Clause for ADO and the IM Column Store

Clause	Description	Examples

`SET INMEMORY`	Sets the `INMEMORY` attribute for the object	`ALTER TABLE sh.sales`
		`  ILM ADD POLICY`
		`  SET INMEMORY`
		`  MEMCOMPRESS FOR QUERY LOW`
		`  PRIORITY HIGH`
		`  SEGMENT`
		`  AFTER 30 DAYS OF CREATION;`

`MODIFY INMEMORY`	Modifies the compression level for the object	`ALTER TABLE sh.customers`
		`  ILM ADD POLICY`
		`  MODIFY INMEMORY`
		`  MEMCOMPRESS FOR QUERY HIGH`
		`  PRIORITY CRITICAL`
		`  SEGMENT`
		`  AFTER 30 DAYS OF CREATION;`

`NO INMEMORY`	Sets the `NO INMEMORY` attribute for the object	`ALTER TABLE sh.products`
		`  ILM ADD POLICY`
		`  NO INMEMORY`
		`  SEGMENT`
		`  AFTER 30 DAYS OF CREATION;`

> **See Also:**
>
> *Oracle Database SQL Language Reference* to learn more about the *ilm_policy_clause* of `CREATE TABLE`

Initialization Parameters

The following table describes initialization parameters that are relevant for ADO and the IM column store.

Table 6-2 Initialization Parameters for ADO and the IM Column Store

Initialization Parameter	Description
`COMPATIBLE`	Specifies the release with which the database must maintain compatibility. For ADO to manage the IM column store, set this parameter to `12.2.0` or higher.
`HEAT_MAP`	Enables both the Heat Map and ADO features. For ADO to manage the IM column store, set this parameter to `ON`.
`INMEMORY_SIZE`	Enables the IM column store. This parameter must be set to a nonzero value.

PL/SQL Packages

The following table describes PL/SQL packages that are relevant for ADO and the IM column store.

Table 6-3 PL/SQL Packages for ADO and the IM Column Store

Package	Description
`DBMS_HEAT_MAP`	Displays detailed Heat Map data at the tablespace, segment, object, extent, and block levels.
`DBMS_ILM`	Implements ILM strategies using ADO policies.
`DBMS_ILM_ADMIN`	Customizes ADO policy execution.

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* to learn more about `DBMS_HEAT_MAP`, `DBMS_ILM`, and `DBMS_ILM_ADMIN` packages

V$ and Data Dictionary Views

The following table describes views that are relevant for ADO and the IM column store.

Table 6-4 Views for ADO and the IM Column Store

View	Description
DBA_HEAT_MAP_SEG_HISTOGRAM	Displays segment access information for all segments visible to the user.
DBA_HEAT_MAP_SEGMENT	Displays the latest segment access time for all segments visible to the user.
DBA_HEATMAP_TOP_OBJECTS	Displays heat map information for the top 10000 objects by default.
DBA_HEATMAP_TOP_TABLESPACES	Displays heat map information for the top 10000 tablespaces.
DBA_ILMDATAMOVEMENTPOLICIES	Displays information specific to data movement-related attributes of an ADO policy in a database. The action_type column describes policies related to the IM column store. Possible values are COMPRESSION, STORAGE, EVICT, and ANNOTATE.
V$HEAT_MAP_SEGMENT	Displays real-time segment access information.

See Also:

Oracle Database Reference to learn more about views

6.1.5 Creating an ADO Policy for the IM Column Store

You can use ADO policies to set, modify, or remove the INMEMORY clause for objects based on Heat Map statistics.

To create an ADO IM column store policy, specify the ILM ADD POLICY clause in an ALTER TABLE statement, followed by one of the following subclauses:

- SET INMEMORY ... SEGMENT

  This option is useful when you want to mark segments with the INMEMORY attribute only when DML activity subsides.

- MODIFY INMEMORY ... MEMCOMPRESS ... SEGMENT

  Storing data uncompressed or at the MEMCOMPRESS FOR DML level is appropriate when it is frequently modified. The alternative compression levels are more suited for queries. If the activity on a segment transitions from mostly writes to mostly reads, then you can use the MODIFY clause to apply a different compression method.

- NO INMEMORY ... SEGMENT

  This option is useful when access to a segment decreases with time (it becomes “cold”), and to prevent population of this segment as a result of random access.

Prerequisites

Before you can use an ADO IM column store policy, you must meet the following prerequisites:

- Enable the IM column store for the database by setting the `INMEMORY_SIZE` initialization parameter to a nonzero value and restarting the database.
- The `HEAT_MAP` initialization parameter must be set to `ON`.

  Heat Map provides data access tracking at the segment-level and data modification tracking at the segment and row level.
- The `COMPATIBLE` initialization parameter must be set to `12.2.0` or higher.

To create an ADO policy:

1. In SQL*Plus or SQL Developer, log in to the database as a user with the necessary privileges.
2. Use an `ALTER TABLE` statement with the `ILM ADD POLICY ... INMEMORY` clause.

Example 6-1 Creating an Eviction Policy

In this example, you create a policy specifying that `oe.order_items` table is evicted from the IM column store if it has not been accessed in three days. An ADO IM column store policy must be a segment-level policy.

```
ALTER TABLE oe.order_items ILM ADD POLICY
   NO INMEMORY SEGMENT
   AFTER 3 DAYS OF NO ACCESS;
```

Example 6-2 Executing an ILM Policy Using DBMS_ILM

You can also evaluate and executes policies manually. Thus, you can programmatically decide when you want an object compressed or tiered. The following example manually executes an ADO task for `sh.sales`:

```
DECLARE
  v_executonid NUMBER;
BEGIN
  DBMS_ILM.EXECUTE_ILM ( owner           => 'SH',
                         object_name     => 'SALES',
                         execution_mode  =>
DBMS_ILM.ILM_EXECUTION_OFFLINE,
                         task_id         => v_executionid);
END;
/
```

See Also:

- *Oracle Database SQL Language Reference* for `CREATE TABLE` syntax and semantics
- *Oracle Database PL/SQL Packages and Types Reference* to learn more about the `DBMS_ILM` package

6.2 Configuring Automatic In-Memory

Automatic In-Memory uses access tracking and column statistics to manage objects in the IM column store.

If the IM column store is full, and if other more frequently accessed segments would benefit from population in the IM column store, then the IM column store evicts inactive segments. If the IM column store is configured to hold all `INMEMORY` segments, however, then Automatic In-Memory takes no action.

> **Note:**
>
> If the `INMEMORY_FORCE` initialization parameter is set to `BASE_LEVEL`, then Automatic In-Memory is disabled even if `INMEMORY_AUTOMATIC_LEVEL` is set. Even if tables have a compression level of `AUTO`, Automatic In-Memory background operations do not run.

This chapter contains the following topics:

6.2.1 Purpose of Automatic In-Memory

To ensure that the working data set is always populated, Automatic In-Memory automatically evicts cold (infrequently accessed) segments.

The IM column store only removes a populated segment if it is dropped or moved, the `INMEMORY` option is removed, or an IM ADO policy acts on it. Memory pressure occurs when the size of the `INMEMORY` data set exceeds the available memory for the IM column store, and some populated segments become inactive. For optimal performance, the IM column store should contain the most frequently queried segments, known as the working data set.

Typically, the working data set changes with time for many applications. Therefore, optimization requires regular monitoring and manual intervention from the DBA to evict IM store elements or create ADO IM policies. Both tasks require a good understanding of the workload.

By automatically evicting cold segments, Automatic In-Memory provides the following benefits:

- Improved performance

  By mitigating memory pressure through eviction of cold segments, Automatic In-Memory improves the performance of workloads because the working data set resides in the IM column store.

- Ease of management

  Management of the IM column store for mitigating memory pressure by eviction of cold segments involves significant user intervention. Automatic In-Memory addresses these issues with minimal user intervention.

6.2.2 How Automatic In-Memory Works

The unit of data eviction is an `INMEMORY` segment.

An `INMEMORY` segment is only eligible for eviction when its priority is `NONE`. The basic process is as follows:

1. A population job fails, which means that IM column store space has been exhausted.
2. The database uses internal statistics of eligible populated segments to define the set of objects to evict. The statistics are similar to those used by Heat Map, but do not require Heat Map to be enabled.
3. For each segment in the set, the database checks whether an ADO policy is enabled for the segment:
   - If an enabled policy requires that the segment remain populated, then the ADO policy overrides Automatic In-Memory. The database does nothing.
   - If no policy prevents eviction, then Automatic In-Memory submits tasks to evict the segments.
4. W*nnn* processes evict any segments that pass the preceding checks, freeing up space in the IM column store.

   The `INMEMORY` attribute is retained for evicted segments.

For example, a nightly batch job loads a `sales` partition (with priority `NONE`), and then queries the partition to trigger population. Because the IM column store is almost at its maximum capacity, only half the rows of the partition are populated. The failure to completely populate the new partition triggers Automatic In-Memory, which evicts a cold segment. A subsequent on-demand populate job for the new partition completely populates the new `sales` partition.

> **See Also:**
>
> "Space Management Worker Processes (Wnnn)"

6.2.3 User Interface for Automatic In-Memory

Enable and disable Automatic In-Memory using the initialization parameter `INMEMORY_AUTOMATIC_LEVEL`.

Initialization Parameters

The system-level initialization parameter `INMEMORY_AUTOMATIC_LEVEL` has the following possible values:

- `OFF` (default)

  This option disables Automatic In-Memory, returning the IM column store to its Oracle Database 12c Release 2 (12.2.0.1) behavior.

- `LOW`

When under memory pressure, the database evicts cold segments from the IM column store.

- `MEDIUM`

  This level includes an additional optimization that ensures that any hot segment that was not populated because of memory pressure is populated first.

> **Note:**
>
> Automatic In-Memory does not require the `HEAT_MAP` initialization parameter to be enabled.

Oracle recommends that you provision enough memory for the working data set to fit in the IM column store. As a rule of thumb for sizing the additional Automatic In-Memory shared pool requirement, multiply 5 KB by the number of `INMEMORY` segments of SGA memory. For example, if 10,000 segments have the `INMEMORY` attribute, then reserve 50 MB of the shared pool for Automatic In-Memory.

> **See Also:**
>
> *Oracle Database Reference* to learn more about `INMEMORY_AUTOMATIC_LEVEL`

DBMS_INMEMORY_ADMIN

Use the `DBMS_INMEMORY_ADMIN` package to control the time window in which Automatic In-Memory considers statistics. For example, you can specify that Automatic In-Memory only consider the past month or the past week.

Use the `DBMS_INMEMORY_ADMIN.AIM_SET_PARAMETER` procedure to set the `AIM_STATWINDOW_DAYS` constant. For example, to set the sliding statistics window to 7 days, execute the following program:

```
EXEC DBMS_INMEMORY_ADMIN.AIM_SET_PARAMETER
( DBMS_INMEMORY_ADMIN.AIM_STATWINDOW_DAYS, 7 );
```

The default value for `AIM_STATWINDOW_DAYS` is `31` days.

The corresponding `DBMS_INMEMORY_ADMIN.AIM_GET_PARAMETER` procedure obtains the current setting for `AIM_STATWINDOW_DAYS`.

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* to learn more about `DBMS_INMEMORY_ADMIN.AIM_SET_PARAMETER` and `DBMS_INMEMORY_ADMIN.AIM_GET_PARAMETER`

V$ VIEWS

The V$IM_ADOTASKS and DBA_INMEMORY_AIMTASKS views enable you to track decisions made by Automatic In-Memory tasks. The V$IM_ADOTASKDETAILS and DBA_INMEMORY_AIMTASKDETAILS views describe details relating to the tasks.

> **See Also:**
>
> *Oracle Database Reference* to learn more about V$IM_ADOTASKS

6.2.4 Controlling Automatic In-Memory

Use the INMEMORY_AUTOMATIC_LEVEL initialization parameter to control Automatic In-Memory.

By default, Automatic In-Memory is set to OFF. Enable it by setting INMEMORY_AUTOMATIC_LEVEL to either MEDIUM or LOW.

Prerequisites

To set this parameter with ALTER SYSTEM, you must have the ALTER SYSTEM privilege.

To change the INMEMORY_AUTOMATIC_LEVEL setting:

1. In SQL*Plus or SQL Developer, log in to the database as a user with the necessary privileges.
2. Specify INMEMORY_AUTOMATIC_LEVEL using the ALTER SYSTEM statement.

   The following example disables Automatic In-Memory:

   ```
   ALTER SYSTEM SET INMEMORY_AUTOMATIC_LEVEL = 'OFF' SCOPE=BOTH;
   ```

> **See Also:**
>
> *Oracle Database Reference* to learn more about INMEMORY_AUTOMATIC_LEVEL

6.2.5 Setting the Time Interval for Automatic In-Memory

Use the DBMS_INMEMORY_ADMIN package to set the time interval for the usage statistics checked by Automatic In-Memory.

By default, Automatic In-Memory checks usage statistics for the past 31 days. You can change the current setting by supplying the AIM_STATWINDOW_DAYS parameter to DBMS_INMEMORY_ADMIN.AIM_SET_PARAMETER.

Prerequisites

You must have administrator privileges to execute the `DBMS_INMEMORY_ADMIN.AIM_SET_PARAMETER` and `DBMS_INMEMORY_ADMIN.AIM_GET_PARAMETER` procedures.

Assumptions

You want to set the interval to 7 days.

To change the Automatic In-Memory interval setting:

1. In SQL*Plus or SQL Developer, log in to the database as a user with administrative privileges.
2. Optionally, check the current setting of the `aim_statwindow_days` parameter.

   The following example calls the `DBMS_INMEMORY_ADMIN.AIM_GET_PARAMETER` procedure:

   ```
   VARIABLE b_interval NUMBER

   BEGIN
     DBMS_INMEMORY_ADMIN.AIM_GET_PARAMETER(
       DBMS_INMEMORY_ADMIN.AIM_STATWINDOW_DAYS, :b_interval);
   END;
   /

   PRINT b_interval

   B_INTERVAL
   ----------------------------
   31
   ```

3. Change the `aim_statwindow_days` setting with the `DBMS_INMEMORY_ADMIN.AIM_SET_PARAMETER` procedure.

   The following code changes the setting to 7 days:

   ```
   BEGIN
     DBMS_INMEMORY_ADMIN.AIM_SET_PARAMETER(
       DBMS_INMEMORY_ADMIN.AIM_STATWINDOW_DAYS, 7);
   END;
   /
   ```

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* to learn more about the `DBMS_INMEMORY_ADMIN.AIM_SET_PARAMETER` and `DBMS_INMEMORY_ADMIN.AIM_GET_PARAMETER` procedures

Part III
Optimizing In-Memory Queries

This Part explains how to optimize queries using In-Memory Expressions, join groups, and In-Memory aggregation. It also explains how the IM column store repopulates modified data.

7

Optimizing Queries with In-Memory Expressions

In the context of the IM column store, an **expression** is a combination of one or more values, operators, and SQL or PL/SQL functions (`DETERMINISTIC` only) that resolve to a value.

The Expression Statistics Store (ESS) automatically tracks the results of frequently evaluated ("hot") expressions. You can use the `DBMS_INMEMORY_ADMIN` package to capture hot expressions and populate them as hidden virtual columns, or drop some or all of them.

7.1 About IM Expressions

By default, the `DBMS_INMEMORY_ADMIN.IME_CAPTURE_EXPRESSIONS` procedure identifies and populates "hot" expressions, called **In-Memory Expressions (IM expressions)**.

An IM expression is materialized as a hidden virtual column in a heap-organized table, but is accessed in the same way as a non-virtual column. To store the materialized expressions, the IM column store uses special compression formats such as fixed-width vectors and dictionary encoding with fixed-width codes.

> **Note:**
>
> IM expressions are not supported for external tables.

Oracle Database automatically identifies the expressions that are candidates for population in the IM column store. In `DBA_IM_EXPRESSIONS.COLUMN_NAME`, IM expression columns have the prefix `SYS_IME`. You cannot create `SYS_IME` columns directly. For example, consider the following query, which specifies two expressions, aliased `weekly_sal` and `ann_comp`:

```
SELECT employee_id, last_name, salary, commission_pct,
       ROUND(salary*12/52,2) as "weekly_sal",
       12*(salary*NVL(commission_pct,0)+salary) as "ann_comp"
FROM   employees
ORDER BY ann_comp;
```

The arithmetical expressions `ROUND(salary*12/52,2)` and `12*(salary*NVL(commission_pct,0)+salary)` are computationally intensive and frequently accessed, which makes them candidates for hidden IM expression columns.

The `DBMS_INMEMORY_ADMIN` package is the primary interface for managing IM expressions:

- To induce the database to identify IM expressions and add them to their respective tables during the next repopulation, use `IME_CAPTURE_EXPRESSIONS`.
- To force immediate population of IM expressions, use `IME_POPULATE_EXPRESSIONS`.

- To drop `SYS_IME` columns, use `DBMS_INMEMORY_ADMIN.IME_DROP_ALL_EXPRESSIONS` or `DBMS_INMEMORY.IME_DROP_EXPRESSIONS`.

> **See Also:**
>
> - "In-Memory External Tables"
> - *Oracle Database SQL Language Reference* to learn more about expressions
> - *Oracle Database PL/SQL Packages and Types Reference* to learn more about `DBMS_INMEMORY_ADMIN`
> - *Oracle Database Reference* to learn more about the `DBA_IM_EXPRESSIONS` view

7.1.1 Purpose of IM Expressions

IM expressions speed queries of large data sets by precomputing computationally intensive expressions.

IM expressions especially benefit frequently executed table joins, projections, and predicate evaluations. The primary advantages of IM expressions are as follows:

- A query does not need to recalculate the expressions every time. If the IM column store does not populate the expression results, then the database must compute them for every row, which can be resource intensive. The database incurs the CPU overhead during the population.
- The materialization of IM expressions enables the database to take advantage of performance-enhancing features such as SIMD vector processing and IMCU pruning.
- The database, rather than the user, tracks which expressions are most active within a user-specified expression capture window.

IM expressions and materialized views address the same problem: how to avoid repeatedly evaluating expressions. However, IM expressions have advantages over materialized views:

- IM expressions can capture data that is not persistently stored.

  For example, the IM column store can automatically cache internal computations based on expressions in the query.
- To be used effectively, a materialized view must have all columns listed in the query, or the query must join the view and the base tables. In contrast, any query containing an IM expression can benefit.
- The database identifies and creates IM expressions automatically, unlike materialized views, which are user-created objects.

> **See Also:**
>
> - "In-Memory Storage Indexes"
> - "Expression Statistics Store (ESS)"
> - *Oracle Database Data Warehousing Guide* to learn more about materialized views

7.1.2 How IM Expressions Work

To identify expressions as candidates for IM expressions, the database queries the ESS. The optimizer uses the ESS to maintain statistics about expression evaluation for a heap-organized table.

7.1.2.1 IM Expressions Infrastructure

The IM expressions infrastructure is responsible for computing and populating the results of IM expressions, IM virtual columns, and any other useful internal computations in the IM column store. These optimizations primarily benefit analytic queries.

Populated results can include function evaluations on columns used in project, scan, or join expressions. The IM column store can automatically cache internal computations based on the expressions evaluated by the SQL runtime engine during query evaluation.

Virtual Columns

Besides populating an IM expression, the IM column store can populate an In-Memory virtual column. The underlying mechanism is the same: an IM expression is a virtual column. However, IM virtual columns are user-created and exposed, whereas IM expressions are database-created and hidden.

> **See Also:**
>
> "Enabling and Disabling Columns for In-Memory Tables"

Static Expressions: Binary JSON Columns

The IM expressions infrastructure supports both dynamic expressions (IM expressions and virtual columns) and static expressions.

Starting in Oracle Database 12c Release 2 (12.2), the IM column store supports OSON, which is a binary JSON format that performs better than row-oriented JSON text storage. Queries access the actual JSON data, but use optimized virtual columns to speed access.

The database uses the IM expression infrastructure to load an efficient binary representation of JSON text columns as virtual columns. If the `MAX_STRING_SIZE` initialization parameter is set to `EXTENDED` for the `VARCHAR2` data type, then the IM column store can store OSON virtual columns up to 32 KB.

Oracle Database supports multiple JSON functions: `JSON_TABLE`, `JSON_VALUE`, and `JSON_EXISTS`. The `INMEMORY_EXPRESSIONS_USAGE` initialization parameter controls the behavior of both dynamic expressions and static expressions.

See Also:

- *Oracle Database JSON Developer's Guide* to learn more about using JSON with Database In-Memory
- *Oracle Database Reference* to learn about `INMEMORY_EXPRESSIONS_USAGE` , `ALL_JSON_COLUMNS` , and `MAX_STRING_SIZE`

7.1.2.2 Capture of IM Expressions

When you invoke the `IME_CAPTURE_EXPRESSIONS` procedure, the database queries the ESS, and identifies the 20 most frequently accessed ("hottest") expressions in the specified time range.

The time range is either a user-specified time window, the past 24 hours, or since database creation. The database only considers expressions on tables that are at least partially populated in the IM column store.

7.1.2.2.1 Expression Capture Interval

The **expression capture interval** is the period in which the database evaluates expressions for possible capture.

Starting in Oracle Database 18c, the `snapshot` parameter of the `DBMS_INMEMORY_ADMIN.IME_CAPTURE_EXPRESSIONS` procedure accepts the following values that define the expression capture interval:

- `CUMULATIVE`

  The database considers all expression statistics since the creation of the database.

- `CURRENT`

  The database considers only expression statistics from the past 24 hours.

- `WINDOW`

  The database adds hidden virtual columns for expressions tracked in the most recent user-specified expression capture window. This window opens with the manual invocation of the `IME_OPEN_CAPTURE_WINDOW` procedure, and then closes with the manual invocation of the `IME_CLOSE_CAPTURE_WINDOW` procedure.

  If the capture window is currently open, then the database considers all expressions tracked in the current window up until this point, and then materializes the hottest expressions. To list the expressions that have been tracked in the current window, query `DBA_EXPRESSION_STATISTICS` with `SNAPSHOT = 'WINDOW'`.

A user-defined time interval (`snapshot='WINDOW'`) is useful for ensuring that only expressions occurring within this window are considered for materialization. This mechanism is especially useful when a short interval is representative of the entire

workload. For example, during the trading window, a brokerage firm can gather the set of expressions, and materialize them in the IM column store to speed-up future query processing for the entire workload.

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* to learn more about `IME_OPEN_CAPTURE_WINDOW`, `IME_CLOSE_CAPTURE_WINDOW`, and `IME_CAPTURE_EXPRESSIONS`

7.1.2.2.2 Hidden SYS_IME Virtual Columns

During capture, the database adds the 20 hottest expressions to their respective tables as hidden `SYS_IME` virtual columns and applies the default `INMEMORY` column compression clause.

If `SYS_IME` columns added during a previous invocation no longer appear in the latest expression list, then their attribute changes to `NO INMEMORY`.

Figure 7-1 Defining Hidden SYS_IME Virtual Columns

The maximum number of `SYS_IME` columns for a table is 50, regardless of whether the attribute is `INMEMORY`. After a table reaches the 50-expression limit, the database does not add new `SYS_IME` columns. To permit new expressions, you must drop `SYS_IME` columns with the `DBMS_INMEMORY.IME_DROP_EXPRESSIONS` or `DBMS_INMEMORY_ADMIN.IME_DROP_ALL_EXPRESSIONS` procedures.

Both `SYS_IME` virtual columns and user-defined virtual columns count toward the 1000-column limit for a table. For example, if a table contains 980 non-virtual (on-disk) columns, then you can add only 20 virtual columns.

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* to learn more about `DBMS_INMEMORY_ADMIN`

7.1.2.3 How the ESS Works

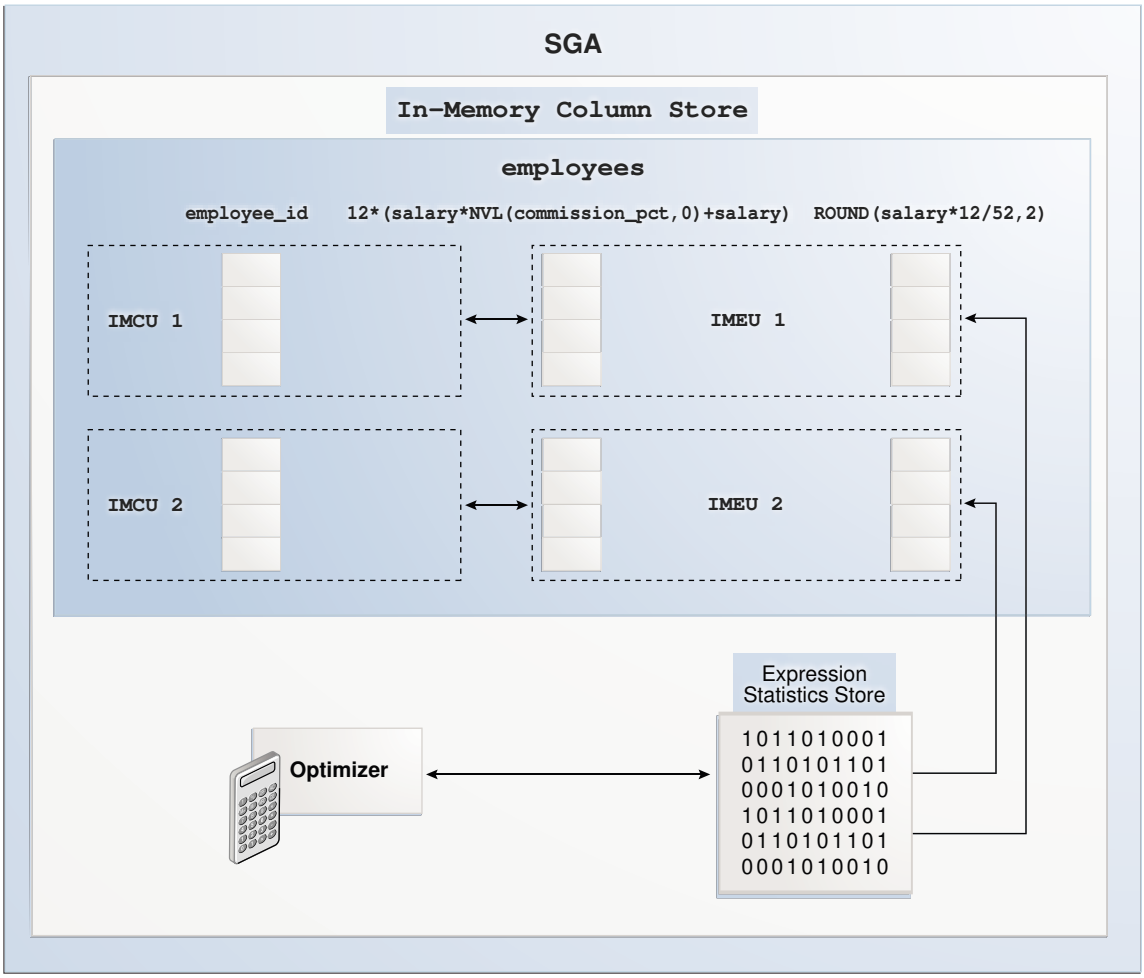
The ESS is a repository maintained by the optimizer to store statistics about expression evaluation.

For each table, the ESS maintains expression statistics such as frequency of execution and cost of evaluation. When evaluating predicates, Oracle Database tracks and provides run-time feedback on evaluation counts and the dynamic costs of expressions. Based on the ESS statistics, the database may decide that queries would perform better if a specific expression were an IM expression.

> **Note:**
>
> Expressions cached in the ESS for a specific table only involve columns of this table. This rule is especially important when Oracle Database identifies deterministic PL/SQL functions as candidates for IM expressions.

Figure 7-2 ESS and IM Expressions

In this graphic, the ESS has determined two commonly used expressions on the `employees` table: `ROUND(salary*12/52,2)` and `12*(salary*NVL(commission_pct,0)+salary)`. When the database populates `employees` in the IM column store, two IMCUs store the columnar data. Each IMCU is associated with its only IMEU, which contains the derived values for the two commonly used expressions for the rows in that IMCU.



Not every expression is a candidate for an IM expression. The database only considers expressions that will be accessed frequently. Because IM expressions are implemented as hidden virtual columns, they must also meet the restrictions for virtual columns.

Although the IM column store is a client of the ESS, the ESS is independent of Database In-Memory features. Other clients can also use ESS statistics, including the optimizer itself.

> **See Also:**
>
> - "Expression Statistics Store (ESS)"
> - *Oracle Database Administrator's Guide* to learn more about virtual columns

7.1.2.4 How the Database Populates IM Expressions

Under the direction of In-Memory Coordinator Process (IMCO), Space Management Worker Processes (W*nnn*) load IM expressions into IMEUs automatically.

The database augments every In-Memory Compression Unit (IMCU) population or repopulation task with information about which virtual columns, either user-defined or IM expressions, to populate. The decision depends on the settings of the `INMEMORY_EXPRESSION_USAGE` and `INMEMORY_VIRTUAL_COLUMNS` initialization parameters.

> **Note:**
>
> The `DBMS_INMEMORY.IME_CAPTURE_EXPRESSIONS` procedure adds automatically detected expressions as hidden virtual columns.

The W*nnn* processes create the IMCUs. To create the IMEUs, the processes perform the following additional steps:

1. Create the expression values
2. Convert the values into columnar format, and compress them into In-Memory Expression Units (IMEUs)
3. Link each IMEU to its associated IMCU

> **Note:**
>
> As the number of expressions to store in IMEUs goes up, the worker processes may consume slightly more CPU to compute the expression values. This overhead may increase population time.

> **See Also:**
>
> - "Expression Statistics Store (ESS)"
> - "Space Management Worker Processes (Wnnn)"
> - "In-Memory Expression Units (IMEUs)"
> - *Oracle Database PL/SQL Packages and Types Reference* to learn more about the `DBMS_INMEMORY.IME_CAPTURE_EXPRESSIONS` procedure

7.1.2.5 How IMEUs Relate to IMCUs

For any row, the physical columns reside in the IMCU, and the virtual columns reside in an associated IMEU. The IMEU is read-only and columnar, just like the IMCU.

Because IMEUs are logical extensions of IMCUs created for a particular `INMEMORY` segment, by default they inherit the `INMEMORY` clause, and Oracle Real Applications Cluster (Oracle RAC) properties such as `DISTRIBUTE` and `DUPLICATE`. An IMEU is associated with one and only one IMCU. The database manages IMEUs as separate structures, making them easier to add and drop.

