Oracle9*i* Advanced Replication
Release 2 (9.2)
Part Number A96567-01

3 Materialized View Concepts and Architecture

This chapter explains the concepts and architecture of Oracle materialized views. This chapter contains these topics:

- Materialized View Concepts
- Materialized View Architecture

Materialized View Concepts

Oracle uses **materialized views** (also known as snapshots in prior releases) to replicate data to non-master sites in a replication environment and to cache expensive queries in a data warehouse environment. This chapter, and this *Oracle9i Replication* manual in general, discusses materialized views for use in a replication environment.

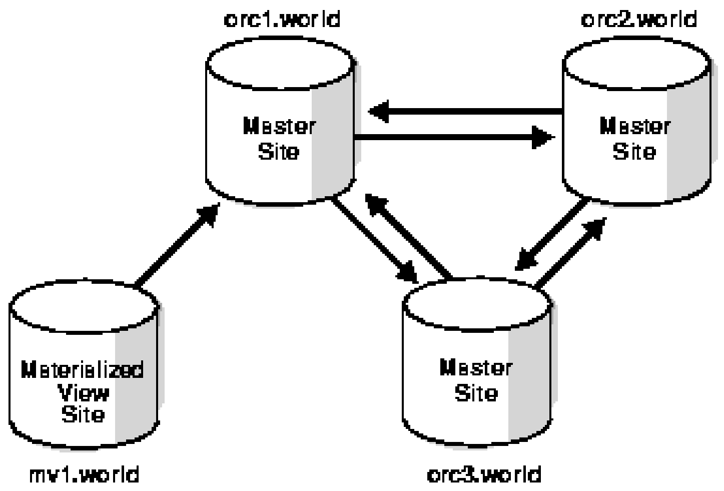
See Also:

Oracle9i Data Warehousing Guide to learn more about materialized views for data warehousing

What is a Materialized View?

A materialized view is a replica of a target master from a single point in time. The master can be either a master table at a master site or a master materialized view at a materialized view site. Whereas in multimaster replication tables are continuously updated by other master sites, materialized views are updated from one or more masters through individual batch updates, known as a **refreshes**, from a single master site or master materialized view site, as illustrated in Figure 3-1. The arrows in Figure 3-1 represent database links.

Figure 3-1 Materialized View Connected to a Single Master Site



Text description of the illustration rep81072.gif

When a materialized view is fast refreshed, Oracle must examine all of the changes to the master table or master materialized view since the last refresh to see if any apply to the materialized view. Therefore, if any changes where made to the master since the last refresh, then a materialized view refresh takes some time to apply the changes to the materialized view. If, however, no changes at all were made to the master since the last refresh of a materialized view, then the materialized view refresh should be very quick.

Why Use Materialized Views?

You can use materialized views to achieve one or more of the following goals:

- Ease Network Loads
- Create a Mass Deployment Environment
- Enable Data Subsetting
- Enable Disconnected Computing

Ease Network Loads

If one of your goals is to reduce network loads, then you can use materialized views to distribute your corporate database to regional sites. Instead of the entire company accessing a single database server, user load is distributed across multiple database servers. Through the use of multitier materialized views, you can create materialized views based on other materialized views, which enables you to distribute user load to an even greater extent because clients can access materialized view sites instead of master sites. To decrease the amount of data that is replicated, a materialized view can be a subset of a master table or master materialized view.

While multimaster replication also distributes a corporate database among multiple sites, the networking requirements for multimaster replication are greater than those for replicating with materialized views because of the transaction by transaction nature of multimaster replication. Further, the ability of multimaster replication to provide real-time or near real-time replication may result in greater network traffic, and might require a dedicated network link.

Materialized views are updated through an efficient batch process from a single master site or master materialized view site. They have lower network requirements and dependencies than multimaster replication because of the point in time nature of materialized view replication. Whereas multimaster replication requires constant communication over the network, materialized view replication requires only periodic refreshes.

In addition to not requiring a dedicated network connection, replicating data with materialized views increases data availability by providing local access to the target data. These benefits, combined with mass deployment and data subsetting (both of which also reduce network loads), greatly enhance the performance and reliability of your replicated database.

Create a Mass Deployment Environment

Deployment templates allow you to precreate a materialized view environment locally. You can then use deployment templates to quickly and easily deploy materialized view environments to support sales force automation and other mass deployment environments. Parameters allow you to create custom data sets for individual users without changing the deployment template. This technology enables you to roll out a database infrastructure to hundreds or thousands of users.

