[Oracle Documentation](http://docs.oracle.com/en/)

* [[Sign In Icon](http://www.oracle.com/webapps/redirect/signon?nexturl=http://docs.oracle.com/cd/B19306_01/server.102/b14223/basicmv.htm) Sign In](http://www.oracle.com/webapps/redirect/signon?nexturl=http://docs.oracle.com/cd/B19306_01/server.102/b14223/basicmv.htm)

[Home](http://docs.oracle.com/) / [Database](http://docs.oracle.com/en/database/database.html) / [Oracle Database Online Documentation, 10g Release 2 (10.2)](http://docs.oracle.com/cd/B19306_01/index.htm) / [Data Warehousing](http://docs.oracle.com/cd/B19306_01/nav/portal_6.htm)

Database Data Warehousing Guide

[Contents](http://docs.oracle.com/cd/B19306_01/server.102/b14223/basicmv.htm)

[Previous](http://docs.oracle.com/cd/B19306_01/server.102/b14223/constra.htm)[Next](http://docs.oracle.com/cd/B19306_01/server.102/b14223/advmv.htm)

Page 15 of 36

Top of Form

Search 

Bottom of Form

FeedbackDownload

Share to:

Categories

* [Home](http://docs.oracle.com/cd/B19306_01/index.htm)

8 Basic Materialized Views

This chapter introduces you to the use of materialized views, and discusses:

* [Overview of Data Warehousing with Materialized Views](http://docs.oracle.com/cd/B19306_01/server.102/b14223/basicmv.htm#i1006193)
* [Types of Materialized Views](http://docs.oracle.com/cd/B19306_01/server.102/b14223/basicmv.htm#i1007880)
* [Creating Materialized Views](http://docs.oracle.com/cd/B19306_01/server.102/b14223/basicmv.htm#i1006803)
* [Registering Existing Materialized Views](http://docs.oracle.com/cd/B19306_01/server.102/b14223/basicmv.htm#i1008170)
* [Choosing Indexes for Materialized Views](http://docs.oracle.com/cd/B19306_01/server.102/b14223/basicmv.htm#i1012761)
* [Dropping Materialized Views](http://docs.oracle.com/cd/B19306_01/server.102/b14223/basicmv.htm#i1008721)
* [Analyzing Materialized View Capabilities](http://docs.oracle.com/cd/B19306_01/server.102/b14223/basicmv.htm#i1008271)

Overview of Data Warehousing with Materialized Views

Typically, data flows from one or more online transaction processing (OLTP) database into a data warehouse on a monthly, weekly, or daily basis. The data is normally processed in a [**staging file**](http://docs.oracle.com/cd/B19306_01/server.102/b14223/glossary.htm#i997125) before being added to the data warehouse. Data warehouses commonly range in size from tens of gigabytes to a few terabytes. Usually, the vast majority of the data is stored in a few very large fact tables.

One technique employed in data warehouses to improve performance is the creation of summaries. Summaries are special types of aggregate views that improve query execution times by precalculating expensive joins and aggregation operations prior to execution and storing the results in a table in the database. For example, you can create a summary table to contain the sums of sales by region and by product.

The summaries or aggregates that are referred to in this book and in literature on data warehousing are created in Oracle Database using a schema object called a [**materialized view**](http://docs.oracle.com/cd/B19306_01/server.102/b14223/glossary.htm#i997008). Materialized views can perform a number of roles, such as improving query performance or providing replicated data.

In the past, organizations using summaries spent a significant amount of time and effort creating summaries manually, identifying which summaries to create, indexing the summaries, updating them, and advising their users on which ones to use. The introduction of summary management eased the workload of the database administrator and meant the user no longer needed to be aware of the summaries that had been defined. The database administrator creates one or more materialized views, which are the equivalent of a summary. The end user queries the tables and views at the detail data level. The query rewrite mechanism in the Oracle server automatically rewrites the SQL query to use the summary tables. This mechanism reduces response time for returning results from the query. Materialized views within the data warehouse are transparent to the end user or to the database application.

Although materialized views are usually accessed through the query rewrite mechanism, an end user or database application can construct queries that directly access the materialized views. However, serious consideration should be given to whether users should be allowed to do this because any change to the materialized views will affect the queries that reference them.

Materialized Views for Data Warehouses

In data warehouses, you can use materialized views to precompute and store aggregated data such as the sum of sales. Materialized views in these environments are often referred to as summaries, because they store summarized data. They can also be used to precompute joins with or without aggregations. A materialized view eliminates the overhead associated with expensive joins and aggregations for a large or important class of queries.

Materialized Views for Distributed Computing

In distributed environments, you can use materialized views to replicate data at distributed sites and to synchronize updates done at those sites with conflict resolution methods. These replica materialized views provide local access to data that otherwise would have to be accessed from remote sites. Materialized views are also useful in remote data marts. See [*Oracle Database Advanced Replication*](http://docs.oracle.com/cd/B19306_01/server.102/b14226/toc.htm) and [*Oracle Database Heterogeneous Connectivity Administrator's Guide*](http://docs.oracle.com/cd/B19306_01/server.102/b14232/toc.htm) for details on distributed and mobile computing.

Materialized Views for Mobile Computing

You can also use materialized views to download a subset of data from central servers to mobile clients, with periodic refreshes and updates between clients and the central servers.

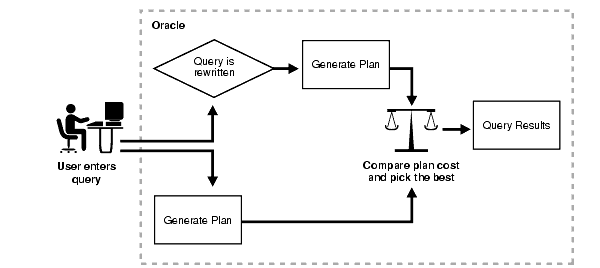
This chapter focuses on the use of materialized views in data warehouses. See [*Oracle Database Advanced Replication*](http://docs.oracle.com/cd/B19306_01/server.102/b14226/toc.htm) and [*Oracle Database Heterogeneous Connectivity Administrator's Guide*](http://docs.oracle.com/cd/B19306_01/server.102/b14232/toc.htm) for details on distributed and mobile computing.

The Need for Materialized Views

You can use materialized views to increase the speed of queries on very large databases. Queries to large databases often involve joins between tables, aggregations such as **SUM**, or both. These operations are expensive in terms of time and processing power. The type of materialized view you create determines how the materialized view is refreshed and used by query rewrite.

Materialized views improve query performance by precalculating expensive join and aggregation operations on the database prior to execution and storing the results in the database. The query optimizer automatically recognizes when an existing materialized view can and should be used to satisfy a request. It then transparently rewrites the request to use the materialized view. Queries go directly to the materialized view and not to the underlying detail tables. In general, rewriting queries to use materialized views rather than detail tables improves response time. [Figure 8-1](http://docs.oracle.com/cd/B19306_01/server.102/b14223/basicmv.htm#CACGIDEI) illustrates how query rewrite works.

***Figure 8-1 Transparent Query Rewrite***

  
[Description of "Figure 8-1 Transparent Query Rewrite"](http://docs.oracle.com/cd/B19306_01/server.102/b14223/img_text/dwhsg027.htm)

When using query rewrite, create materialized views that satisfy the largest number of queries. For example, if you identify 20 queries that are commonly applied to the detail or fact tables, then you might be able to satisfy them with five or six well-written materialized views. A materialized view definition can include any number of aggregations (**SUM**, **COUNT(x)**, **COUNT(\*)**, **COUNT(DISTINCT x)**, **AVG**, **VARIANCE**,**STDDEV**, **MIN**, and **MAX**). It can also include any number of joins. If you are unsure of which materialized views to create, Oracle provides the SQL Access Advisor, which is a set of advisory procedures in the **DBMS\_ADVISOR** package to help in designing and evaluating materialized views for query rewrite.

If a materialized view is to be used by query rewrite, it must be stored in the same database as the detail tables on which it relies. A materialized view can be partitioned, and you can define a materialized view on a partitioned table. You can also define one or more indexes on the materialized view.

Unlike indexes, materialized views can be accessed directly using a **SELECT** statement. However, it is recommended that you try to avoid writing SQL statements that directly reference the materialized view, because then it is difficult to change them without affecting the application. Instead, let query rewrite transparently rewrite your query to use the materialized view.

Note that the techniques shown in this chapter illustrate how to use materialized views in data warehouses. Materialized views can also be used by Oracle Replication. See [*Oracle Database Advanced Replication*](http://docs.oracle.com/cd/B19306_01/server.102/b14226/toc.htm) for further information.

Components of Summary Management

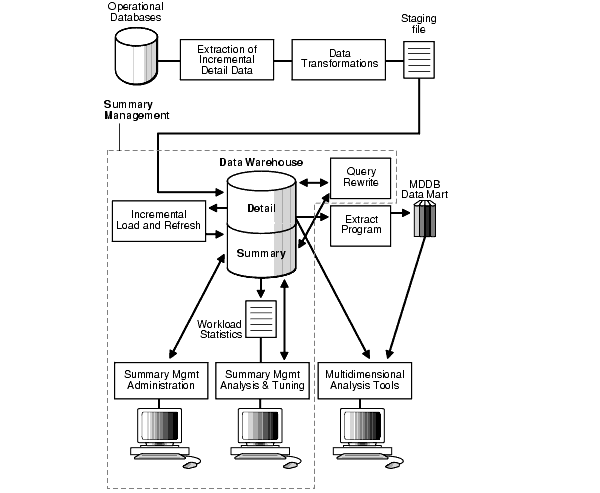
Summary management consists of:

* Mechanisms to define materialized views and dimensions.
* A refresh mechanism to ensure that all materialized views contain the latest data.
* A query rewrite capability to transparently rewrite a query to use a materialized view.
* The SQL Access Advisor, which recommends materialized views and indexes to create.
* **TUNE\_MVIEW**, which shows you how to make your materialized view fast refreshable and use general query rewrite.

The use of summary management features imposes no schema restrictions, and can enable some existing DSS database applications to improve performance without the need to redesign the database or the application.

[Figure 8-2](http://docs.oracle.com/cd/B19306_01/server.102/b14223/basicmv.htm#i1012580) illustrates the use of summary management in the warehousing cycle. After the data has been transformed, staged, and loaded into the detail data in the warehouse, you can invoke the summary management process. First, use the SQL Access Advisor to plan how you will use materialized views. Then, create materialized views and design how queries will be rewritten. If you are having problems trying to get your materialized views to work then use **TUNE\_MVIEW** to obtain an optimized materialized view.

***Figure 8-2 Overview of Summary Management***

  
[Description of "Figure 8-2 Overview of Summary Management"](http://docs.oracle.com/cd/B19306_01/server.102/b14223/img_text/dwhsg071.htm)

Understanding the summary management process during the earliest stages of data warehouse design can yield large dividends later in the form of higher performance, lower summary administration costs, and reduced storage requirements.

Data Warehousing Terminology

Some basic data warehousing terms are defined as follows:

* **Dimension** **tables** describe the business entities of an enterprise, represented as hierarchical, categorical information such as time, departments, locations, and products. Dimension tables are sometimes called lookup or reference tables.

Dimension tables usually change slowly over time and are not modified on a periodic schedule. They are used in long-running decision support queries to aggregate the data returned from the query into appropriate levels of the dimension hierarchy.