> **Note:**
>
> The IMEUs also contain user-created IM virtual columns.

If the source data changes, then the database changes the derived data in the IM expression during repopulation. For example, if a transaction updates 100 salary values in a table, then the Space Management Worker Processes (W*nnn*) automatically update all IM expression values that are derived from these 100 changed values. The database repopulates an IMCU and its associated IMEUs together rather than first repopulating all IMCUs and then repopulating all IMEUs. IMEUs remain available for queries during IMCU repopulation.

7.1.3 User Interfaces for IM Expressions

The `DBMS_INMEMORY_ADMIN` package, `DBMS_INMEMORY` package, and `INMEMORY_EXPRESSIONS_USAGE` initialization parameter control the behavior of IM expressions.

7.1.3.1 INMEMORY_EXPRESSIONS_USAGE

The `INMEMORY_EXPRESSIONS_USAGE` initialization parameter determines which type of IM expression is populated. The `INMEMORY_VIRTUAL_COLUMNS` initialization parameter controls the population of normal (non-hidden) virtual columns.

When the IM column store is enabled (`INMEMORY_SIZE` is nonzero), `INMEMORY_EXPRESSIONS_USAGE` controls the type of IM expression that the database populates. The `INMEMORY_EXPRESSIONS_USAGE` initialization parameter has the following options:

- ENABLE

  The database populates both static and dynamic IM expressions into the IM column store. Setting this value increases the In-Memory footprint for some tables. This is the default.

- STATIC_ONLY

  A static configuration enables the IM column store to cache OSON (binary JSON) columns, which are marked with an IS_JSON check constraint. Internally, an OSON column is a hidden virtual column named SYS_IME_OSON.

- DYNAMIC_ONLY

  The database only populates frequently used or “hot” expressions that have been added to the table as SYS_IME hidden virtual columns. Setting this value increases the In-Memory footprint for some tables.

- DISABLE

  The database does not populate any IM expressions, whether static or dynamic, into the IM column store.

Changing the value of INMEMORY_EXPRESSIONS_USAGE does not have an immediate effect on the IM expressions currently populated in the IM column store. For example, if you change INMEMORY_EXPRESSIONS_USAGE from DYNAMIC_ONLY to DISABLE, then the database does not immediately remove the stored IM expressions. Rather, the next repopulation excludes the disabled IM expressions, which effectively removes them.

See Also:

- *Oracle Database JSON Developer's Guide* to learn more about using JSON with Database In-Memory
- *Oracle Database Reference* to learn about INMEMORY_EXPRESSIONS_USAGE and ALL_JSON_COLUMNS

7.1.3.2 DBMS_INMEMORY_ADMIN and DBMS_INMEMORY

To manage IM expressions, use the DBMS_INMEMORY_ADMIN and DBMS_INMEMORY packages.

PL/SQL Procedures for Managing IM Expressions

Package	Procedure	Description
DBMS_INMEMORY_ADMIN	IME_OPEN_CAPTURE_WINDOW	This procedure signals the beginning of an expression capture window.
DBMS_INMEMORY_ADMIN	IME_CLOSE_CAPTURE_WINDOW	This procedure signals the end of the current expression capture window.

Package	Procedure	Description
DBMS_INMEMORY_ADMIN	IME_GET_CAPTURE_STATE	This procedure returns the current capture state of the expression capture window and the timestamp of the most recent modification.
DBMS_INMEMORY_ADMIN	IME_CAPTURE_EXPRESSIONS	This procedure captures the 20 most frequently accessed (“hottest”) expressions in the database in the specified time range.
DBMS_INMEMORY_ADMIN	IME_POPULATE_EXPRESSIONS	This procedure forces the population of IM expressions captured in the latest invocation of the IME_CAPTURE_EXPRESSIONS procedure.
DBMS_INMEMORY_ADMIN	IME_DROP_ALL_EXPRESSIONS	This procedure drops all SYS_IME virtual columns in the database.
DBMS_INMEMORY	IME_DROP_EXPRESSIONS	This procedure drops a specified set of SYS_IME virtual columns from a table.

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn more about the DBMS_INMEMORY and DBMS_INMEMORY_ADMIN packages

7.1.4 Basic Tasks for IM Expressions

The default setting of INMEMORY_EXPRESSIONS_USAGE enables the database to use both dynamic and static IM expressions. You must use DBMS_INMEMORY_ADMIN to populate the expressions in the IM column store.

Typically, you perform IM expression tasks in the following sequence:

1. Optionally, change the type of IM expression that the database can use.

   See "Configuring IM Expression Usage".

2. Capture and populate IM expressions.

   See "Capturing and Populating IM Expressions".

3. Optionally, drop some or all IM expressions.

   See "Dropping IM Expressions".

7.2 Configuring IM Expression Usage

Optionally, use `INMEMORY_EXPRESSIONS_USAGE` to choose which types of IM expressions are eligible for population, or to disable population of all IM expressions.

Prerequisites

To enable the database to use IM expressions, the following conditions must be true:

- The `INMEMORY_SIZE` initialization parameter is set to a non-zero value.
- The value for the initialization parameter `COMPATIBLE` is set to 12.2.0 or higher.

> **Note:**
>
> In an Oracle Real Applications Cluster (RAC) database, the `INMEMORY_EXPRESSIONS_USAGE` initialization parameter does not require the same value on every database instance. Each IMCU independently lists virtual columns. Each IMCU could materialize different expressions based on the initialization parameter value and the virtual columns that existed when the IMCU was populated or repopulated.

To configure IM expression usage:

1. Log in to the database as a user with the appropriate privileges.
2. To configure IM expression usage, use an `ALTER SYSTEM` statement to set `INMEMORY_EXPRESSIONS_USAGE` to any of the following values:
   - `ENABLE` (default) — Enable dynamic and static IM expressions
   - `STATIC_ONLY` — Enable only static IM expressions
   - `DYNAMIC_ONLY` — Enable only dynamic IM expressions
   - `DISABLE` — Disable all IM expressions

Example 7-1 Disabling IM Expressions

The following statement disables storage of IM expressions in the IM column store:

```
ALTER SYSTEM SET INMEMORY_EXPRESSIONS_USAGE='DISABLE' SCOPE=BOTH;
```

> **See Also:**
>
> *Oracle Database Reference* to learn more about `INMEMORY_EXPRESSIONS_USAGE`

7.3 Capturing and Populating IM Expressions

The `IME_CAPTURE_EXPRESSIONS` procedure captures the 20 most frequently accessed ("hottest") expressions in the database in the specified time interval. The

IME_POPULATE_EXPRESSIONS procedure forces the population of expressions captured in the latest invocation of DBMS_INMEMORY_ADMIN.IME_CAPTURE_EXPRESSIONS.

Whenever you invoke the IME_CAPTURE_EXPRESSIONS procedure, the database queries the Expression Statistics Store (ESS), and considers only expressions on heap-organized tables that are at least partially populated in the IM column store. The database adds the 20 hottest expressions to their respective tables as hidden virtual columns, prefixed with the string SYS_IME, and applies the default INMEMORY column compression clause. If SYS_IME columns added during a previous invocation are no longer in the latest top 20 list, then their attribute changes to NO INMEMORY.

If you do not invoke IME_POPULATE_EXPRESSIONS, then the database gradually repopulates SYS_IME columns when their parent IMCUs are repopulated. If a table is not repopulated, then the database does not repopulate new SYS_IME columns captured by the IME_CAPTURE_EXPRESSIONS procedure. IME_POPULATE_EXPRESSIONS solves this problem by forcing repopulation.

Internally, the IME_POPULATE_EXPRESSIONS procedure invokes DBMS_INMEMORY.REPOPULATE for all tables that have SYS_IME columns with the INMEMORY attribute. To populate SYS_IME columns in a specified subset of tables, use DBMS_INMEMORY.REPOPULATE instead of DBMS_INMEMORY_ADMIN.IME_POPULATE_EXPRESSIONS.

Prerequisites

To enable the database to capture IM expressions, the following conditions must be true:

- The INMEMORY_EXPRESSIONS_USAGE initialization parameter must be set to a value other than DISABLE.
- The INMEMORY_SIZE initialization parameter is set to a non-zero value.
- The value for the initialization parameter COMPATIBLE must be set to 12.2.0 or higher.
- To specify an expression capture window of arbitrary length (rather than the predetermined time span of 24 hours or forever), you must open the window with DBMS_INMEMORY_ADMIN.IME_OPEN_CAPTURE_WINDOW and close it with IME_CLOSE_CAPTURE_WINDOW. The window is global across all instances in an Oracle RAC database.

To capture and populate IM expressions:

1. In SQL*Plus or SQL Developer, log in to the database as a user with the necessary privileges.
2. If you want to specify the length of the window (rather than a predetermined time span), then execute DBMS_INMEMORY_ADMIN.IME_OPEN_CAPTURE_WINDOW as follows:

```
EXEC DBMS_INMEMORY_ADMIN.IME_OPEN_CAPTURE_WINDOW();
```

> **Note:**
>
> To check whether any expression capture windows are currently open, execute `DBMS_INMEMORY_ADMIN.IME_GET_CAPTURE_STATE`.

3. If you opened a window with `IME_OPEN_CAPTURE_WINDOW` in the preceding step, then close it with `IME_CLOSE_CAPTURE_WINDOW` as follows:

```
EXEC DBMS_INMEMORY_ADMIN.IME_CLOSE_CAPTURE_WINDOW();
```

4. Execute `DBMS_INMEMORY_ADMIN.IME_CAPTURE_EXPRESSIONS`, setting the `interval` parameter to any of the following values:

   - `CUMULATIVE`

     The database considers all expression statistics since the creation of the database.

   - `CURRENT`

     The database considers only expression statistics from the past 24 hours.

   - `WINDOW`

     The database adds hidden virtual columns for expressions tracked in the most recent user-specified expression capture window. This window opens with the manual invocation of the `IME_OPEN_CAPTURE_WINDOW` procedure, and then closes with the manual invocation of the `IME_CLOSE_CAPTURE_WINDOW` procedure.

     If the capture window is currently open, then the database considers all expressions tracked in the current window up until this point, and then materializes the hottest expressions. To list the expressions that have been tracked in the current window, query `DBA_EXPRESSION_STATISTICS` with `SNAPSHOT = 'WINDOW'`.

5. Optionally, to force immediate population of all captured IM expressions, execute `DBMS_INMEMORY_ADMIN.IME_POPULATE_EXPRESSIONS` as follows:

```
EXEC DBMS_INMEMORY_ADMIN.IME_POPULATE_EXPRESSIONS();
```

Example 7-2 Capturing Expressions in a User-Defined Window

This example demonstrates use of the `WINDOW` capture mode. Your goal is to open and close an expression capture window, and then capture all expressions that the database tracked during this window. You perform the following steps:

1. Open an expression capture window, generate expressions, and then close the window:

```
EXEC DBMS_INMEMORY_ADMIN.IME_OPEN_CAPTURE_WINDOW();
-- Generate expressions for the database to track
EXEC DBMS_INMEMORY_ADMIN.IME_CLOSE_CAPTURE_WINDOW();
```

2. Query `DBA_EXPRESSION_STATICS` (sample output included):

```
COL OWNER FORMAT A6
COL TABLE_NAME FORMAT A9
COL COUNT FORMAT 99999
COL CREATED FORMAT A10
COL EXPRESSION_TEXT FORMAT A29
```

```
SELECT OWNER, TABLE_NAME, EVALUATION_COUNT AS COUNT,
       CREATED, EXPRESSION_TEXT
FROM   DBA_EXPRESSION_STATISTICS
WHERE  SNAPSHOT = 'WINDOW'
AND    OWNER = 'SH';

OWNER  TABLE_NAM  COUNT CREATED    EXPRESSION_TEXT
------ --------- ------ ---------- -------------------------
SH     SALES       4702 09-OCT-17  "QUANTITY_SOLD"
SH     SALES       4702 09-OCT-17  "QUANTITY_SOLD"*"AMOUNT_SOLD"
SH     SALES       4702 09-OCT-17  "PROD_ID"
SH     SALES       4702 09-OCT-17  "CUST_ID"
SH     SALES       4702 09-OCT-17  "CHANNEL_ID"
SH     SALES       4702 09-OCT-17  "AMOUNT_SOLD"
```

The preceding query shows both the columns tracked in the ESS and the expressions captured during the window for queries in the `sh` schema. During the most recent window, the database captured one expression: `QUANTITY_SOLD*AMOUNT_SOLD`.

3. Use `IME_CAPTURE_EXPRESSIONS` to make the database consider all expressions in the current window for materialization:

```
EXEC DBMS_INMEMORY_ADMIN.IME_CAPTURE_EXPRESSIONS('WINDOW');
```

4. Query `DBA_IM_EXPRESSIONS` (sample output included):

```
COL OWNER FORMAT a6
COL TABLE_NAME FORMAT a9
COL COLUMN_NAME FORMAT a25
SET LONG 50
SET LINESIZE 150

SELECT OWNER, TABLE_NAME, COLUMN_NAME, SQL_EXPRESSION
FROM   DBA_IM_EXPRESSIONS;

OWNER  TABLE_NAM COLUMN_NAME               SQL_EXPRESSION
------ --------- -------------------------
----------------------------
SH     SALES     SYS_IME000100000025201B
"QUANTITY_SOLD"*"AMOUNT_SOLD"
```

The preceding output shows all virtual columns that were added to the table and marked `INMEMORY` as part of the latest `IME_CAPTURE_EXPRESSIONS` invocation. The database gradually populates the captured expressions into the IM column store when it repopulates different IMCUs of the table.

5. Execute the following procedure to explicitly force a population of all captured IM expressions:

```
EXEC DBMS_INMEMORY_ADMIN.IME_POPULATE_EXPRESSIONS();
```

Note that you can populate IM expressions from a specific table by executing the `DBMS_INMEMORY.REPOPULATE` procedure with the `force` parameter set to `TRUE`.

Example 7-3 Determining the State of an Expression Capture Window

This example opens an expression capture window, and then determines its capture state.

```
EXEC DBMS_INMEMORY_ADMIN.IME_OPEN_CAPTURE_WINDOW();

VARIABLE b_state VARCHAR2(25)
VARIABLE b_time  VARCHAR2(10)
EXECUTE DBMS_INMEMORY_ADMIN.IME_GET_CAPTURE_STATE(:b_state, :b_time)
PRINT b_state b_time
```

The following sample output indicates that an expression capture window is currently open:

```
B_STATE
--------------------------------------------------
OPEN

B_TIME
--------------------------------------------------
09-OCT-17
```

Example 7-4 Capturing the Top 20 IM Expressions in the Past 24 Hours

This example captures IM expressions using only the statistics gathered during the last day, and then forces immediate population:

```
EXEC DBMS_INMEMORY_ADMIN.IME_CAPTURE_EXPRESSIONS('CURRENT');
EXEC DBMS_INMEMORY_ADMIN.IME_POPULATE_EXPRESSIONS();
```

The following query of `DBA_IM_EXPRESSIONS` shows all IME virtual columns that are marked `INMEMORY` (sample output provided):

```
COL OWNER FORMAT a6
COL TABLE_NAME FORMAT a9
COL COLUMN_NAME FORMAT a25
SET LONG 50
SET LINESIZE 150

SELECT OWNER, TABLE_NAME,
       COLUMN_NAME, SQL_EXPRESSION
FROM   DBA_IM_EXPRESSIONS;

OWNER  TABLE_NAM COLUMN_NAME              SQL_EXPRESSION
------ --------- ------------------------ --------------------------------------------
HR     EMPLOYEES SYS_IME00010000001746FD  12*("SALARY"*NVL("COMMISSION_PCT",0)+"SALARY")
HR     EMPLOYEES SYS_IME00010000001746FE  ROUND("SALARY"*12/52,2)
```

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* to learn more about `IME_CAPTURE_EXPRESSIONS` and `IME_POPULATE_EXPRESSIONS`
- *Oracle Database Reference* to learn more about the `DBA_IM_EXPRESSIONS` view

7.4 Dropping IM Expressions

The `DBMS_INMEMORY_ADMIN.IME_DROP_ALL_EXPRESSIONS` procedure drops all `SYS_IME` expression virtual columns in the database. The `DBMS_INMEMORY.IME_DROP_EXPRESSIONS` procedure drops a specified set of `SYS_IME` virtual columns from a table.

Typical reasons for dropping `SYS_IME` columns are space and performance. The maximum number of `SYS_IME` columns for a table, regardless of whether the attribute is `INMEMORY` or `NO INMEMORY`, is 50. After the 50-expression limit is reached for a table, the database will not add new `SYS_IME` columns. To make space for new expressions, you must manually drop `SYS_IME` columns with the `DBMS_INMEMORY.IME_DROP_EXPRESSIONS` or `DBMS_INMEMORY_ADMIN.IME_DROP_ALL_EXPRESSIONS` procedures.

The `IME_DROP_ALL_EXPRESSIONS` procedure drops all `SYS_IME` columns from all tables, regardless of whether they have the `INMEMORY` attribute. In effect, the procedure acts as a database-wide reset button.

Using `IME_DROP_ALL_EXPRESSIONS` triggers a drop of all IMEUs *and* IMCUs for segments that have `SYS_IME` columns. For example, if 50 populated tables have one `SYS_IME` column each, then `IME_DROP_ALL_EXPRESSIONS` removes all 50 tables from the IM column store. To populate these segments again, you must use the `DBMS_INMEMORY.POPULATE` procedure or perform a full table scan.

Prerequisites

To drop IM expressions, the following conditions must be true:

- The `INMEMORY_EXPRESSIONS_USAGE` initialization parameter is set to a value other than `DISABLE`.
- The `INMEMORY_SIZE` initialization parameter is set to a nonzero value.
- The `COMPATIBLE` initialization parameter is set to 12.2.0 or higher.

To drop IM expressions:

1. In SQL*Plus or SQL Developer, log in to the database as a user with the necessary privileges.
2. Execute either `DBMS_INMEMORY_ADMIN.IME_DROP_ALL_EXPRESSIONS` or `DBMS_INMEMORY.IME_DROP_EXPRESSIONS`.

   If you execute `IME_DROP_EXPRESSIONS`, then specify the following parameters:

   - `schema_name` — The name of the schema that contains the In-Memory table

- `table_name` — The name of the In-Memory table
- `column_name` — The name of the `SYS_IME` column. By default, this value is null, which specifies all `SYS_IME` columns in this table.

Example 7-5 Dropping All IM Expressions in a Table

This example drops all IM expressions in the `hr.employees` table:

```
EXEC DBMS_INMEMORY.IME_DROP_EXPRESSIONS('hr', 'employees');
```

See Also:

- *Oracle Database Reference* to learn more about the `INMEMORY_EXPRESSIONS_USAGE` initialization parameter
- *Oracle Database PL/SQL Packages and Types Reference* to learn more about `DBMS_INMEMORY.IME_DROP_EXPRESSIONS` and `DBMS_INMEMORY_ADMIN.IME_DROP_ALL_EXPRESSIONS`
- *Oracle Database Reference* to learn more about the `DBA_IM_EXPRESSIONS` data dictionary view

8

Optimizing Joins with Join Groups 6.3

A **join group** is a user-created dictionary object that lists one or more columns that can be meaningfully joined.

This chapter contains the following topics:

8.1 About In-Memory Joins

Joins are an integral part of data warehousing workloads. The IM column store enhances the performance of joins when the tables being joined are stored in memory.

Because of faster scan and join processing, complex multitable joins and simple joins that use Bloom filters benefit from the IM column store. In a data warehousing environment, the most frequently-used joins involved a fact table and one or more dimension tables.

The following joins run faster when the tables are populated in the IM column store:

- Joins that are amenable to using Bloom filters
- Joins of multiple small dimension tables with one fact table
- Joins between two tables that have a primary key-foreign key relationship

8.2 About Join Groups

When the IM column store is enabled, the database can use join groups to optimize joins of tables populated in the IM column store.

A join group is a set of columns on which a set of tables is frequently joined. The column set contains one or more columns, with a maximum of 255 columns. The table set includes one or more internal tables. External tables are not supported.

The columns in the join group can be in the same or different tables. For example, if the `sales` and `times` tables frequently join on the `time_id` column, then you might create a join group for `(times(time_id), sales(time_id))`. If the `employees` table often joins to itself on the `employee_id` column, then a join group could be `(employees(employee_id))`.

> **Note:**
>
> The same column cannot be a member of multiple join groups.

When you create a join group, the database invalidates the current In-Memory contents of the tables referenced in the join group. Subsequent repopulation causes the database to re-encode the IMCUs of the tables with the common dictionary. For this reason, Oracle recommends that you first create the join group, and then populate the tables.

Create join groups using the `CREATE INMEMORY JOIN GROUP` statement. To add columns to or drop columns from a join group, use an `ALTER INMEMORY JOIN GROUP` statement. Drop a join group using the `DROP INMEMORY JOIN GROUP` statement.

> **Note:**
>
> In Oracle Active Data Guard, a standby database ignores join group definitions. A standby database does not use common dictionaries, and executes queries as if join groups did not exist.

Example 8-1 Creating a Join Group

This example creates a join group named `deptid_jg` that includes the `department_id` column in the `hr.employees` and `hr.departments` tables.

```
CREATE INMEMORY JOIN GROUP deptid_jg
(hr.employees(department_id),hr.departments(department_id));
```

8.3 Purpose of Join Groups

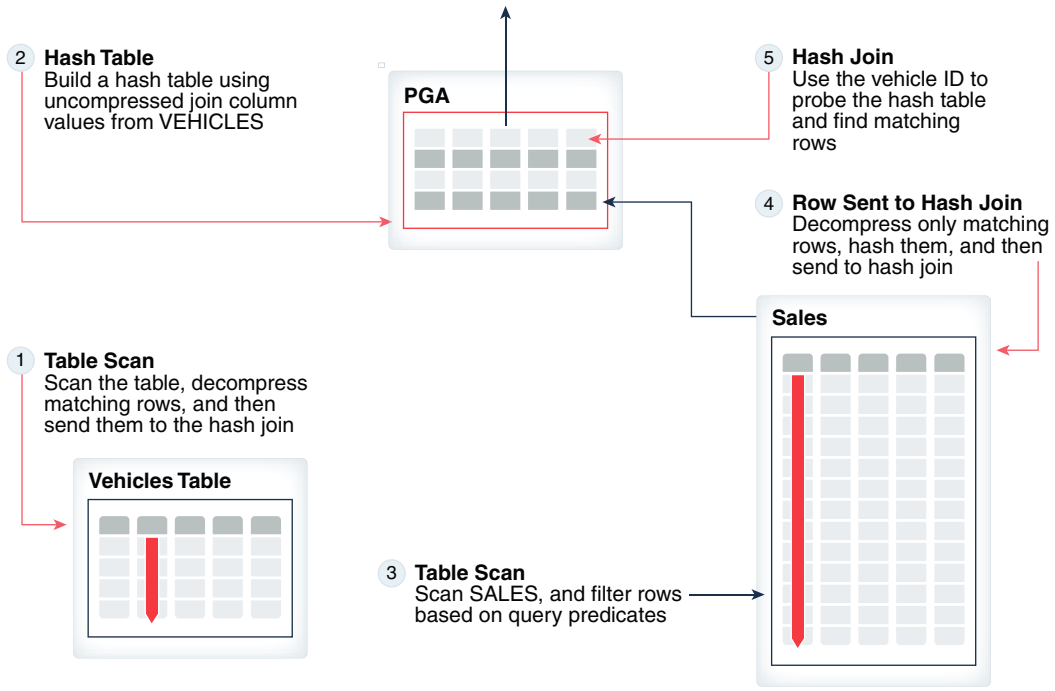
In certain queries, join groups eliminate the performance overhead of decompressing and hashing column values.

Without join groups, if the optimizer uses a hash join but cannot use a Bloom filter, or if the Bloom filter does not filter rows effectively, then the database must decompress IMCUs and use an expensive hash join. To illustrate the problem, assume a star schema has a `sales` fact table and a `vehicles` dimension table. The following query joins these tables, but does not filter the output, which means that the database cannot use a Bloom filter:

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

The following figure illustrates how the database joins the two data sets.

Figure 8-1 Hash Join without Join Group



The database performs a hash join as follows:

1. Scans the `vehicles` table, decompresses the rows that satisfy the predicate (in this case, all rows satisfy the predicate because no filters exist), and sends the rows to the hash join
2. Builds a hash table in the PGA based on the decompressed rows
3. Scans the `sales` table and applies any filters (in this case, the query does not specify filters)
4. Processes matching rows from the IMCUs and then sends the rows to the join

   When the hash join can consume row sets from the probe side (in this case, the `sales` table), the row sets sent by the table scan are in compressed form. Depending on whether the local dictionary or join group is leveraged to find matching rows from the build side, the hash join either decompresses the rows or leaves them uncompressed.
5. Probes the hash table using the join column, which in this case is the vehicle name

If a join group exists on the `v.name` and `s.name` columns, then the database can make the preceding steps more efficient, eliminating the decompression and filtering overhead. The benefits of join groups are:

- The database operates on compressed data.
- The database avoids hashing on the join key and probing the hash table, which requires comparing the join keys of the probe rows and hashed rows.

  When a join group exists, the database stores codes for each join column value in a common dictionary. The database builds a join group array using dictionary codes. Every array element points to a build-side row stored in the hash area (typically, PGA memory). During the probe, each probe row has a code associated with the join key. The database

uses this code to search the array to determine whether a pointer exists in the array element. If a pointer exists, then there is a match; otherwise, there is no match.

- The dictionary codes are dense and have a fixed length, which makes them space efficient.
- Optimizing a query with a join group is sometimes possible when it is not possible to use a Bloom filter.

8.4 How Join Groups Work

In a join group, the database compresses all columns in the join group using the same common dictionary.

8.4.1 How a Join Group Uses a Common Dictionary

A **common dictionary** is a table-level, instance-specific set of dictionary codes.

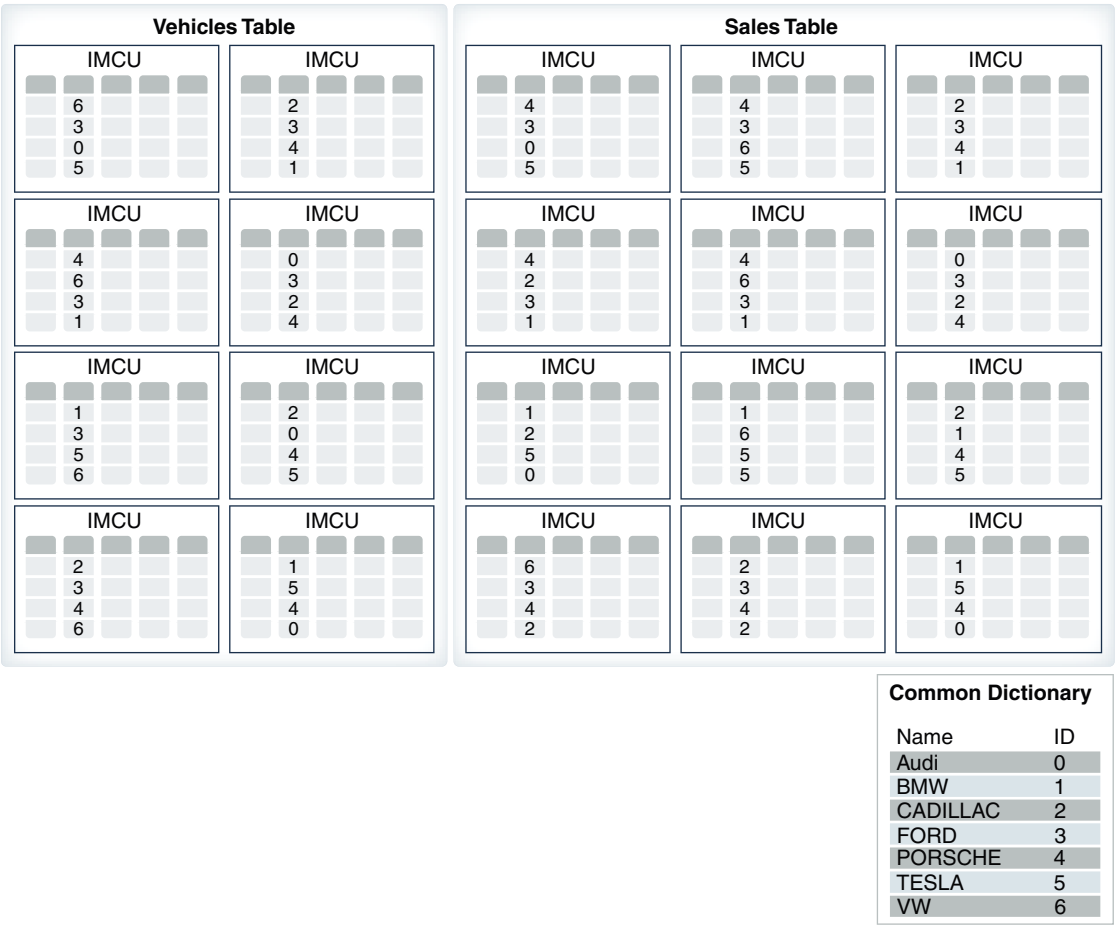
The database automatically creates a common dictionary in the IM column store when a join group is defined on the underlying columns. The common dictionary enables the join columns to share the same dictionary codes.

A common dictionary provides the following benefits:

- Encodes the values in the local dictionaries with codes from the common dictionary, which provides compression and increases the cache efficiency of the IMCU
- Enables joins to use dictionary codes to construct and probe the data structures used during hash joins
- Enables the optimizer to obtain statistics such as cardinality, distribution of column values, and so on

The following figure illustrates a common dictionary that corresponds to a join group created on the `sales.name` and `vehicles.name` columns.

Figure 8-2 Common Dictionary for a Join Group



When the database uses a common dictionary, the local dictionary for each CU does not store the original values: `AUDI`, `BMW`, `CADILLAC`, `FORD`, and so on. Instead, the local dictionary stores *references* to the values stored in the common dictionary. For example, the *local* dictionary might store the value `101` for `Audi` and `220` for `BMW`. The *common* dictionary might store the value `0` for `Audi` and `1` for `BMW`. The `101` (AUDI) in the local dictionary is a pointer to the `0` (AUDI) in the common dictionary.