Enable Data Subsetting

Materialized views allow you to replicate data based on column- and row-level subsetting, while multimaster replication requires replication of the entire table. Data subsetting enables you to replicate information that pertains only to a particular site. For example, if you have a regional sales office, then you might replicate only the data that is needed in that region, thereby cutting down on unnecessary network traffic.

Enable Disconnected Computing

Materialized views do not require a dedicated network connection. Though you have the option of automating the refresh process by scheduling a job, you can manually refresh your materialized view on-demand, which is an ideal solution for sales applications running on a laptop. For example, a developer can integrate the replication management API for refresh on-demand into the sales application. When the salesperson has completed the day's orders, the salesperson simply dials up the network and uses the integrated mechanism to refresh the database, thus transferring the orders to the main office.

Read-Only, Updatable, and Writeable Materialized Views

A materialized view can be either read-only, updatable, or writeable. Users cannot perform data manipulation language (DML) statements on read-only materialized views, but they can perform DML on updatable and writeable materialized views.

Note:

For read-only, updatable, and writeable materialized views, the defining query of the materialized view must reference all of the primary key columns in the master.

See Also:

- "Materialized View Replication" for an introduction to read-only and updatable materialized views
- "Datatype Considerations" for information about datatype considerations for materialized views

Read-Only Materialized Views

You can make a materialized view read-only during creation by omitting the `FOR UPDATE` clause or disabling the equivalent option in the Replication Management tool. Read-only materialized views use many of the same mechanisms as updatable materialized views, except that they do not need to belong to a materialized view group.

In addition, using read-only materialized views eliminates the possibility of a materialized view introducing data conflicts at the master site or master materialized view site, although this convenience means that updates cannot be made at the remote materialized view site. The following is an example of a read-only materialized view:

```
CREATE MATERIALIZED VIEW hr.employees AS
  SELECT * FROM hr.employees@orc1.world;
```

Updatable Materialized Views

You can make a materialized view updatable during creation by including the `FOR UPDATE` clause or enabling the equivalent option in the Replication Management tool. For changes made to an updatable materialized view to be pushed back to the master during refresh, the updatable materialized view must belong to a materialized view group.

Updatable materialized views enable you to decrease the load on master sites because users can make changes to the data at the materialized view site. The following is an example of an updatable materialized view:

```
CREATE MATERIALIZED VIEW hr.departments FOR UPDATE AS
  SELECT * FROM hr.departments@orc1.world;
```

The following statement creates a materialized view group:

```
BEGIN
   DBMS_REPCAT.CREATE_MVIEW_REPGROUP (
      gname => 'hr_repg',
      master => 'orc1.world',
      propagation_mode => 'ASYNCHRONOUS');
END;
/
```

The following statement adds the `hr.departments` materialized view to the materialized view group, making the materialized view updatable:

```
BEGIN
   DBMS_REPCAT.CREATE_MVIEW_REPOBJECT (
      gname => 'hr_repg',
      sname => 'hr',
      oname => 'departments',
      type => 'SNAPSHOT',
      min_communication => TRUE);
END;

/
```

You can also use the Replication Management tool to create a materialized view group and add a materialized view to it.

Note:

- Do not use column aliases when you are creating an updatable materialized view. Column aliases cause an error when you attempt to add the materialized view to a materialized view group using the `CREATE_MVIEW_REPOBJECT` procedure.

- An updatable materialized view based on a master table or master materialized view that has defined column default values does not automatically use the master's default values.
- Updatable materialized views do not support the `DELETE CASCADE` constraint.

See Also:

- "Materialized View Groups" for more information
- *Oracle9i SQL Reference* for more information about column aliases

Writeable Materialized Views

A writeable materialized view is one that is created using the `FOR UPDATE` clause but is not part of a materialized view group. Users can perform DML operations on a writeable materialized view, but if you refresh the materialized view, then these changes are not pushed back to the master and the changes are lost in the materialized view itself. Writeable materialized views are typically allowed wherever fast-refreshable read-only materialized views are allowed.

Note:

Most of the documentation about materialized views only refers to read-only and updatable materialized views because writeable materialized views are rarely used.

Available Materialized Views

Oracle offers several types of materialized views to meet the needs of many different replication (and non-replication) situations. The following sections describe each type of materialized view and also describe some environments for which they are best suited.