* **Hierarchies** describe the business relationships and common access patterns in the database. An analysis of the dimensions, combined with an understanding of the typical work load, can be used to create materialized views. See [Chapter 10, "Dimensions"](http://docs.oracle.com/cd/B19306_01/server.102/b14223/dimen.htm#g1011059) for more information.
* **Fact** **tables** describe the business transactions of an enterprise.

The vast majority of data in a data warehouse is stored in a few very large fact tables that are updated periodically with data from one or more operational OLTP databases.

Fact tables include facts (also called measures) such as sales, units, and inventory.

* + A simple measure is a numeric or character column of one table such as **fact.sales**.
  + A computed measure is an expression involving measures of one table, for example, **fact.revenues** - **fact.expenses**.
  + A multitable measure is a computed measure defined on multiple tables, for example, **fact\_a.revenues** - **fact\_b.expenses**.

Fact tables also contain one or more foreign keys that organize the business transactions by the relevant business entities such as time, product, and market. In most cases, these foreign keys are non-null, form a unique compound key of the fact table, and each foreign key joins with exactly one row of a [**dimension table**](http://docs.oracle.com/cd/B19306_01/server.102/b14223/glossary.htm#i996883).

* A materialized view is a precomputed table comprising aggregated and joined data from fact and possibly from dimension tables. Among builders of data warehouses, a materialized view is also known as a [**summary**](http://docs.oracle.com/cd/B19306_01/server.102/b14223/glossary.htm#i997158).

Materialized View Schema Design

Summary management can perform many useful functions, including query rewrite and materialized view refresh, even if your data warehouse design does not follow these guidelines. However, you will realize significantly greater query execution performance and materialized view refresh performance benefits and you will require fewer materialized views if your schema design complies with these guidelines.

A materialized view definition includes any number of aggregates, as well as any number of joins. In several ways, a materialized view behaves like an index:

* The purpose of a materialized view is to increase query execution performance.
* The existence of a materialized view is transparent to SQL applications, so that a database administrator can create or drop materialized views at any time without affecting the validity of SQL applications.
* A materialized view consumes storage space.
* The contents of the materialized view must be updated when the underlying detail tables are modified.

Schemas and Dimension Tables

In the case of normalized or partially normalized dimension tables (a dimension that is stored in more than one table), identify how these tables are joined. Note whether the joins between the dimension tables can guarantee that each child-side row joins with one and only one parent-side row. In the case of denormalized dimensions, determine whether the child-side columns uniquely determine the parent-side (or attribute) columns. These relationships can be enabled with constraints, using the **NOVALIDATE** and **RELY** options if the relationships represented by the constraints are guaranteed by other means. Note that if the joins between fact and dimension tables do not support the parent-child relationship described previously, you still gain significant performance advantages from defining the dimension with the **CREATEDIMENSION** statement. Another alternative, subject to some restrictions, is to use outer joins in the materialized view definition (that is, in the**CREATE** **MATERIALIZED** **VIEW** statement).

You must not create dimensions in any schema that does not satisfy these relationships. Incorrect results can be returned from queries otherwise.

Materialized View Schema Design Guidelines

Before starting to define and use the various components of summary management, you should review your schema design to abide by the following guidelines wherever possible.

Guidelines 1 and 2 are more important than guideline 3. If your schema design does not follow guidelines 1 and 2, it does not then matter whether it follows guideline 3. Guidelines 1, 2, and 3 affect both query rewrite performance and materialized view refresh performance.

***Table 8-1 Schema Design Guidelines***

| **Schema Guideline** | **Description** |
| --- | --- |
| Guideline 1  Dimensions | Dimensions should either be denormalized (each dimension contained in one table) or the joins between tables in a normalized or partially normalized dimension should guarantee that each child-side row joins with exactly one parent-side row. The benefits of maintaining this condition are described in ["Creating Dimensions"](http://docs.oracle.com/cd/B19306_01/server.102/b14223/dimen.htm#i1006266).  You can enforce this condition by adding **FOREIGN** **KEY** and **NOT NULL** constraints on the child-side join keys and**PRIMARY** **KEY** constraints on the parent-side join keys. |
| Guideline 2  Dimensions | If dimensions are denormalized or partially denormalized, hierarchical integrity must be maintained between the key columns of the dimension table. Each child key value must uniquely identify its parent key value, even if the dimension table is denormalized. Hierarchical integrity in a denormalized dimension can be verified by calling the**VALIDATE\_DIMENSION** procedure of the **DBMS\_DIMENSION** package. |
| Guideline 3  Dimensions | Fact and dimension tables should similarly guarantee that each fact table row joins with exactly one dimension table row. This condition must be declared, and optionally enforced, by adding **FOREIGN** **KEY** and **NOT** **NULL** constraints on the fact key column(s) and **PRIMARY** **KEY** constraints on the dimension key column(s), or by using outer joins. In a data warehouse, constraints are typically enabled with the **NOVALIDATE** and **RELY** clauses to avoid constraint enforcement performance overhead. See [*Oracle Database SQL Reference*](http://docs.oracle.com/cd/B19306_01/server.102/b14200/toc.htm) for further details. |
| Guideline 4  Incremental Loads | Incremental loads of your detail data should be done using the SQL\*Loader direct-path option, or any bulk loader utility that uses Oracle's direct-path interface. This includes **INSERT** ... **AS SELECT** with the **APPEND** or **PARALLEL** hints, where the hints cause the direct loader log to be used during the insert. See [*Oracle Database SQL Reference*](http://docs.oracle.com/cd/B19306_01/server.102/b14200/toc.htm) and ["Types of Materialized Views"](http://docs.oracle.com/cd/B19306_01/server.102/b14223/basicmv.htm#i1007880) for more information. |
| Guideline 5  Partitions | Range/composite partition your tables by a monotonically increasing time column if possible (preferably of type **DATE**). |
| Guideline 6  Dimensions | After each load and before refreshing your materialized view, use the **VALIDATE\_DIMENSION** procedure of the**DBMS\_DIMENSION** package to incrementally verify dimensional integrity. |
| Guideline 7  Time Dimensions | If a time dimension appears in the materialized view as a time column, partition and index the materialized view in the same manner as you have the fact tables. |

If you are concerned with the time required to enable constraints and whether any constraints might be violated, then use the **ENABLENOVALIDATE** with the **RELY** clause to turn on constraint checking without validating any of the existing constraints. The risk with this approach is that incorrect query results could occur if any constraints are broken. Therefore, as the designer, you must determine how clean the data is and whether the risk of incorrect results is too great.

Loading Data into Data Warehouses

A popular and efficient way to load data into a data warehouse or data mart is to use SQL\*Loader with the **DIRECT** or **PARALLEL** option, Data Pump, or to use another loader tool that uses the Oracle direct-path API. See [*Oracle Database Utilities*](http://docs.oracle.com/cd/B19306_01/server.102/b14215/ldr_concepts.htm#SUTIL003) for the restrictions and considerations when using SQL\*Loader with the **DIRECT** or **PARALLEL** keywords.

Loading strategies can be classified as one-phase or two-phase. In one-phase loading, data is loaded directly into the target table, quality assurance tests are performed, and errors are resolved by performing DML operations prior to refreshing materialized views. If a large number of deletions are possible, then storage utilization can be adversely affected, but temporary space requirements and load time are minimized.

In a two-phase loading process:

* Data is first loaded into a temporary table in the warehouse.
* Quality assurance procedures are applied to the data.
* Referential integrity constraints on the target table are disabled, and the local index in the target partition is marked unusable.
* The data is copied from the temporary area into the appropriate partition of the target table using **INSERT** **AS** **SELECT** with the **PARALLEL**or **APPEND** hint. The temporary table is then dropped. Alternatively, if the target table is partitioned, you can create a new (empty) partition in the target table and use **ALTER TABLE ... EXCHANGE PARTITION** to incorporate the temporary table into the target table. See [*Oracle Database SQL Reference*](http://docs.oracle.com/cd/B19306_01/server.102/b14200/toc.htm) for more information.
* The constraints are enabled, usually with the **NOVALIDATE** option.

Immediately after loading the detail data and updating the indexes on the detail data, the database can be opened for operation, if desired. You can disable query rewrite at the system level by issuing an **ALTER** **SYSTEM** **SET** **QUERY\_REWRITE\_ENABLED** = **FALSE** statement until all the materialized views are refreshed.

If **QUERY\_REWRITE\_INTEGRITY** is set to **STALE\_TOLERATED**, access to the materialized view can be allowed at the session level to any users who do not require the materialized views to reflect the data from the latest load by issuing an **ALTER** **SESSION** **SETQUERY\_REWRITE\_ENABLED** = **TRUE** statement. This scenario does not apply when **QUERY\_REWRITE\_INTEGRITY** is either **ENFORCED** or**TRUSTED** because the system ensures in these modes that only materialized views with updated data participate in a query rewrite.

Overview of Materialized View Management Tasks

The motivation for using materialized views is to improve performance, but the overhead associated with materialized view management can become a significant system management problem. When reviewing or evaluating some of the necessary materialized view management activities, consider some of the following:

* Identifying what materialized views to create initially.
* Indexing the materialized views.
* Ensuring that all materialized views and materialized view indexes are refreshed properly each time the database is updated.
* Checking which materialized views have been used.
* Determining how effective each materialized view has been on workload performance.
* Measuring the space being used by materialized views.
* Determining which new materialized views should be created.
* Determining which existing materialized views should be dropped.
* Archiving old detail and materialized view data that is no longer useful.

After the initial effort of creating and populating the data warehouse or data mart, the major administration overhead is the update process, which involves:

* Periodic extraction of incremental changes from the operational systems.
* Transforming the data.
* Verifying that the incremental changes are correct, consistent, and complete.
* Bulk-loading the data into the warehouse.
* Refreshing indexes and materialized views so that they are consistent with the detail data.

The update process must generally be performed within a limited period of time known as the [**update window**](http://docs.oracle.com/cd/B19306_01/server.102/b14223/glossary.htm#i997213). The update window depends on the [**update frequency**](http://docs.oracle.com/cd/B19306_01/server.102/b14223/glossary.htm#i997216) (such as daily or weekly) and the nature of the business. For a daily update frequency, an update window of two to six hours might be typical.

You need to know your update window for the following activities:

* Loading the detail data
* Updating or rebuilding the indexes on the detail data
* Performing quality assurance tests on the data
* Refreshing the materialized views
* Updating the indexes on the materialized views

Types of Materialized Views

The **SELECT** clause in the materialized view creation statement defines the data that the materialized view is to contain. Only a few restrictions limit what can be specified. Any number of tables can be joined together. However, they cannot be remote tables if you wish to take advantage of query rewrite. Besides tables, other elements such as views, inline views (subqueries in the **FROM** clause of a **SELECT**statement), subqueries, and materialized views can all be joined or referenced in the **SELECT** clause. You cannot, however, define a materialized view with a subquery in the **SELECT** list of the defining query. You can, however, include subqueries elsewhere in the defining query, such as in the **WHERE** clause.