8.4.2 How a Join Group Optimizes Scans

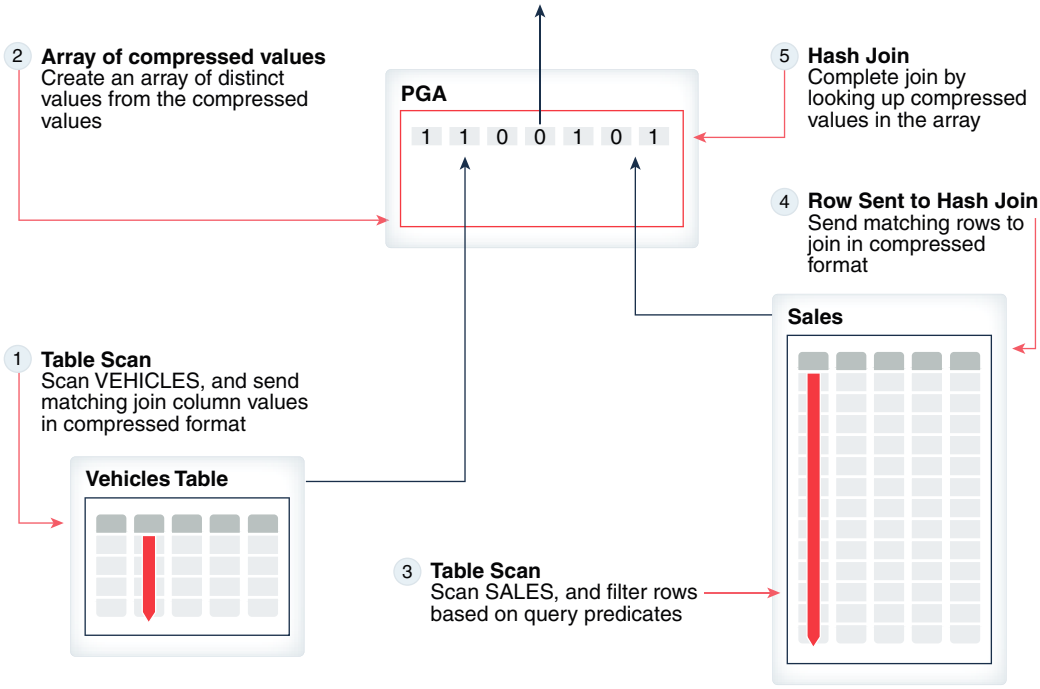
The key optimization is joining on common dictionary codes instead of column values, thereby avoiding the use of a hash table for the join.

Consider the following query, which uses a join group to join `vehicles` and `sales` on the `name` column:

```
SELECT v.year, v.name, s.sales_price
FROM   vehicles v, sales s
WHERE  v.name = s.name
AND    v.name IN ('Audi', 'BMW', 'Porsche', 'VW');
```

The following figure illustrates how the join benefits from the common dictionary created on the join group.

Figure 8-3 Hash Join with Join Group



As illustrated in the preceding diagram, the database performs a hash join on the compressed data as follows:

1. Scans the `vehicles` table, and sends the dictionary codes (not the original column values) to the hash join: `0` (Audi), `1` (BMW), `2` (Cadillac), and so on
2. Builds an array of distinct common dictionary codes in the PGA
3. Scans the `sales` table and applies any filters (in this case, the filter is for German cars only)
4. Sends matching rows to the join in compressed format
5. Looks up corresponding values in the array rather than probing a hash table, thus avoiding the need to compute a hash function on the join key columns

In this example, the `vehicles` table has only seven rows. The `vehicles.name` column has the following values:

```
Audi
BMW
Cadillac
Ford
Porsche
Tesla
VW
```

The common dictionary assigns a dictionary code to each distinct value. Conceptually, the common dictionary looks as follows:

```
Audi     0
BMW      1
Cadillac 2
Ford     3
Porsche  4
Tesla    5
VW       6
```

The database scans `vehicles.name`, starting at the first dictionary code in the first IMCU and ending at the last code in the last IMCU. It stores a `1` for every row that matches the filter (German cars only), and `0` for every row that does not match the filter. Conceptually, the array might look as follows:

```
array[0]: 1
array[1]: 1
array[2]: 0
array[3]: 0
array[4]: 1
array[5]: 0
array[6]: 1
```

The database now scans the `sales` fact table. To simplify the example, assume that the `sales` table only has 6 rows. The database scans the rows as follows (the common dictionary code for each value is shown in parentheses):

```
Cadillac (2)
Cadillac (2)
BMW      (1)
Ford     (3)
Audi     (0)
Tesla    (5)
```

The database then proceeds through the `vehicles.name` array, looking for matches. If a row matches, then the database sends the matching row with its associated common dictionary code, and retrieves the corresponding column value from the `vehicles.name` and `sales.name` IMCUs:

```
2  -> array[2] is 0, so no join
2  -> array[2] is 0, so no join
1  -> array[1] is 1, so join
3  -> array[3] is 0, so no join
0  -> array[0] is 1, so join
5  -> array[5] is 0, so no join
```

8.5 When a Hash Join Uses Common Dictionary Encodings

Joins on columns in a join group typically see a performance benefit.

At join group creation, the database does the following:

- Caches the hash of the dictionary values for the join key columns
- Caches the binary representation of the `NUMBER` data for the join key columns
- Encodes columns with the same common dictionary

A join on columns in a join group *always* uses the first two optimizations to improve performance. For example, if the optimizer chooses a hash join, then the query uses the cached hash values to probe the bloom filter. If the query uses an IM aggregation join, then the query uses the cached binary number to index into the key vector.

A hash join may or may not use dictionary encodings. When dictionary encodings are present in at least one column of the hash join, the query can leverage the encodings in the following ways:

- Join group-aware hash join

  Both columns in the hash join carry common dictionary encoding data during runtime. The execution plan must show either a parallel hash join plan without any distribution involved from *both* sides of the hash join, or a serial hash join plan.

- Encoding-aware hash join

  One fact table column in the hash join carries dictionary encoding data during runtime. The execution plan must show either a parallel hash join without any distribution from the right side of the hash join, or a serial hash join plan. In some cases, if the common dictionary has good compression ratio, and if a parallel hash join plan cannot leverage a join group-aware hash join (for example, in a parallel broadcast-none plan), then the query can use an encoding-aware hash join for the common dictionary.

In a SQL Monitor report, the following fields show dictionary usage: `Columnar Encodings Observed`, and `Columnar Encodings Leveraged`. The statistics are cumulative. In a parallel hash join, the fields summarize statistics collected from all slave processes involved in executing a row source. In the context of the local dictionary in an IMCU, the statistics show the number of encoding IDs observed from the right child row source and the number of encodings leveraged by the join. If a hash join on a single process leverages the common dictionary, then `Columnar Encodings Leveraged` shows the number of encodings leveraged in the join.

The following table indicates the possible values for `Columnar Encodings Observed` and `Columnar Encodings Leveraged`, and what the combinations mean.

Table 8-1 Join Group Usage in a SQL Monitor Report

Columnar Encodings Observed	Columnar Encodings Leveraged	Encoding-Aware Hash Join Used?	Join Group-Aware Hash Join Used?
Not present	Not present	No	No
Positive value	Not present	No	No
Positive value	Positive value	Yes	No
Not present	Positive value	No	Yes

For example, if the report shows that the `Columnar Encodings Leveraged` field is `4` (for example, because the parallel degree is 4) but the `Columnar Encodings Observed`

field is absent, then the query leveraged the join group for the hash join. If the `Columnar Encodings Observed` field is `4` but the `Columnar Encodings Leveraged` field is absent, then dictionary encodings existed, but the query did not use them.

Various factors can prevent a query from engaging an encoding-aware hash join. Factors include the following:

- The compression ratio of the common dictionary is suboptimal.
- The query observes too many row sets passed from the table scan without a common dictionary.
- The build-side row length is too large.
- The build-side rows cannot fit into PGA memory.
- The build side has duplicate join keys.

> **See Also:**
>
> "Monitoring Join Group Usage"

8.6 Creating Join Groups

Define join groups using the `CREATE INMEMORY JOIN GROUP` statement.

Candidates for join groups are columns that are frequently paired in a join predicate. Typical examples include a column joining a fact and dimension table, or a column joining a table to itself.

The `CREATE INMEMORY JOIN GROUP` statement immediately defines a join group, which means that its metadata is visible in the data dictionary. The database does not immediately construct the common dictionary. Rather, the database builds the common dictionary the next time that a table referenced in the join group is populated or repopulated in the IM column store.

Guidelines

Creating, modifying, or dropping a join group typically invalidates all the underlying tables referenced in the join group. Thus, Oracle recommends that you create join groups *before* initially populating the tables.

To create a join group:

1. In SQL*Plus or SQL Developer, log in to the database as a user with the necessary privileges.
2. Create a join group by using a statement in the following form:

```
CREATE INMEMORY JOIN GROUP join_group_name ( table1(col1), table2(col2) );
```

For example, the following statement creates a join group named `sales_products_jg`:

```
CREATE INMEMORY JOIN GROUP sales_products_jg (sales(prod_id),
products(prod_id));
```

3. Optionally, view the join group definition by querying the data dictionary (sample output included):

```
COL JOINGROUP_NAME FORMAT a18
COL TABLE_NAME FORMAT a8
COL COLUMN_NAME FORMAT a7

SELECT JOINGROUP_NAME, TABLE_NAME, COLUMN_NAME, GD_ADDRESS
FROM   DBA_JOINGROUPS;

JOINGROUP_NAME     TABLE_NA COLUMN_ GD_ADDRESS
------------------ -------- ------- ----------------
SALES_PRODUCTS_JG  SALES    PROD_ID 00000000A142AE50
SALES_PRODUCTS_JG  PRODUCTS PROD_ID 00000000A142AE50
```

4. Populate the tables referenced in the join group, or repopulate them if they are currently populated.

Example 8-2 Optimizing a Query Using a Join Group

In this example, you log in to the database as `SYSTEM`, and then create a join group on the `prod_id` column of `sales` and `products`, which are not yet populated in the IM column store:

```
CREATE INMEMORY JOIN GROUP
  sh.sales_products_jg (sh.sales(prod_id), sh.products(prod_id));
```

You enable the `sh.sales` and `sh.products` tables for population in the IM column store:

```
ALTER TABLE sh.sales INMEMORY;
ALTER TABLE sh.products INMEMORY;
```

The following query indicates the tables are not yet populated in the IM column store (sample output included):

```
COL OWNER FORMAT a3
COL NAME FORMAT a10
COL STATUS FORMAT a20

SELECT OWNER, SEGMENT_NAME NAME,
       POPULATE_STATUS STATUS
FROM   V$IM_SEGMENTS;

no rows selected
```

Query both tables to populate them in the IM column store:

```
SELECT /*+ FULL(s) NO_PARALLEL(s) */ COUNT(*) FROM sh.sales s;
SELECT /*+ FULL(p) NO_PARALLEL(p) */ COUNT(*) FROM sh.products p;
```

The following query indicates the tables are now populated in the IM column store (sample output included):

```
COL OWNER FORMAT a3
COL NAME FORMAT a10
COL PARTITION FORMAT a13
COL STATUS FORMAT a20

SELECT OWNER, SEGMENT_NAME NAME, PARTITION_NAME PARTITION,
       POPULATE_STATUS STATUS, BYTES_NOT_POPULATED
FROM   V$IM_SEGMENTS;

OWN NAME       PARTITION     STATUS               BYTES_NOT_POPULATED
--- ---------- ------------- -------------------- -------------------
SH  SALES      SALES_Q3_1998 COMPLETED            0
SH  SALES      SALES_Q4_2001 COMPLETED            0
SH  SALES      SALES_Q4_1999 COMPLETED            0
SH  PRODUCTS                 COMPLETED            0
SH  SALES      SALES_Q1_2001 COMPLETED            0
SH  SALES      SALES_Q1_1999 COMPLETED            0
SH  SALES      SALES_Q2_2000 COMPLETED            0
SH  SALES      SALES_Q2_1998 COMPLETED            0
SH  SALES      SALES_Q3_2001 COMPLETED            0
SH  SALES      SALES_Q3_1999 COMPLETED            0
SH  SALES      SALES_Q4_2000 COMPLETED            0
SH  SALES      SALES_Q4_1998 COMPLETED            0
SH  SALES      SALES_Q1_2000 COMPLETED            0
SH  SALES      SALES_Q1_1998 COMPLETED            0
SH  SALES      SALES_Q2_2001 COMPLETED            0
SH  SALES      SALES_Q2_1999 COMPLETED            0
SH  SALES      SALES_Q3_2000 COMPLETED            0
```

Query `DBA_JOINGROUPS` to get information about the join group (sample output included):

```
COL JOINGROUP_NAME FORMAT a18
COL TABLE_NAME FORMAT a8
COL COLUMN_NAME FORMAT a7

SELECT JOINGROUP_NAME, TABLE_NAME, COLUMN_NAME, GD_ADDRESS
FROM   DBA_JOINGROUPS;

JOINGROUP_NAME     TABLE_NA COLUMN_ GD_ADDRESS
------------------ -------- ------- ----------------
SALES_PRODUCTS_JG  SALES    PROD_ID 00000000A142AE50
SALES_PRODUCTS_JG  PRODUCTS PROD_ID 00000000A142AE50
```

The preceding output shows that the join group `sales_products_jg` joins on the same common dictionary address.

> **See Also:**
>
> - *Oracle Database SQL Language Reference* to learn about the `CREATE INMEMORY JOIN GROUP` statement
> - *Oracle Database Reference* to learn about the `DBA_JOINGROUPS` view

8.7 Monitoring Join Group Usage

To determine whether queries are using the join group, you can use either a graphical SQL Monitor report (recommended) or a SQL query that uses the `DBMS_SQLTUNE.REPORT_SQL_MONITOR_XML` function.

"When a Hash Join Uses Common Dictionary Encodings" explains how to interpret the SQL Monitor output.

Prerequisites

To monitor join groups, you must meet the following prerequisites:

- A join group must exist.
- The columns referenced by the join group must have been populated *after* join group creation.
- You must execute a join query that could potentially use the join group.

To monitor join group usage:

1. Log in to the database as a user with the necessary privileges.
2. Create a SQL*Plus variable to store the SQL ID as follows:

   ```
   VAR b_sqlid VARCHAR2(13)
   ```

3. Execute a query that joins on the columns in the join group.
4. Use either following techniques:
   - Graphical SQL Monitor Report

     SQL Monitor reports are available in Enterprise Manager. In SQL*Plus, you can use `DBMS_SQL_MONITOR.REPORT_SQL_MONITOR` to generate a SQL Monitor report as follows:

     ```
     SET TRIMSPOOL ON
     SET TRIM ON
     SET PAGES 0
     SET LINESIZE 1000
     SET LONG 1000000
     SET LONGCHUNKSIZE 1000000
     SPOOL /tmp/long_sql.htm
     SELECT DBMS_SQL_MONITOR.REPORT_SQL_MONITOR(
              sql_id       => :b_sqlid,
              report_level => 'ALL',
              TYPE         => 'active')
     ```

```
FROM   DUAL;
SPOOL OFF
```

Access the report in a browser, and then click the binoculars icon on the hash join to view the join group statistics.

- Command-Line Query

  Use the `DBMS_SQLTUNE.REPORT_SQL_MONITOR_XML` function in a query, as shown in the following example:

```
SELECT
  encoding_hj.rowsource_id row_source_id,
    CASE
      WHEN encoding_hj.encodings_observed IS NULL
      AND encoding_hj.encodings_leveraged IS NOT NULL
      THEN
        'join group was leveraged on ' ||
encoding_hj.encodings_leveraged || ' processes'
      ELSE
        'join group was NOT leveraged'
    END columnar_encoding_usage_info
FROM
  (SELECT
DBMS_SQLTUNE.REPORT_SQL_MONITOR_XML(session_id=>-1,sql_id=>:b_sqlid).
          EXTRACT(q'#//operation[@name='HASH JOIN' and @parent_id]#')
xmldata
   FROM   DUAL) hj_operation_data,
  XMLTABLE('/operation'
    PASSING hj_operation_data.xmldata
    COLUMNS
     "ROWSOURCE_ID"        NUMBER PATH '@id',
     "ENCODINGS_LEVERAGED" NUMBER PATH 'rwsstats/stat[@id="9"]',
     "ENCODINGS_OBSERVED"  NUMBER PATH 'rwsstats/stat[@id="10"]')
encoding_hj;
```

8.7.1 Monitoring Join Groups Using a SQL Monitor Report: Example

Your goal is to use a graphical SQL Monitor report to determine whether a query leveraged a join group.

In this example, you create a join group on the `prod_id` columns of `sh.products` and `sh.sales` tables, and then join these tables on this column. You grant the `sh` account administrative privileges.

Example 8-3 Monitoring a Join Group Using a SQL Monitor Report

1. In SQL*Plus, log in to the database as user `sh`.
2. Create a SQL*Plus variable to store the SQL ID as follows:

```
VAR b_sqlid VARCHAR2(13)
```

3. Apply the `INMEMORY` attribute to the `sh.products` and `sh.sales` tables as follows:

```
ALTER TABLE sales NO INMEMORY;
ALTER TABLE products NO INMEMORY;

ALTER TABLE sales INMEMORY MEMCOMPRESS FOR QUERY;
ALTER TABLE products INMEMORY MEMCOMPRESS FOR QUERY;
```

4. Create a join group on `prod_id`:

```
CREATE INMEMORY JOIN GROUP jgrp_products_sales (products(prod_id),
sales(prod_id));
```

5. Scan the tables to populate them in the IM column store:

```
SELECT /*+ FULL(s) */ COUNT(*) FROM sales s;
SELECT /*+ FULL(p) */ COUNT(*) FROM products p;
```

6. Execute a query that joins on the `prod_id` column, and then aggregates product sales:

```
SELECT /*+ USE_HASH(sales) LEADING(products sales) MONITOR */
products.prod_id,
       products.prod_category_id, SUM(sales.amount_sold)
FROM   products, sales
WHERE  products.prod_id = sales.prod_id
GROUP BY products.prod_category_id, products.prod_id;
```

7. Generate an HTML-based SQL Monitor report by using `DBMS_SQLTUNE.REPORT_SQL_MONITOR`.

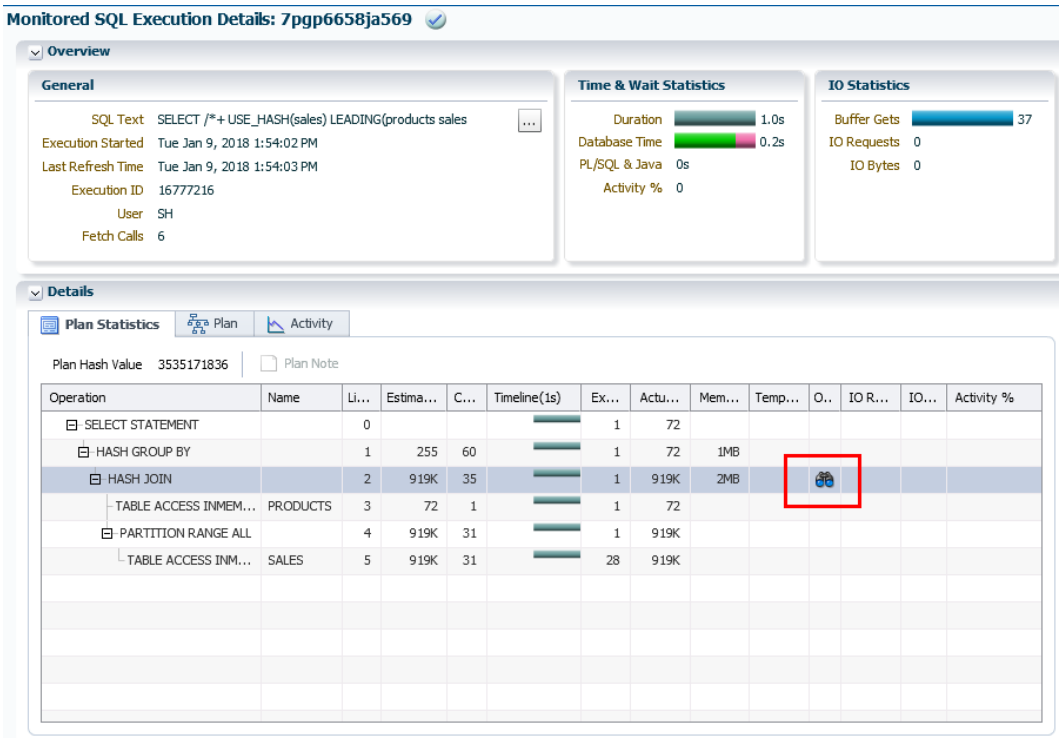
   For example, create a SQL script with the following contents, and run it in SQL*Plus:

```
SET TRIMSPOOL ON
SET TRIM ON
SET PAGES 0
SET LINESIZE 1000
SET LONG 1000000
SET LONGCHUNKSIZE 1000000
SPOOL /tmp/jg_report.htm
SELECT DBMS_SQL_MONITOR.REPORT_SQL_MONITOR(
         sql_id       => :b_sqlid,
         report_level => 'ALL',
         TYPE         => 'active')
FROM   DUAL;
SPOOL OFF
```

8. Open the HTML report in a browser.

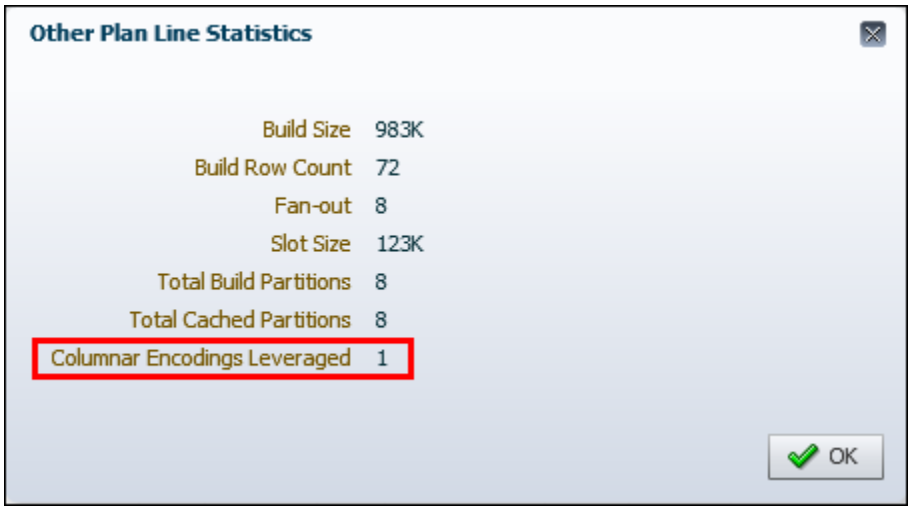
   The following sample report shows the execution plan for the join. The binoculars in the hash join open a window that shows additional statistics.

Figure 8-4 Monitored SQL Execution Details Page



9. Click the binoculars icon to open a window that shows join group statistics.

   The following sample window shows the statistics:



   Because `Columnar Encodings Leveraged` is a positive value and `Columnar Encodings Observed` is not present, the join group was leveraged.

10. Optionally, clean up after the example:

```
DROP INMEMORY JOIN GROUP jgrp_products_sales;
ALTER TABLE sales NO INMEMORY;
ALTER TABLE products NO INMEMORY;
```

See Also:

- "When a Hash Join Uses Common Dictionary Encodings"
- *Oracle Database PL/SQL Packages and Types Reference* to learn about the `DBMS_SQLTUNE.REPORT_SQL_MONITOR_XML` function
- *Oracle Database Reference* to learn about the `V$SESSION` view

8.7.2 Monitoring Join Groups from the Command Line: Example

Your goal is to use command-line tools to determine whether a query leveraged a join group.

In this example, you create a join group on the `prod_id` columns of `sh.products` and `sh.sales` tables, and then join these tables on this column. You grant the `sh` account administrative privileges.

Example 8-4 Monitoring a Join Group from the Command Line

1. Log in to the database as `sh`.
2. Create a SQL*Plus variable to store the SQL ID as follows:

```
VAR b_sqlid VARCHAR2(13)
```

3. Apply the `INMEMORY` attribute to the `sh.products` and `sh.sales` tables as follows:

```
ALTER TABLE sales NO INMEMORY;
ALTER TABLE products NO INMEMORY;

ALTER TABLE sales INMEMORY MEMCOMPRESS FOR QUERY;
ALTER TABLE products INMEMORY MEMCOMPRESS FOR QUERY;
```

4. Create a join group on `prod_id`:

```
CREATE INMEMORY JOIN GROUP jgrp_products_sales (products(prod_id),
sales(prod_id));
```

5. Scan the tables to populate them in the IM column store:

```
SELECT /*+ FULL(s) */ COUNT(*) FROM sales s;
SELECT /*+ FULL(p) */ COUNT(*) FROM products p;
```

6. Execute a query that joins on the `prod_id` column, and then aggregates product sales:

```
SELECT /*+ USE_HASH(sales) LEADING(products sales) MONITOR */
products.prod_id,
       products.prod_category_id, SUM(sales.amount_sold)
FROM   products, sales
```

```
WHERE  products.prod_id = sales.prod_id
GROUP BY products.prod_category_id, products.prod_id;
```

7. Obtain the SQL ID of the preceding aggregation query:

```
BEGIN
  SELECT PREV_SQL_ID
    INTO :b_sqlid
  FROM   V$SESSION
  WHERE  SID=USERENV('SID');
END;
```

8. Use `DBMS_SQLTUNE.REPORT_SQL_MONITOR_XML` to determine whether the database used the join group.

   For example, execute the following query:

```
COL row_source_id FORMAT 999
COL columnar_encoding_usage_info FORMAT A40

SELECT
  encoding_hj.rowsource_id row_source_id,
    CASE
      WHEN encoding_hj.encodings_observed IS NULL
      AND encoding_hj.encodings_leveraged IS NOT NULL
      THEN
        'join group was leveraged on ' || encoding_hj.encodings_leveraged
|| ' processes'
      ELSE
        'join group was NOT leveraged'
    END columnar_encoding_usage_info
FROM
  (SELECT
DBMS_SQLTUNE.REPORT_SQL_MONITOR_XML(session_id=>-1,sql_id=>:b_sqlid).
          EXTRACT(q'#//operation[@name='HASH JOIN' and @parent_id]#')
xmldata
   FROM   DUAL
  ) hj_operation_data,
  XMLTABLE('/operation'
    PASSING hj_operation_data.xmldata
    COLUMNS
     "ROWSOURCE_ID"       NUMBER PATH '@id',
     "ENCODINGS_LEVERAGED" NUMBER PATH 'rwsstats/stat[@id="9"]',
     "ENCODINGS_OBSERVED"  NUMBER PATH 'rwsstats/stat[@id="10"]'
     ) encoding_hj;
```

   The following sample output shows that the join group was leveraged in the query:

```
ROW_SOURCE_ID COLUMNAR_ENCODING_USAGE_INFO
------------- ----------------------------------------
            2 join group was leveraged on 1 processes
```

9. Optionally, clean up after the example:

```
DROP INMEMORY JOIN GROUP jgrp_products_sales;
ALTER TABLE sales NO INMEMORY;
ALTER TABLE products NO INMEMORY;
```

See Also:

- "When a Hash Join Uses Common Dictionary Encodings"
- *Oracle Database PL/SQL Packages and Types Reference* to learn about the `DBMS_SQLTUNE.REPORT_SQL_MONITOR_XML` function
- *Oracle Database Reference* to learn about the `V$SESSION` view

9
Optimizing Aggregation 6.3

Database In-Memory provides optimizations to speed up aggregation and arithmetic.

9.1 Optimizing In-Memory Aggregation with VECTOR GROUP BY

Starting with Oracle Database 12c Release 1 (12.1.0.2), **In-Memory Aggregation (IM aggregation)** enables queries to aggregate while scanning.

9.1.1 About IM Aggregation

IM aggregation optimizes query blocks involving aggregation and joins from a large table to multiple small tables.

The `KEY VECTOR` and `VECTOR GROUP BY` operations use efficient arrays for joins and aggregation. The optimizer chooses `VECTOR GROUP BY` for `GROUP BY` operations based on cost. The optimizer does not choose `VECTOR GROUP BY` aggregations for `GROUP BY ROLLUP`, `GROUPING SETS`, or `CUBE` operations.

> **Note:**
>
> IM aggregation is also called *vector aggregation* and *`VECTOR GROUP BY` aggregation*.

IM aggregation requires `INMEMORY_SIZE` to be set to a nonzero value. However, IM aggregation does not require that the referenced tables be *populated* in the IM column store.

> **See Also:**
>
> - "Enabling the IM Column Store for a Database"
> - *Oracle Database Data Warehousing Guide* to learn more about SQL aggregation

9.1.2 Purpose of IM Aggregation

IM aggregation preprocesses the small tables to accelerate the per-row work performed on the large table.

A typical analytic query aggregates from a fact table, and joins it to dimension tables. This type of query scans a large volume of data, with optional filtering, and performs a `GROUP BY` of between 1 and 40 columns. The first aggregation on the fact table processes the most rows.

Before Oracle Database 12c, the only `GROUP BY` operations were `HASH` and `SORT`. The `VECTOR GROUP BY` is an additional cost-based transformation that transforms a join between a dimension and fact table into a filter. The database can apply this filter during the fact table scan. The joins use key vectors, which are similar to Bloom filters, and the aggregation uses a `VECTOR GROUP BY`.

> **Note:**
>
> Although vector transformations are independent of the IM column store, they can be applied very efficiently to In-Memory data through SIMD vector processing.

IM aggregation enables vector joins and `GROUP BY` operations to occur *simultaneously* with the scan of the large table. Thus, these operations aggregate as they scan, and do not need to wait for table scans and join operations to complete. IM aggregation optimizes CPU usage, especially the CPU cache.

IM aggregation can greatly improve query performance. The database can create a report outline dynamically, and then fill in report details during the scan of the fact table.

> **See Also:**
>
> - "CPU Architecture: SIMD Vector Processing"
> - *Oracle Database SQL Tuning Guide* to learn more about query transformations

9.1.2.1 When IM Aggregation Is Useful

IM aggregation improves performance of queries that join relatively small tables to a relatively large fact table, and aggregate data in the fact table. This typically occurs in a star or snowflake query.