The following sections contain examples of creating different types of materialized views:

- Primary Key Materialized Views
- Object Materialized Views
- ROWID Materialized Views
- Complex Materialized Views

Whenever you create a materialized view, regardless of its type, always specify the schema name of the table owner in the query for the materialized view. For example, consider the following `CREATE MATERIALIZED VIEW` statement:

```
CREATE MATERIALIZED VIEW hr.employees
  AS SELECT * FROM hr.employees@orc1.world;
```

Here, the schema `hr` is specified in the query.

Primary Key Materialized Views

Primary key materialized views are the default type of materialized view. They are updatable if the materialized view was created as part of a materialized view group and `FOR UPDATE` was specified when defining the materialized view. An updatable materialized view must belong to a materialized view group that has the same name as the replication group at its master site or master materialized view site. In addition, an updatable materialized view must reside in a different database than the master replication group.

Changes are propagated according to the row-level changes that have occurred, as identified by the primary key value of the row (not the `ROWID`). The following is an example of a SQL statement for creating an updatable, primary key materialized view:

```
CREATE MATERIALIZED VIEW oe.customers FOR UPDATE AS
  SELECT * FROM oe.customers@orc1.world;
```

Primary key materialized views may contain a subquery so that you can create a subset of rows at the remote materialized view site. A subquery is a query imbedded within the primary query, so that you have more than one `SELECT` statement in the `CREATE MATERIALIZED VIEW` statement. This subquery may be as simple as a basic `WHERE` clause or as complex as a multilevel `WHERE EXISTS` clause. Primary key materialized views that contain a selected class of subqueries can still be incrementally (or fast) refreshed, if each master referenced has a materialized view log. A fast refresh uses materialized view logs to update only the rows that have changed since the last refresh.

The following materialized view is created with a `WHERE` clause containing a subquery:

```
CREATE MATERIALIZED VIEW oe.orders REFRESH FAST AS
 SELECT * FROM oe.orders@orc1.world o
 WHERE EXISTS
   (SELECT * FROM oe.customers@orc1.world c
    WHERE o.customer_id = c.customer_id AND c.credit_limit > 10000);
```

This type of materialized view is called a subquery materialized view.

Note:

To create this `oe.orders` materialized view, `credit_limit` must be logged in the master's materialized view log. See "Logging Columns in the Materialized View Log" for more information.

See Also:

- "Materialized View Groups" for more information about materialized view groups
- "Materialized Views with Subqueries" for more information about materialized views with subqueries
- "Refresh Types" for more information about fast refresh
- "Materialized View Log" for more information about materialized view logs
- *Oracle9i SQL Reference* for more information about subqueries

Object Materialized Views

If a materialized view is based on an object table and is created using the `OF` *type* clause, then the materialized view is called an **object materialized view**. An object materialized view is structured in the same way as an object table. That is, an object materialized view is composed of row objects, and each row object is identified by an object identifier (OID) column.

See Also:

"Materialized Views Based on Object Tables"

ROWID Materialized Views

For backward compatibility, Oracle supports `ROWID` materialized views in addition to the default primary key materialized views. A `ROWID` materialized view is based on the physical row identifiers (rowids) of the rows in a master. `ROWID` materialized views should be used only for materialized views based on master tables from an Oracle7 database, and should not be used when creating new materialized views based on masters from Oracle8 or higher databases.

The following is an example of a `CREATE MATERIALIZED VIEW` statement that creates a `ROWID` materialized view:

```
CREATE MATERIALIZED VIEW oe.orders REFRESH WITH ROWID AS
 SELECT * FROM oe.orders@orc1.world;
```

See Also:

"Materialized View Log" for more information on the differences between a ROWID and Primary Key materialized view

Complex Materialized Views

To be fast refreshed, the defining query for a materialized view must observe certain restrictions. If you require a materialized view whose defining query is more general and cannot observe the restrictions, then the materialized view is complex and cannot be fast refreshed.