The types of materialized views are:

* [Materialized Views with Aggregates](http://docs.oracle.com/cd/B19306_01/server.102/b14223/basicmv.htm#i1006519)
* [Materialized Views Containing Only Joins](http://docs.oracle.com/cd/B19306_01/server.102/b14223/basicmv.htm#i1006674)
* [Nested Materialized Views](http://docs.oracle.com/cd/B19306_01/server.102/b14223/basicmv.htm#i1006734)

Materialized Views with Aggregates

In data warehouses, materialized views normally contain aggregates as shown in [Example 8-1](http://docs.oracle.com/cd/B19306_01/server.102/b14223/basicmv.htm#i1006540). For fast refresh to be possible, the **SELECT**list must contain all of the **GROUP** **BY** columns (if present), and there must be a **COUNT(\*)** and a **COUNT(column)** on any aggregated columns. Also, materialized view logs must be present on all tables referenced in the query that defines the materialized view. The valid aggregate functions are: **SUM**, **COUNT(x)**, **COUNT(\*)**, **AVG**, **VARIANCE**, **STDDEV**, **MIN**, and **MAX**, and the expression to be aggregated can be any SQL value expression. See ["Restrictions on Fast Refresh on Materialized Views with Aggregates"](http://docs.oracle.com/cd/B19306_01/server.102/b14223/basicmv.htm#i1007028).

Fast refresh for a materialized view containing joins and aggregates is possible after any type of DML to the base tables (direct load or conventional **INSERT**, **UPDATE**, or **DELETE**). It can be defined to be refreshed **ON** **COMMIT** or **ON** **DEMAND**. A **REFRESH ON** **COMMIT**materialized view will be refreshed automatically when a transaction that does DML to one of the materialized view's detail tables commits. The time taken to complete the commit may be slightly longer than usual when this method is chosen. This is because the refresh operation is performed as part of the commit process. Therefore, this method may not be suitable if many users are concurrently changing the tables upon which the materialized view is based.

Here are some examples of materialized views with aggregates. Note that materialized view logs are only created because this materialized view will be fast refreshed.

***Example 8-1 Example 1: Creating a Materialized View***

CREATE MATERIALIZED VIEW LOG ON products WITH SEQUENCE, ROWID

(prod\_id, prod\_name, prod\_desc, prod\_subcategory, prod\_subcategory\_desc,

prod\_category, prod\_category\_desc, prod\_weight\_class, prod\_unit\_of\_measure,

prod\_pack\_size, supplier\_id, prod\_status, prod\_list\_price, prod\_min\_price)

INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW LOG ON sales

WITH SEQUENCE, ROWID

(prod\_id, cust\_id, time\_id, channel\_id, promo\_id, quantity\_sold, amount\_sold)

INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW product\_sales\_mv

PCTFREE 0 TABLESPACE demo

STORAGE (INITIAL 8k NEXT 8k PCTINCREASE 0)

BUILD IMMEDIATE

REFRESH FAST

ENABLE QUERY REWRITE

AS SELECT p.prod\_name, SUM(s.amount\_sold) AS dollar\_sales,

COUNT(\*) AS cnt, COUNT(s.amount\_sold) AS cnt\_amt

FROM sales s, products p

WHERE s.prod\_id = p.prod\_id GROUP BY p.prod\_name;

This example creates a materialized view **product\_sales\_mv** that computes total number and value of sales for a product. It is derived by joining the tables **sales** and **products** on the column **prod\_id**. The materialized view is populated with data immediately because the build method is immediate and it is available for use by query rewrite. In this example, the default refresh method is **FAST**, which is allowed because the appropriate materialized view logs have been created on tables **products** and **sales**.

***Example 8-2 Example 2: Creating a Materialized View***

CREATE MATERIALIZED VIEW product\_sales\_mv

PCTFREE 0 TABLESPACE demo

STORAGE (INITIAL 16k NEXT 16k PCTINCREASE 0)

BUILD DEFERRED

REFRESH COMPLETE ON DEMAND

ENABLE QUERY REWRITE AS

SELECT p.prod\_name, SUM(s.amount\_sold) AS dollar\_sales

FROM sales s, products p WHERE s.prod\_id = p.prod\_id

GROUP BY p.prod\_name;

This example creates a materialized view **product\_sales\_mv** that computes the sum of sales by **prod\_name**. It is derived by joining the tables **sales** and **products** on the column **prod\_id**. The materialized view does not initially contain any data, because the build method is**DEFERRED**. A complete refresh is required for the first refresh of a build deferred materialized view. When it is refreshed and once populated, this materialized view can be used by query rewrite.

***Example 8-3 Example 3: Creating a Materialized View***

CREATE MATERIALIZED VIEW LOG ON sales WITH SEQUENCE, ROWID

(prod\_id, cust\_id, time\_id, channel\_id, promo\_id, quantity\_sold, amount\_sold)

INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW sum\_sales

PARALLEL

BUILD IMMEDIATE

REFRESH FAST ON COMMIT AS

SELECT s.prod\_id, s.time\_id, COUNT(\*) AS count\_grp,

SUM(s.amount\_sold) AS sum\_dollar\_sales,

COUNT(s.amount\_sold) AS count\_dollar\_sales,

SUM(s.quantity\_sold) AS sum\_quantity\_sales,

COUNT(s.quantity\_sold) AS count\_quantity\_sales

FROM sales s

GROUP BY s.prod\_id, s.time\_id;

This example creates a materialized view that contains aggregates on a single table. Because the materialized view log has been created with all referenced columns in the materialized view's defining query, the materialized view is fast refreshable. If DML is applied against the**sales** table, then the changes will be reflected in the materialized view when the commit is issued.

Requirements for Using Materialized Views with Aggregates

[Table 8-2](http://docs.oracle.com/cd/B19306_01/server.102/b14223/basicmv.htm#g1014280) illustrates the aggregate requirements for materialized views. If aggregate **X** is present, aggregate **Y** is required and aggregate **Z**is optional.

***Table 8-2 Requirements for Materialized Views with Aggregates***

| **X** | **Y** | **Z** |
| --- | --- | --- |
| **COUNT(expr)** | **-** | **-** |
| **MIN(expr)** |  |  |
| **MAX(expr)** |  |  |
| **SUM(expr)** | **COUNT(expr)** | **-** |
| **SUM(col)**, **col** has **NOT** **NULL** constraint | - |  |
| **AVG(expr)** | **COUNT(expr)** | **SUM(expr)** |
| **STDDEV(expr)** | **COUNT(expr) SUM(expr)** | **SUM(expr \* expr)** |
| **VARIANCE(expr)** | **COUNT(expr) SUM(expr)** | **SUM(expr \* expr)** |

Note that **COUNT(\*)** must always be present to guarantee all types of fast refresh. Otherwise, you may be limited to fast refresh after inserts only. Oracle recommends that you include the optional aggregates in column **Z** in the materialized view in order to obtain the most efficient and accurate fast refresh of the aggregates.

Materialized Views Containing Only Joins

Some materialized views contain only joins and no aggregates, such as in [Example 8-4](http://docs.oracle.com/cd/B19306_01/server.102/b14223/basicmv.htm#i1006694), where a materialized view is created that joins the**sales** table to the **times** and **customers** tables. The advantage of creating this type of materialized view is that expensive joins will be precalculated.

Fast refresh for a materialized view containing only joins is possible after any type of DML to the base tables (direct-path or conventional**INSERT**, **UPDATE**, or **DELETE**).

A materialized view containing only joins can be defined to be refreshed **ON COMMIT** or **ON** **DEMAND**. If it is **ON** **COMMIT**, the refresh is performed at commit time of the transaction that does DML on the materialized view's detail table.

If you specify **REFRESH** **FAST**, Oracle performs further verification of the query definition to ensure that fast refresh can be performed if any of the detail tables change. These additional checks are:

* A materialized view log must be present for each detail table unless the table supports PCT. Also, when a materialized view log is required, the **ROWID** column must be present in each materialized view log.
* The rowids of all the detail tables must appear in the **SELECT** list of the materialized view query definition.

If some of these restrictions are not met, you can create the materialized view as **REFRESH** **FORCE** to take advantage of fast refresh when it is possible. If one of the tables did not meet all of the criteria, but the other tables did, the materialized view would still be fast refreshable with respect to the other tables for which all the criteria are met.

Materialized Join Views FROM Clause Considerations

If the materialized view contains only joins, the **ROWID** columns for each table (and each instance of a table that occurs multiple times in the**FROM** list) must be present in the **SELECT** list of the materialized view.

If the materialized view has remote tables in the **FROM** clause, all tables in the **FROM** clause must be located on that same site. Further, **ONCOMMIT** refresh is not supported for materialized view with remote tables. Materialized view logs must be present on the remote site for each detail table of the materialized view and **ROWID** columns must be present in the **SELECT** list of the materialized view, as shown in the following example.

***Example 8-4 Materialized View Containing Only Joins***

CREATE MATERIALIZED VIEW LOG ON sales WITH ROWID;

CREATE MATERIALIZED VIEW LOG ON times WITH ROWID;

CREATE MATERIALIZED VIEW LOG ON customers WITH ROWID;

CREATE MATERIALIZED VIEW detail\_sales\_mv

PARALLEL BUILD IMMEDIATE

REFRESH FAST AS

SELECT s.rowid "sales\_rid", t.rowid "times\_rid", c.rowid "customers\_rid",

c.cust\_id, c.cust\_last\_name, s.amount\_sold, s.quantity\_sold, s.time\_id

FROM sales s, times t, customers c

WHERE s.cust\_id = c.cust\_id(+) AND s.time\_id = t.time\_id(+);

Alternatively, if the previous example did not include the columns **times\_rid** and **customers\_rid**, and if the refresh method was **REFRESHFORCE**, then this materialized view would be fast refreshable only if the sales table was updated but not if the tables **times** or **customers**were updated.

CREATE MATERIALIZED VIEW detail\_sales\_mv

PARALLEL

BUILD IMMEDIATE

REFRESH FORCE AS

SELECT s.rowid "sales\_rid", c.cust\_id, c.cust\_last\_name, s.amount\_sold,

s.quantity\_sold, s.time\_id

FROM sales s, times t, customers c

WHERE s.cust\_id = c.cust\_id(+) AND s.time\_id = t.time\_id(+);

Nested Materialized Views

A nested materialized view is a materialized view whose definition is based on another materialized view. A nested materialized view can reference other relations in the database in addition to referencing materialized views.

Why Use Nested Materialized Views?

In a data warehouse, you typically create many aggregate views on a single join (for example, rollups along different dimensions). Incrementally maintaining these distinct materialized aggregate views can take a long time, because the underlying join has to be performed many times.

Using nested materialized views, you can create multiple single-table materialized views based on a joins-only materialized view and the join is performed just once. In addition, optimizations can be performed for this class of single-table aggregate materialized view and thus refresh is very efficient.

***Example 8-5 Nested Materialized View***

You can create a nested materialized view on materialized views, but all parent and base materialized views must contain joins or aggregates. If the defining queries for a materialized view do not contain joins or aggregates, it cannot be nested. All the underlying objects (materialized views or tables) on which the materialized view is defined must have a materialized view log. All the underlying objects are treated as if they were tables. In addition, you can use all the existing options for materialized views.

Using the tables and their columns from the **sh** sample schema, the following materialized views illustrate how nested materialized views can be created.