Both row-store tables and tables in the IM column store can benefit from IM aggregation.

Example 9-1 VECTOR GROUP BY

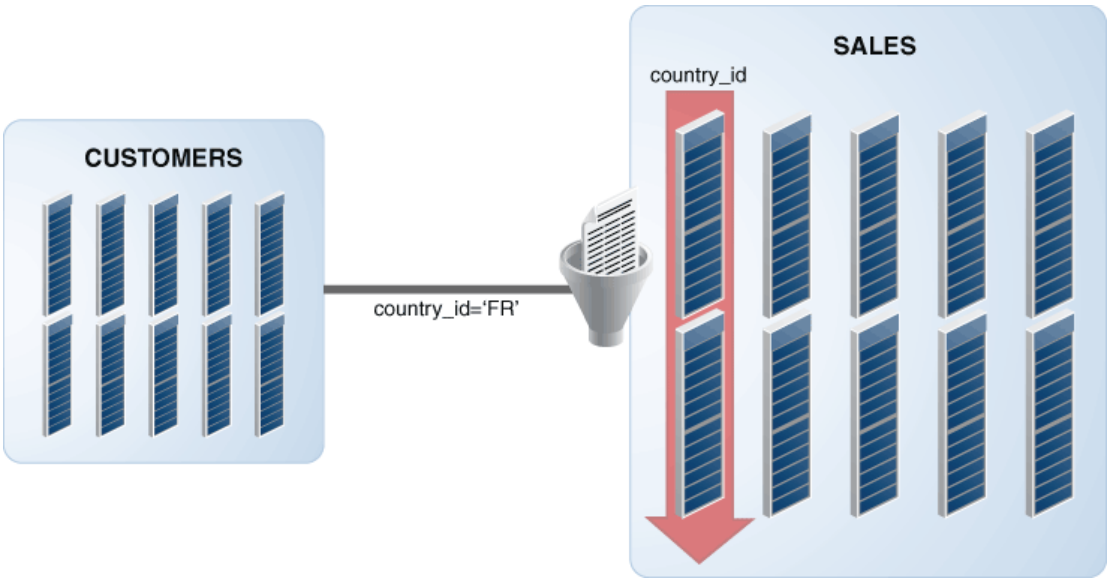
Consider the following query, which performs a join of the `customers` dimension table with the `sales` fact table:

```
SELECT c.customer_id, s.quantity_sold, s.amount_sold
FROM   customers c, sales s
```

```
WHERE   c.customer_id = s.customer_id
AND     c.country_id = 'FR';
```

When both tables are populated in the IM column store, the database can use SIMD vector processing to scan the row sets and apply filters. The following figure shows how the query uses vector joins. The optimizer converts the predicate on the `customers` table, `c.country_id='FR'` into a filter on the `sales` fact table. The filter is `country_id='FR'`. Because `sales` is stored in columnar format, the query only needs to scan one column to determine the result.

Figure 9-1 Vector Joins Using In-Memory Column Store



9.1.2.2 When IM Aggregation Is Not Beneficial

IM aggregation benefits certain star queries when sufficient system resources exist. Other queries may receive little or no benefit.

Situations Where VECTOR GROUP BY Aggregation Is Not Advantageous

Specifically, `VECTOR GROUP BY` aggregation does not benefit performance in the following scenarios:

- Joins are performed between two very large tables.

  By default, the optimizer chooses a `VECTOR GROUP BY` transformation only if a relatively small table is joined to a relatively large table.
- Dimensions contain more than 2 billion rows.

  The `VECTOR GROUP BY` transformation is not used if a dimension contains more than 2 billion rows.
- The system does not have sufficient memory.

  Most databases that use the IM column store benefit from IM aggregation.

9.1.3 How IM Aggregation Works

A typical analytic query distributes rows among processing stages.

The stages are as follows:

1. Filtering tables and producing row sets
2. Joining row sets
3. Aggregating rows

The `VECTOR GROUP BY` transformation combines the work in the different stages, converting joins to filters and aggregating while scanning the fact table.

The unit of work between stages is called a data flow operator (DFO). `VECTOR GROUP BY` aggregation uses a DFO for each dimension to create a key vector structure and temporary table. When aggregating measure columns from the fact table, the database uses this key vector to translate a fact join key to its dense grouping key. The late materialization step joins on the dense grouping keys to the temporary tables.

9.1.3.1 When the Optimizer Chooses IM Aggregation

The optimizer decides whether to use vector transformation based on the size of the key vector (that is, the distinct join keys), the number of distinct grouping keys, and other factors.

The optimizer tends to choose this transformation when dimension join keys have low cardinality. Oracle Database uses `VECTOR GROUP BY` aggregation to perform data aggregation when the following conditions are met:

- The queries or subqueries aggregate data from a fact table and join the fact table to one or more dimensions.

  Multiple fact tables joined to the same dimensions are also supported assuming that these fact tables are connected only through joins to the dimension. In this case, `VECTOR GROUP BY` aggregates fact table separately and then joins the results on the grouping keys.
- The dimensions and fact table are connected to each other only through join columns.

  Specifically, the query must not have any other predicates that refer to columns across multiple dimensions or from both a dimension and the fact table. If a query performs a join between two or more tables and then joins the result to the fact, then `VECTOR GROUP BY` aggregation treats the multiple dimensions as a single dimension.

> **Note:**
>
> You can direct the database to use `VECTOR GROUP BY` aggregation for a query by using query block hints or table hints.

`VECTOR GROUP BY` aggregation does not support the following:

- Semi-joins and anti-joins across multiple dimensions or between a dimension and the fact table
- Equijoins across multiple dimensions
- Aggregations performed using the `DISTINCT` function

> **Note:**
>
> Bloom filters and `VECTOR GROUP BY` aggregation are mutually exclusive. Therefore, if a query uses Bloom filters to join row sets, then `VECTOR GROUP BY` aggregation is not applicable to the processing of this query.

> **See Also:**
>
> *Oracle Database Data Warehousing Guide* to learn more about SQL aggregation

9.1.3.2 Key Vector

A **key vector** is a data structure that maps between dense join keys and dense grouping keys.

A dense key is a numeric key that is stored as a native integer and has a range of values. A dense join key represents all join keys whose join columns come from a particular fact table or dimension. A dense grouping key represents all grouping keys whose grouping columns come from a particular fact table or dimension. A key vector enables fast lookups.

Example 9-2 Key Vector

Assume that the `hr.locations` table has values for `country_id` as shown (only the first few results are shown):

```
SQL> SELECT country_id FROM locations;

CO
--
IT
IT
JP
JP
US
US
US
US
CA
CA
CN
```

A complex analytic query applies the filter `WHERE country_id='US'` to the `locations` table. A key vector for this filter might look like the following one-dimensional array:

```
0
0
0
0
1
1
1
1
0
0
0
```

In the preceding array, `1` is the dense grouping key for `country_id='US'`. The `0` values indicate rows in `locations` that do not match this filter. If a query uses the filter `WHERE country_id IN ('US','JP')`, then the array might look as follows, where `2` is the dense grouping key for `JP` and `1` is the dense grouping key for `US`:

```
0
0
2
2
1
1
1
1
0
0
0
```

9.1.3.3 Two Phases of IM Aggregation

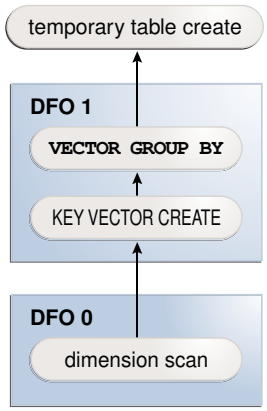
Typically, `VECTOR GROUP BY` aggregation processes each dimension in sequence, and then processes the fact table.

When performing IM aggregation, the database proceeds as follows:

1. Process each dimension sequentially as follows:
   a. Find the unique dense grouping keys.
   b. Create a key vector.
   c. Create a temporary table (`CURSOR DURATION MEMORY`).

   The following figure illustrates the steps in this phase, beginning with the scan of the dimension table in DFO 0, and ending with the creation of a temporary table. In the simplest form of parallel `GROUP BY` or join processing, the database processes each join or `GROUP BY` in its own DFO.

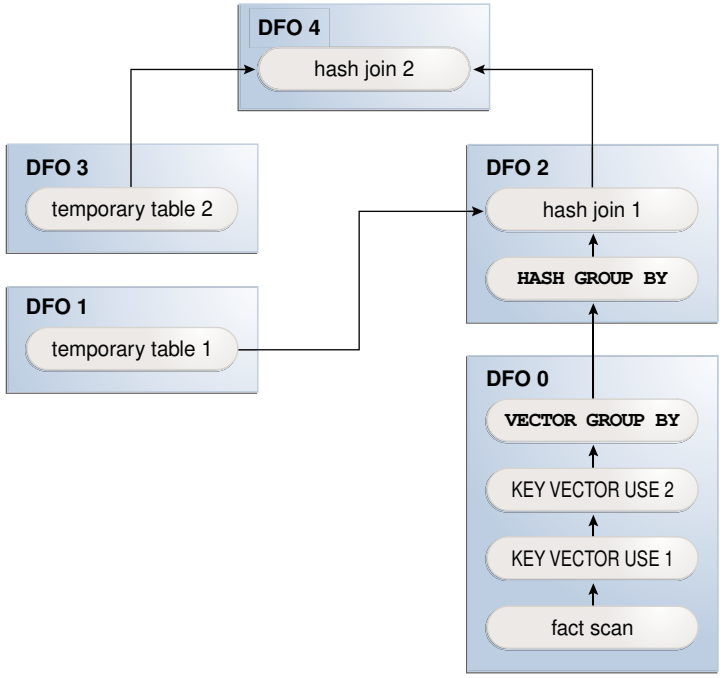
Figure 9-2 Phase 1 of In-Memory Aggregation



2. Process the fact table.
   a. Process all the joins and aggregations using the key vectors created in the preceding phase.
   b. Join back the results to each temporary table.

Figure 9-3 illustrates phase 2 in a join of the fact table with two dimensions. In DFO 0, the database performs a full scan of the fact table, and then uses the key vectors for each dimension to filter out nonmatching rows. DFO 2 joins the results of DFO 0 with DFO 1. DFO 4 joins the result of DFO 2 with DFO 3.

Figure 9-3 Phase 2 of In-Memory Aggregation



9.1.3.4 IM Aggregation: Scenario

This section gives a conceptual example of how `VECTOR GROUP BY` aggregation works.

> **Note:**
>
> The scenario does not use the sample schema tables or show an actual execution plan.

9.1.3.4.1 Sample Analytic Query of a Star Schema

This sample star schema in this scenario contains the `sales_online` fact table and two dimension tables: `geography` and `products`.

Each row in `geography` is uniquely identified by the `geog_id` column. Each row in `products` is uniquely identified by the `prod_id` column. Each row in `sales_online` is uniquely identified by the `geog_id`, `prod_id`, and amount sold.

Table 9-1 Sample Rows in geography Table

country	state	city	geog_id
USA	WA	seattle	2
USA	WA	spokane	3
USA	CA	SF	7
USA	CA	LA	8

Table 9-2 Sample Rows in products Table

manuf	category	subcategory	prod_id
Acme	sport	bike	4
Acme	sport	ball	3
Acme	electric	bulb	1
Acme	electric	switch	8

Table 9-3 Sample Rows in sales_online Table

prod_id	geog_id	amount
8	1	100
9	1	150
8	2	100
4	3	110
2	30	130
6	20	400
3	1	100
1	7	120
3	8	130
4	3	200

A manager asks the business question, "How many Acme products in each subcategory were sold online in Washington, and how many were sold in California?" To answer this question, an analytic query of the `sales_online` fact table joins the `products` and `geography` dimension tables as follows:

```
SELECT p.category, p.subcategory, g.country, g.state, SUM(s.amount)
FROM   sales_online s, products p, geography g
WHERE  s.geog_id = g.geog_id
AND    s.prod_id = p.prod_id
AND    g.state IN ('WA','CA')
AND    p.manuf = 'ACME'
GROUP BY category, subcategory, country, state
```

9.1.3.4.2 Step 1: Key Vector and Temporary Table Creation for geography Dimension

In the first phase of `VECTOR GROUP BY` aggregation for this query, the database creates a dense grouping key for each city/state combination for cities in the states of Washington or California.

In Table 9-6, the `1` is the `USA,WA` grouping key, and the `2` is the `USA,CA` grouping key.

Table 9-4 Dense Grouping Key for geography

country	state	city	geog_id	dense_gr_key_geog
USA	WA	seattle	2	1
USA	WA	spokane	3	1
USA	CA	SF	7	2
USA	CA	LA	8	2

A key vector for the `geography` table looks like the array represented by the final column in Table 9-5. The values are the `geography` dense grouping keys. Thus, the key vector indicates which rows in `sales_online` meet the `geography.state` filter criteria (a sale made in the state of `CA` or `WA`) and which country/state group each row belongs to (either the `USA,WA` group or `USA,CA` group).

Table 9-5 Online Sales

prod_id	geog_id	amount	key vector for geography
8	1	100	0
9	1	150	0
8	2	100	1
4	3	110	1
2	30	130	0
6	20	400	0
3	1	100	0
1	7	120	2
3	8	130	2

Table 9-5 (Cont.) Online Sales

prod_id	geog_id	amount	key vector for geography
4	3	200	1

Internally, the database creates a temporary table similar to the following:

```
CREATE TEMPORARY TABLE tt_geography AS
SELECT MAX(country), MAX(state), KEY_VECTOR_CREATE(...) dense_gr_key_geog
FROM   geography
WHERE  state IN ('WA','CA')
GROUP BY country, state
```

Table 9-6 shows rows in the `tt_geography` temporary table. The dense grouping key for the `USA,WA` combination is `1`, and the dense grouping key for the `USA,CA` combination is `2`.

Table 9-6 tt_geography

country	state	dense_gr_key_geog
USA	WA	1
USA	CA	2

9.1.3.4.3 Step 2: Key Vector and Temporary Table Creation for products Dimension

The database creates a dense grouping key for each distinct category/subcategory combination of an Acme product.

For example, in Table 9-7, the `4` is the dense grouping key for an Acme electric switch.

Table 9-7 Sample Rows in products Table

manuf	category	subcategory	prod_id	dense_gr_key_prod
Acme	sport	bike	4	1
Acme	sport	ball	3	2
Acme	electric	bulb	1	3
Acme	electric	switch	8	4

A key vector for the `products` table might look like the array represented by the final column in Table 9-8. The values represent the `products` dense grouping key. For example, the `4` represents the online sale of an Acme electric switch. Thus, the key vector indicates which rows in `sales_online` meet the `products` filter criteria (a sale of an Acme product).

Table 9-8 Key Vector

prod_id	geog_id	amount	key vector for products
8	1	100	4

Table 9-8 (Cont.) Key Vector

prod_id	geog_id	amount	key vector for products
9	1	150	0
8	2	100	4
4	3	110	1
2	30	130	0
6	20	400	0
3	1	100	2
1	7	120	3
3	8	130	2
4	3	200	1

Internally, the database creates a temporary table similar to the following:

```
CREATE TEMPORTARY TABLE tt_products AS
SELECT MAX(category), MAX(subcategory), KEY_VECTOR_CREATE(...) dense_gr_key_prod
FROM   products
WHERE  manuf = 'ACME'
GROUP BY category, subcategory
```

Table 9-9 shows rows in this temporary table.

Table 9-9 tt_products

category	subcategory	dense_gr_key_prod
sport	bike	1
sport	ball	2
electric	bulb	3
electric	switch	4

9.1.3.4.4 Step 3: Key Vector Query Transformation

In this phase, the database processes the fact table.

The optimizer transforms the original query into the following equivalent query, which accesses the key vectors:

```
SELECT KEY_VECTOR_PROD(prod_id),
       KEY_VECTOR_GEOG(geog_id),
       SUM(amount)
FROM   sales_online
WHERE  KEY_VECTOR_PROD_FILTER(prod_id) IS NOT NULL
AND    KEY_VECTOR_GEOG_FILTER(geog_id) IS NOT NULL
GROUP BY KEY_VECTOR_PROD(prod_id), KEY_VECTOR_GEOG(geog_id)
```

The preceding transformation is not an exact rendition of the internal SQL, which is much more complicated, but a conceptual representation designed to illustrate the basic concept.

9.1.3.4.5 Step 4: Row Filtering from Fact Table

This phase obtains the amount sold for each combination of grouping keys.

The database uses the key vectors to filter out unwanted rows from the fact table. In Table 9-10, the first three columns represent the `sales_online` table. The last two columns provide the dense grouping keys for the `geography` and `products` tables.

Table 9-10 Dense Grouping Keys for the sales_online Table

prod_id	geog_id	amount	dense_gr_key_prod	dense_gr_key_geog
7	1	100	4	
9	1	150		
8	2	100	4	1
4	3	110	1	1
2	30	130		
6	20	400		
3	1	100	2	
1	7	120	3	2
3	8	130	2	2
4	3	200	1	1

As shown in Table 9-11, the database retrieves only those rows from `sales_online` with non-null values for both dense grouping keys, indicating rows that satisfy all the filtering criteria.

Table 9-11 Filtered Rows from sales_online Table

geog_id	prod_id	amount	dense_gr_key_prod	dense_gr_key_geog
2	8	100	4	1
3	4	110	1	1
3	4	200	1	1
7	1	120	3	2
8	3	130	2	2

9.1.3.4.6 Step 5: Aggregation Using an Array

The database uses a multidimensional array to perform the aggregation.

In Table 9-12, the `geography` grouping keys are horizontal, and the `products` grouping keys are vertical. The database adds the values in the intersection of each dense grouping key combination. For example, for the intersection of the `geography` grouping key `1` and the `products` grouping key `1`, the sum of `110` and `200` is `310`.

Table 9-12 Aggregation Array

dgkp/dgkg	1	2
1	110,200	
2		130
3		120
4	100	

9.1.3.4.7 Step 6: Join Back to Temporary Tables

In the final stage of processing, the database uses the dense grouping keys to join back the rows to the temporary tables to obtain the names of the regions and categories.

The results look as follows:

```
CATEGORY SUBCATEGORY COUNTRY STATE AMOUNT
-------- ----------- ------- ----- ------
electric bulb        USA     CA    120
electric switch      USA     WA    100
sport    ball        USA     CA    130
sport    bike        USA     WA    310
```

9.1.4 Controls for IM Aggregation

IM aggregation is integrated with the optimizer.

No new SQL or initialization parameters are required. IM aggregation does not need additional indexes, foreign keys, or dimensions.

To control IM aggregation manually, you can use the following pairs of hints:

- Query block hints

  `VECTOR_TRANSFORM` enables the vector transformation on the specified query block, regardless of costing. `NO_VECTOR_TRANSFORM` disables the vector transformation from engaging on the specified query block.

- Table hints

  You can use the following pairs of hints:

  - `VECTOR_TRANSFORM_FACT` includes the specified `FROM` expressions in the fact table generated by the vector transformation. `NO_VECTOR_TRANSFORM_FACT` excludes the specified `FROM` expressions from the fact table generated by the vector transformation.
  - `VECTOR_TRANSFORM_DIMS` includes the specified `FROM` expressions in enabled dimensions generated by the vector transformation. `NO_VECTOR_TRANSFORM_DIMS` excludes the specified `FROM` expressions from enabled dimensions generated by the vector transformation.

> **See Also:**
>
> *Oracle Database SQL Language Reference* to learn more about the `VECTOR_TRANSFORM_FACT` and `VECTOR_TRANSFORM_DIMS` hints

9.1.5 In-Memory Aggregation: Example

In this example, the business question is "How many products were sold in each category in each calendar year?"

You write the following query, which joins the `times`, `products`, and `sales` tables:

```
SELECT t.calendar_year, p.prod_category, SUM(quantity_sold)
FROM   times t, products p, sales s
WHERE  t.time_id = s.time_id
AND    p.prod_id = s.prod_id
GROUP BY t.calendar_year, p.prod_category;
```

Example 9-3 VECTOR GROUP BY Execution Plan

The following example shows the execution plan contained in the current cursor. Steps 4 and 8 show the creation of the key vectors for the dimension tables `times` and `products`. Steps 17 and 18 show the use of the previously created key vectors. Step 15 shows the `VECTOR GROUP BY` operation.

```
----------------------------------------------------
Execution Plan
----------------------------------------------------
Plan hash value: 2093829546

-------------------------------------------------------------------------------------------------------------------
------
|Id| Operation                              | Name                   |Rows|Bytes|Cost(%CPU)|Time |Pstart|
Pstop |
-------------------------------------------------------------------------------------------------------------------
------
|0 |SELECT STATEMENT                        |                        | 18 | 1116|302(90)|0:00:01|
|       |
|1 | TEMP TABLE TRANSFORMATION              |                        |    |     |       |       |
|       |
|2 |  LOAD AS SELECT (CURSOR DURATION MEMORY)|SYS_TEMP_0FD9D6608_F6A13|    |     |       |       |
|       |
|3 |   HASH GROUP BY                        |                        |  5 |  80 | 3 (67)|0:00:01|
|       |
|4 |    KEY VECTOR CREATE BUFFERED          | :KV0000                |  5 |  80 | 2 (50)|0:00:01|
|       |
|5 |     TABLE ACCESS INMEMORY FULL         | TIMES                  |1826|21912| 2 (50)|0:00:01|
|       |
|6 |  LOAD AS SELECT (CURSOR DURATION MEMORY)|SYS_TEMP_0FD9D6607_F6A13|    |     |       |       |
|       |
|7 |   HASH GROUP BY                        |                        |  5 | 125 | 2 (50)|0:00:01|
|       |
|8 |    KEY VECTOR CREATE BUFFERED          | :KV0001                |  5 | 125 | 1  (0)|0:00:01|
|       |
|9 |     TABLE ACCESS INMEMORY FULL         | PRODUCTS               | 72 |1512 | 1  (0)|0:00:01|
|       |
|10|  HASH GROUP BY                         |                        | 18 |1116 |297(91)|0:00:01|
|       |
```

```
|11|   HASH JOIN                          |                        | 18 |1116 |296(91)|0:00:01|       |       |
|12|    HASH JOIN                         |                        | 18 | 666 |294(91)|0:00:01|       |       |
|13|     TABLE ACCESS FULL                |SYS_TEMP_0FD9D6608_F6A13|  5 |  80 | 2  (0)|0:00:01|       |       |
|14|     VIEW                             | VW_VT_0737CF93         | 18 | 378 |291(92)|0:00:01|       |       |
|15|      VECTOR GROUP BY                 |                        | 18 | 414 |291(92)|0:00:01|       |       |
|16|       HASH GROUP BY                  |                        | 18 | 414 |291(92)|0:00:01|       |       |
|17|        KEY VECTOR USE                | :KV0000                |918K| 20M |285(92)|0:00:01|       |       |
|18|         KEY VECTOR USE               | :KV0001                |918K| 16M |284(92)|0:00:01|       |       |
|19|          PARTITION RANGE ITERATOR    |                        |918K| 13M |282(92)|0:00:01|:KV0000|:KV0000|
|20|           TABLE ACCESS INMEMORY FULL | SALES                  |918K| 13M |282(92)|0:00:01|:KV0000|:KV0000|
|21|    TABLE ACCESS FULL                 |SYS_TEMP_0FD9D6607_F6A13|  5 | 125 | 2  (0)|0:00:01|       |       |
-------------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

11 - access("ITEM_9"=INTERNAL_FUNCTION("C0"))
12 - access("ITEM_8"=INTERNAL_FUNCTION("C0"))
20 - inmemory(SYS_OP_KEY_VECTOR_FILTER("S"."PROD_ID",:KV0001) AND SYS_OP_KEY_VECTOR_FILTER("S"."TIME_ID",:KV0000))
       filter(SYS_OP_KEY_VECTOR_FILTER("S"."PROD_ID",:KV0001) AND SYS_OP_KEY_VECTOR_FILTER("S"."TIME_ID",:KV0000))

Note
-----
   - vector transformation used for this statement


Statistics
----------------------------------------------------------
         26  recursive calls
         13  db block gets
        124  consistent gets
         67  physical reads
       2200  redo size
       1454  bytes sent via SQL*Net to client
        634  bytes received via SQL*Net from client
          3  SQL*Net roundtrips to/from client
          2  sorts (memory)
          0  sorts (disk)
         20  rows processed
```

9.2 Optimizing In-Memory Arithmetic

In-Memory Optimized Arithmetic uses an optimized `NUMBER` format for fast calculations using SIMD hardware.

9.2.1 About In-Memory Optimized Arithmetic

The In-Memory optimized `NUMBER` format enables fast calculations using SIMD hardware.

For tables compressed with `QUERY LOW`, `NUMBER` columns are encoded using an optimized format that enables native calculations in hardware. SIMD vector processing enables simple aggregations, `GROUP BY` aggregations, and arithmetic operations to benefit significantly. The performance improvement depends on the amount of time the aggregation spends on arithmetic computation. Some aggregations may benefit by up to a factor of 9.

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

> **See Also:**
>
> "Estimating the Required Size of the IM Column Store"

9.2.2 Enabling and Disabling In-Memory Optimized Arithmetic

Control the feature by setting the initialization parameter `INMEMORY_OPTIMIZED_ARITHMETIC` to `DISABLE` (default) or `ENABLE`.

When set to `ENABLE`, Oracle Database uses an In-Memory optimized encoding for `NUMBER` columns in tables that use `FOR QUERY LOW` compression. When set to `DISABLE`, the database does not use the optimized encoding.

Switching from `ENABLE` to `DISABLE` does not drop the optimized number encoding for existing IMCUs immediately. Instead, as the IM column store repopulates IMCUs, the new IMCUs do not use the optimized encoding.

To enable and disable In-Memory Optimized Arithmetic:

1. In SQL*Plus or SQL Developer, log in to the database as a user with the necessary privileges.
2. Specify `INMEMORY_OPTIMIZED_ARITHMETIC` using the `ALTER SYSTEM` statement.

   The following example enables In-Memory Optimized Arithmetic:

   ```
   ALTER SYSTEM SET INMEMORY_OPTIMIZED_ARITHMETIC = 'ENABLE'
   SCOPE=BOTH;
   ```

> **See Also:**
>
> *Oracle Database Reference* to learn more about the `INMEMORY_OPTIMIZED_ARITHMETIC` initialization parameter

10

Optimizing Repopulation of the IM Column Store

The IM column store periodically refreshes objects that have been modified. You can control this behavior using initialization parameters and the `DBMS_INMEMORY` package.

10.1 About Repopulation of the IM Column Store

The automatic refresh of columnar data after significant modifications is called **repopulation**.

10.1.1 Row Modifications and the Transaction Journal

An In-Memory Compression Unit (IMCU) is a read-only structure that does not modify data in place when DML occurs on an internal table.

The Snapshot Metadata Unit (SMU) associated with each IMCU tracks row modifications in a transaction journal. If a query accesses the data, and discovers modified rows, then it can obtain the corresponding rowids from the transaction journal, and then retrieve the modified rows from the buffer cache.

As the number of modifications increase, so do the size of SMUs, and the amount of data that must be fetched from the transaction journal or database buffer cache. To avoid degrading query performance through journal access, background processes repopulate modified objects.

10.1.2 Automatic Repopulation

When DML occurs for objects in the IM column store, the database repopulates them automatically.

Automatic repopulation takes the following forms:

- threshold-based repopulation

  This form depends on the percentage of stale entries in the transaction journal for an IMCU.

- trickle repopulation

  This form supplements threshold-based repopulation by periodically refreshing columnar data even when the staleness threshold has not been reached.

During automatic repopulation, traditional access mechanisms are available. Data is always accessible from the buffer cache or disk. Additionally, the IM column store is always transactionally consistent with the data on disk. No matter where the query accesses the data, the database always returns consistent results.

> **See Also:**
>
> - "Transaction Journal"
> - "In-Memory Process Architecture"

10.1.3 Manual Repopulation of External Tables

External tables are not eligible for automatic repopulation.

The IM column store manages external tables differently from internal tables. Because external tables are read-only, they are not updated through DML, and thus do not rely on the transaction journal. For this reason, the database does not repopulate external tables automatically. However, you can refresh external tables manually by using `DBMS_INMEMORY.REPOPULATE`. In-Memory scans of external tables are only supported when the tables are completely populated in the IM column store.

> **Note:**
>
> Sessions that query In-Memory external tables must have the initialization parameter `QUERY_REWRITE_INTEGRITY` set to `stale_tolerated`.
>
> It is important to keep in mind that if an external table is modified, then the results from the IM column store are undefined. Results are also undefined if a partition is altered (by dropping or adding values). This may lead to differences in results between IM and non-IM based scans. You can run `DBMS_INMEMORY.REPOPULATE` to refresh the IM store so that it is resynchronized with the table data.

> **See Also:**
>
> - "Populating an In-Memory External Table Using DBMS_INMEMORY.POPULATE: Example"
> - *Oracle Database Reference* to learn more about the `QUERY_REWRITE_INTEGRITY` initialization parameter

10.2 How Data Loading Works with the IM Column Store

The IM column store uses different mechanisms depending on the type of data loading: conventional DML, direct path loads, and partition exchange loads.

10.2.1 How Conventional DML Works with the IM Column Store

Conventional DML processes one row or array of rows at a time, and inserts rows below the high water mark. Regardless of whether the IM column store is enabled, the database processes DML using the buffer cache.

IMCUs are read-only. When a statement modifies a row in an IMCU, the IM column store records the rowid in the associated SMU.

A Column Compression Unit (CU) entry becomes stale when its value differs from the value in its corresponding journal entry. For example, a transaction may change an employee's weekly salary from `1000` to `1200`, but the actual value in the IMCU is still `1000`. The transaction journal records the rowid of the stale row and its SCN.

> **Note:**
>
> The transaction journal does not record the new value. Rather, it indicates the corresponding row as stale as of a specific SCN.

10.2.1.1 Staleness Threshold

As the number of stale entries in an IMCU increases, the speed of the IMCU scan decreases.

Performance decreases because the database must fetch the modified rows from the buffer cache or disk, rather than from the IM column store. For this reason, Oracle Database repopulates an IMCU when the number of stale entries in an IMCU reaches an internal staleness threshold.