Specifically, a materialized view is considered complex when the defining query of the materialized view contains:

- A CONNECT BY clause

  For example, the following statement creates a complex materialized view:

  ```
  CREATE MATERIALIZED VIEW hr.emp_hierarchy AS
    SELECT LPAD(' ', 4*(LEVEL-1))||email USERNAME
      FROM hr.employees@orc1.world START WITH manager_id IS NULL
      CONNECT BY PRIOR employee_id = manager_id;
  ```

- An INTERSECT, MINUS, or UNION ALL set operation

  For example, the following statement creates a complex materialized view because it has a UNION ALL set operation:

  ```
  CREATE MATERIALIZED VIEW hr.mview_employees AS
    SELECT employees.employee_id, employees.email
    FROM hr.employees@orc1.world
  UNION ALL
    SELECT new_employees.employee_id, new_employees.email
    FROM hr.new_employees@orc1.world;
  ```

- In some cases, the DISTINCT or UNIQUE keyword, although it is possible to have the DISTINCT or UNIQUE keyword in the defining query and still have a simple materialized view

  For example, the following statement creates a complex materialized view:

  ```
  CREATE MATERIALIZED VIEW hr.employee_depts AS
    SELECT DISTINCT department_id FROM hr.employees@orc1.world
    ORDER BY department_id;
  ```

- An aggregate function

  For example, the following statement creates a complex materialized view:

  ```
  CREATE MATERIALIZED VIEW hr.average_sal AS
    SELECT AVG(salary) "Average" FROM hr.employees@orc1.world;
  ```

- Joins other than those in a subquery

  For example, the following statement creates a complex materialized view:

  ```
  CREATE MATERIALIZED VIEW hr.emp_join_dep AS
    SELECT last_name
    FROM hr.employees@orc1.world e, hr.departments@orc1.world d
    WHERE e.department_id = d.department_id;
  ```

- In some cases, a `UNION` operation. Specifically, a materialized view with a `UNION` operation is complex if any one of these conditions is true:
  - Any query within the `UNION` is complex. The previous bullet items specify when a query makes a materialized view complex.
  - The outermost `SELECT` list columns do not match for the queries in the `UNION`. In the following example, the first query only has `order_total` in the outermost `SELECT` list while the second query has `customer_id` in the outermost `SELECT` list. Therefore, the materialized view is complex.

    ```
    CREATE MATERIALIZED VIEW oe.orders AS
      SELECT order_total
      FROM oe.orders@orc1.world o
      WHERE EXISTS
        (SELECT cust_first_name, cust_last_name
           FROM oe.customers@orc1.world c
           WHERE o.customer_id = c.customer_id
           AND c.credit_limit > 50)
    UNION
      SELECT customer_id
      FROM oe.orders@orc1.world o
      WHERE EXISTS
        (SELECT cust_first_name, cust_last_name
           FROM oe.customers@orc1.world c
           WHERE o.customer_id = c.customer_id
           AND c.account_mgr_id = 30);
    ```

The innermost `SELECT` list has no bearing on whether a materialized view is complex. In the previous example, the innermost `SELECT` list is `cust_first_name` and `cust_last_name` for both queries in the `UNION`.

- Clauses that do not comply with the requirements detailed in "Restrictions for Materialized Views with Subqueries".

See Also:

- *Oracle9i Data Warehousing Guide* for more information about materialized views with aggregate functions and complex materialized views
- *Oracle9i SQL Reference* for more information about the `CONNECT BY` clause, set operations, the `DISTINCT` keyword, and aggregate functions