CREATE MATERIALIZED VIEW LOG ON sales WITH ROWID;

CREATE MATERIALIZED VIEW LOG ON customers WITH ROWID;

CREATE MATERIALIZED VIEW LOG ON times WITH ROWID;

/\*create materialized view join\_sales\_cust\_time as fast refreshable at

COMMIT time \*/

CREATE MATERIALIZED VIEW join\_sales\_cust\_time

REFRESH FAST ON COMMIT AS

SELECT c.cust\_id, c.cust\_last\_name, s.amount\_sold, t.time\_id,

t.day\_number\_in\_week, s.rowid srid, t.rowid trid, c.rowid crid

FROM sales s, customers c, times t

WHERE s.time\_id = t.time\_id AND s.cust\_id = c.cust\_id;

To create a nested materialized view on the table **join\_sales\_cust\_time***,* you would have to create a materialized view log on the table. Because this will be a single-table aggregate materialized view on **join\_sales\_cust\_time**, you need to log all the necessary columns and use the **INCLUDING** **NEW** **VALUES** clause.

/\* create materialized view log on join\_sales\_cust\_time \*/

CREATE MATERIALIZED VIEW LOG ON join\_sales\_cust\_time

WITH ROWID (cust\_last\_name, day\_number\_in\_week, amount\_sold)

INCLUDING NEW VALUES;

/\* create the single-table aggregate materialized view sum\_sales\_cust\_time

on join\_sales\_cust\_time as fast refreshable at COMMIT time \*/

CREATE MATERIALIZED VIEW sum\_sales\_cust\_time

REFRESH FAST ON COMMIT AS

SELECT COUNT(\*) cnt\_all, SUM(amount\_sold) sum\_sales, COUNT(amount\_sold)

cnt\_sales, cust\_last\_name, day\_number\_in\_week

FROM join\_sales\_cust\_time

GROUP BY cust\_last\_name, day\_number\_in\_week;

Nesting Materialized Views with Joins and Aggregates

Some types of nested materialized views cannot be fast refreshed. Use **EXPLAIN\_MVIEW** to identify those types of materialized views. You can refresh a tree of nested materialized views in the appropriate dependency order by specifying the **nested = TRUE** parameter with the**DBMS\_MVIEW.REFRESH** parameter. For example, if you call **DBMS\_MVIEW.REFRESH ('SUM\_SALES\_CUST\_TIME', nested => TRUE)**, the**REFRESH** procedure will first refresh the **join\_sales\_cust\_time** materialized view, and then refresh the **sum\_sales\_cust\_time**materialized view.

Nested Materialized View Usage Guidelines

You should keep the following in mind when deciding whether to use nested materialized views:

* If you want to use fast refresh, you should fast refresh all the materialized views along any chain.
* If you want the highest level materialized view to be fresh with respect to the detail tables, you need to ensure that all materialized views in a tree are refreshed in the correct dependency order before refreshing the highest-level. You can automatically refresh intermediate materialized views in a nested hierarchy using the **nested = TRUE** parameter, as described in ["Nesting Materialized Views with Joins and Aggregates"](http://docs.oracle.com/cd/B19306_01/server.102/b14223/basicmv.htm#CHDIDDEJ). If you do not specify **nested = TRUE** and the materialized views under the highest-level materialized view are stale, refreshing only the highest-level will succeed, but makes it fresh only with respect to its underlying materialized view, not the detail tables at the base of the tree.
* When refreshing materialized views, you need to ensure that all materialized views in a tree are refreshed. If you only refresh the highest-level materialized view, the materialized views under it will be stale and you must explicitly refresh them. If you use the **REFRESH** procedure with the **nested** parameter value set to **TRUE**, only specified materialized views and their child materialized views in the tree are refreshed, and not their top-level materialized views. Use the **REFRESH\_DEPENDENT** procedure with the nested parameter value set to**TRUE** if you want to ensure that all materialized views in a tree are refreshed.
* Freshness of a materialized view is calculated relative to the objects directly referenced by the materialized view. When a materialized view references another materialized view, the freshness of the topmost materialized view is calculated relative to changes in the materialized view it directly references, not relative to changes in the tables referenced by the materialized view it references.

Restrictions When Using Nested Materialized Views

You cannot create both a materialized view and a prebuilt materialized view on the same table. For example, If you have a table **costs** with a materialized view **cost\_mv** based on it, you cannot then create a prebuilt materialized view on table **costs**. The result would make**cost\_mv** a nested materialized view and this method of conversion is not supported.

Creating Materialized Views

A materialized view can be created with the **CREATE** **MATERIALIZED** **VIEW** statement or using Enterprise Manager. [Example 8-6](http://docs.oracle.com/cd/B19306_01/server.102/b14223/basicmv.htm#i1006810) illustrates creating an materialized view called **cust\_sales\_mv**.

***Example 8-6 Creating a Materialized View***

CREATE MATERIALIZED VIEW cust\_sales\_mv

PCTFREE 0 TABLESPACE demo

STORAGE (INITIAL 16k NEXT 16k PCTINCREASE 0)

PARALLEL

BUILD IMMEDIATE

REFRESH COMPLETE

ENABLE QUERY REWRITE AS

SELECT c.cust\_last\_name, SUM(amount\_sold) AS sum\_amount\_sold

FROM customers c, sales s WHERE s.cust\_id = c.cust\_id

GROUP BY c.cust\_last\_name;

It is not uncommon in a data warehouse to have already created summary or aggregation tables, and you might not wish to repeat this work by building a new materialized view. In this case, the table that already exists in the database can be registered as a prebuilt materialized view. This technique is described in ["Registering Existing Materialized Views"](http://docs.oracle.com/cd/B19306_01/server.102/b14223/basicmv.htm#i1008170).

Once you have selected the materialized views you want to create, follow these steps for each materialized view.

1. Design the materialized view. Existing user-defined materialized views do not require this step. If the materialized view contains many rows, then, if appropriate, the materialized view should be partitioned (if possible) and should match the partitioning of the largest or most frequently updated detail or fact table (if possible). Refresh performance benefits from partitioning, because it can take advantage of parallel DML capabilities and possible PCT-based refresh.
2. Use the **CREATE** **MATERIALIZED** **VIEW** statement to create and, optionally, populate the materialized view. If a user-defined materialized view already exists, then use the **ON** **PREBUILT** **TABLE** clause in the **CREATE** **MATERIALIZED** **VIEW** statement. Otherwise, use the **BUILDIMMEDIATE** clause to populate the materialized view immediately, or the **BUILD** **DEFERRED** clause to populate the materialized view later. A **BUILD** **DEFERRED** materialized view is disabled for use by query rewrite until the first **COMPLETE** **REFRESH**, after which it will be automatically enabled, provided the **ENABLE** **QUERY** **REWRITE** clause has been specified.

**See Also:**

[*Oracle Database SQL Reference*](http://docs.oracle.com/cd/B19306_01/server.102/b14200/toc.htm) for descriptions of the SQL statements **CREATE** **MATERIALIZED** **VIEW**, **ALTERMATERIALIZED** **VIEW**, and **DROP** **MATERIALIZED** **VIEW**

Creating Materialized Views with Column Alias Lists

Currently, when a materialized view is created, if its defining query contains same-name columns in the **SELECT** list, the name conflicts need to be resolved by specifying unique aliases for those columns. Otherwise, the **CREATE** **MATERIALIZED** **VIEW** statement will fail with the error messages of columns ambiguously defined. However, the standard method of attaching aliases in the **SELECT** clause for name resolution restricts the use of the full text match query rewrite and it will occur only when the text of the materialized view's defining query and the text of user input query are identical. Thus, if the user specifies select aliases in the materialized view's defining query while there is no alias in the query, the full text match comparison will fail. This is particularly a problem for queries from Discoverer, which makes extensive use of column aliases.

The following is an example of the problem. **sales\_mv** is created with column aliases in the **SELECT** clause but the input query **Q1** does not have the aliases. The full text match rewrite will fail. The materialized view is as follows:

CREATE MATERIALIZED VIEW sales\_mv

ENABLE QUERY REWRITE AS

SELECT s.time\_id sales\_tid, c.time\_id costs\_tid

FROM sales s, products p, costs c

WHERE s.prod\_id = p.prod\_id AND c.prod\_id = p.prod\_id AND

p.prod\_name IN (SELECT prod\_name FROM products);

Input query statement **Q1** is as follows:

SELECT s.time\_id, c.time\_id

FROM sales s, products p, costs c

WHERE s.prod\_id = p.prod\_id AND c.prod\_id = p.prod\_id AND

p.prod\_name IN (SELECT prod\_name FROM products);

Even though the materialized view's defining query is almost identical and logically equivalent to the user's input query, query rewrite does not happen because of the failure of full text match that is the only rewrite possibility for some queries (for example, a subquery in the **WHERE**clause).

You can add a column alias list to a **CREATE** **MATERIALIZED** **VIEW** statement. The column alias list explicitly resolves any column name conflict without attaching aliases in the **SELECT** clause of the materialized view. The syntax of the materialized view column alias list is illustrated in the following example:

CREATE MATERIALIZED VIEW sales\_mv (sales\_tid, costs\_tid)

ENABLE QUERY REWRITE AS

SELECT s.time\_id, c.time\_id

FROM sales s, products p, costs c

WHERE s.prod\_id = p.prod\_id AND c.prod\_id = p.prod\_id AND

p.prod\_name IN (SELECT prod\_name FROM products);

In this example, the defining query of **sales\_mv** now matches exactly with the user query **Q1**, so full text match rewrite will take place.

Note that when aliases are specified in both the **SELECT** clause and the new alias list clause, the alias list clause supersedes the ones in the**SELECT** clause.

Naming Materialized Views

The name of a materialized view must conform to standard Oracle naming conventions. However, if the materialized view is based on a user-defined prebuilt table, then the name of the materialized view must exactly match that table name.

If you already have a naming convention for tables and indexes, you might consider extending this naming scheme to the materialized views so that they are easily identifiable. For example, instead of naming the materialized view **sum\_of\_sales**, it could be called**sum\_of\_sales\_mv** to denote that this is a materialized view and not a table or view.

Storage And Table Compression

Unless the materialized view is based on a user-defined prebuilt table, it requires and occupies storage space inside the database. Therefore, the storage needs for the materialized view should be specified in terms of the tablespace where it is to reside and the size of the extents.

If you do not know how much space the materialized view will require, then the **DBMS\_MVIEW.ESTIMATE\_SIZE** package can estimate the number of bytes required to store this uncompressed materialized view. This information can then assist the design team in determining the tablespace in which the materialized view should reside.

You should use table compression with highly redundant data, such as tables with many foreign keys. This is particularly useful for materialized views created with the **ROLLUP** clause. Table compression reduces disk use and memory use (specifically, the buffer cache), often leading to a better scaleup for read-only operations. Table compression can also speed up query execution at the expense of update cost.

**See Also:**

[*Oracle Database SQL Reference*](http://docs.oracle.com/cd/B19306_01/server.102/b14200/statements_10.htm#SQLRF017) for a complete description of **STORAGE** semantics, [*Oracle Database Performance Tuning Guide*](http://docs.oracle.com/cd/B19306_01/server.102/b14211/toc.htm), and [Chapter 5, "Partitioning in Data Warehouses"](http://docs.oracle.com/cd/B19306_01/server.102/b14223/parpart.htm#g1020112) for table compression examples

Build Methods

Two build methods are available for creating the materialized view, as shown in [Table 8-3](http://docs.oracle.com/cd/B19306_01/server.102/b14223/basicmv.htm#g1014320). If you select **BUILD** **IMMEDIATE**, the materialized view definition is added to the schema objects in the data dictionary, and then the fact or detail tables are scanned according to the **SELECT**expression and the results are stored in the materialized view. Depending on the size of the tables to be scanned, this build process can take a considerable amount of time.