The database determines the threshold using heuristics that consider the frequency of IMCU access and the number of stale rows. Repopulation is more frequent for IMCUs that are accessed frequently or have a higher percentage of stale rows.

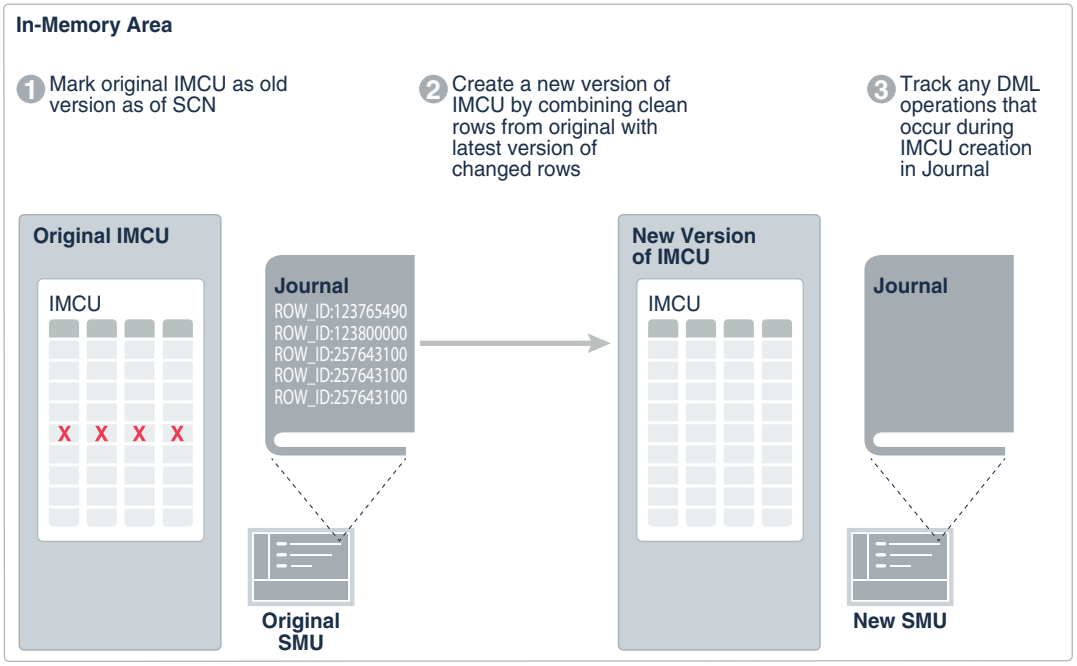
> **See Also:**
>
> - "In-Memory Compression Units (IMCUs)"
> - *Oracle Database Concepts* to learn more about the database buffer cache

10.2.1.2 Double Buffering

In double buffering, background processes create new IMCU versions by combining the original rows with the latest modified rows.

When the database begins either threshold-based repopulation or trickle repopulation, the IM column store uses double buffering. As shown in the following figure, the IM column store maintains two versions of an IMCU simultaneously, with the original stale IMCU remaining accessible to queries.

Figure 10-1 Double Buffering



The basic steps of double buffering are:

1. In the original SMU, the database marks the existing IMCU as the original version as of a specific SCN.
2. Background processes create a new version of the IMCU by combining the original rows with the latest versions of the modified rows.
3. In the journal of the new SMU, the database tracks DML operations that occur during IMCU creation.

In this way, the original IMCU stays online. The database keeps both old and new IMCUs versions for as long as they are useful, or until the IM column store is under space pressure.

10.2.2 How Direct Path Loads Work with the IM Column Store

A direct path load is an `INSERT /*+APPEND*/` statement or a SQL*Loader operation in which `DIRECT=true` .

In a direct path load, the database writes formatted data blocks directly to the data files, bypassing the database buffer cache. The database appends the data above the high water mark, which is the boundary between used and unused space in a segment. Direct path loads operate are “all or nothing” operations: the operation either inserts all data or no data.

Figure 10-2 Direct Path Loads and the High Water Mark

When the segment is populated in the IM column store, a direct path load works as follows:

1. You load data using a `CREATE TABLE AS SELECT` or `INSERT /*+APPEND*/` statement. Only the current session is aware of the DML.
2. You commit the statement.
3. The high water mark moves to encompass the new data, which alerts the IMCU that data is missing. `V$IM_SEGMENTS.BYTES_NOT_POPULATED` now indicates the size of the newly inserted data.
4. The IM column store manages repopulation based on the following algorithm:
   - If the affected object has a `PRIORITY` set to a value other than `NONE`, then the database repopulates the data.
   - If the affected object has a `PRIORITY` set to `NONE`, then the database repopulates at the next full scan of the object.

10.2.3 How a Partition Exchange Load Works with the IM Column Store

A **partition exchange load** is a technique that exchanges a table for a partition. An exchange load is almost instantaneous because it modifies metadata instead of data.

To perform an exchange load, follow these steps:

1. Create a nonpartitoned table, called a *source table*.
2. Load rows into the source table.
3. Exchange an existing table partition, called the *target partition*, with the table.

For the target partition to be populated in the IM column store after the exchange, the source table must be populated in the IM column store *before* the exchange. The following scenarios are possible, depending on the whether the target partition is populated:

- Before the exchange, the target partition is not populated in the IM column store. For example, the partition is empty.

After the exchange, the source table is no longer populated in the IM column store. The source IMCUs are now associated with the target partition.

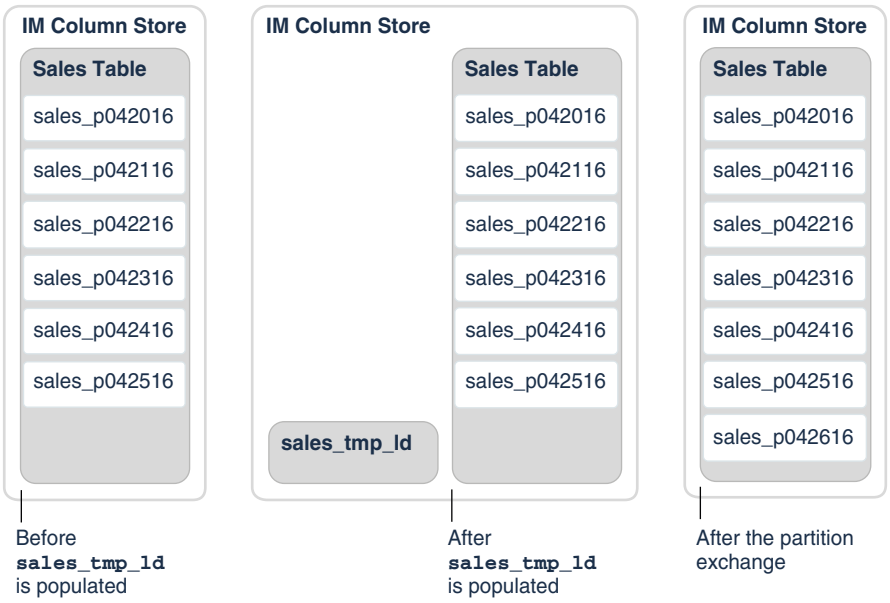
- Before the exchange, the target partition is populated in the IM column store.

  After the exchange, the source table remains populated in the IM column store.

Example 10-1 INMEMORY Partition Exchange Load

In this example, the `sales` table, which is partitioned, has the `INMEMORY` attribute set at the table level. All non-empty partitions in this table are currently populated. The `sales_p042616` partition is currently empty. Your goal is to populate the empty partition `sales_p042616` with data contained in text files. The following figure illustrates the before and after scenarios.

Figure 10-3 Partition Exchange



To perform the exchange, do the following:

1. Create an external table `sales_tmp_ext` using the `CREATE TABLE ... ORGANIZATION EXTERNAL` statement.

   The external table does not reside in the database, and can be in any format for which an access driver is provided. The table is read-only.

2. Create a nonpartitioned table named `sales_tmp_ld` using `CREATE TABLE ... AS SELECT * FROM sales_tmp_ext`.

   The `sales_tmp_ld` table is not external, which means it stores rows in the data files.

3. Set the `INMEMORY` attribute in `sales_tmp_ld` using an `ALTER TABLE` statement.

   The `sales_tmp_ld` table is now marked as `INMEMORY`, but it is not yet populated into the IM column store.

4. Populate `sales_tmp_ld` into the IM column store by forcing a full table scan.

   For example, the following query forces a full scan:

   ```
   SELECT /*+ FULL(s) NO_PARALLEL(s) */ COUNT(*) FROM sales_tmp_ld s;
   ```

5. Exchange the `sales_p042616` partition with the `sales_tmp_ld` table.

   For example, alter the `sales` table as follows:

   ```
   ALTER TABLE sales EXCHANGE PARTITION sales_p042616 WITH TABLE
   sales_tmp_ld;
   ```

After the exchange completes, the `sales_p042616` partition is populated in the IM column store, and the `sales_tmp_ld` is no longer populated.

> **See Also:**
>
> *Oracle Database VLDB and Partitioning Guide* to learn more about partition exchange loads

10.3 When the Database Repopulates the IM Column Store

The database repopulates the IM column store automatically according to an internal algorithm. You can manually disable repopulation, and influence its aggressiveness.

> **Note:**
>
> This section describes automatic repopulation. You can force repopulation manually by using the `DBMS_INMEMORY.REPOPULATE` procedure.

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* to learn about the `DBMS_INMEMORY.REPOPULATE` procedure

10.3.1 Threshold-Based and Trickle Repopulation

Automatic repopulation takes two forms: **threshold-based repopulation** and **trickle repopulation**.

Automatic repopulation always checks stale journal entries and uses double buffering. However, repopulation has different triggers:

- Threshold-based repopulation

The database repopulates IMCUs when the number of changes recorded in the transaction journal reaches an internal staleness threshold. Threshold-based repopulation occurs automatically when `INMEMORY_MAX_POPULATE_SERVERS` initialization parameter is set to a value other than `0`.

- Trickle repopulation

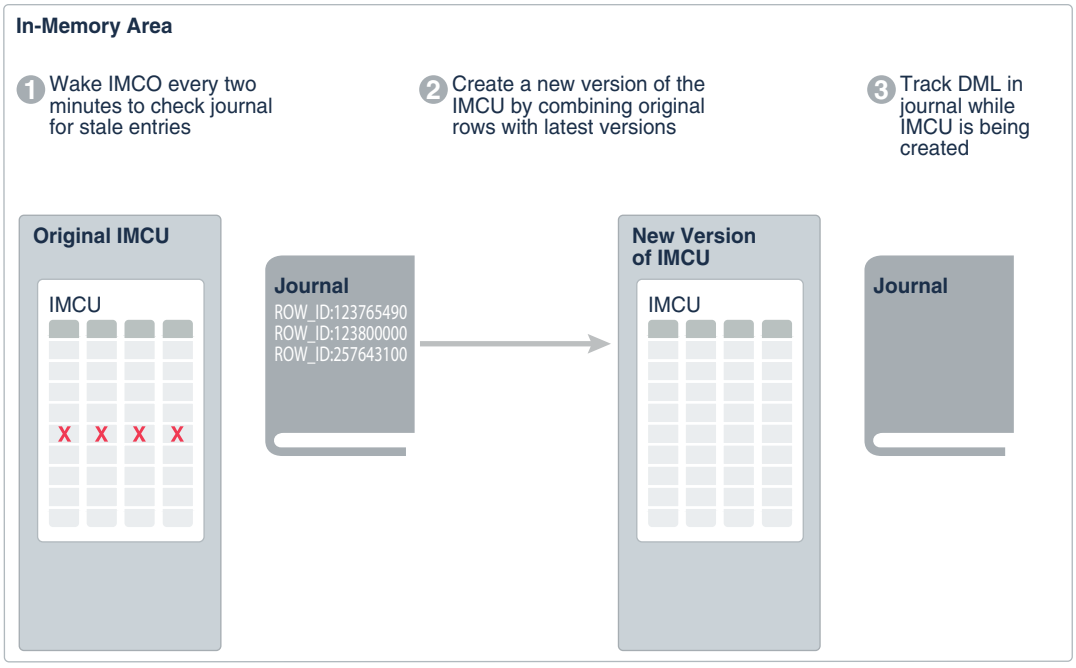
  The IMCO (In-Memory Coordinator) background process periodically checks whether stale rows exist, and then adds IMCUs to a repopulation queue. This mechanism does *not* depend on meeting the staleness threshold. The `INMEMORY_TRICKLE_REPOPULATE_SERVERS_PERCENT` initialization parameter limits the number of background processes used for trickle repopulation. Setting this parameter to `0` disables trickle repopulation.

Trickle repopulation is analogous to Java garbage collection. The mechanism works as follows:

1. IMCO wakes up.
2. IMCO determines whether any population tasks need to be performed, including whether any stale entries exist in the transaction journal associated with an IMCU.
3. If IMCO finds stale entries, then it triggers a Space Management Worker Process (W*nnn*) to create a new version of the IMCU.

   During IMCU creation, the database records the rowids of modified rows in the transaction journal.
4. IMCO sleeps for two minutes, and then returns to Step 1.

Figure 10-4 Trickle Repopulation



For example, a database may be busy for 8 hours per day. Most SMUs contain a small number of transaction journal entries (below the staleness threshold). When the

database is quiet, IMCO wakes up, checks the journals to determine which IMCUs have stale entries, and then uses trickle repopulation to refresh the IMCUs.

See Also:

- "In-Memory Process Architecture"
- "Transaction Journal"
- *Oracle Database Reference* to learn about In-Memory background processes

10.3.2 Factors Affecting Repopulation

The algorithm that triggers repopulation is internal, and depends on several factors.

The principal factors affecting repopulation are as follows:

- Rate of DML changes

  As the number of modified rows increases, the percentage of stale columnar data increases. The transaction journal grows, increasing the need to use the buffer cache to satisfy queries.

- Type of DML operations

  Typically, inserts have less performance overhead than deletes and updates because inserts often go into a new data block.

- Location of modified rows within a data block

  Changes grouped within the same database block or table partition have less effect then changes distributed across an entire table. Versioning every IMCU has a greater impact than versioning a small number of IMCUs.

- Compression level applied to `INMEMORY` objects

  Because of double buffering, tables with higher compression levels incur more query and DML overhead during repopulation. For example, `MEMCOMPRESS FOR CAPACITY HIGH` incurs more overhead than `MEMCOMPRESS FOR DML`.

- Number of active worker processes

  As the number of worker processes increases, more work occurs in parallel. Consequently, the rate of repopulation increases.

See Also:

- "IM Column Store Compression Methods"
- *Oracle Database Reference* to learn about the `INMEMORY_TRICKLE_REPOPULATE_SERVERS_PERCENT` initialization parameter

10.4 Controls for Repopulation of the IM Column Store

Repopulation occurs automatically by default, but you can control its aggressiveness, or disable it altogether.

Initialization Parameters

The following initialization parameters influence background process behavior:

- `INMEMORY_MAX_POPULATE_SERVERS`

  This parameter limits the maximum number of W*nnn* processes available for population and repopulation (threshold-based and trickle). The default value is half the `CPU_COUNT`. This parameter acts as a throttle, preventing these server processes from overloading the rest of the database. Setting this parameter to `0` disables both population and repopulation.

  > **Caution:**
  >
  > Be careful not to set the value of this parameter too high. If it is set close to the number of cores or higher, then no CPU may be available for the rest of the system to run.

- `INMEMORY_TRICKLE_REPOPULATE_SERVERS_PERCENT`

  This parameter limits the percentage of the total population and repopulation processes that perform trickle repopulation. Its effect is to limit the number of IMCUs repopulated through trickle repopulation within a two-minute interval.

  The value for this parameter is a percentage of the `INMEMORY_MAX_POPULATE_SERVERS` value. For example, if `INMEMORY_TRICKLE_REPOPULATE_SERVERS_PERCENT` is `5` percent, and if `INMEMORY_MAX_POPULATE_SERVERS` is `20`, then the IM column store uses an average of 1 core (.05 * 20) for trickle repopulation.

  To increase throughput at the expense of increased background CPU, set this parameter to higher values such as `5` or `10`. A value greater than `50` is not allowed, so that at least half of the `INMEMORY_MAX_POPULATE_SERVERS` processes are available for other tasks.

  Setting this parameter to `0` disables trickle population.

> **See Also:**
>
> - *Oracle Database Reference* to learn about `INMEMORY_TRICKLE_REPOPULATE_SERVERS_PERCENT`
> - *Oracle Database Reference* to learn about `INMEMORY_MAX_POPULATE_SERVERS`

DBMS_INMEMORY.REPOPULATE Procedure

To manually repopulate a table, partition, or subpartition, use the `DBMS_INMEMORY.REPOPULATE` procedure. Only objects that are currently populated in the IM column store are eligible for repopulation.

The following values are possible for the `force` parameter:

- `FALSE` — The database repopulates only IMCUs containing modified rows. This is the default.
- `TRUE` — The database drops the segment, and then rebuilds it. The database increments the statistics and performs all other tasks related to initial population.

For example, IMCU 1 contains rows 1 to 500,000, and IMCU 2 contains rows 500,001 to 1,000,000. A statement modifies row 600,000. When `force` is `FALSE`, the database only repopulates IMCU 2. When `force` is `TRUE`, the database repopulates both IMCUs.

Consider further that the `INMEMORY_VIRTUAL_COLUMNS` initialization parameter is set to `ENABLE`, and an application creates a new virtual column. When `force` is `FALSE`, the database only repopulates IMCU 2 with the new column. When `force` is `TRUE`, the database repopulates both IMCUs with the new column.

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* to learn more about `DBMS_INMEMORY.REPOPULATE`

10.5 Optimizing Trickle Repopulation: Tutorial

In this tutorial, you increase the percentage of background processes available for trickle repopulation.

Assumptions

This tutorial assumes the following:

- The IM column store is enabled.
- You want to devote more CPU to the Space Management Worker Processes (W*nnn*) that perform trickle repopulation.
- The database server has 12 CPU cores.

To increase the aggressiveness of repopulation:

1. In SQL*Plus or SQL Developer, log in to the database as a user with administrative privileges.
2. Show the settings for the initialization parameters relating to repopulation (sample output included):

```
SHOW PARAMETER POPULATE_SERVERS

NAME                                 TYPE        VALUE
```

```
-----------------------------------        ----------- -----------
inmemory_max_populate_servers              integer     12
inmemory_trickle_repopulate_servers_percent integer     1
```

The preceding output indicates that 12 cores are available for population and repopulation tasks. The `INMEMORY_TRICKLE_REPOPULATE_SERVERS_PERCENT` is 1% of the `INMEMORY_MAX_POPULATE_SERVERS` value. Of the server processes available for population and repopulation tasks, the IM column store can use a maximum of .12 CPU cores (.01 * 12) for trickle repopulation.

3. Increase the trickle repopulation maximum to 25% of the `INMEMORY_MAX_POPULATE_SERVERS` initialization parameter value.

   For example, use the following statement:

```
ALTER SYSTEM
  SET INMEMORY_TRICKLE_REPOPULATE_SERVERS_PERCENT=25
  SCOPE=BOTH;
```

   As a result, the IM column store now uses a maximum of 3 CPU cores (.25 * 12) for trickle repopulation, out of a total of 12 that are available for population and repopulation work.

See Also:

- *Oracle Database Reference* to learn about `INMEMORY_TRICKLE_REPOPULATE_SERVERS_PERCENT`
- *Oracle Database Reference* to learn about `INMEMORY_MAX_POPULATE_SERVERS`

Part IV

High Availability and the IM Column Store

This part explains how to use the IM column store with high availability features such as In-Memory FastStart (IM FastStart), Oracle Data Guard, and Oracle Real Application Clusters (Oracle RAC).

This section contains the following topics:

> **See Also:**
>
> The MAA white paper Oracle Database In-Memory High Availability Best Practices

11

Managing IM FastStart for the IM Column Store

When the IM column store is enabled, In-Memory FastStart (IM FastStart) enables the database to open faster by storing columnar data on disk.

This chapter contains the following topics:

11.1 About IM FastStart

IM FastStart optimizes the population of database objects in the IM column store by storing IMCUs directly on disk.

The database can read from the IM FastStart area after instance failure and recovery, or during duplication to a different Oracle RAC instance.

> **Note:**
>
> IM FastStart is not supported in a standby database, which is read-only.

This section contains the following topics:

11.1.1 Purpose of IM FastStart

The IM column store is populated whenever a database instance restarts, which can be a slow operation that is I/O-intensive and CPU-intensive.

When IM FastStart is enabled, the database periodically saves a copy of columnar data to disk for faster repopulation during instance restarts. If the database re-opens after being closed, then the database reads columnar data from the FastStart area, and then populates it into the IM column store, ensuring that all transactional consistencies are maintained.

An IM FastStart tablespace requires intermittent I/O while the database is open and operational. The performance gain occurs when the database re-opens because the database avoids the CPU-intensive compression and formatting of data.

11.1.2 How IM FastStart Works

A FastStart area is a designated tablespace where IM FastStart stores and manages data for `INMEMORY` objects. Oracle Database manages the FastStart tablespace without DBA intervention.

Only one FastStart area, and one designated FastStart tablespace, is allowed for each PDB. You cannot alter or drop the tablespace while it is the designated IM FastStart tablespace. In an Oracle RAC database, all nodes share the FastStart data.

Enable a FastStart tablespace using the `DBMS_INMEMORY_ADMIN.FASTSTART_ENABLE` procedure. The Space Management Worker Processes (W*nnn*) creates an empty SecureFiles LOB named `SYSDB`*`instance_name`*`_LOBSEG$`.

> **Note:**
>
> Enabling the IM FastStart area is not sufficient to create the FastStart area. Data population or repopulation is required.

> **See Also:**
>
> - "Space Management Worker Processes (Wnnn)"
> - "About Repopulation of the IM Column Store"
> - *Oracle Database PL/SQL Packages and Types Reference* to learn about the `DBMS_INMEMORY_ADMIN.FASTSTART_ENABLE` procedure

11.1.2.1 How the Database Manages the FastStart Area

During the first population or repopulation after the FastStart area is enabled, the database creates the FastStart area.

The database manages the FastStart area automatically as follows:

- Whenever population or repopulation of an object occurs, the database writes its columnar data to the FastStart area.

  > **Note:**
  >
  > The database writes segments from encrypted tablespaces to the FastStart area only if the FastStart tablespace is also encrypted.

  The Space Management Worker Processes (W*nnn*) write IMCUs (not IMEUs or SMUs) to the SecureFiles LOB named `SYSDB`*`instance_name`*`_LOBSEG$`. The database writes FastStart metadata to the `SYSAUX` tablespace, which must be online.

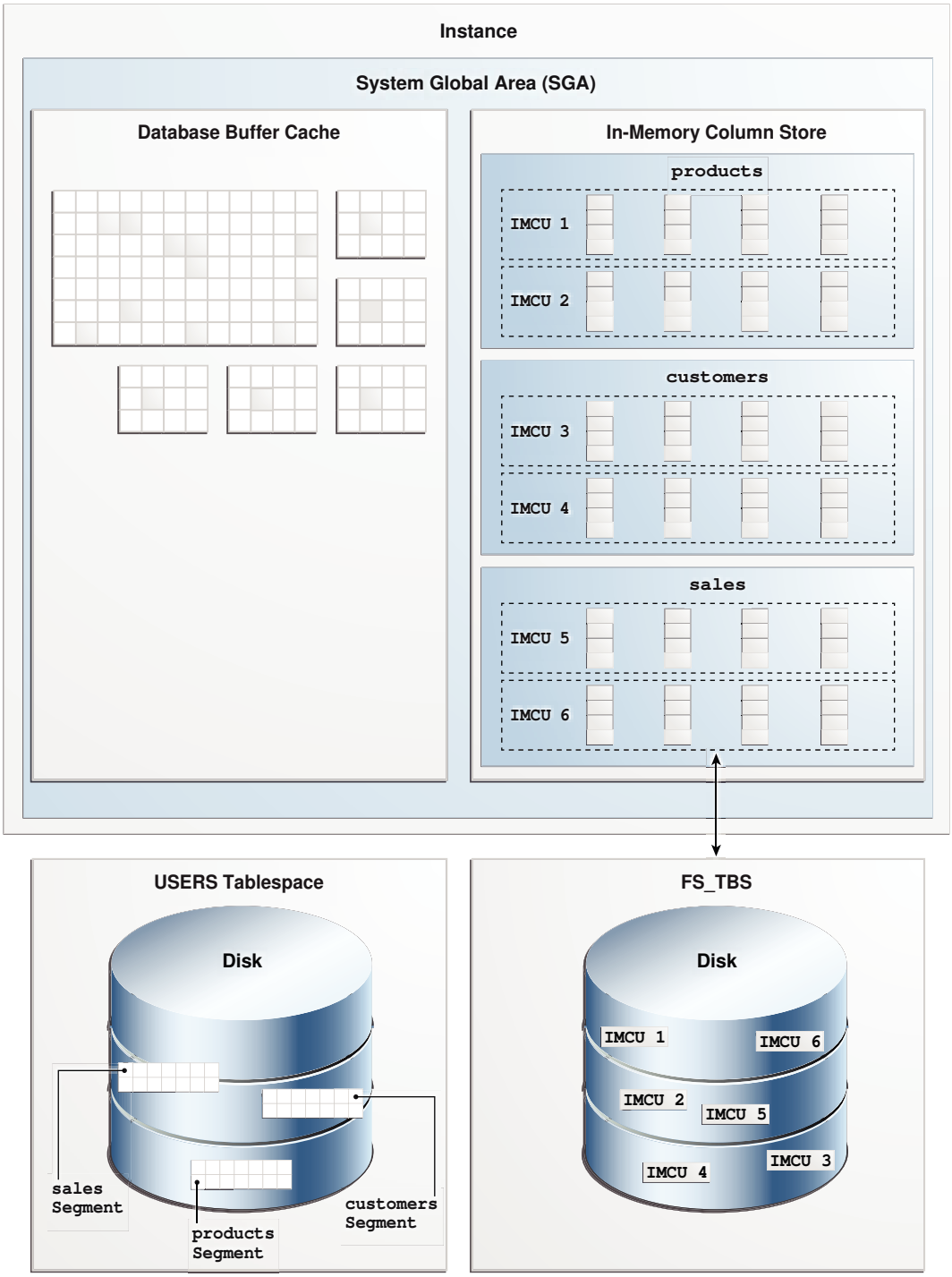
  Depending on how much DML activity occurs for a CU, a lag can exist between the CUs in the FastStart area and the CUs in the IM column store. The “hotter” a CU is, the less frequently the database populates it in the IM column store and writes it to the FastStart area. If the database crashes, then some CUs that were populated in the IM column store may not exist in the FastStart area.

> **Note:**
>
> If the FastStart area becomes temporarily inaccessible, then In-Memory population is unaffected.

- If you define an ADO policy on a segment, then the database manages the segment in the FastStart area based on the rule in the policy. For example, if ADO specifies that an object changes its attribute to `NO INMEMORY` based on a policy, then the IM column store removes its data from the FastStart area.
- If the attribute of a populated object is changed to `NOINMEMORY`, then the database automatically removes its IMCUs from the FastStart area.
- If the FastStart tablespace runs out of space, then the database uses an internal algorithm to drop the oldest segments, and continues writing to the FastStart area. If no space remains, then the database stops writing to the FastStart area.

The following figure shows `products`, `customers`, and `sales` populated in the IM column store.

Figure 11-1 FastStart Area



When the FastStart area is enabled, the database also writes the IMCUs for these segments to the FastStart area in `fs_tbs`. If the database re-opens or if the instance restarts, then the database can validate the IMCUs for modifications to ensure the transactional consistency, and reuse the IMCUs. Regardless of whether the FastStart

area is enabled, the database stores data blocks and segments on disk in the `users` tablespace.

> **Note:**
>
> You cannot manually force the IM column store to write data to the FastStart tablespace.

> **See Also:**
>
> - "Enabling ADO for the IM Column Store"
> - "FastStart Area in Oracle RAC"
> - "About Repopulation of the IM Column Store"
> - *Oracle Database VLDB and Partitioning Guide* to learn more about ADO

11.1.2.2 How the Database Reads from the FastStart Area

The FastStart area defines *what* data is loaded when the database reopens, but not *when* it is loaded. Population is controlled by the priority settings.

When the database reopens, the standard `PRIORITY` rules determine population. For example, the database populates objects with `PRIORITY NONE` on demand. Objects with priority `CRITICAL` are higher in the automatic population queue than objects with priority `LOW`.

For example, in a single-instance database, the `sales`, `customers`, and `product` tables are populated with `PRIORITY NONE` in the IM column store. At every repopulation, the database saves the IMCUs for these tables to the FastStart area. Assume that the instance unexpectedly terminates. When you reopen the database, the IM column store is empty. If a query scans the `sales`, `customers`, or `product` table, then the database loads the IMCUs for this table from the FastStart area into the IM column store.

In most cases, the FastStart area increases the speed of population. However, if any CU stored in the FastStart area reaches an internal threshold of DML activity, then the database populates the row data from data files instead of from the FastStart area.

> **See Also:**
>
> - "Prioritization of In-Memory Population"
> - "FastStart Area in Oracle RAC"
> - *Oracle Database SQL Language Reference* for `INMEMORY` clause semantics

11.2 Enabling IM FastStart for the IM Column Store

Specify a tablespace for the FastStart area using the `DBMS_INMEMORY_ADMIN.FASTSTART_ENABLE` procedure.

Optionally, set the logging mode of the LOB created for the FastStart area. If the `nologging` parameter is set to `TRUE` (default), then the database creates the LOB with the `NOLOGGING` option. If `nologging` is set to `FALSE`, then the database creates the FastStart LOB with the `LOGGING` option.

Prerequisites

To create a FastStart area, you must meet the following prerequisites:

- The tablespace that will be designated as the FastStart area must exist.
- This tablespace must have enough space to store data for the IM column store, and it must not contain any other data before you designate it as the FastStart area. Oracle recommends that you create the FastStart tablespace with twice the size of the `INMEMORY_SIZE` setting.
- You must have administrator privileges.

To create the IM FastStart area:

1. In SQL*Plus or SQL Developer, log in to the database as a user with the necessary privileges.
2. Use the `DBMS_INMEMORY_ADMIN.FASTSTART_ENABLE` procedure.

