|  |
| --- |
|  |
|  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| REVISION HISTORY | | | | | |
| Ver. | Description of Change | Author | Date | Approved | |
| Name | Effective Date |
| 1.0 | Initial status | [Elias Nema](mailto:Elias_Nema@epam.com) | 20-JAN-2014 |  |  |
|  |  |  |  |  |  |

Contents

[1. Overview of Data Warehousing with Materialized Views 3](#_Toc383666514)

[1.1. Materialized Views for Data Warehouses 3](#_Toc383666515)

[1.2. Materialized Views for Distributed Computing 3](#_Toc383666516)

[1.3. Materialized Views for Mobile Computing 3](#_Toc383666517)

[1.4. The Need for Materialized Views 3](#_Toc383666518)

[1.5. Components of Summary Management 4](#_Toc383666519)

[2. Types of Materialized Views 5](#_Toc383666520)

[2.1. Materialized Views with Aggregates 6](#_Toc383666521)

[2.2. Materialized Views Containing Only Joins 6](#_Toc383666522)

[2.3. Nested Materialized Views 7](#_Toc383666523)

[2.4. Build Methods 7](#_Toc383666524)

[2.5. Refresh Options 7](#_Toc383666525)

[2.6. ORDER BY Clause 8](#_Toc383666526)

[2.7. Materialized View Logs 8](#_Toc383666527)

[3. Partitioning and Materialized Views 8](#_Toc383666528)

[3.1. Partition Change Tracking 9](#_Toc383666529)

[3.2. Benefits of Partitioning a Materialized View 9](#_Toc383666530)

[4. Source Books and Articles 9](#_Toc383666531)

# 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 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. 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 affects 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 and Oracle Database Heterogeneous Connectivity Administrator's Guide for details on distributed and mobile computing.

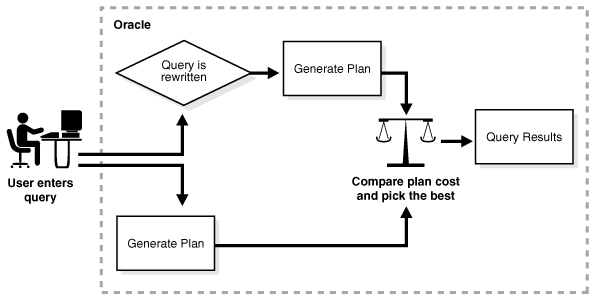
## Materialized Views for Mobile Computing

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.

## 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 1 illustrates how query rewrite works.



**Figure 1 Transparent Query Rewrite**

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.

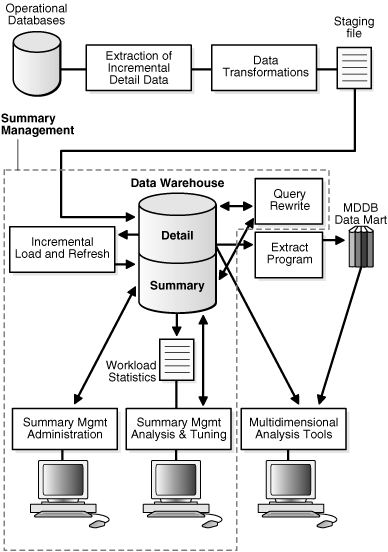
## Components of Summary Management

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, partitions, 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 2 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 2 Overview of Summary Management**

# Types of Materialized Views

A materialized view can be created with the CREATE MATERIALIZED VIEW statement or using Enterprise Manager.

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. 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
* Materialized Views Containing Only Joins
* Nested Materialized Views

## Materialized Views with Aggregates

In data warehouses, materialized views normally contain aggregates. 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".

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 is 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 is fast refreshed.

CREATE MATERIALIZED VIEW LOG ON sales WITH ROWID(prod\_id, cust\_id, time\_id),

COMMIT SCN INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW product\_sales\_mv

PCTFREE 0 TABLESPACE demo

STORAGE (INITIAL 8M)

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;

## Materialized Views Containing Only Joins

Some materialized views contain only joins and no aggregates, 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 are 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.

To achieve an optimally efficient refresh, you should ensure that the defining query does not use an outer join that behaves like an inner join. If the defining query contains such a join, consider rewriting the defining query to contain an inner join. See "Restrictions on Fast Refresh on Materialized Views with Joins Only" for more information regarding the conditions that cause refresh performance to degrade.

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(+);

## 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.

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.

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;

## Build Methods

Two build methods are available for creating the materialized view. 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 16, "Maintaining the Data Warehouse".

* *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.

## 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 options may not be available.

* *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.

## 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. You can typically achieve better fast refresh performance of local materialized views containing aggregates or joins by using a WITH COMMIT SCN clause.

# Partitioning and Materialized Views

Because of the large volume of data held in a data warehouse, partitioning is an extremely useful option when designing a database. Partitioning the fact tables improves scalability, simplifies system administration, and makes it possible to define local indexes that can be efficiently rebuilt. Partitioning the fact tables also improves the opportunity of fast refreshing the materialized view because this may enable Partition Change Tracking (PCT) refresh on the materialized view. Partitioning a materialized view also has benefits for refresh, because the refresh procedure can then use parallel DML in more scenarios and PCT-based refresh can use truncate partition to efficiently maintain the materialized view. See Oracle Database VLDB and Partitioning Guide for further details about partitioning.

## Partition Change Tracking

It is possible and advantageous to track freshness to a finer grain than the entire materialized view. The ability to identify which rows in a materialized view are affected by a certain detail table partition, is known as Partition Change Tracking. When one or more of the detail tables are partitioned, it may be possible to identify the specific rows in the materialized view that correspond to a modified detail partition(s); those rows become stale when a partition is modified while all other rows remain fresh.

You can use PCT to identify which materialized view rows correspond to a particular partition. PCT is also used to support fast refresh after partition maintenance operations on detail tables. For instance, if a detail table partition is truncated or dropped, the affected rows in the materialized view are identified and deleted.

## Benefits of Partitioning a Materialized View

When a materialized view is partitioned on the partitioning key column or join dependent expressions of the detail table, it is more efficient to use a TRUNCATE PARTITION statement to remove one or more partitions of the materialized view during refresh and then repopulate the partition with new data. Oracle Database uses this variant of fast refresh (called PCT refresh) with partition truncation if the following conditions are satisfied in addition to other conditions described in "Partition Change Tracking".

* The materialized view is partitioned on the partitioning key column or join dependent expressions of the detail table.
* If PCT is enabled using either the partitioning key column or join expressions, the materialized view should be range or list partitioned.
* PCT refresh is nonatomic.

# Source Books and Articles

1. Kyte T. Expert Oracle Database Architecture: Oracle Database 9i, 10g, and 11g Programming Techniques and Solutions, Second Edition. Apress, 2010.
2. Morton K., & Osborne K., & Sands R., & Shamsudeen R., & Still J. Pro Oracle SQL. Apress, 2013.
3. Lane, P. Oracle Database Data Warehousing Guide, 11g Release 2 (11.2). Redwood City: Oracle, 2013.