An alternative approach is to use the **BUILD** **DEFERRED** clause, which creates the materialized view without data, thereby enabling it to be populated at a later date using the **DBMS\_MVIEW.REFRESH** package described in [Chapter 15, "Maintaining the Data Warehouse"](http://docs.oracle.com/cd/B19306_01/server.102/b14223/refresh.htm#g1018554).

***Table 8-3 Build Methods***

| **Build Method** | **Description** |
| --- | --- |
| **BUILD** **IMMEDIATE** | Create the materialized view and then populate it with data. |
| **BUILD** **DEFERRED** | Create the materialized view definition but do not populate it with data. |

Enabling Query Rewrite

Before creating a materialized view, you can verify what types of query rewrite are possible by calling the procedure**DBMS\_MVIEW.EXPLAIN\_MVIEW**, or use **DBMS\_ADVISOR.TUNE\_MVIEW** to optimize the materialized view so that many types of query rewrite are possible. Once the materialized view has been created, you can use **DBMS\_MVIEW.EXPLAIN\_REWRITE** to find out if (or why not) it will rewrite a specific query.

Even though a materialized view is defined, it will not automatically be used by the query rewrite facility. Even though query rewrite is enabled by default, you also must specify the **ENABLE** **QUERY** **REWRITE** clause if the materialized view is to be considered available for rewriting queries.

If this clause is omitted or specified as **DISABLE** **QUERY** **REWRITE** when the materialized view is created, the materialized view can subsequently be enabled for query rewrite with the **ALTER** **MATERIALIZED** **VIEW** statement.

If you define a materialized view as **BUILD** **DEFERRED**, it is not eligible for query rewrite until it is populated with data through a complete refresh.

Query Rewrite Restrictions

Query rewrite is not possible with all materialized views. If query rewrite is not occurring when expected, **DBMS\_MVIEW.EXPLAIN\_REWRITE**can help provide reasons why a specific query is not eligible for rewrite. If this shows that not all types of query rewrite are possible, use the procedure **DBMS\_ADVISOR.TUNE\_MVIEW** to see if the materialized view can be defined differently so that query rewrite is possible. Also, check to see if your materialized view satisfies all of the following conditions.

Materialized View Restrictions

You should keep in mind the following restrictions:

* The defining query of the materialized view cannot contain any non-repeatable expressions (**ROWNUM**, **SYSDATE**, non-repeatable PL/SQL functions, and so on).
* The query cannot contain any references to **RAW** or **LONG** **RAW** datatypes or object **REF**s.
* If the materialized view was registered as **PREBUILT**, the precision of the columns must agree with the precision of the corresponding**SELECT** expressions unless overridden by the **WITH** **REDUCED** **PRECISION** clause.

General Query Rewrite Restrictions

You should keep in mind the following restrictions:

* If a query has both local and remote tables, only local tables will be considered for potential rewrite.
* Neither the detail tables nor the materialized view can be owned by **SYS**.
* If a column or expression is present in the **GROUP** **BY** clause of the materialized view, it must also be present in the **SELECT** list.
* Aggregate functions must occur only as the outermost part of the expression. That is, aggregates such as **AVG(AVG(x))** or **AVG(x)**+**AVG(x)** are not allowed.
* **CONNECT** **BY** clauses are not allowed.

Refresh Options

When you define a materialized view, you can specify three refresh options: how to refresh, what type of refresh, and can trusted constraints be used. If unspecified, the defaults are assumed as **ON** **DEMAND**, **FORCE**, and **ENFORCED** constraints respectively.

The two refresh execution modes are **ON** **COMMIT** and **ON** **DEMAND**. Depending on the materialized view you create, some of the options may not be available. [Table 8-4](http://docs.oracle.com/cd/B19306_01/server.102/b14223/basicmv.htm#g1014263) describes the refresh modes.

***Table 8-4 Refresh Modes***

| **Refresh Mode** | **Description** |
| --- | --- |
| **ON COMMIT** | Refresh occurs automatically when a transaction that modified one of the materialized view's detail tables commits. This can be specified as long as the materialized view is fast refreshable (in other words, not complex). The **ON** **COMMIT**privilege is necessary to use this mode. |
| **ON DEMAND** | Refresh occurs when a user manually executes one of the available refresh procedures contained in the **DBMS\_MVIEW**package (**REFRESH**, **REFRESH\_ALL\_MVIEWS**, **REFRESH\_DEPENDENT**). |

When a materialized view is maintained using the **ON** **COMMIT** method, the time required to complete the commit may be slightly longer than usual. This is because the refresh operation is performed as part of the commit process. Therefore this method may not be suitable if many users are concurrently changing the tables upon which the materialized view is based.

If you anticipate performing insert, update or delete operations on tables referenced by a materialized view concurrently with the refresh of that materialized view, and that materialized view includes joins and aggregation, Oracle recommends you use **ON** **COMMIT** fast refresh rather than **ON** **DEMAND** fast refresh.

If you think the materialized view did not refresh, check the alert log or trace file.

If a materialized view fails during refresh at **COMMIT** time, you must explicitly invoke the refresh procedure using the **DBMS\_MVIEW** package after addressing the errors specified in the trace files. Until this is done, the materialized view will no longer be refreshed automatically at commit time.

You can specify how you want your materialized views to be refreshed from the detail tables by selecting one of four options: **COMPLETE**,**FAST**, **FORCE**, and **NEVER**. [Table 8-5](http://docs.oracle.com/cd/B19306_01/server.102/b14223/basicmv.htm#g1014333) describes the refresh options.

***Table 8-5 Refresh Options***

| **Refresh Option** | **Description** |
| --- | --- |
| **COMPLETE** | Refreshes by recalculating the materialized view's defining query. |
| **FAST** | Applies incremental changes to refresh the materialized view using the information logged in the materialized view logs, or from a SQL\*Loader direct-path or a partition maintenance operation. |
| **FORCE** | Applies **FAST** refresh if possible; otherwise, it applies **COMPLETE** refresh. |
| **NEVER** | Indicates that the materialized view will not be refreshed with refresh mechanisms. |

Whether the fast refresh option is available depends upon the type of materialized view. You can call the procedure**DBMS\_MVIEW.EXPLAIN\_MVIEW** to determine whether fast refresh is possible.

You can also specify if it is acceptable to use trusted constraints and **QUERY\_REWRITE\_INTEGRITY = TRUSTED** during refresh. Any nonvalidated **RELY** constraint is a trusted constraint. For example, nonvalidated foreign key/primary key relationships, functional dependencies defined in dimensions or a materialized view in the **UNKNOWN** state. If query rewrite is enabled during refresh, these can improve the performance of refresh by enabling more performant query rewrites. Any materialized view that can use **TRUSTED** constraints for refresh is left in a state of trusted freshness (the **UNKNOWN** state) after refresh.

This is reflected in the column **STALENESS** in the view **USER\_MVIEWS**. The column **UNKNOWN\_TRUSTED\_FD** in the same view is also set to **Y**, which means yes.

You can define this property of the materialized view either during create time by specifying **REFRESH** **USING** **TRUSTED** **[ENFORCED]CONSTRAINTS** or by using **ALTER** **MATERIALIZED** **VIEW** DDL.

***Table 8-6 Constraints***

| **Constraints to Use** | **Description** |
| --- | --- |
| TRUSTED CONSTRAINTS | Refresh can use trusted constraints and **QUERY\_REWRITE\_INTEGRITY** **= TRUSTED** during refresh.This allows use of non-validated **RELY** constraints and rewrite against materialized views in **UNKNOWN** or **FRESH**state during refresh. |
| ENFORCED CONSTRAINTS | Refresh can use validated constraints and **QUERY\_REWRITE\_INTEGRITY = ENFORCED** during refresh. This allows use of only validated, enforced constraints and rewrite against materialized views in **FRESH** state during refresh. |

General Restrictions on Fast Refresh

The defining query of the materialized view is restricted as follows:

* The materialized view must not contain references to non-repeating expressions like **SYSDATE** and **ROWNUM**.
* The materialized view must not contain references to **RAW** or **LONG** **RAW** data types.
* It cannot contain a **SELECT** list subquery.
* It cannot contain analytical functions (for example, **RANK**) in the **SELECT** clause.
* It cannot contain a **MODEL** clause.
* It cannot contain a **HAVING** clause with a subquery.
* It cannot contain nested queries that have **ANY**, **ALL**, or **NOT** **EXISTS**.
* It cannot contain a **[START WITH …] CONNECT BY** clause.
* It cannot contain multiple detail tables at different sites.
* On-commit materialized view cannot have remote detail tables.
* Nested materialized views must have a join or aggregate.

Restrictions on Fast Refresh on Materialized Views with Joins Only

Defining queries for materialized views with joins only and no aggregates have the following restrictions on fast refresh:

* All restrictions from ["General Restrictions on Fast Refresh"](http://docs.oracle.com/cd/B19306_01/server.102/b14223/basicmv.htm#i1007007).
* They cannot have **GROUP** **BY** clauses or aggregates.
* Rowids of all the tables in the **FROM** list must appear in the **SELECT** list of the query.
* Materialized view logs must exist with rowids for all the base tables in the **FROM** list of the query.

Restrictions on Fast Refresh on Materialized Views with Aggregates

Defining queries for materialized views with aggregates or joins have the following restrictions on fast refresh:

* All restrictions from ["General Restrictions on Fast Refresh"](http://docs.oracle.com/cd/B19306_01/server.102/b14223/basicmv.htm#i1007007).

Fast refresh is supported for both **ON** **COMMIT** and **ON** **DEMAND** materialized views, however the following restrictions apply:

* All tables in the materialized view must have materialized view logs, and the materialized view logs must:
  + Contain all columns from the table referenced in the materialized view. However, none of these columns in the base table can be encrypted.
  + Specify with **ROWID** and **INCLUDING** **NEW** **VALUES**.
  + Specify the **SEQUENCE** clause if the table is expected to have a mix of inserts/direct-loads, deletes, and updates.
* Only **SUM**, **COUNT**, **AVG**, **STDDEV**, **VARIANCE**, **MIN** and **MAX** are supported for fast refresh.
* **COUNT(\*)** must be specified.
* For each aggregate such as **AVG(expr)**, the corresponding **COUNT(expr)** must be present. Oracle recommends that **SUM(expr)** be specified. See [Table 8-2](http://docs.oracle.com/cd/B19306_01/server.102/b14223/basicmv.htm#g1014280) for further details.
* If **VARIANCE(expr)** or **STDDEV(expr**) is specified, **COUNT(expr)** and **SUM(expr)** must be specified. Oracle recommends that**SUM(expr \*expr)** be specified. See [Table 8-2](http://docs.oracle.com/cd/B19306_01/server.102/b14223/basicmv.htm#g1014280) for further details.
* The **SELECT** column in the defining query cannot be a complex expression with columns from multiple base tables. A possible workaround to this is to use a nested materialized view.
* The **SELECT** list must contain all **GROUP** **BY** columns.
* If the materialized view has one of the following, then fast refresh is supported only on conventional DML inserts and direct loads.
  + Materialized views with **MIN** or **MAX** aggregates
  + Materialized views which have **SUM(expr)** but no **COUNT(expr)**
  + Materialized views without **COUNT(\*)**

Such a materialized view is called an insert-only materialized view.