Example 11-1 Designating an IM FastStart Area

This example creates a tablespace and designates it as the FastStart area.

1. In SQL*Plus or SQL Developer, log in to the database as a user with administrative privileges.
2. Create a tablespace named `fs_tbs`:

```
CREATE TABLESPACE fs_tbs
  DATAFILE 'fs_tbs.dbf' SIZE 500M REUSE
  AUTOEXTEND ON NEXT 500K MAXSIZE 1G;
```

3. Enable IM FastStart, and designate the `fs_tbs` tablespace as the FastStart area, using the default `NOLOGGING` option for the FastStart LOB:

```
EXEC DBMS_INMEMORY_ADMIN.FASTSTART_ENABLE('fs_tbs');
```

4. Query the status and size of the FastStart area:

```
COL TABLESPACE_NAME FORMAT a15

SELECT TABLESPACE_NAME, STATUS,
       ( (ALLOCATED_SIZE/1024) / 1024 ) AS ALLOC_MB,
       ( (USED_SIZE/1024) / 1024 ) AS USED_MB
FROM   V$INMEMORY_FASTSTART_AREA;
```

```
TABLESPACE_NAME STATUS                ALLOC_MB    USED_MB
--------------- -------------------- ---------- ----------
FS_TBS          ENABLE                     500      .0625
```

At this stage, no user data is in the FastStart area.

5. Query the logging mode of the FastStart LOB:

```
COL SEGMENT_NAME FORMAT a20
SELECT SEGMENT_NAME, LOGGING
FROM   DBA_LOBS
WHERE  TABLESPACE_NAME = 'FS_TBS';

SEGMENT_NAME         LOGGING
-------------------- -------
SYSDBIMFS_LOBSEG$    NO
```

6. Force the IM column store to repopulate any currently populated objects.

   The following queries force the repopulation of the `sales`, `products`, and `customers` tables:

```
SELECT /*+ FULL(s) NO_PARALLEL(s) */ COUNT(*) FROM sh.sales s;
SELECT /*+ FULL(p) NO_PARALLEL(p) */ COUNT(*) FROM sh.products p;
SELECT /*+ FULL(c) NO_PARALLEL(c) */ COUNT(*) FROM sh.customers c;
```

7. Query the size of the FastStart area:

```
COL TABLESPACE_NAME FORMAT a15

SELECT TABLESPACE_NAME, STATUS,
       ( (ALLOCATED_SIZE/1024) / 1024 ) AS ALLOC_MB,
       ( (USED_SIZE/1024) / 1024 ) AS USED_MB
FROM   V$INMEMORY_FASTSTART_AREA;

TABLESPACE_NAME STATUS                ALLOC_MB    USED_MB
--------------- -------------------- ---------- ----------
FS_TBS          ENABLE                     500       2.25
```

Now the same query shows that 2.25 MB of the FastStart area has been filled.

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* to learn about the `DBMS_INMEMORY_ADMIN` package

11.3 Retrieving the Name of the Current IM FastStart Tablespace

Obtain the name of the tablespace that is currently designated as the FastStart area by querying `V$INMEMORY_FASTSTART_AREA` view.

If no FastStart tablespace is enabled, then the `STATUS` column shows `NOT ENABLED`; otherwise, the column shows the tablespace name.

Prerequisites

To retrieve the name of the FastStart tablespace, you must have administrator privileges.

To retrieve the name of the FastStart tablespace:

1. In SQL*Plus or SQL Developer, log in to the database as a user with the necessary privileges.
2. Query the `V$INMEMORY_FASTSTART_AREA` view.

Example 11-2 Getting the Name of the Current IM FastStart Tablespace

This example queries the name and status of the FastStart tablespace (sample output included):

```
COL TABLESPACE_NAME FORMAT a20

SELECT TABLESPACE_NAME, STATUS
FROM   V$INMEMORY_FASTSTART_AREA;

TABLESPACE_NAME      STATUS
-------------------- ----------
FS_TBS               ENABLE
```

> **See Also:**
>
> *Oracle Database Reference* to learn about the `V$INMEMORY_FASTSTART_AREA` view

11.4 Migrating the FastStart Area to a Different Tablespace

You can migrate the FastStart area to a different tablespace by running the `FASTSTART_MIGRATE_STORAGE` procedure in the `DBMS_INMEMORY_ADMIN` package.

In a non-CDB or PDB, you can designate only one tablespace at a time as the FastStart area.

Prerequisites

To migrate a FastStart area, you must meet the following prerequisites:

- The tablespace that will be designated as the new FastStart area must exist.
- This tablespace must have enough space to store data for the IM column store, and it must not contain any other data before it is designated as the FastStart area.
- You must have administrator privileges.

To migrate the IM FastStart area:

1. In SQL*Plus or SQL Developer, log in to the database as a user with the necessary privileges.
2. Run the `DBMS_INMEMORY_ADMIN.FASTSTART_MIGRATE_STORAGE` procedure.

Example 11-3 Migrating the FastStart Area to a Different Tablespace

This example migrates the IM FastStart area to the `new_fs_tbs` tablespace.

1. In SQL*Plus or SQL Developer, log in to the database as a user with administrative privileges.
2. Query the name of the current FastStart tablespace:

```
COL TABLESPACE_NAME FORMAT a15

SELECT TABLESPACE_NAME, STATUS
FROM   V$INMEMORY_FASTSTART_AREA;

TABLESPACE_NAME STATUS
--------------- -----------
FS_TBS          ENABLE
```

3. Create a tablespace named `new_fs_tbs`:

```
CREATE TABLESPACE new_fs_tbs
  DATAFILE 'new_fs_tbs.dbf' SIZE 500M REUSE
  AUTOEXTEND ON NEXT 500K MAXSIZE 1G;
```

4. Migrate the FastStart area to the new tablespace:

```
EXEC DBMS_INMEMORY_ADMIN.FASTSTART_MIGRATE_STORAGE('new_fs_tbs');
```

5. Query the name of the current FastStart tablespace:

```
TABLESPACE_NAME      STATUS
-------------------- --------------------
NEW_FS_TBS           ENABLE
```

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* to learn about the `FASTSTART_MIGRATE_STORAGE` procedure

11.5 Disabling IM FastStart for the IM Column Store

When you disable IM FastStart, the database no longer maintains the FastStart area. The database does not use IM FastStart to populate the IM column store when the database reopens.

Prerequisites

To disable the FastStart area, the following conditions must be true:

- The FastStart area must be enabled.
- You must have administrator privileges.

To disable the FastStart tablespace:

1. In SQL*Plus or SQL Developer, log in to the database as a user with the necessary privileges.
2. Query `V$INMEMORY_FASTSTART_AREA` to confirm that the IM FastStart area is enabled.
3. Execute the `DBMS_INMEMORY_ADMIN.FASTSTART_DISABLE` procedure.
4. Optionally, drop the FastStart tablespace.

Example 11-4 Disabling IM FastStart

This example disables the IM FastStart area, and then drops the `fs_tbs` tablespace.

1. In SQL*Plus or SQL Developer, log in to the database as a user with administrative privileges.
2. Query the status of the FastStart area:

```
COL TABLESPACE_NAME FORMAT a15

SELECT TABLESPACE_NAME, STATUS
FROM   V$INMEMORY_FASTSTART_AREA;

TABLESPACE_NAME STATUS
--------------- -----------
FS_TBS          ENABLE
```

3. Disable the FastStart area:

```
EXEC DBMS_INMEMORY_ADMIN.FASTSTART_DISABLE;
```

4. Query the status of the FastStart area:

```
SELECT TABLESPACE_NAME, STATUS
FROM   V$INMEMORY_FASTSTART_AREA;

TABLESPACE_NAME      STATUS
-------------------- --------------------
INVALID_TABLESPACE   DISABLE
```

When IM FastStart is not enabled, the value of `TABLESPACE_NAME` is `INVALID_TABLESPACE` and the value of `STATUS` is `DISABLE`.

5. Drop the former FastStart tablespace:

```
DROP TABLESPACE fs_tbs INCLUDING CONTENTS AND DATAFILES;
```

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* to learn about the `FASTSTART_DISABLE` procedure

12
Deploying IM Column Stores in Oracle RAC

This chapter explains how to enable IM column stores in an Oracle Real Application Clusters (Oracle RAC) environment, and configure objects for population.

This section contains the following topics:

12.1 Overview of Database In-Memory and Oracle RAC

Every Oracle RAC node has its own In-Memory (IM) column store. By default, populated objects are distributed across all IM column stores in the cluster.

Oracle recommends that you size the IM column stores equally on every Oracle RAC node. If an Oracle RAC node does not require an IM column store, then set the `INMEMORY_SIZE` parameter to `0`.

It is possible to have completely different objects populated on every node, or to have larger objects distributed across all of the IM column stores in the cluster. On Oracle Engineered Systems, it is also possible for the same objects to appear in the IM column store on every node. The distribution of objects across the IM column stores in a cluster is controlled by two subclauses to the `INMEMORY` attribute: `DISTRIBUTE` and `DUPLICATE`.

In an Oracle RAC environment, an object that only has the `INMEMORY` attribute specified is automatically distributed across the IM column stores in the cluster. You can use the `DISTRIBUTE` clause to specify how an object is distributed across the cluster. By default, the type of partitioning used (if any) determines how the object is distributed. If the object is not partitioned, then it is distributed by rowid range. Alternatively, you can specify the `DISTRIBUTE` clause to override the default behavior.

On an Oracle Engineered System, you can duplicate or mirror populated objects across the IM column stores in the cluster. This technique provides the highest level of redundancy. The `DUPLICATE` clause controls how an object is duplicated. If you specify only `DUPLICATE`, then one mirrored copy of the data is distributed across the IM column stores in the cluster. To duplicate the entire object in each IM column store, specify `DUPLICATE ALL`.

> **Note:**
>
> When you deploy Oracle RAC on a non-Engineered System, the `DUPLICATE` clause is treated as `NO DUPLICATE`.

12.1.1 Multiple IM Column Stores

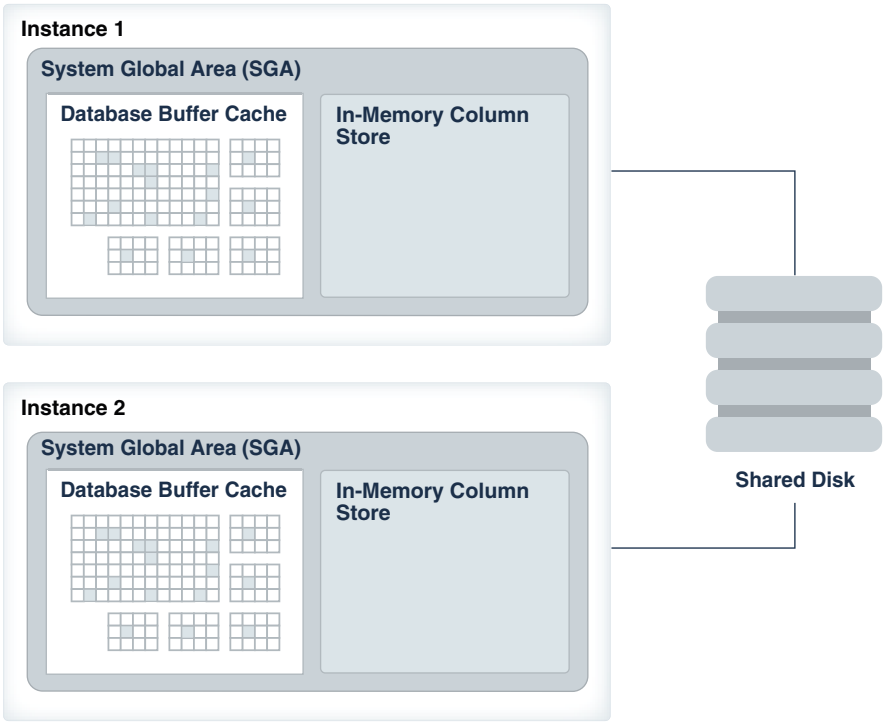
In Oracle RAC, each database instance has its own IM column store.

Conceptually, the IM column store in Oracle RAC environment uses a shared-nothing architecture. On each database instance, you size and manage the IM column stores

separately. The database instances do *not* use Cache Fusion to transfer IMCUs back and forth.

Figure 12-1 IM Column Stores in an Oracle RAC Database

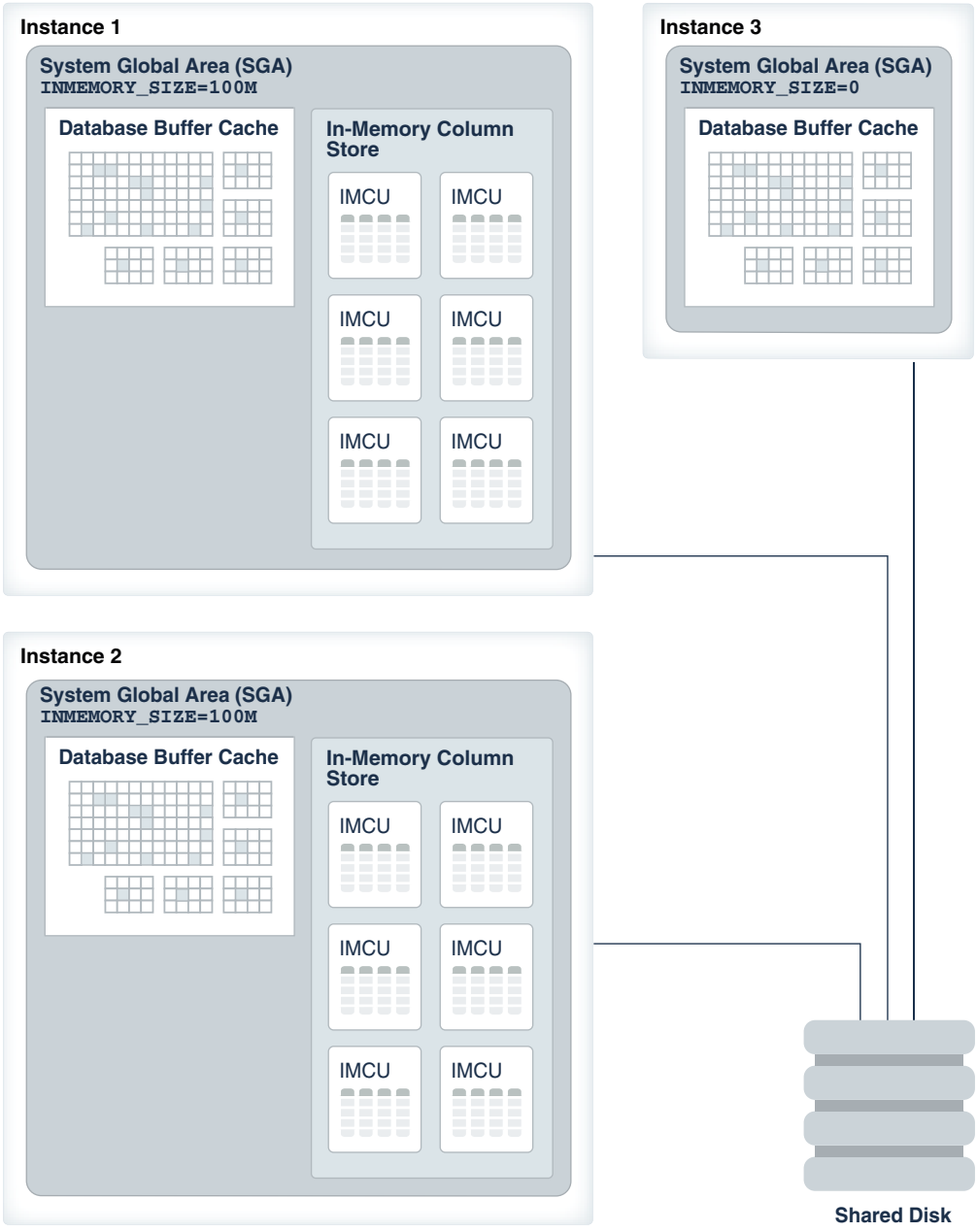
This figure shows a two-node Oracle RAC cluster. Each instance has a separately configured IM column store.



Oracle recommends that you set the size of the IM column stores on every Oracle RAC node to an equal value. For example, you might assign every IM column store 100 GB of memory. For any node that does not require an IM column store, set the `INMEMORY_SIZE` initialization parameter on this node to 0.

Figure 12-2 Three-Node Oracle RAC Database with Two IM Column Stores

In this example, instance 1 and instance 2 both have IM column stores. Instance 3 does not require an IM column store, so the `INMEMORY_SIZE` initialization parameter on this node is set to 0.



See Also:

- "Enabling the IM Column Store for a Database"
- *Oracle Database Reference* for more information about the `INMEMORY_SIZE` initialization parameter
- *Oracle Real Application Clusters Administration and Deployment Guide* for an introduction to Oracle RAC

12.1.2 Distribution and Duplication of Columnar Data in Oracle RAC

When `INMEMORY` is specified, the `DISTRIBUTE` and `DUPLICATE` keywords control the distribution of objects.

Oracle RAC provides multiple distribution options. You can have different objects populated on each node, or have larger objects distributed across all IM column stores in the Oracle RAC cluster. You can also have the same objects populated in the IM column store on every node (Oracle Engineered Systems only).

> **Note:**
>
> If a table is currently populated in the IM column store, and if you change any `INMEMORY` attribute of the table *other than* `PRIORITY`, then the database evicts the table from the IM column store. The repopulation behavior depends on the `PRIORITY` setting.

This section contains the following topics:

12.1.2.1 Distribution of Columnar Data in Oracle RAC

The `DISTRIBUTE` clause of `INMEMORY` controls how table data in the IM column store is distributed across Oracle RAC instances.

When the default option of `AUTO` is set, the Oracle RAC instances distribute data automatically. While populating a segment, Space Management Slave Processes (W*nnn*) processes attempt to put an equal amount of data on each instance. Distribution depends on access patterns and object size. Alternatively, you can manually specify how the database must distribute partitions, subpartitions, or rowid ranges across instances.

Equal data distribution is important for performance. The goal is for parallel query processes to work on equal data set sizes so that they all finish in the minimum amount of time. If data distribution is skewed, then a long-running process delays the completion of the query.

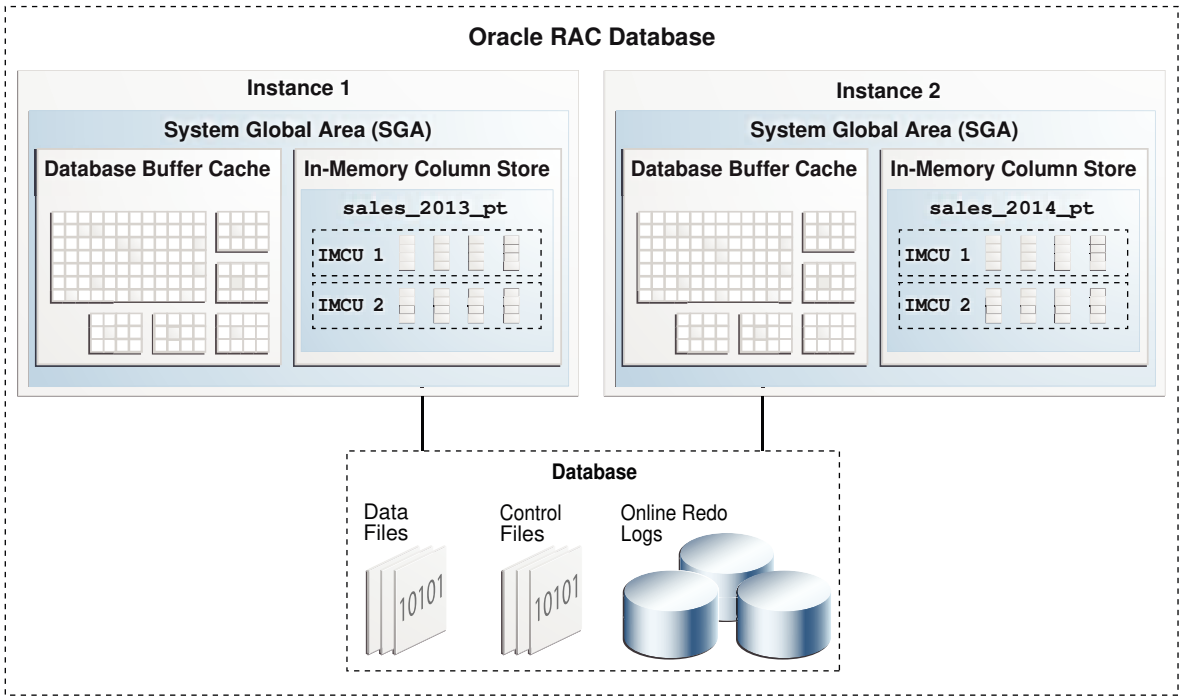
If an Oracle RAC instance fails, then the IMCUs on the failed instance are unavailable. Consequently, a query that needs data stored in the inaccessible IMCUs must read it from somewhere else: the database buffer cache, flash storage, disk, or mirrored IMCUs in other IM column stores.

The `DBA_TABLES.INMEMORY_DISTRIBUTE` column indicates how IMCUs are distributed. When the `AUTO` option is set, the column value is `AUTO-DISTRIBUTE`.

Example 12-1 Default Distribution

This example shows the database distributing a `sales` table that contains only partitions: `sales_2013_pt` and `sales_2014_pt`. The database automatically places the `sales_2013_pt` partition in Instance 1, and `sales_2014_pt` in Instance 2.

Figure 12-3 Automatic In-Memory Distribution in Oracle RAC



This section contains the following topics:

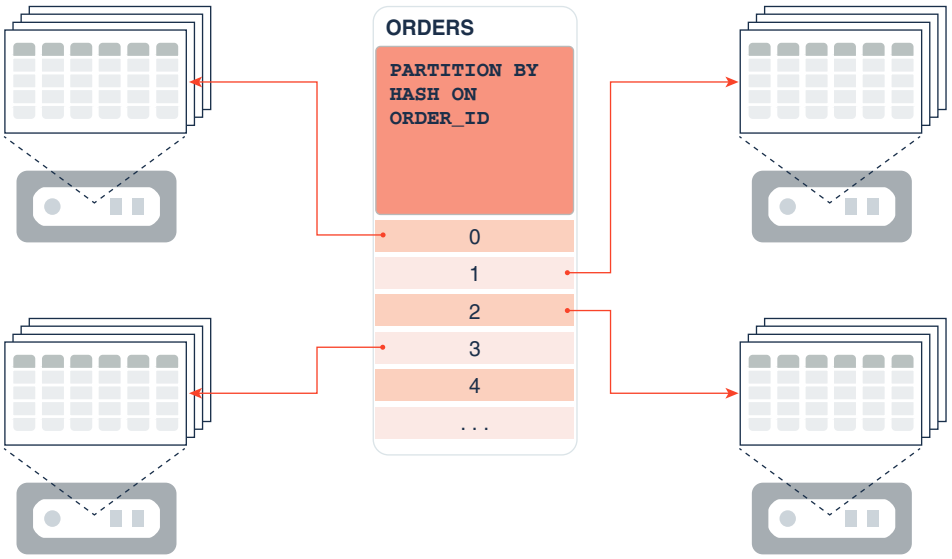
> **See Also:**
>
> - "Space Management Worker Processes (Wnnn)"
> - *Oracle Real Application Clusters Administration and Deployment Guide* to learn how to manage the IM column store on Oracle RAC
> - *Oracle Database SQL Language Reference* to learn more about the `DISTRIBUTE` clause
> - *Oracle Database Reference* to learn about the W*nnn* background processes

12.1.2.1.1 Distribution by Partition

You can use the `DISTRIBUTE BY PARTITION` clause to distribute data in partitions to different Oracle RAC instances.

This technique is ideal for hash partitions. For example, to distribute partitions in the `orders` table equally, you could partition by hash on the `order_id` column. As shown in the following figure, Oracle Database distributes partitions among four instances by hashing on the `order_id` column.

Figure 12-4 Distributing Partitions by Hash



This technique is suitable for other partitioning schemes when the partitions are uniformly accessed. The `DISTRIBUTE BY PARTITION` clause also supports partition-wise joins.

> **Note:**
>
> If your partitioned strategy results in a large data skew, that is, one partition is much larger than the others, then Oracle recommends that you override the default distribution (`BY PARTITION`) by manually specifying `DISTRIBUTE BY ROWID RANGE`.

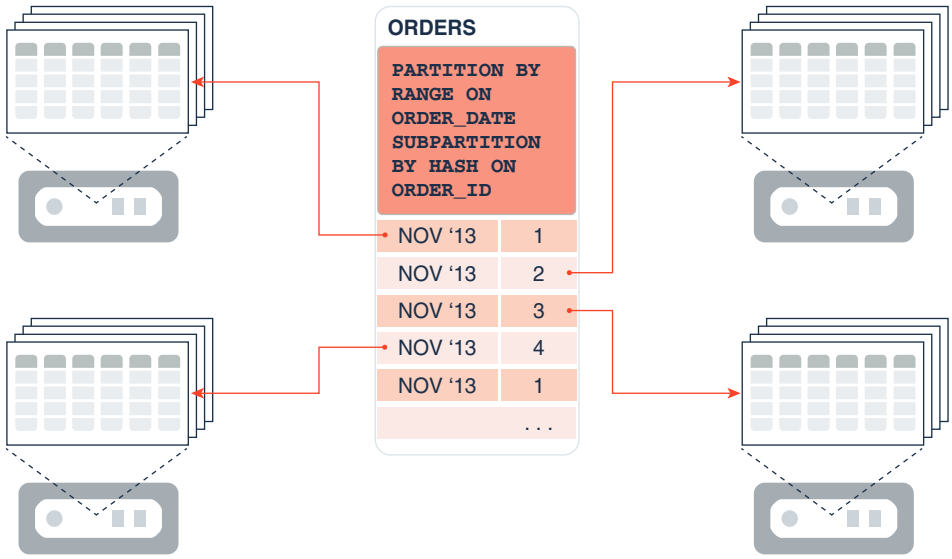
> **See Also:**
>
> - *Oracle Database SQL Language Reference* to learn more about the `DISTRIBUTE BY PARTITION` subclause
> - *Oracle Database VLDB and Partitioning Guide* for an introduction to partitioned tables

12.1.2.1.2 Distribution by Subpartition

In tables with a composite partitioning scheme, you can use the `DISTRIBUTE BY SUBPARTITION` clause to distribute data in subpartitions to different instances.

This technique is ideal for hash subpartitions. For example, to distribute partitions in the `orders` table equally, you could partition by range on the `order_date` column and by hash on the `order_id` column.

Figure 12-5 Distributing Partitions by Range and Subpartitions by Hash



This technique is suitable for other partitioning schemes when the subpartitions are uniformly accessed. The `DISTRIBUTE BY PARTITION ... SUBPARTITION` clause also supports partition-wise joins.

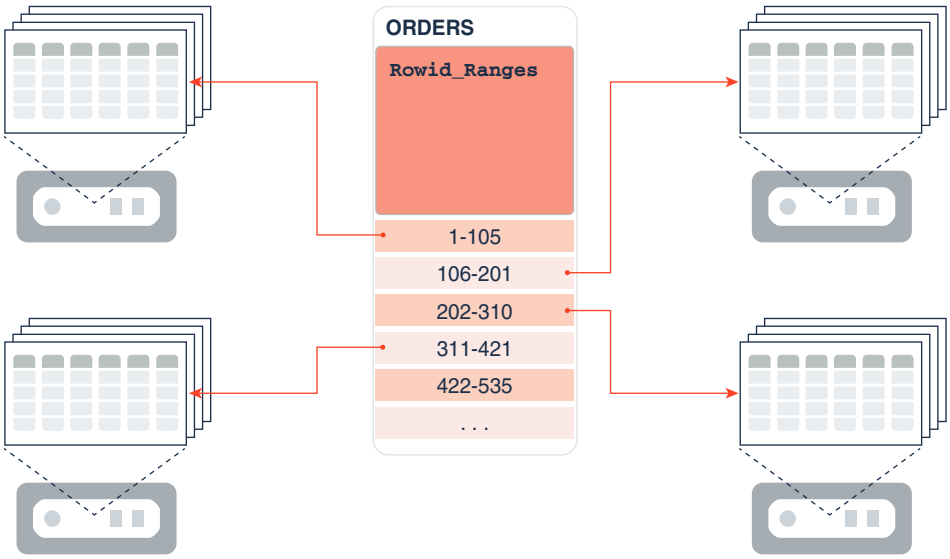
> **See Also:**
>
> - *Oracle Database VLDB and Partitioning Guide* to learn more about composite partitioning
> - *Oracle Database SQL Language Reference* to learn more about the `DISTRIBUTE BY PARTITION ... SUBPARTITION` subclause

12.1.2.1.3 Distribution by Rowid Range

You can use the `DISTRIBUTE BY ROWID RANGE` clause to distribute data in specific ranges of rowids to different Oracle RAC instances.

This technique distributes IMCUs by uniform hash on the first rowid. For example, Oracle Database might distribute rows 1-105 in the `orders` table to one database instance, rows 106-121 to a different instance, and so on.

Figure 12-6 Distribution by Rowid Range



The rowid distribution technique is most useful for nonpartitioned tables. However, if your partitioned strategy results in a large data skew, for example, one partition is much larger than the others, then Oracle recommends overriding the default distribution (`BY PARTITION`) by manually specifying `DISTRIBUTE BY ROWID RANGE`.

> **See Also:**
>
> *Oracle Database SQL Language Reference* to learn more about the `DISTRIBUTE BY ROWID RANGE` subclause

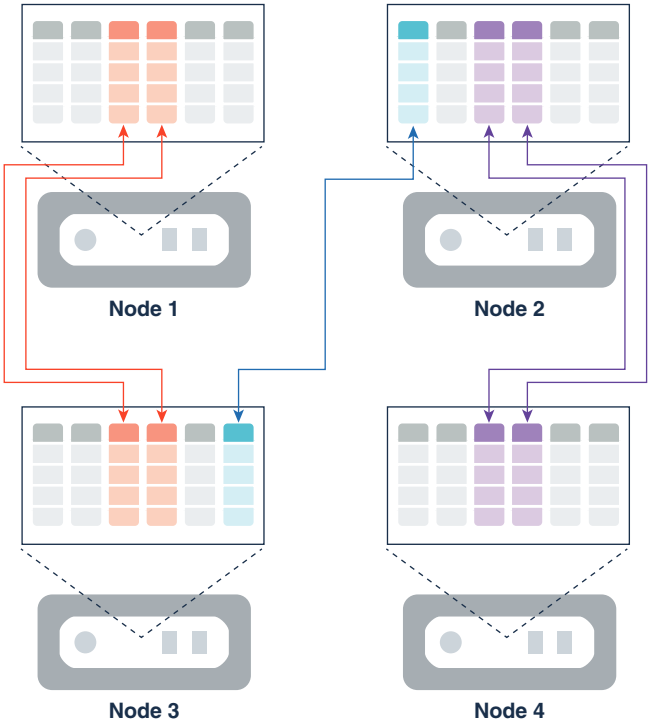
12.1.2.2 Duplication of Columnar Data in Oracle RAC

The `DUPLICATE` clause controls how an Oracle RAC database duplicates columnar data across Oracle RAC instances.