* A materialized view with **MAX** or **MIN** is fast refreshable after delete or mixed DML statements if it does not have a **WHERE** clause.
* Materialized views with named views or subqueries in the **FROM** clause can be fast refreshed provided the views can be completely merged. For information on which views will merge, refer to the [*Oracle Database Performance Tuning Guide*](http://docs.oracle.com/cd/B19306_01/server.102/b14211/toc.htm).
* If there are no outer joins, you may have arbitrary selections and joins in the **WHERE** clause.
* Materialized aggregate views with outer joins are fast refreshable after conventional DML and direct loads, provided only the outer table has been modified. Also, unique constraints must exist on the join columns of the inner join table. If there are outer joins, all the joins must be connected by **AND**s and must use the equality (**=**) operator.
* For materialized views with **CUBE**, **ROLLUP**, grouping sets, or concatenation of them, the following restrictions apply:
  + The **SELECT** list should contain grouping distinguisher that can either be a **GROUPING\_ID** function on all **GROUP** **BY** expressions or**GROUPING** functions one for each **GROUP** **BY** expression. For example, if the **GROUP** **BY** clause of the materialized view is "**GROUP** **BYCUBE(a, b)**", then the **SELECT** list should contain either "**GROUPING\_ID(a, b)**" or "**GROUPING(a)** **AND** **GROUPING(b)**" for the materialized view to be fast refreshable.
  + **GROUP** **BY** should not result in any duplicate groupings. For example, "**GROUP BY a, ROLLUP(a, b)**" is not fast refreshable because it results in duplicate groupings "**(a), (a, b), AND (a)**".

Restrictions on Fast Refresh on Materialized Views with UNION ALL

Materialized views with the **UNION** **ALL** set operator support the **REFRESH** **FAST** option if the following conditions are satisfied:

* The defining query must have the **UNION** **ALL** operator at the top level.

The **UNION** **ALL** operator cannot be embedded inside a subquery, with one exception: The **UNION** **ALL** can be in a subquery in the **FROM**clause provided the defining query is of the form **SELECT \* FROM** (view or subquery with **UNION** **ALL**) as in the following example:

CREATE VIEW view\_with\_unionall AS

(SELECT c.rowid crid, c.cust\_id, 2 umarker

FROM customers c WHERE c.cust\_last\_name = 'Smith'

UNION ALL

SELECT c.rowid crid, c.cust\_id, 3 umarker

FROM customers c WHERE c.cust\_last\_name = 'Jones');

CREATE MATERIALIZED VIEW unionall\_inside\_view\_mv

REFRESH FAST ON DEMAND AS

SELECT \* FROM view\_with\_unionall;

Note that the view **view\_with\_unionall** satisfies the requirements for fast refresh.

* Each query block in the **UNION** **ALL** query must satisfy the requirements of a fast refreshable materialized view with aggregates or a fast refreshable materialized view with joins.

The appropriate materialized view logs must be created on the tables as required for the corresponding type of fast refreshable materialized view.

Note that the Oracle Database also allows the special case of a single table materialized view with joins only provided the **ROWID** column has been included in the **SELECT** list and in the materialized view log. This is shown in the defining query of the view**view\_with\_unionall**.

* The **SELECT** list of each query must include a maintenance column, called a **UNION** **ALL** marker. The **UNION** **ALL** column must have a distinct constant numeric or string value in each **UNION** **ALL** branch. Further, the marker column must appear in the same ordinal position in the **SELECT** list of each query block.
* Some features such as outer joins, insert-only aggregate materialized view queries and remote tables are not supported for materialized views with **UNION** **ALL**.
* Partiton Change Tracking (PCT)-based refresh is not supported for **UNION** **ALL** materialized views.
* The compatibility initialization parameter must be set to 9.2.0 or higher to create a fast refreshable materialized view with **UNION** **ALL**.

Achieving Refresh Goals

In addition to the **EXPLAIN\_MVIEW** procedure, which is discussed throughout this chapter, you can use the **DBMS\_ADVISOR.TUNE\_MVIEW**procedure to optimize a **CREATE** **MATERIALIZED** **VIEW** statement to achieve **REFRESH** **FAST** and **ENABLE** **QUERY** **REWRITE** goals.

Refreshing Nested Materialized Views

A nested materialized view is considered to be fresh as long as its data is synchronized with the data in its detail tables, even if some of its detail tables could be stale materialized views.

You can refresh nested materialized views in two ways: **DBMS\_MVIEW.REFRESH** with the **nested** flag set to **TRUE** and **REFRESH\_DEPENDENT**with the **nested** flag set to **TRUE** on the base tables. If you use **DBMS\_MVIEW.REFRESH**, the entire materialized view chain is refreshed and the coverage starting from the specified materialized view in top-down fashion. That is, the specified materialized view and all its child materialized views in the dependency hierarchy are refreshed in order. With **DBMS\_MVIEW.REFRESH\_DEPENDENT**, the entire chain is refreshed from the bottom up. That is, all the parent materialized views in the dependency hierarchy starting from the specified table are refreshed in order.

***Example 8-7 Example of Refreshing a Nested Materialized View***

The following statement shows an example of refreshing a nested materialized view:

DBMS\_MVIEW.REFRESH('SALES\_MV,COST\_MV', nested => TRUE);

This statement will first refresh all child materialized views of **sales\_mv** and **cost\_mv** based on the dependency analysis and then refresh the two specified materialized views.

You can query the **STALE\_SINCE** column in the **\*\_MVIEWS** views to find out when a materialized view became stale.

ORDER BY Clause

An **ORDER** **BY** clause is allowed in the **CREATE** **MATERIALIZED** **VIEW** statement. It is used only during the initial creation of the materialized view. It is not used during a full refresh or a fast refresh.

To improve the performance of queries against large materialized views, store the rows in the materialized view in the order specified in the**ORDER** **BY** clause. This initial ordering provides physical clustering of the data. If indexes are built on the columns by which the materialized view is ordered, accessing the rows of the materialized view using the index often reduces the time for disk I/O due to the physical clustering.

The **ORDER** **BY** clause is not considered part of the materialized view definition. As a result, there is no difference in the manner in which Oracle Database detects the various types of materialized views (for example, materialized join views with no aggregates). For the same reason, query rewrite is not affected by the **ORDER** **BY** clause. This feature is similar to the **CREATE** **TABLE** ... **ORDER** **BY** capability.

Materialized View Logs

Materialized view logs are required if you want to use fast refresh, with the exception of PCT refresh. That is, if a detail table supports PCT for a materialized view, the materialized view log on that detail table is not required in order to do fast refresh on that materialized view. As a general rule, though, you should create materialized view logs if you want to use fast refresh. Materialized view logs are defined using a**CREATE** **MATERIALIZED** **VIEW** **LOG** statement on the base table that is to be changed. They are not created on the materialized view unless there is another materialized view on top of that materialized view, which is the case with nested materialized views. For fast refresh of materialized views, the definition of the materialized view logs must normally specify the **ROWID** clause. In addition, for aggregate materialized views, it must also contain every column in the table referenced in the materialized view, the **INCLUDING** **NEW** **VALUES** clause and the**SEQUENCE** clause.

An example of a materialized view log is shown as follows where one is created on the table **sales**:

CREATE MATERIALIZED VIEW LOG ON sales WITH ROWID

(prod\_id, cust\_id, time\_id, channel\_id, promo\_id, quantity\_sold, amount\_sold)

INCLUDING NEW VALUES;

Alternatively, a materialized view log can be amended to include the rowid, as in the following:

ALTER MATERIALIZED VIEW LOG ON sales ADD ROWID

(prod\_id, cust\_id, time\_id, channel\_id, promo\_id, quantity\_sold, amount\_sold)

INCLUDING NEW VALUES;

Oracle recommends that the keyword **SEQUENCE** be included in your materialized view log statement unless you are sure that you will never perform a mixed DML operation (a combination of **INSERT**, **UPDATE**, or **DELETE** operations on multiple tables). The **SEQUENCE** column is required in the materialized view log to support fast refresh with a combination of **INSERT**, **UPDATE**, or **DELETE** statements on multiple tables. You can, however, add the **SEQUENCE** number to the materialized view log after it has been created.

The boundary of a mixed DML operation is determined by whether the materialized view is **ON** **COMMIT** or **ON** **DEMAND**.

* For **ON** **COMMIT**, the mixed DML statements occur within the same transaction because the refresh of the materialized view will occur upon commit of this transaction.
* For **ON** **DEMAND**, the mixed DML statements occur between refreshes. The following example of a materialized view log illustrates where one is created on the table **sales** that includes the **SEQUENCE** keyword:
* CREATE MATERIALIZED VIEW LOG ON sales WITH SEQUENCE, ROWID
* (prod\_id, cust\_id, time\_id, channel\_id, promo\_id,
* quantity\_sold, amount\_sold) INCLUDING NEW VALUES;

Using the FORCE Option with Materialized View Logs

If you specify **FORCE** and any items specified with the **ADD** clause have already been specified for the materialized view log, Oracle does not return an error, but silently ignores the existing elements and adds to the materialized view log any items that do not already exist in the log. For example, if you used a filter column such as **cust\_id** and this column already existed, Oracle Database ignores the redundancy and does not return an error.

Using Oracle Enterprise Manager

A materialized view can also be created using Enterprise Manager by selecting the materialized view object type. There is no difference in the information required if this approach is used.

Using Materialized Views with NLS Parameters

When using certain materialized views, you must ensure that your NLS parameters are the same as when you created the materialized view. Materialized views with this restriction are as follows:

* Expressions that may return different values, depending on NLS parameter settings. For example, (**date > "01/02/03"**) or (**rate <= "2.150"**) are NLS parameter dependent expressions.
* Equijoins where one side of the join is character data. The result of this equijoin depends on collation and this can change on a session basis, giving an incorrect result in the case of query rewrite or an inconsistent materialized view after a refresh operation.
* Expressions that generate internal conversion to character data in the **SELECT** list of a materialized view, or inside an aggregate of a materialized aggregate view. This restriction does not apply to expressions that involve only numeric data, for example, **a+b** where **a** and**b** are numeric fields.

Adding Comments to Materialized Views

You can add a comment to a materialized view. For example, the following statement adds a comment to data dictionary views for the existing materialized view:

COMMENT ON MATERIALIZED VIEW sales\_mv IS 'sales materialized view';

To view the comment after the preceding statement execution, the user can query the catalog views, **{USER**, **DBA}** **ALL\_MVIEW\_COMMENTS**. For example:

SELECT MVIEW\_NAME, COMMENTS

FROM USER\_MVIEW\_COMMENTS WHERE MVIEW\_NAME = 'SALES\_MV';

The output will resemble the following:

MVIEW\_NAME COMMENTS

----------- -----------------------

SALES\_MV sales materialized view

Note: If the compatibility is set to 10.0.1 or higher, **COMMENT** **ON** **TABLE** will not be allowed for the materialized view container table. The following error message will be thrown if it is issued.

ORA-12098: cannot comment on the materialized view.

In the case of a prebuilt table, if it has an existing comment, the comment will be inherited by the materialized view after it has been created. The existing comment will be prefixed with **'(from table)'**. For example, table **sales\_summary** was created to contain sales summary information. An existing comment **'Sales summary data'** was associated with the table. A materialized view of the same name is created to use the prebuilt table as its container table. After the materialized view creation, the comment becomes**'(from table) Sales summary data'**.

However, if the prebuilt table, **sales\_summary**, does not have any comment, the following comment is added: **'Sales summary data'**. Then, if we drop the materialized view, the comment will be passed to the prebuilt table with the comment:**'(from materialized view) Sales summary data'**.

Registering Existing Materialized Views

Some data warehouses have implemented materialized views in ordinary user tables. Although this solution provides the performance benefits of materialized views, it does not:

* Provide query rewrite to all SQL applications.
* Enable materialized views defined in one application to be transparently accessed in another application.
* Generally support fast parallel or fast materialized view refresh.

Because of these limitations, and because existing materialized views can be extremely large and expensive to rebuild, you should register your existing materialized view tables whenever possible. You can register a user-defined materialized view with the **CREATE** **MATERIALIZEDVIEW** ... **ON** **PREBUILT** **TABLE** statement. Once registered, the materialized view can be used for query rewrites or maintained by one of the refresh methods, or both.

The contents of the table must reflect the materialization of the defining query at the time you register it as a materialized view, and each column in the defining query must correspond to a column in the table that has a matching datatype. However, you can specify **WITHREDUCED** **PRECISION** to allow the precision of columns in the defining query to be different from that of the table columns.

The table and the materialized view must have the same name, but the table retains its identity as a table and can contain columns that are not referenced in the defining query of the materialized view. These extra columns are known as unmanaged columns. If rows are inserted during a refresh operation, each unmanaged column of the row is set to its default value. Therefore, the unmanaged columns cannot have**NOT** **NULL** constraints unless they also have default values.

Materialized views based on prebuilt tables are eligible for selection by query rewrite provided the parameter **QUERY\_REWRITE\_INTEGRITY** is set to **STALE\_TOLERATED** or **TRUSTED**. See [Chapter 17, "Basic Query Rewrite"](http://docs.oracle.com/cd/B19306_01/server.102/b14223/qrbasic.htm#g1045551) for details about integrity levels.

When you drop a materialized view that was created on a prebuilt table, the table still exists—only the materialized view is dropped.

The following example illustrates the two steps required to register a user-defined table. First, the table is created, then the materialized view is defined using exactly the same name as the table. This materialized view **sum\_sales\_tab** is eligible for use in query rewrite.

CREATE TABLE sum\_sales\_tab

PCTFREE 0 TABLESPACE demo

STORAGE (INITIAL 16k NEXT 16k PCTINCREASE 0) AS

SELECT s.prod\_id, SUM(amount\_sold) AS dollar\_sales,

SUM(quantity\_sold) AS unit\_sales

FROM sales s GROUP BY s.prod\_id;

CREATE MATERIALIZED VIEW sum\_sales\_tab

ON PREBUILT TABLE WITHOUT REDUCED PRECISION

ENABLE QUERY REWRITE AS

SELECT s.prod\_id, SUM(amount\_sold) AS dollar\_sales,

SUM(quantity\_sold) AS unit\_sales

FROM sales s GROUP BY s.prod\_id;

You could have compressed this table to save space. See ["Storage And Table Compression"](http://docs.oracle.com/cd/B19306_01/server.102/b14223/basicmv.htm#i1006854) for details regarding table compression.

In some cases, user-defined materialized views are refreshed on a schedule that is longer than the update cycle. For example, a monthly materialized view might be updated only at the end of each month, and the materialized view values always refer to complete time periods. Reports written directly against these materialized views implicitly select only data that is not in the current (incomplete) time period. If a user-defined materialized view already contains a time dimension:

* It should be registered and then fast refreshed each update cycle.
* You can create a view that selects the complete time period of interest.
* The reports should be modified to refer to the view instead of referring directly to the user-defined materialized view.

If the user-defined materialized view does not contain a time dimension, then:

* Create a new materialized view that does include the time dimension (if possible).
* The view should aggregate over the time column in the new materialized view.

Choosing Indexes for Materialized Views

The two most common operations on a materialized view are query execution and fast refresh, and each operation has different performance requirements. Query execution might need to access any subset of the materialized view key columns, and might need to join and aggregate over a subset of those columns. Consequently, query execution usually performs best if a single-column bitmap index is defined on each materialized view key column.

In the case of materialized views containing only joins using fast refresh, Oracle recommends that indexes be created on the columns that contain the rowids to improve the performance of the refresh operation.

If a materialized view using aggregates is fast refreshable, then an index appropriate for the fast refresh procedure is created unless **USINGNO** **INDEX** is specified in the **CREATE** **MATERIALIZED** **VIEW** statement.

If the materialized view is partitioned, then, after doing a partition maintenance operation on the materialized view, the indexes become unusable, and they need to be rebuilt for fast refresh to work.

See [*Oracle Database Performance Tuning Guide*](http://docs.oracle.com/cd/B19306_01/server.102/b14211/toc.htm) for information on using the SQL Access Advisor to determine what indexes are appropriate for your materialized view.

Dropping Materialized Views

Use the **DROP** **MATERIALIZED** **VIEW** statement to drop a materialized view. For example. the following statement:

DROP MATERIALIZED VIEW sales\_sum\_mv;

This statement drops the materialized view **sales\_sum\_mv**. If the materialized view was prebuilt on a table, then the table is not dropped, but it can no longer be maintained with the refresh mechanism or used by query rewrite. Alternatively, you can drop a materialized view using Oracle Enterprise Manager.

Analyzing Materialized View Capabilities

You can use the **DBMS\_MVIEW**.**EXPLAIN\_MVIEW** procedure to learn what is possible with a materialized view or potential materialized view. In particular, this procedure enables you to determine:

* If a materialized view is fast refreshable
* What types of query rewrite you can perform with this materialized view
* Whether PCT refresh is possible

Using this procedure is straightforward. You simply call **DBMS\_MVIEW.EXPLAIN\_MVIEW**, passing in as a single parameter the schema and materialized view name for an existing materialized view. Alternatively, you can specify the **SELECT** string for a potential materialized view or the complete **CREATE** **MATERIALIZED** **VIEW** statement. The materialized view or potential materialized view is then analyzed and the results are written into either a table called **MV\_CAPABILITIES\_TABLE**, which is the default, or to an array called **MSG\_ARRAY**.

Note that you must run the **utlxmv.sql** script prior to calling **EXPLAIN\_MVIEW** except when you are placing the results in **MSG\_ARRAY**. The script is found in the **admin** directory. It is to create the **MV\_CAPABILITIES\_TABLE** in the current schema. An explanation of the various capabilities is in [Table 8-7](http://docs.oracle.com/cd/B19306_01/server.102/b14223/basicmv.htm#g1014360), and all the possible messages are listed in [Table 8-8](http://docs.oracle.com/cd/B19306_01/server.102/b14223/basicmv.htm#g1014409).

Using the DBMS\_MVIEW.EXPLAIN\_MVIEW Procedure

The **EXPLAIN\_MVIEW** procedure has the following parameters:

* **stmt\_id**

An optional parameter. A client-supplied unique identifier to associate output rows with specific invocations of **EXPLAIN\_MVIEW**.

* **mv**

The name of an existing materialized view or the query definition or the entire **CREATE** **MATERIALIZED** **VIEW** statement of a potential materialized view you want to analyze.

* **msg-array**

The PL/SQL **VARRAY** that receives the output.

**EXPLAIN\_MVIEW** analyzes the specified materialized view in terms of its refresh and rewrite capabilities and inserts its results (in the form of multiple rows) into **MV\_CAPABILITIES\_TABLE** or **MSG\_ARRAY**.

**See Also:**

[*Oracle Database PL/SQL Packages and Types Reference*](http://docs.oracle.com/cd/B19306_01/appdev.102/b14258/d_mview.htm#ARPLS027) for further information about the **DBMS\_MVIEW** package

DBMS\_MVIEW.EXPLAIN\_MVIEW Declarations

The following PL/SQL declarations that are made for you in the **DBMS\_MVIEW** package show the order and datatypes of these parameters for explaining an existing materialized view and a potential materialized view with output to a table and to a **VARRAY**.

Explain an existing or potential materialized view with output to **MV\_CAPABILITIES\_TABLE**:

DBMS\_MVIEW.EXPLAIN\_MVIEW (mv IN VARCHAR2,

stmt\_id IN VARCHAR2:= NULL);

Explain an existing or potential materialized view with output to a **VARRAY**:

DBMS\_MVIEW.EXPLAIN\_MVIEW (mv IN VARCHAR2,

msg\_array OUT SYS.ExplainMVArrayType);

Using MV\_CAPABILITIES\_TABLE

One of the simplest ways to use **DBMS\_MVIEW.EXPLAIN\_MVIEW** is with the **MV\_CAPABILITIES\_TABLE**, which has the following structure:

CREATE TABLE MV\_CAPABILITIES\_TABLE

(STMT\_ID VARCHAR(30), -- client-supplied unique statement identifier

MV VARCHAR(30), -- NULL for SELECT based EXPLAIN\_MVIEW

CAPABILITY\_NAME VARCHAR(30), -- A descriptive name of particular

-- capabilities, such as REWRITE.

-- See [Table 8-7](http://docs.oracle.com/cd/B19306_01/server.102/b14223/basicmv.htm#g1014360)

POSSIBLE CHARACTER(1), -- Y = capability is possible

-- N = capability is not possible

RELATED\_TEXT VARCHAR(2000), -- owner.table.column, and so on related to

-- this message

RELATED\_NUM NUMBER, -- When there is a numeric value

-- associated with a row, it goes here.

MSGNO INTEGER, -- When available, message # explaining

-- why disabled or more details when

-- enabled.

MSGTXT VARCHAR(2000), -- Text associated with MSGNO

SEQ NUMBER); -- Useful in ORDER BY clause when

-- selecting from this table.

You can use the **utlxmv.sql** script found in the **admin** directory to create **MV\_CAPABILITIES\_TABLE**.

***Example 8-8 DBMS\_MVIEW.EXPLAIN\_MVIEW***

First, create the materialized view. Alternatively, you can use **EXPLAIN\_MVIEW** on a potential materialized view using its **SELECT** statement or the complete **CREATE** **MATERIALIZED** **VIEW** statement.

CREATE MATERIALIZED VIEW cal\_month\_sales\_mv

BUILD IMMEDIATE

REFRESH FORCE

ENABLE QUERY REWRITE AS

SELECT t.calendar\_month\_desc, SUM(s.amount\_sold) AS dollars

FROM sales s, times t WHERE s.time\_id = t.time\_id

GROUP BY t.calendar\_month\_desc;

Then, you invoke **EXPLAIN\_MVIEW** with the materialized view to explain. You need to use the **SEQ** column in an **ORDER** **BY** clause so the rows will display in a logical order. If a capability is not possible, **N** will appear in the **P** column and an explanation in the **MSGTXT** column. If a capability is not possible for more than one reason, a row is displayed for each reason.