> **Note:**
>
> The `DUPLICATE` clause is only available with Oracle RAC on an Oracle Engineered System. When Oracle RAC does *not* run on an Oracle Engineered System, the `DUPLICATE` clause is functionally equivalent to `NO DUPLICATE`.

To provide IM column store fault tolerance, you may choose to mirror the IMCUs. With IMCU mirroring, the same IMCU resides in multiple IM column stores. This technique is analogous to storage mirroring.

Figure 12-7 Duplication of IMCUs in Oracle RAC



Using the `DUPLICATE` clause to mirror data at the tablespace or object (table, partition, or subpartition) level provides the following benefits:

- Provides fault tolerance because if one node fails, then the mirrored columnar data is accessible from a different node
- Improves performance because queries can access data locally, thus avoiding buffer cache or disk access

  For example, in a star query, the fact table might be partitioned, whereas the dimension tables use `DUPLICATE ALL`. In this scenario, all joins take place fully on the local nodes.
- Enhances manageability because you can duplicate a subset of objects

  For example, you can duplicate this year's partitions while leaving others partitions from the same table not duplicated.

A disadvantage of IMCU mirroring is that when an object is duplicated *n* times, its memory requirements increase by a factor of *n*. For example, a 500 MB table that is duplicated in 4 instances occupies a total of 2000 MB of memory.

This section contains the following topics:

> **See Also:**
>
> *Oracle Database SQL Language Reference* to learn more about the `DUPLICATE` clause

12.1.2.2.1 DUPLICATE Clause in Oracle RAC

The `DUPLICATE` clause specifies that the database maintain a copy of every IMCU in a second database instance. Thus, the same segment is populated in exactly two Oracle RAC instances.

For each object, one IMCU is primary. A secondary IMCU resides on a different database instance. The database can use either copy to satisfy a query. If the database instance with the primary copy of the IMCU fails, then the database can use the surviving IMCU to satisfy the query.

For example, you might specify `DUPLICATE` for the partition `sales_q1_2014`. The IM column stores in instance 1 and instance 2 both have an identical copy of the data. If instance 1 terminates, then the IM column store on instance 2 can satisfy requests for `sales_q1_2014`.

> **See Also:**
>
> - "In-Memory Compression Units (IMCUs)"
> - *Oracle Database SQL Language Reference* to learn more about the `DUPLICATE` clause

12.1.2.2.2 DUPLICATE ALL Clause in Oracle RAC

The `DUPLICATE ALL` clause specifies that every In-Memory object is mirrored on every database instance.

This setting provides the highest level of redundancy and provides linear scalability because queries can execute completely within a single node. For example, every IMCU for the `sales` table is populated in the IM column store in instance 1, instance 2, and instance 3. Thus, any database instance can retrieve the data requested by a query of `sales`.

A consequence of the `DUPLICATE ALL` clause is that the `DISTRIBUTE` subclause has no application because all IMCUs for the object are distributed. You specify duplication at the object level, which means that not all objects in the IM column store required the `DUPLICATE ALL` clause.

The primary advantages of the `DUPLICATE ALL` technique are:

- High availability

  When you use `DUPLICATE ALL` clause for all In-Memory objects, an Oracle RAC database with *n* instances can sustain *n*-1 Oracle RAC instance failures. If you need take one database instance out of service for maintenance, then critical data is available in at least one IM column store. The only scenario in which all data is inaccessible is a failure of all database instances in the cluster.

- Performance of star queries

  If queries join smaller dimension tables to a large partitioned fact table, then you can use `DUPLICATE ALL` to mirror dimension tables in every Oracle RAC instance. The fact table is distributed by partition or subpartition. In this strategy, the IM

column store in every database instance has the data necessary for a star join. This technique is analogous to a partition-wise join because the entire dimension table is populated in every IM column store.

> **See Also:**
>
> *Oracle Database SQL Language Reference* to learn more about the `DUPLICATE ALL` clause

12.1.2.2.3 NO DUPLICATE Clause in Oracle RAC

The default `NO DUPLICATE` clause specifies that the database maintain only one copy of an object.

For example, a three-node Oracle RAC database might store the 2012 partition of a sales table in instance 1, the 2013 partition in instance 2, and the 2014 partition in instance 3. Each table partition resides in exactly one database instance.

If an Oracle RAC node does not duplicate the columnar data, then the columnar data on the failed node is not available in the IM column store on the cluster. Queries issued against missing data do not fail. Instead, queries access the data either from the database buffer cache or permanent storage, which may negatively affect performance. If the node remains down for some time, and if space exists in the surviving IM column stores, then Oracle RAC populates the missing objects or pieces of the objects on the remaining nodes in the cluster.

> **See Also:**
>
> *Oracle Database SQL Language Reference* to learn more about the `NO DUPLICATE` clause

12.1.3 Parallelism in Oracle RAC

A database instance must access the IMCUs in the IM column store in which they reside. Population and access of IM column stores in Oracle RAC must occur in parallel so that all IM column stores are accessible from any instance.

12.1.3.1 Serial and Parallel Queries in Oracle RAC

Database In-Memory in Oracle RAC is a shared-nothing architecture. Unless at least one parallel server process runs on every active instance, a query is not guaranteed to access all necessary data from the IM column stores.

Serial Queries

A serial query that runs on one node in an Oracle RAC database cannot access IMCUs in the other IM column stores. For example, a serial query running on instance 1 requests a full scan of `sales`. Some `sales` partitions are populated in the IM column store in instance 1, whereas the others are populated in the IM column store in instance 2. The query can only access the IMCUs in the IM column store on instance 1: the remaining data must come from on-disk storage.

Parallel Queries

When parallel execution is enabled, but the `PARALLEL_DEGREE_POLICY` initialization parameter is not set to `AUTO`, the situation is similar to the serial query case. The query coordinator runs on the database instance where the query executes. The PQ processes send data to the coordinator. In this case, the database starts multiple PQ processes. However, unless the DOP is greater than or equal to the number of IM column stores containing IMCUs populated for objects referenced in the query, not all data is accessible from the IM column store. When Auto DOP is not enabled, ensure that the DOP is at least as great as the IM column stores with IMCUs for the populated objects in the query.

In-Memory Dynamic Scans

Both serial and parallel queries can perform an In-Memory Dynamic Scan (IM dynamic scan) and use the lightweight thread infrastructure. The parallel execution infrastructure co-exists with the new thread infrastructure, which is dynamically managed by Oracle Database Resource Manager (the Resource Manager). The Resource Manager is enabled by default when `INMEMORY_SIZE` is greater than `0`.

A table scan process can be either a foreground process in a serial query or a parallel server process in a parallel query. When a parallel query performs an IM dynamic scan, every table scan process can own a pool of threads.

> **See Also:**
>
> - "In-Memory Dynamic Scans"
> - *Oracle Database Administrator's Guide* to learn more about the Resource Manager
> - *Oracle Database VLDB and Partitioning Guide* to learn more about parallel queries in Oracle RAC
> - *Oracle Database Reference* to learn more about the `PARALLEL_DEGREE_POLICY` initialization parameter

12.1.3.2 Auto DOP in Oracle RAC

With Automatic Degree of Parallelism (Auto DOP), the optimizer performs a cost-based calculation to determine the degree of parallelism for a SQL statement.

Enable Auto DOP by setting the `PARALLEL_DEGREE_POLICY` initialization parameter to `AUTO`. When the optimizer parses a SQL statement, it estimates the execution time. It checks this estimate against the setting of the `PARALLEL_MIN_TIME_THRESHOLD` initialization parameter, which is automatically set when the IM column store is enabled. The optimizer then makes the following cost-based decision:

- If the estimated time is less than `PARALLEL_MIN_TIME_THRESHOLD`, then the statement executes serially.
- If the estimated time is greater than `PARALLEL_MIN_TIME_THRESHOLD`, then the statement executes in parallel.

The optimizer calculates the degree of parallelism based on resource requirements. The calculation is limited by the `PARALLEL_DEGREE_LIMIT` initialization parameter and, if configured, the Database Resource Manager.

When using IM column stores in an Oracle RAC environment, the goal is to avoid disk or buffer cache access. To this end, you must *guarantee* that at least one parallel server process runs on every active database instance. Auto DOP is the recommended way to achieve this goal.

> **Note:**
>
> If you do not use Auto DOP, then you must ensure that the DOP is greater than or equal to the number of IM column stores containing the IMCUs required by the query.

Auto DOP guarantees an adequate distribution of processes because every shared pool stores metadata that indicates where all the IMCUs are located, how large they are, and so on. The same map is in every shared pool. No matter where the query originates in the cluster, the parallel query coordinator is aware of the home location (instance of residence) of the IMCUs.

For example, the PQ coordinator knows that the sales partitions for 2016 are in instance 1, whereas the partitions for 2015 are in instance 2. If a query running on instance 1 requests both 2015 and 2016 partitions, then the query coordinator uses the home location to determine which IM column stores to access. If the DOP has been set sufficiently high, then the coordinator automatically starts PQ processes on both instances, and the processes send the requested data back to the query coordinator.

> **See Also:**
>
> *Oracle Database VLDB and Partitioning Guide* to learn about Auto DOP

12.1.4 FastStart Area in Oracle RAC

The FastStart area is shared across all Oracle RAC nodes. This feature enables maximum sharing and reusability across the cluster.

Only one copy of an IMCU resides in the FastStart area. For example, if `DUPLICATE ALL` is specified for an object in a four-node cluster, then four copies of the object exist in the IM column stores. However, the database saves only one copy to the FastStart area.

Any database instance in an Oracle RAC cluster can use an IMCU in the FastStart area. This feature improves performance of instance restarts in an Oracle RAC environment.

For example, the `sales` table might have three partitions: `sales_2014`, `sales_2015`, and `sales_2016`, with each partition populated in a different instance. An instance failure occurs, with one instance unable to restart. If sufficient space is available in the IM column stores, then the surviving instances can read the IMCUs that were previously populated in the inaccessible instance. Thus, all three `sales` table partitions are available to applications.

> **See Also:**
>
> - "FastStart Area in Oracle RAC"
> - *Oracle Database SQL Language Reference* to learn more about the `DUPLICATE ALL` clause

12.2 Configuring In-Memory Services in Oracle RAC

A **service** represents a set of instances. In Oracle RAC, you can use services to direct connections or applications to a subset of nodes in the cluster.

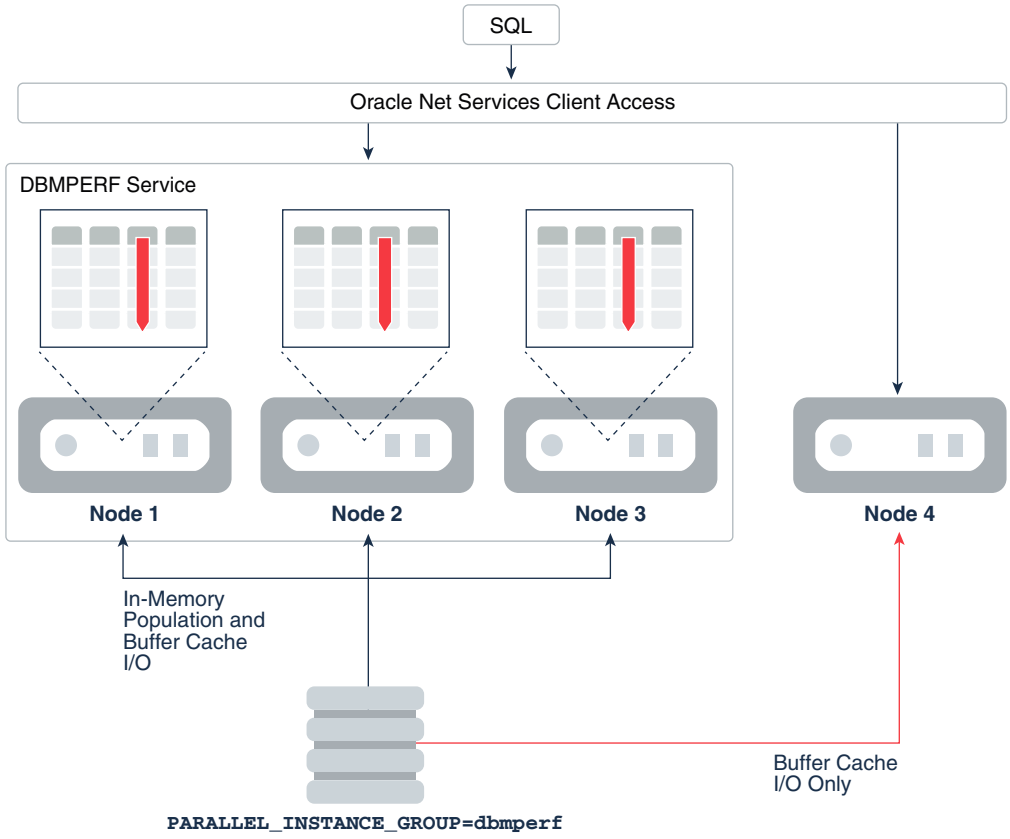
> **See Also:**
>
> *Oracle Real Application Clusters Administration and Deployment Guide* to learn more about services in Oracle RAC

12.2.1 Instance-Level Service Controls

In Oracle RAC, the population and access of IM column stores must occur in parallel so that all IM column stores are accessible from any database instance.

The `PARALLEL_INSTANCE_GROUP` initialization parameter restricts parallel query operations to the specified service. For example, if three out of four database instances in a cluster have an IM column store, then you might create a service named `dbmperf` and use `PARALLEL_INSTANCE_GROUP` to assign these three instances to this service. You can then restrict all client connections to the `dbmperf` service. Parallel operations spawn parallel execution processes only on the instances defined in the service.

Figure 12-8 Assigning a Subset of Instances to a Service



> **See Also:**
>
> *Oracle Database Reference* to learn more about the `PARALLEL_INSTANCE_GROUP` initialization parameter

12.2.2 Object-Level Service Controls

For an individual object, the `INMEMORY ... DISTRIBUTE` clause has a `FOR SERVICE` subclause that limits population to the database instance where this service can run.

The `PARALLEL_INSTANCE_GROUP` initialization parameter controls segments at the service level, where a service represents one or more instances. In contrast, `INMEMORY ... DISTRIBUTE FOR SERVICE` controls distribution at the segment level. For example, you can configure an `INMEMORY` object to be populated in the IM column store on instance 1 only, or on instance 2 only, or in both instances.

The `DISTRIBUTE FOR SERVICE` options are:

- `DEFAULT` - If `PARALLEL_INSTANCE_GROUP` is set, then the object is populated in all database instances that have an IM column store specified in `PARALLEL_INSTANCE_GROUP`. If `PARALLEL_INSTANCE_GROUP` is not set, then the object is populated in all instances that have an IM column store.

Specifying `FOR SERVICE` is equivalent to specifying `FOR SERVICE DEFAULT`.

- `ALL` - The database populates the object in all instances that have an IM column store.

> **Note:**
>
> If `PARALLEL_INSTANCE_GROUP` is not set, then `DEFAULT` and `ALL` are functionally equivalent.

- *service_name* - As part of its duties, IMCO triggers the removal of the object from the database instances assigned to the previous service, and populates it into the instances assigned to the new service.

  When redistributing segments, the database does the minimum work necessary. For example, service `dbmperf` is assigned to instance 1 and instance 2. The `sales` partitions are evenly distributed between instance 1 and instance 2. You add instance 3 to this service. The database only populates IMCUs in instance 3 and then removes them from instance 1 or instance 2 when necessary for even distribution. Some IMCUs remain in their original location.

- `NONE` - IMCO removes the object from the IM column stores of the currently specified services.

If the object has a `PRIORITY` value other than `NONE`, then W*nnn* processes populate the object during the next IMCO cycle after the DDL executes or the service starts. If the object has `PRIORITY` set to `NONE`, however, then the object is only populated during a full table scan. The scan triggers In-Memory population on all database instances on which the specified service for the table is active and not blocked. Note that this service can be different from the scan of the issuing service.

If a service used for In-Memory population stops, then the database removes the segment from the IM column stores represented by this service. In this respect, stopping the service is like shutting down the instances. The `INMEMORY` attributes of this object do not change. If the service starts again, then the database populates the object according to its `INMEMORY` attributes. To remove an object from the IM column store, specify `NO INMEMORY` in a DDL statement.

You can combine `DUPLICATE` with `DISTRIBUTE FOR SERVICE`. For example, you might specify that an object use `DUPLICATE ALL` for service `dbmperf`, which is assigned to three nodes out of four. In this case, the IM column store on each of these three nodes has a copy of the object.

> **See Also:**
>
> - "Prioritization of In-Memory Population"
> - *Oracle Database SQL Language Reference* to learn more about the `DISTRIBUTE FOR SERVICE` subclause

12.2.3 Benefits of Services for Database In-Memory in Oracle RAC

The combination of services and `DUPLICATE` enables you to control node access and In-Memory population.

Benefits of services include the following:

- Rolling patches and upgrades

  Suppose you set up an Oracle RAC service to direct client queries to the instances that contain an IM column store. If you use the `DUPLICATE` clause, then you can selectively remove an instance without affecting query response time. This approach assumes that sufficient resources exist on the other instances in the service to handle the workload of the removed instance.

  For example, in a four-node cluster, you could remove each node in turn, patch it, and then make it available again. The IMCUs for the temporarily inaccessible node are available on at least one other node, depending on whether you use the `DUPLICATE` or `DUPLICATE ALL` clause. Thus, application access to columnar data remains uninterrupted.

- Application affinity

  You can restrict application access to a single node based on service name. For example, service `dbmperf1` is restricted to node 1, service `dbmperf2` is restricted to node 2, and so on. When an application connects to a specific service and submits a parallel query, the query uses processes on the nodes belonging to the same service. For example, an application that connects to service `dbmperf1` only uses processes on node 1.

  Applications can coexist in an Oracle RAC database independently and access columnar data. Completely different objects can be populated in each node. For example, you could direct an HR application to service `dbmperf1`, and direct a sales history application to service `dbmperf2`.

> **See Also:**
>
> - "About Manually Enabling Objects for In-Memory Population"
> - *Oracle Database SQL Language Reference* for `DUPLICATE` semantics

12.2.4 Configuring an In-Memory Service for a Subset of Nodes: Example

This task explains how to assign an In-Memory service to a subset of nodes in an Oracle RAC database.

The goal is the following:

- Create IM column stores on a subset of nodes in a RAC database
- Define a service to allow access to only the nodes that have an IM column store

Assumptions

This task assumes the following:

- The Oracle RAC database named `dbmm` has four instances: `dbm1`, `dbm2`, `dbm3`, and `dbm4`. See "Figure 12-8".
- All instances except `dbm4` have `INMEMORY_SIZE` set to a nonzero value.
- You want to add a service named `dbmperf` and assign it to the three nodes that have an IM column store.
- You want to populate the `sales` table in the IM column stores attached to the service.

To configure an In-Memory service for a subset of nodes:

1. Create a service that represents the three nodes running IM column stores.

   On the operating system command line, use the `srvctl` command using the following form:

   ```
   srvctl add service -db db_name -s service_name
     -preferred "instance_names"
   ```

   For example, enter the following command:

   ```
   srvctl add service -db dbmm -s dbmperf -preferred "dbm1, dbm2, dbm3"
   ```

2. Start the service.

   For example, to start the `dbmperf` service, use the following command:

   ```
   srvctl start service -db dbmm -service "dbmperf"
   ```

3. Create a net service name for a connection to the service.

   For example, update the `tnsnames.ora` file as follows:

   ```
   DBMPERF =
     (DESCRIPTION =
        (ADDRESS =
          (PROTOCOL = TCP)
          (HOST = host_name)
          (PORT = listener_port))
        (CONNECT_DATA =
          (SERVER = DEDICATED)
          (SERVICE_NAME = DBMPERF)
        )
     )
   ```

4. Assign the `INMEMORY` attribute to the tables that you intend to populate, using the `DISTRIBUTE FOR SERVICE` subclause.

   For example, alter `sales` as follows:

   ```
   ALTER TABLE sales INMEMORY DISTRIBUTE FOR SERVICE "dbmperf";
   ```

The preceding statement uses the default `PRIORITY` setting of `NONE` for the `sales` table. Therefore, population of this table occurs on demand rather than automatically.

5. To populate the `sales` table, connect to the `dbmperf` service, and then initiate a full scan of the table.

   For example, force a full scan by querying `sales` as follows:

```
SELECT /*+ FULL(s) */ COUNT(*) FROM sales s;
```

See Also:

- *Oracle Database SQL Language Reference* to learn more about the `INMEMORY DISTRIBUTE FOR SERVICE` clause of `ALTER TABLE`
- *Oracle Database Administrator's Guide* to learn more about the `srvctl` utility
- *Oracle Database Net Services Administrator's Guide* to learn more about net service names

13

Deploying an IM Column Store with Oracle Active Data Guard

This chapter explains how Database In-Memory feature works in an Oracle Active Data Guard environment.

13.1 About Database In-Memory and Active Data Guard

Starting in Oracle Database 12c Release 2 (12.2.0.1), Oracle Database In-Memory is supported in an Oracle Active Data Guard environment using Oracle Engineered Systems or Oracle Cloud Platform as a Service.

> **See Also:**
>
> *Oracle Data Guard Concepts and Administration* for an introduction to Oracle Active Data Guard

13.1.1 Purpose of IM Column Stores in Oracle Active Data Guard

You can configure an IM column store only on the primary database, only on a standby database, or on both the primary and standby databases.

If you configure an IM column store for both databases, then you can populate the same or a different set of objects on the two instances. This technique effectively increases the IM column store size.

13.1.1.1 Identical IM Column Stores in Primary and Standby Databases

In the simplest scenario, the primary and standby databases both contain an IM column store with the same size (which is not required). The IM column stores contain the same objects.

The advantage of this scenario is that analytic queries can access the IM column store on either database. Therefore, you can direct analytic queries to the standby database and not consume resources on the primary database. As a result, the primary database can support the transactional workload, while the standby database supports the analytic workload.

The primary tasks are as follows:

- Set the `INMEMORY_SIZE` initialization parameter on both the primary and standby database instances.
- Ensure that the `INMEMORY_ADG_ENABLED` initialization parameter is set to `TRUE` (default) on the standby database instance.
- Set the `INMEMORY` attribute on all objects to be populated in the two IM column stores.

If you change the INMEMORY attributes of an object, then the primary database propagates the change to the standby database. For example, if you set the NO INMEMORY attribute on the sales table, then both IM column stores evict sales.

On the primary database, you can enable a subset of columns of a table for population into the IM column store. You can also specify different compression levels for different columns. Enabling specific columns involves a dictionary change. DDL on the primary database is propagated to the Oracle Active Data Guard database.

See Also:

- "Enabling the IM Column Store for a Database"
- *Oracle Database SQL Language Reference* for information about the CREATE TABLE statement
- *Oracle Database Reference* for more information about the INMEMORY_SIZE and INMEMORY_ADG_ENABLED initialization parameters

13.1.1.2 IM Column Store in Standby Database Only

In this scenario, an IM column store exists in the standby database, but not in the primary database.

In this scenario, the primary database can function as a pure OLTP database. No extra memory is required in the primary database for an IM column store. You can direct analytic reporting applications to the standby database without sacrificing performance or consuming resources on the primary database.

The primary tasks are as follows:

- Set the INMEMORY_SIZE initialization parameter to a non-zero value in the standby database instance, and set it to 0 in the primary database instance.
- Ensure that the INMEMORY_ADG_ENABLED initialization parameter is set to TRUE (default) on the standby database instance.
- Set the INMEMORY attribute with the DISTRIBUTE FOR SERVICE clause on all objects to be populated in the IM column store in the standby database.

  For example, if you log in to the primary database, and if you set the INMEMORY attribute on the sh.sales table, then this table will not be populated in the IM column store on the primary database—because no IM column store exists on this database. However, the standby database will inherit the INMEMORY attribute on the sh.sales table. The table will be populated in the IM column store in the standby database.

13.1.1.3 Different Objects in the Primary and Standby IM Column Stores

The most flexible scenario is separately configuring the IM column stores for primary and standby databases.

The advantage of this scenario is that you can run different workloads in each database. For example, an HR application runs reports in the primary database, while

a sales history application runs reports in the standby database. Thus, neither database bears the full burden of analytic reporting.

The primary tasks are as follows:

- Set the `INMEMORY_SIZE` initialization parameter to a non-zero value on the standby and primary database instance. The values do not need to be identical.
- Ensure that the `INMEMORY_ADG_ENABLED` initialization parameter is set to `TRUE` (default) on the standby database instance.
- Set the `INMEMORY ... DISTRIBUTE FOR SERVICE` clause on all objects to be populated in the two IM column stores. The service specifies the instance into which the object is populated.

Three-Service Configuration

In a typical configuration, you create three services: standby-only, primary-only, and primary-and-standby. For example, you may want the latest month of `sales` fact table data in the primary instance, but the previous `sales` data in the standby instance. You want the dimension tables populated in both instances. For each `sales` partition, you use `INMEMORY ... DISTRIBUTE FOR SERVICE` to specify either the standby or primary service. For each dimension table, you specify the service that includes both primary and standby database instances.

> **Note:**
>
> As long as the service name is defined for both the primary and standby instances, you can specify the same service name in `DISTRIBUTE FOR SERVICE` to populate the same tables in the primary and standby databases.

Oracle RAC and Oracle Active Data Guard

In Oracle RAC, you can combine the `FOR SERVICE` clause, which specifies the instance for population, with the `DISTRIBUTE AUTO` or `DISTRIBUTE BY` clause, which controls the distribution of IMCUs. However, in Oracle Active Data Guard, the `FOR SERVICE` clause specifies the primary or standby instances in which to *populate* the specified object: you cannot use `DISTRIBUTE AUTO` or `DISTRIBUTE BY` to distribute IMCUs *between* the primary and standby instances. For example, you cannot divide the population of the `sales` table between the primary instance and standby instance, so that half the IMCUs are in the primary instance and half the IMCUs are in the standby instance.

> **See Also:**
>
> - "Object-Level Service Controls"
> - *Oracle Database SQL Language Reference* to learn more about the `DISTRIBUTE FOR SERVICE` subclause

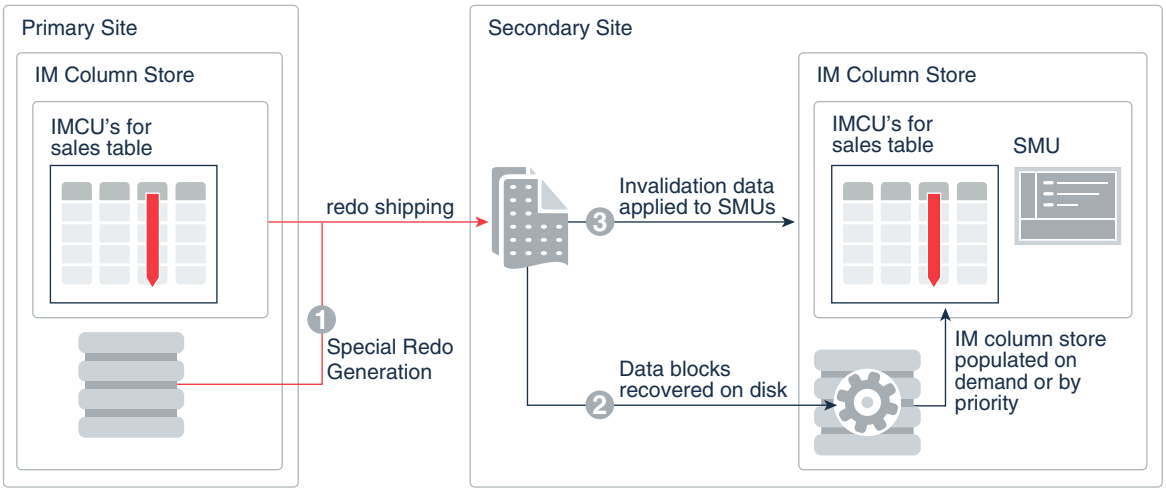
13.1.2 How IM Column Stores Work in Oracle Active Data Guard

In an Oracle Active Data Guard environment, the object-level `PRIORITY` attribute governs population. An object is only populated in the database instances on which the service is active.

Population is either on-demand or priority-based, depending on the `PRIORITY` value. When a role change or switchover occurs, the database repopulates the tables according to the set of database instances to which the service is newly mapped.

The following graphic illustrates the internal mechanism for updating a standby database with redo from the primary database.

Figure 13-1 Updating a Standby Database



The process is as follows:

1. The primary database generates redo, and then transfers the redo to the standby database.

   The redo generated on the primary database for all DML statements includes metadata indicating whether the change is to an `INMEMORY` object.

2. The standby database applies the redo to the data blocks stored in disk.

   As the standby database applies redo generated from ongoing operations on the primary database, the standby database keeps them transactionally consistent.

3. If an `INMEMORY` object is modified, then the standby database invalidates the modified rows just as it does on the primary database, using the transaction journal and Snapshot Metadata Unit (SMU) to track the changes.

The repopulation mechanism works the same way in a standby database as it does in a primary database. When sufficient DML occurs on an object to reach an internal threshold, the standby database repopulates the object in the IM column store.

> **See Also:**
>
> - "About Repopulation of the IM Column Store"
> - "About Join Groups"
> - *Oracle Data Guard Concepts and Administration* to learn more about multi-instance redo apply

13.1.3 In-Memory Restrictions in Active Data Guard

Most In-Memory features are supported in Active Data Guard.