EXECUTE DBMS\_MVIEW.EXPLAIN\_MVIEW ('SH.CAL\_MONTH\_SALES\_MV');

SELECT capability\_name, possible, SUBSTR(related\_text,1,8)

AS rel\_text, SUBSTR(msgtxt,1,60) AS msgtxt

FROM MV\_CAPABILITIES\_TABLE

ORDER BY seq;

CAPABILITY\_NAME P REL\_TEXT MSGTXT

--------------- - -------- ------

PCT N

REFRESH\_COMPLETE Y

REFRESH\_FAST N

REWRITE Y

PCT\_TABLE N SALES no partition key or PMARKER in select list

PCT\_TABLE N TIMES relation is not a partitioned table

REFRESH\_FAST\_AFTER\_INSERT N SH.TIMES mv log must have new values

REFRESH\_FAST\_AFTER\_INSERT N SH.TIMES mv log must have ROWID

REFRESH\_FAST\_AFTER\_INSERT N SH.TIMES mv log does not have all necessary columns

REFRESH\_FAST\_AFTER\_INSERT N SH.SALES mv log must have new values

REFRESH\_FAST\_AFTER\_INSERT N SH.SALES mv log must have ROWID

REFRESH\_FAST\_AFTER\_INSERT N SH.SALES mv log does not have all necessary columns

REFRESH\_FAST\_AFTER\_ONETAB\_DML N DOLLARS SUM(expr) without COUNT(expr)

REFRESH\_FAST\_AFTER\_ONETAB\_DML N see the reason why

REFRESH\_FAST\_AFTER\_INSERT is disabled

REFRESH\_FAST\_AFTER\_ONETAB\_DML N COUNT(\*) is not present in the select list

REFRESH\_FAST\_AFTER\_ONETAB\_DML N SUM(expr) without COUNT(expr)

REFRESH\_FAST\_AFTER\_ANY\_DML N see the reason why

REFRESH\_FAST\_AFTER\_ONETAB\_DML is disabled

REFRESH\_FAST\_AFTER\_ANY\_DML N SH.TIMES mv log must have sequence

REFRESH\_FAST\_AFTER\_ANY\_DML N SH.SALES mv log must have sequence

REFRESH\_PCT N PCT is not possible on any of the detail

tables in the materialized view

REWRITE\_FULL\_TEXT\_MATCH Y

REWRITE\_PARTIAL\_TEXT\_MATCH Y

REWRITE\_GENERAL Y

REWRITE\_PCT N PCT is not possible on any detail tables

**See Also:**

[Chapter 15, "Maintaining the Data Warehouse"](http://docs.oracle.com/cd/B19306_01/server.102/b14223/refresh.htm#g1018554) and [Chapter 18, "Advanced Query Rewrite"](http://docs.oracle.com/cd/B19306_01/server.102/b14223/qradv.htm#g1045552) for further details about PCT

MV\_CAPABILITIES\_TABLE.CAPABILITY\_NAME Details

[Table 8-7](http://docs.oracle.com/cd/B19306_01/server.102/b14223/basicmv.htm#g1014360) lists explanations for values in the **CAPABILITY\_NAME** column.

***Table 8-7 CAPABILITY\_NAME Column Details***

| **CAPABILITY\_NAME** | **Description** |
| --- | --- |
| **PCT** | If this capability is possible, Partition Change Tracking (PCT) is possible on at least one detail relation. If this capability is not possible, PCT is not possible with any detail relation referenced by the materialized view. |
| **REFRESH\_COMPLETE** | If this capability is possible, complete refresh of the materialized view is possible. |
| **REFRESH\_FAST** | If this capability is possible, fast refresh is possible at least under certain circumstances. |
| **REWRITE** | If this capability is possible, at least full text match query rewrite is possible. If this capability is not possible, no form of query rewrite is possible. |
| **PCT\_TABLE** | If this capability is possible, it is possible with respect to a particular partitioned table in the top level **FROM** list. When possible, PCT applies to the partitioned table named in the**RELATED\_TEXT** column.  PCT is needed to support fast fresh after partition maintenance operations on the table named in the **RELATED\_TEXT** column.  PCT may also support fast refresh with regard to updates to the table named in the**RELATED\_TEXT** column when fast refresh from a materialized view log is not possible.  PCT is also needed to support query rewrite in the presence of partial staleness of the materialized view with regard to the table named in the **RELATED\_TEXT** column.  When disabled, PCT does not apply to the table named in the **RELATED\_TEXT** column. In this case, fast refresh is not possible after partition maintenance operations on the table named in the **RELATED\_TEXT** column. In addition, PCT-based refresh of updates to the table named in the**RELATED\_TEXT** column is not possible. Finally, query rewrite cannot be supported in the presence of partial staleness of the materialized view with regard to the table named in the**RELATED\_TEXT** column. |
| **PCT\_TABLE\_REWRITE** | If this capability is possible, it is possible with respect to a particular partitioned table in the top level **FROM** list. When possible, PCT applies to the partitioned table named in the**RELATED\_TEXT** column.  This capability is needed to support query rewrite against this materialized view in partial stale state with regard to the table named in the **RELATED\_TEXT** column.  When disabled, query rewrite cannot be supported if this materialized view is in partial stale state with regard to the table named in the **RELATED\_TEXT** column. |
| **REFRESH\_FAST\_AFTER\_INSERT** | If this capability is possible, fast refresh from a materialized view log is possible at least in the case where the updates are restricted to **INSERT** operations; complete refresh is also possible. If this capability is not possible, no form of fast refresh from a materialized view log is possible. |
| **REFRESH\_FAST\_AFTER\_ONETAB\_DML** | If this capability is possible, fast refresh from a materialized view log is possible regardless of the type of update operation, provided all update operations are performed on a single table. If this capability is not possible, fast refresh from a materialized view log may not be possible when the update operations are performed on multiple tables. |
| **REFRESH\_FAST\_AFTER\_ANY\_DML** | If this capability is possible, fast refresh from a materialized view log is possible regardless of the type of update operation or the number of tables updated. If this capability is not possible, fast refresh from a materialized view log may not be possible when the update operations (other than**INSERT**) affect multiple tables. |
| **REFRESH\_FAST\_PCT** | If this capability is possible, fast refresh using PCT is possible. Generally, this means that refresh is possible after partition maintenance operations on those detail tables where PCT is indicated as possible. |
| **REWRITE\_FULL\_TEXT\_MATCH** | If this capability is possible, full text match query rewrite is possible. If this capability is not possible, full text match query rewrite is not possible. |
| **REWRITE\_PARTIAL\_TEXT\_MATCH** | If this capability is possible, at least full and partial text match query rewrite are possible. If this capability is not possible, at least partial text match query rewrite and general query rewrite are not possible. |
| **REWRITE\_GENERAL** | If this capability is possible, all query rewrite capabilities are possible, including general query rewrite and full and partial text match query rewrite. If this capability is not possible, at least general query rewrite is not possible. |
| **REWRITE\_PCT** | If this capability is possible, query rewrite can use a partially stale materialized view even in**QUERY\_REWRITE\_INTEGRITY** = **ENFORCED** or **TRUSTED** modes. When this capability is not possible, query rewrite can use a partially stale materialized view only in**QUERY\_REWRITE\_INTEGRITY** = **STALE\_TOLERATED** mode. |

MV\_CAPABILITIES\_TABLE Column Details

[Table 8-8](http://docs.oracle.com/cd/B19306_01/server.102/b14223/basicmv.htm#g1014409) lists the semantics for **RELATED\_TEXT** and **RELATED\_NUM** columns.

***Table 8-8 MV\_CAPABILITIES\_TABLE Column Details***

| **MSGNO** | **MSGTXT** | **RELATED\_NUM** | **RELATED\_TEXT** |
| --- | --- | --- | --- |
| **NULL** | **NULL** |  | For PCT capability only: **[*owner*.]*name*** of the table upon which PCT is enabled |
| 2066 | This statement resulted in an Oracle error | Oracle error number that occurred |  |
| 2067 | No partition key or **PMARKER** or join dependent expression in **SELECT**list |  | **[*owner*.]*name*** of relation for which PCT is not supported |
| 2068 | Relation is not partitioned |  | **[*owner*.]*name*** of relation for which PCT is not supported |
| 2069 | PCT not supported with multicolumn partition key |  | **[*owner*.]*name*** of relation for which PCT is not supported |
| 2070 | PCT not supported with this type of partitioning |  | **[*owner*.]*name*** of relation for which PCT is not supported |
| 2071 | Internal error: undefined PCT failure code | The unrecognized numeric PCT failure code | **[*owner*.]*name*** of relation for which PCT is not supported |
| 2072 | Requirements not satisfied for fast refresh of nested materialized view |  |  |
| 2077 | Materialized view log is newer than last full refresh |  | **[*owner*.]*table\_name*** of table upon which the matrialized view log is needed |
| 2078 | Materialized view log must have new values |  | **[*owner*.]*table\_name*** of table upon which the materialized view log is needed |
| 2079 | Materialized view log must have**ROWID** |  | **[*owner*.]*table\_name*** of table upon which the materialized view log is needed |
| 2080 | Materialized view log must have primary key |  | **[*owner*.]*table\_name*** of table upon which the materialized view log is needed |
| 2081 | Materialized view log does not have all necessary columns |  | **[*owner*.]*table\_name*** of table upon which the materialized view log is needed |
| 2082 | Problem with materialized view log |  | **[*owner*.]*table\_name*** of table upon which the materialized view log is needed |
| 2099 | Materialized view references a remote table or view in the **FROM**list | Offset from the **SELECT** keyword to the table or view in question | **[*owner*.]*name*** of the table or view in question |
| 2126 | Multiple master sites |  | Name of the first different node, or **NULL** if the first different node is local |
| 2129 | Join or filter condition(s) are complex |  | **[*owner*.]*name*** of the table involved with the join or filter condition (or **NULL** when not available) |
| 2130 | Expression not supported for fast refresh | Offset from the **SELECT** keyword to the expression in question | The alias name in the **SELECT** list of the expression in question |
| 2150 | **SELECT** lists must be identical across the **UNION** operator | Offset from the **SELECT** keyword to the first different select item in the**SELECT** list | The alias name of the first different select item in the **SELECT** list |
| 2182 | PCT is enabled through a join dependency |  | **[*owner*.]*name*** of relation for which**PCT\_TABLE\_REWRITE** is not enabled |
| 2183 | Expression to enable PCT not in**PARTITION** **BY** of analytic function or model | The unrecognized numeric PCT failure code | **[*owner*.]*name*** of relation for which PCT is not enabled |
| 2184 | Expression to enable PCT cannot be rolled up |  | **[*owner*.]*name*** of relation for which PCT is not enabled |
| 2185 | No partition key or **PMARKER** in the**SELECT** list |  | **[*owner*.]*name*** of relation for which**PCT\_TABLE\_REWRITE** is not enabled |
| 2186 | **GROUP** **OUTER** **JOIN** is present |  |  |
| 2187 | Materialized view on external table |  |  |

* [Choose your language](http://docs.oracle.com/cd/B19306_01/server.102/b14223/basicmv.htm)
* [About Oracle](http://www.oracle.com/corporate/index.html)
* [Contact Us](http://www.oracle.com/us/corporate/contact/index.html)
* [Legal Notices](http://www.oracle.com/us/legal/index.html)
* [Terms of Use](http://www.oracle.com/us/legal/terms/index.html)
* [Your Privacy Rights](http://www.oracle.com/us/legal/privacy/index.html)

Copyright © 2015, Oracle and/or its affiliates. All rights reserved.