Standby databases do not support the following In-Memory features:

- IM FastStart
- Join groups
- IM expression capture
- Heat Map (which reflects the primary database only)

Data Guard Multi-Instance Redo Apply is supported for the IM column store on an Active Data Guard standby database, but only when the initialization parameter `ENABLE_IMC_WITH_MIRA` is set to `TRUE`. By default, `ENABLE_IMC_WITH_MIRA` is `FALSE`.

> **See Also:**
>
> *Oracle Database Reference* to learn more about `ENABLE_IMC_WITH_MIRA`

13.2 Configuring IM Column Stores in an Oracle Active Data Guard Environment

Configuring IM column stores in Oracle Active Data Guard requires setting `INMEMORY_SIZE`, and setting the `INMEMORY` attribute appropriately for the objects to be populated.

This task assumes knowledge of Oracle Active Data Guard concepts and procedures.

Prerequisites

You must meet the following requirements:

- The standby database must run on an Oracle Engineered System or in Oracle Cloud Platform as a Service.
- The `COMPATIBLE` initialization parameter must be `12.2.0` or greater.
- To populate different objects in each database, configure the appropriate services.

To configure IM column stores in an Oracle Active Data Guard environment:

1. Set the `INMEMORY_SIZE` initialization parameter on the database instances that will contain an IM column store.

   Follow these guidelines:

   - To configure IM column stores on the primary and standby databases, set `INMEMORY_SIZE` on both database instances.
   - To configure IM column stores on the standby database only, set `INMEMORY_SIZE` on the standby database instance.

2. Ensure that the `INMEMORY_ADG_ENABLED` initialization parameter is set to `TRUE` (default) on the standby database instance.
3. Optionally, if you want to enable Multi-Instance Redo Apply with the IM column store, set the `ENABLE_IMC_WITH_MIRA` initialization parameter to `TRUE`.
4. On the primary database, execute DDL statements with the `INMEMORY` attribute.

   The task depends on where the IM column stores exist, and in which IM column stores you want the objects to be populated:

   - To populate an object on the standby database only, then set the `INMEMORY` attribute with a `DISTRIBUTE FOR SERVICE` clause that specifies a valid service running only on the standby database.

     During redo transfer, the standby database receives this DDL statement from the primary database. Population occurs on the standby database in the normal way. For example, if `sales` has the `INMEMORY` attribute and priority `NONE`, then the table must undergo a full scan for population to occur.
   - To populate an object on the primary database only, then set the `INMEMORY` attribute with a `DISTRIBUTE FOR SERVICE` clause that specifies a valid service running only on the primary database.
   - To populate an object on both primary and standby databases, then perform one of the following actions:
     - Do not set the `DISTRIBUTE FOR SERVICE` clause.
     - Set `DISTRIBUTE FOR SERVICE` *`servicename`*, where *`servicename`* is a service that is running on both the primary and standby databases.
     - Set `DISTRIBUTE FOR SERVICE DEFAULT` so that the object is eligible for population on all instances specified with the `PARALLEL_INSTANCE_GROUP` initialization parameter.
     - Set `DISTRIBUTE FOR SERVICE ALL` so that the object is eligible for population on all instances, regardless of the setting of the `PARALLEL_INSTANCE_GROUP` initialization parameter.

     Population of an object occurs on the primary or standby database according to the priority setting. For example, if `sales` on the standby database has priority `NONE`, then a query of the standby database that triggers a full scan of `sales` populates this table in the standby IM column store.

> **Note:**
>
> A full scan of `sales` on the *standby* database does not populate this table in the IM column store in the *primary* database.

> **See Also:**
>
> - "Prioritization of In-Memory Population"
> - *Oracle Data Guard Concepts and Administration*
> - *Oracle Database SQL Language Reference* to learn more about the `DISTRIBUTE FOR SERVICE` subclause
> - *Oracle Database Reference* for more information about the `INMEMORY_SIZE`, `INMEMORY_ADG_ENABLED`, and `ENABLE_IMC_WITH_MIRA` initialization parameters

Part V
Database In-Memory Reference

The Part contains initialization parameters and views relevant for the In-Memory Column Store (IM column store).

14
In-Memory Initialization Parameters

Several initialization parameters control the behavior of the IM column store.

This chapter is a summary only. *Oracle Database Reference* contains complete information for all database views.

Table 14-1 Initialization Parameters Related to the IM Column Store

Initialization Parameter	Description
INMEMORY_ADG_ENABLED	Indicates whether In-Memory for Oracle Active Data Guard is enabled (TRUE) or disabled (FALSE) on the standby database. The default is TRUE. For Active Data Guard, media recovery must retrieve In-Memory objects when applying redo and invalidate the related objects after the query advance. This parameter controls whether media recovery performs the retrieving and invalidating. You can only modify this system-level parameter when standby recovery is not running. If the standby database uses Oracle RAC, then this parameter must be set to the same value on every instance.
INMEMORY_AUTOMATIC_LEVEL	Automates the management of the IM column store by helping to ensure that the working data set is always in the IM column store. You can set the following values: • OFF: This is the default. When this value is set, Automatic In-Memory is disabled. This value returns the IM column store to the behavior that existed before Oracle Database 18c. If you do not expect a stable working data set, then set the parameter to DISABLE. • LOW: When this value is set, the database evicts cold segments from the IM column store when it is under memory pressure. • MEDIUM: This level includes an additional optimization that ensures that any hot segment that was not populated because of memory pressure is populated first.

Table 14-1 (Cont.) Initialization Parameters Related to the IM Column Store

Initialization Parameter	Description
INMEMORY_CLAUSE_DEFAULT	Specifies a default IM column store clause for new tables and materialized views. This parameter supports the following options: • To specify that there is no default IM column store clause for new tables and materialized views, leave this parameter unset or set it to an empty string. Setting this parameter to NO INMEMORY has the same effect as setting it to the default value (the empty string). • To specify that the clause is the default for all new tables and materialized views, set this parameter to a valid INMEMORY clause. The clause can include valid clauses for IM column store compression methods and data population options. The options are: – If the clause starts with INMEMORY, then all new tables and materialized views, including those without an INMEMORY clause, are populated in the IM column store. – If the clause omits INMEMORY, then it only applies to new tables and materialized views that are enabled for the IM column store with an INMEMORY clause during creation.

Table 14-1 (Cont.) Initialization Parameters Related to the IM Column Store

Initialization Parameter	Description
INMEMORY_EXPRESSIONS_USAGE	Controls which IM expressions are eligible to be populated in the IM column store. This parameter supports the following options: • ENABLE The database populates both static and dynamic IM expressions into the IM column store. Setting this value increases the In-Memory footprint for some tables. This is the default. • STATIC_ONLY A static configuration enables the IM column store to cache OSON (binary JSON) columns, which are marked with an IS_JSON check constraint. Internally, an OSON column is a hidden virtual column named SYS_IME_OSON. • DYNAMIC_ONLY The database only populates frequently used or “hot” expressions that have been added to the table as SYS_IME hidden virtual columns. Setting this value increases the In-Memory footprint for some tables. • DISABLE The database does not populate any IM expressions, whether static or dynamic, into the IM column store. **Note:** IM expressions do not support NLS-dependent data types.

Table 14-1 (Cont.) Initialization Parameters Related to the IM Column Store

Initialization Parameter	Description
INMEMORY_FORCE	Enables or disables tables and materialized views for the IM column store. This parameter supports the following options: • To allow the INMEMORY or NO INMEMORY attributes to determine population, set this parameter to DEFAULT (the default value). You can set this value dynamically. • To disable all tables and materialized views for the IM column store, set this parameter to OFF. You can set this value dynamically. • To enable the Database In-Memory Base Level, set this parameter to BASE_LEVEL in the CDB root initialization parameter file (not at the PDB level). You cannot set this value dynamically. When the CDB uses the Base Level, Automatic In-Memory is disabled, and the compression level for INMEMORY objects and columns is automatically set to QUERY LOW. • To use the CellMemory feature without incurring the overhead of creating an IM column store, set this parameter to CELLMEMORY_LEVEL. You cannot set this value dynamically. Note that if the value of INMEMORY_SIZE is greater than 0, then setting INMEMORY_FORCE=CELLMEMORY_LEVEL is equivalent to setting INMEMORY_FORCE=DEFAULT. The IM column store is enabled, even if you use only CellMemory.
INMEMORY_MAX_POPULATE_SERVERS	Specifies the maximum number of Space Management Worker Processes (W*nnn*) to use for population so that the processes do not overload the rest of the system. Set this parameter to an appropriate value based on the number of cores in the system. The default is half the effective CPU thread count or the PGA_AGGREGATE_TARGET value divided by 512 MB, whichever is less. **Note:** When INMEMORY_MAX_POPULATE_SERVERS is set to 0, objects cannot be populated in the IM column store

Table 14-1 (Cont.) Initialization Parameters Related to the IM Column Store

Initialization Parameter	Description
INMEMORY_OPTIMIZED_ARITHMETIC	Controls whether NUMBER columns are stored in an In-Memory optimized format. This parameter supports the following options: • DISABLE (the default) does not use the optimized encoding. • ENABLE encodes numbers in an optimized format that enables native calculations using SIMD hardware. This optimization is available for tables with FOR QUERY LOW compression.
INMEMORY_QUERY	Specifies whether In-Memory queries are allowed. This parameter supports the following options: • ENABLE (the default) allows queries to access populated objects. • DISABLE blocks access to populated objects.
INMEMORY_SIZE	Sets the size of the IM column store in a database instance. The default is 0, which disables the IM column store. For the In-Memory Free Tier option, the size must not exceed 16 GB for a non-CDB or CDB. In an Oracle RAC database, the INMEMORY_SIZE setting in each instance must not exceed 16 GB.
INMEMORY_TRICKLE_REPOPULATE_SERVERS_PERCENT	Limits the percentage of the total population and repopulation processes that perform trickle repopulation. The value for this parameter is a percentage of the INMEMORY_MAX_POPULATE_SERVERS initialization parameter value. For example, if INMEMORY_TRICKLE_REPOPULATE_SERVERS_PERCENT is 5 percent, and if INMEMORY_MAX_POPULATE_SERVERS is 20, then the IM column store uses an average of 1 core (.05 * 20) for trickle repopulation.
INMEMORY_VIRTUAL_COLUMNS	Controls which expressions are populated in the IM column store. This parameter supports the following options: • MANUAL (the default) permits population of IM virtual columns explicitly specified as INMEMORY. • ENABLE permits population of all IM virtual columns in an INMEMORY table, unless these columns have been explicitly excluded from the IM column store. • DISABLE specifies that no IM virtual columns are eligible for population.

Table 14-1 (Cont.) Initialization Parameters Related to the IM Column Store

Initialization Parameter	Description
OPTIMIZER_INMEMORY_AWARE	Controls the optimizer cost model enhancements for Database In-Memory. This parameter supports the following options: • TRUE (default) optimizes SQL statements that reference INMEMORY objects. • FALSE ignores the INMEMORY attribute of tables during optimization.

15
In-Memory Views

This topic describes data dictionary and dynamic performance views related to the In-Memory Column Store (IM column store).

Table 15-1 In-Memory Views

Initialization Parameter	Description
DBA_EXPRESSION_STATISTICS	Provides expression usage tracking statistics for all the tables in the database.
DBA_EXTERNAL_TABLES	Describes all external tables in the database. The INMEMORY column indicates whether the table is enabled for INMEMORY, and INMEMORY_COMPRESSION indicates the compression level. Related views include ALL_XTERNAL_PART_TABLES, ALL_XTERNAL_TAB_PARTITIONS, and ALL_XTERNAL_TAB_SUBPARTITIONS.
DBA_FEATURE_USAGE_STATISTICS	Displays information about database feature usage statistics. When the IM column store is accessed, the NAME column shows In-Memory Column Store.
DBA_HEAT_MAP_SEGMENT	Displays the latest segment access time for all segments. The timestamps in the view are coarse with a granularity of a day reflecting the flush times of the heat map.
DBA_INMEMORY_AIMTASKS	Describes the status of tasks created by Automatic In-Memory.
DBA_INMEMORY_AIMTASKDETAILS	Describes the object and action taken by Automatic In-Memory.
DBA_ILMDATAMOVEMENTPOLICIES	Contains information specific to data movement-related attributes of an Automatic Data Optimization policy in a database.
DBA_IM_EXPRESSIONS	Describes the IM expressions, which have the column prefix SYS_IME, that have the INMEMORY attribute.
DBA_JOINGROUPS	Describes join groups in the database. A **join group** is a user-created object that lists two columns that can be meaningfully joined.
DBA_SEGMENTS	Describes the storage allocated for all segments in the database. Several columns, including INMEMORY and INMEMORY_PRIORITY, describe In-Memory attributes of the segment.
DBA_TABLES	Indicates which tables have the INMEMORY attribute set (the INMEMORY column is ENABLED) or not set (DISABLED).
V$ACTIVE_SESSION_HISTORY	Displays sampled session activity. Several columns, including INMEMORY_QUERY and INMEMORY_POPULATE, describe session activity relating to the In-Memory Column Store at the time of sampling.
V$HEAT_MAP_SEGMENT	Displays real-time segment access information.

Table 15-1 (Cont.) In-Memory Views

Initialization Parameter	Description
V$IM_COLUMN_LEVEL	Presents the selective column compression levels that are defined using the INMEMORY clause of the CREATE TABLE statement. SYS_IME hidden virtual columns are not shown in this view.
V$IM_SEGMENTS	Presents information about all In-Memory segments in the database. Only segments that have an In-Memory representation are displayed. If a segment is marked for the IM column store but is not populated, the view does not contain a corresponding row for this segment.
V$INMEMORY_AREA	Displays information about the space allocation inside the In-Memory Area.
V$INMEMORY_FASTSTART_AREA	Provides information about the In-Memory FastStart (IM FastStart) area.
V$SGA	Displays the size of the In-Memory Area.

A

Using IM Column Store in Cloud Control

You can configure and manage the IM column store in Oracle Enterprise Manager Cloud Control (Cloud Control).

A.1 Meeting Prerequisites for Using IM Column Store in Cloud Control

Before you can enable a database to use the IM column store, ensure that the `COMPATIBLE` is set to 12.1.0.0 or higher.

To set the compatibility level, follow these steps:

1. From the Database Home page in Enterprise Manager, choose **Initialization Parameters** from the **Administration** menu to navigate to the Initialization Parameters page.

   You can use this page to set or change the compatibility level.
2. Search for the `COMPATIBLE` initialization parameter.

   The category for the parameter is Miscellaneous.
3. Change the value to `12.1.0.0` and click **Apply**.

   Cloud Control prompts you to restart the database. After the database is restarted, the new value takes effect.

To set or change the size of the IM column store, follow these steps:

1. From the Database Home page in Enterprise Manager, choose **Initialization Parameters** from the **Administration** menu to navigate to the Initialization Parameters page.
2. Search for the parameter `INMEMORY_SIZE`. The category for the parameter is In-Memory.
3. Change the value and click **Apply**.

   You can set the value to any value above the minimum size of 100 MB.

   You will then be prompted to restart the database.

A.2 Using the In-Memory Column Store Central Home Page to Monitor In-Memory Support for Database Objects

Use the In-Memory Column Store Central Home page to monitor In-Memory support for database objects such as tables, indexes, partitions and tablespaces. You can view in-memory functionality for objects and monitor their In-Memory usage statistics.

You can complete the following actions on the In-Memory Column Store Central Home page:

- The In-Memory Object Access Heatmap displays the top 100 objects in the IM column store with their relative sizes and shows you how frequently objects are accessed, represented by different colors. To activate the heat map, you must turn on the option for the Heat Map in the initialization parameter file. Typically, there is a one day wait period before the map is activated. You can use the date selector to pick the date range for objects displayed in the Heat Map. You can also use the slider to control the granularity of the color.
- Use the Configuration section to view the status settings such as In-Memory Query, In-Memory Force, and Default In-Memory Clause. Click Edit to navigate to the Initialization Parameters page where you can change the values and settings displayed in this section. Use the Performance section to view the metrics for Active Sessions.
- Use the Objects Summary section to view the Compression Factor and data about the memory used by the populated objects. The In-Memory Enabled Object Statistics are available in a pop-up window through a drill-down from the View In-Memory Enabled Object Statistics link on the page.
- Use the In-Memory Objects Distribution section to view the distribution on a percentage basis of the various objects used in memory. The section includes a chart showing the distribution of Partitions, Subpartitions, Non-partitioned Tables, and Non-partitioned Materialized Views. The numerical values for each are displayed above the chart.
- Use the In-Memory Objects Search section to search for objects designated for In-Memory use. Click Search after you enter the parameters by which you want to search. The results table shows the Name of each object found along with its Size, Size in Memory, Size on Disk, In-Memory percentage, and its In-Memory parameters. You can also search for accessed objects that are either in-memory or not in-memory. If the Heat Map is enabled, then the Accessed Objects option appears in the drop-down list in the View field of the In-Memory Objects Search box. When you select Accessed Objects, you can filter based on the top 100 objects with access data that are either in-memory or not in-memory. You can select a time range and search for objects within that range. If you select the All Objects In-Memory option, you can view the list of top 100 objects that are in-memory based on their in-memory size.

If you are working in an Oracle RAC environment, you can quickly move between instances by selecting the instance in the Instances selection box above and on the right side of the Heat Map.

A.3 Specifying In-Memory Details When Creating a Table or Partition

You can specify IM column store details when creating a table or partition.

1. From the **Schema** menu, choose **Database Objects**, then select the **Tables** option.
2. Click **Create** to create a table.

   The Create Table page is shown. Select the In-Memory Column Store tab to specify the in-memory options for the table.
3. Choose to override the column level in-memory details (if required) in the table below where the columns are specified.

4. Optionally, you can click on the **Partitions** tab.
5. Create table partitions as needed using the wizard.

   To specify IM column store details for a partition, select it from the table in the Partitions tab, and then click **Advanced Options**.
6. After entering all necessary IM column store details at the table level, column level, and partitions level, click **Show SQL** to see the generated SQL. Click **OK** to create the table.

A.4 Viewing or Editing IM Column Store Details of a Table

You can view or edit IM column store details of a table.

1. From the **Schema** menu, choose **Database Objects**, and then select the **Tables** option.
2. Search for the desired table, and then click **View** to view its details.
3. Click **Edit** to launch the Edit Table page.

   Alternatively, on the Search page, click **Edit**. Click the **In-Memory Column Store** tab to specify In-Memory options for the table.
4. Edit the required details.
5. Click **Apply**.

A.5 Viewing or Editing IM Column Store Details of a Partition

You can view or edit IM column store details of a partition.

1. From the **Schema** menu, choose **Database Objects**, then select the **Tables** option.
2. Search for the table that contains the desired partition, select it, and then click **View**.
3. Click **Edit** to launch the Edit Table page.

   Alternatively, on the Table Search page, click **Edit**.
4. Click the **Partitions** tab, and then select the desired partition.
5. Click **Advanced Options**.
6. Edit the required details.
7. Click **OK** to return to the Partitions tab.
8. After making similar changes to all desired partitions of the table, click **Apply**.

A.6 Specifying IM Column Store Details During Tablespace Creation

You can specify IM column store details when creating a tablespace.

1. From the **Administration** menu, choose **Storage**, and then select **Tablespaces**.
2. Click **Create** to create a tablespace.
3. Enter the details that appear on the General tab.
4. Click the **In-Memory Column Store** tab.
5. Enter all required IM column store details for the tablespace.

6. Click **Show SQL**.
7. In the Show SQL page, click **Return**.

   Another page appears.
8. Click **OK**.
9. Click **OK** to create the tablespace.

The IM column store settings of a tablespace apply for any new table created in the tablespace. IM column store configuration details must be specified at the individual table level if a table must override the configuration of the tablespace.

A.7 Viewing and Editing IM Column Store Details of a Tablespace

You can view or edit IM column store details of a tablespace.

1. From the **Administration** menu, choose **Storage**, then select the **Tablespaces** option.
2. Search for the desired tablespace, select it, then click **View**.
3. Click **Edit** to launch the Edit Tablespace page, then click the **In-Memory Column Store** tab.
4. Edit the required details and click **Apply**.

A.8 Specifying IM Column Store Details During Materialized View Creation

You can specify IM column store details when creating a materialized view.

1. From the **Schema** menu, choose **Materialized Views**, then select the **Materialized Views** option.
2. Click **Create** to create a materialized view.
3. Enter the materialized view name, and specify its query.
4. Click the **In-Memory Column Store** tab to specify IM column store options for the materialized view.
5. After entering all necessary IM column store details, click **Show SQL**. Click **Return** from the Show SQL page, and then in the resulting page click **OK**.
6. Click **OK** to create the materialized view.

A.9 Viewing or Editing IM Column Store Details of a Materialized View

You can view or edit IM column store details of a materialized view.

1. From the **Schema** menu, choose **Materialized Views**, then select the **Materialized Views** option.

2. Search for the desired materialized view, and click **View** to view its details.
3. Click **Edit** to launch the Edit Materialized View page.
4. Click the **In-Memory Column Store** tab to specify IM column store options for the materialized view.
5. Edit the required details, and click **Apply**.

Glossary

ADO policy

A policy that specifies a rule and condition for Automatic Data Optimization (ADO). For example, an ADO policy may specify that an object is marked `NOINMEMORY` (action) 30 days after creation (condition). Specify ADO policies using the `ILM` clause of `CREATE TABLE` and `ALTER TABLE` statements.

Automatic Data Optimization (ADO)

A technology that creates policies, and automates actions based on those policies, to implement an Information Lifecycle Management (ILM) strategy.

Automatic In-Memory

A feature that automatically evicts cold (infrequently accessed) segments from the IM column store to ensure that the working data set is always populated.

availability

The degree to which an application, service, or function is accessible on demand.

Bloom filter

A low-memory data structure that tests membership in a set. The database uses Bloom filters to improve the performance of hash joins.

Column Compression Unit (CU)

Contiguous storage for a column in an In-Memory Compression Unit (IMCU).

columnar data pool

The subpool in the In-Memory Area that stores columnar data. It is also known as the *1 MB pool*.

columnar format

The column-based format for objects that reside in the In-Memory Column Store. The columnar format contrasts with the row format used in data blocks.

common dictionary

A segment-level, instance-specific set of common dictionary codes, created from local dictionaries. A local dictionary is a sorted list of dictionary codes specific to a Column Compression Unit (CU). A join group uses a common dictionary to optimize joins.

compression tiering

The application of different levels of compression to data based on its access pattern. For example, administrators may compress inactive data at a higher rate of compression at the cost of slower access.

data flow operator (DFO)

The unit of work between data redistribution stages in a parallel query.

dense grouping key

A key that represents all grouping keys whose grouping columns come from a specific fact table or dimension.

dense join key

A key that represents all join keys whose join columns come from a particular fact table or dimension.

dense key

A numeric key that is stored as a native integer and has a range of values.

double buffering

A repopulation mechanism in which background processes create new In-Memory Compression Unit (IMCU) versions by combining the original rows with the latest modified rows. During repopulation, the stale IMCUs remain accessible for queries.

expression

A combination of one or more values, operators, and SQL functions that resolves to a value.

expression capture interval

The time interval within which the database considers IM expressions for possible capture.

expression capture window

An expression capture interval defined by invocation of the `IME_OPEN_CAPTURE_WINDOW` and `IME_OPEN_CAPTURE_WINDOW` procedures in the `DBMS_INMEMORY_ADMIN` package.

Expression Statistics Store (ESS)

A repository maintained by the optimizer to store statistics about expression evaluation. For each segment, the ESS monitors statistics such as frequency of execution, cost of evaluation, timestamp evaluation, and so on. The ESS is persistent in nature and has an SGA representation for fast lookup of expressions.

FastStart area

A designated tablespace to which the database periodically writes In-Memory columnar data.

Heat Map

Heat Map shows the popularity of data blocks and rows. Automatic Data Optimization (ADO) to decide which segments are candidates for movement to a different storage tier.

home location

The database instance in which an IMCU resides. When auto DOP is enabled on Oracle RAC, the parallel query coordinator uses home location to determine where each IMCU is located, how large it is, and so on.

hybrid partitioned table

A table in which some partitions are stored in data file segments and some are stored in external data source.

IM aggregation

An optimization that accelerates aggregation for queries that join from a single large table to multiple small tables. The transformation uses `KEY VECTOR` and `VECTOR GROUP BY` operators, which is why it is also known as *`VECTOR GROUP BY` aggregation*.

IM column store

An optional SGA area that stores copies of tables and partitions in a columnar format optimized for rapid scans.

IM dynamic scan

The use of lightweight threads to automatically parallelize In-Memory table scans.

IM expression

A SQL expression whose results are stored in the In-Memory Column Store. If `last_name` is a column stored in the IM column store, then an IM expression might be `UPPER(last_name)`.

IMCU mirroring

In Oracle RAC, the duplication of an IMCU in multiple IM column stores. For example, the IM column stores on instance 1 and instance 2 are populated with the same `sales` table.

IMCU pruning

In a query of the In-Memory Column Store, the elimination of IMCUs based on the high and low values in each IMCU. For example, if a statements filters product IDs greater than 100, then the database avoids scanning IMCUs that contain values less than 100.

IM storage index

A data structure in an IMCU header that stores the minimum and maximum for all columns within the IMCU.

In-Memory Advisor

A downloadable PL/SQL package that analyzes the analytical processing workload in your database. This advisor recommends a size for the IM column store and a list of objects that would benefit from In-Memory population.

In-Memory Aggregation

See IM aggregation.

In-Memory Area

An optional SGA component that contains the IM column store.

In-Memory Column Store

See IM column store.

In-Memory Compression Unit (IMCU)

A storage unit in the In-Memory Column Store that is optimized for faster scans. The In-Memory Column Store stores each column in table separately and compresses it. Each IMCU contains all columns for a subset of rows in a specific table segment.
A one-to-many mapping exists between an IMCU and a set of database blocks. For example, if a table contains columns `c1` and `c2`, and if its rows are stored in 100 database blocks on disk, then IMCU 1 might store the values for both columns for blocks 1-50, and IMCU 2 might store the values for both columns for blocks 51-100.

In-Memory Coordinator Process (IMCO)

A background process whose primary task is to initiate background population and repopulation of columnar data.

In-Memory Dynamic Scan

See IM dynamic scan.

In-Memory Expression

See IM expression.

In-Memory Expression Unit (IMEU)

A container that stores the computed result of an In-Memory Expression (IM expression). Each IMEU is linked to its own parent In-Memory Compression Unit (IMCU).

In-Memory FastStart

A feature that significantly reduces the time to populate data into the IM column store when a database instance restarts.

In-Memory population

See population.

In-Memory virtual column

A virtual column that is eligible to be populated in the In-Memory Column Store.

Information Lifecycle Management (ILM)

A set of processes and policies for managing data throughout its useful life.

join group

A user-defined object that specifies frequently joined columns from the same table or different tables. External tables are not supported.

A typical join group candidate is a set of columns used to join fact and dimension tables. Join groups are only supported when `INMEMORY_SIZE` is a nonzero value.

key vector

A data structure that maps between dense join keys and dense grouping keys.

local dictionary

A sorted list of dictionary codes specific to a Column Compression Unit (CU).

lightweight thread

An execution entity used in an In-Memory Dynamic Scan. Lightweight threads help to parallelize scans of IMCUs.

metadata pool

A subpool of the In-Memory Area that stores metadata about the objects that reside in the IM column store. The metadata pool is also known as the *64 KB pool*.

on-demand population

When `INMEMORY PRIORITY` is set to `NONE`, the IM column store *only* populates the object when it is accessed through a full scan. If the object is never accessed, or if it is accessed only through an index scan or fetch by rowid, then it is never populated.

OSON

Oracle's optimized binary JSON format. OSON enables fast queries and updates of the JSON data model in Oracle database server and Oracle database clients.

OZIP

A proprietary compression technique that offers extremely fast decompression. OZIP is tuned specifically for Oracle Database.

partition exchange load

A technique in which you create a table, load data into it, and then exchange an existing table partition with the table. This exchange process is a DDL operation with no actual data movement.

population

The operation of reading existing data blocks from data files, transforming the rows into columnar format, and then writing the columnar data to the IM column store. In contrast, *loading* refers to bringing new data into the database using DML or DDL.

priority-based population

When `PRIORITY` is set to a value other than `NONE`, Oracle Database adds the object to a prioritized population queue. The database populates objects based on their queue position, from `CRITICAL` to `LOW`. It is "priority-based" because the IM column store automatically populates objects using the prioritized list whenever the database re-opens. Unlike in on-demand population, objects do not require a full scan to be populated.

repopulation

The automatic refresh of a currently populated In-Memory Compression Unit (IMCU) after its data has been significantly modified. In contrast, population is the initial creation of IMCUs in the IM column store.

service

The logical representation of an application workload that shares common attributes, performance thresholds, and priorities. A single service can be associated with one or more instances of an Oracle RAC database, and a single instance can support multiple services.

Snapshot Metadata Unit (SMU)

A storage unit in the In-Memory Area that contains metadata and transactional information for an associated In-Memory Compression Unit (IMCU).

Space Management Worker Process (W*nnn*)

A process that populates or repopulates data in the IM column store on behalf of In-Memory Coordinator Process (IMCO).

staleness threshold

An internally set percentage of entries in the transaction journal for an IMCU that initiates repopulation.

storage tiering

The deployment of data on different tiers of storage depending on its level of access. For example, administrators migrate inactive data from high-performance, high-cost storage to low-cost storage.

table scan process

A foreground or PQ process that coordinates an IM dynamic scan.

threshold-based repopulation

The automatic repopulation of an IMCU when the number of stale entries in an IMCU reaches an internal staleness threshold.

transaction journal

Metadata in a Snapshot Metadata Unit (SMU) that keeps the IM column store transactionally consistent.

trickle repopulation

A supplement to threshold-based repopulation. The In-Memory Coordinator Process (IMCO) may instigate trickle repopulation automatically for any IMCU in the IM column store that has stale entries but does not meet the staleness threshold.

vector aggregation

See IM aggregation.

virtual column

A column that is not stored on disk. The database derives the values in virtual columns on demand by computing a set of expressions or functions.

working data set

The subset of `INMEMORY` objects that is actively queried at a given time. Typically, the work working data set changes over time.

Index

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

V

W