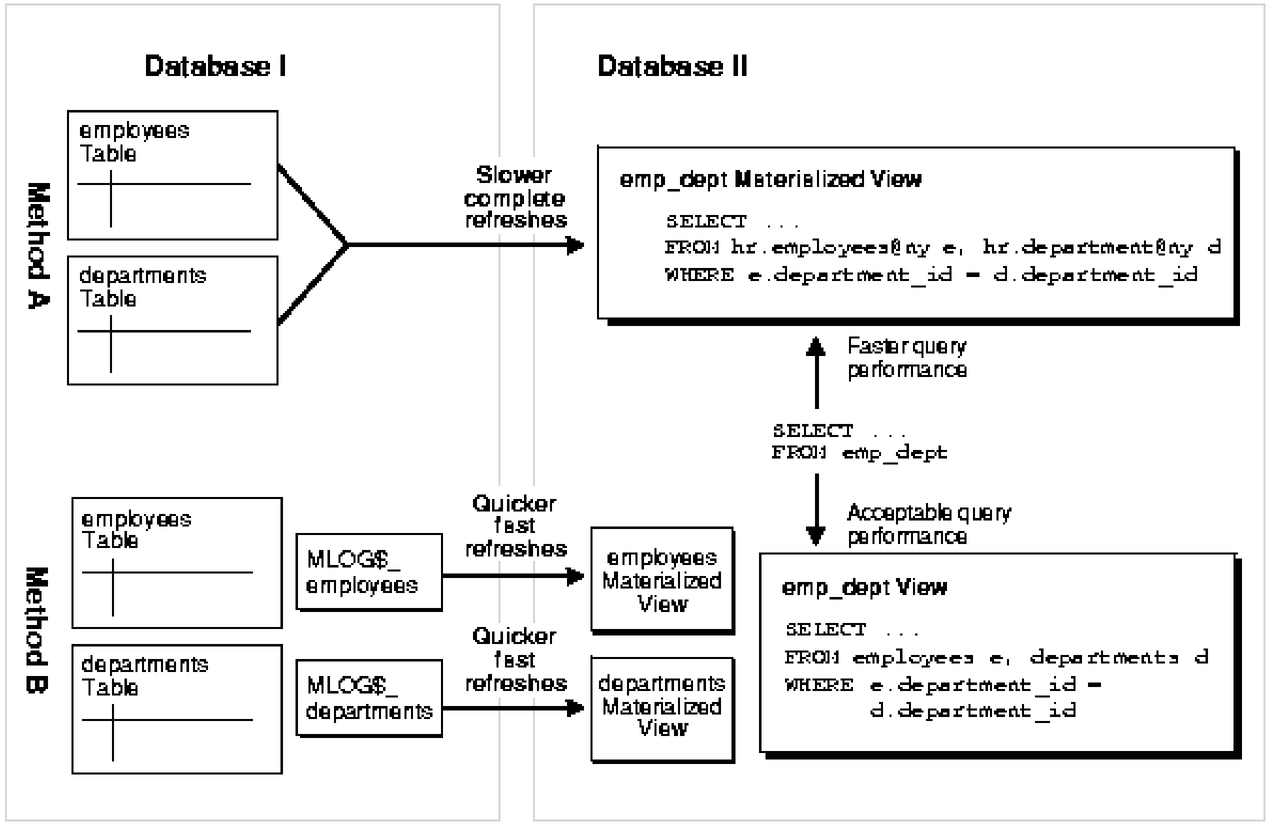
Note:

If possible, you should avoid using complex materialized views because they cannot be fast refreshed, which may degrade network performance (see "Refresh Process" for information).

A Comparison of Simple and Complex Materialized Views

For certain applications, you may want to consider using a complex materialized view. Figure 3-2 and the following text discuss some issues that you should consider.

Figure 3-2 Comparison of Simple and Complex Materialized Views


Database I
Database II
Method A
Method B
employees Table
departments Table
Slower complete refreshes
emp_dept Materialized View
SELECT ...
FROM hr.employees@ny e, hr.department@ny d
WHERE e.department_id = d.department_id
Faster query performance
SELECT ...
FROM emp_dept
Acceptable query performance
employees Table
departments Table
MLOG$_ employees
MLOG$_ departments
Quicker fast refreshes
Quicker fast refreshes
employees Materialized View
departments Materialized View
emp_dept View
SELECT ...
FROM employees e, departments d
WHERE e.department_id = d.department_id

Text description of the illustration rep81023.gif

- **Complex Materialized View**: Method A in Figure 3-2 shows a complex materialized view. The materialized view in Database II exhibits efficient query performance because the join operation was completed during the materialized view's refresh. However, complete refreshes must be performed because the materialized view is complex, and these refreshes will probably be slower than fast refreshes.
- **Simple Materialized Views with a Joined View**: Method B in Figure 3-2 shows two simple materialized views in Database II, as well as a view that performs the join in the materialized view's database. Query performance against the view would not be as good as the query performance against the complex materialized view in Method A. However, the simple materialized views can be refreshed more efficiently using fast refresh and materialized view logs.

In summary, to decide which method to use:

- If you refresh rarely and want faster query performance, then use Method A (complex materialized view).
- If you refresh regularly and can sacrifice query performance, then use Method B (simple materialized view).

Required Privileges for Materialized View Operations

Three distinct types of users perform operations on materialized views:

- **Creator:** the user who creates the materialized view
- **Refresher:** the user who refreshes the materialized view
- **Owner:** the user who owns the materialized view. The materialized view resides in this user's schema.

One user may perform all of these operations on a particular materialized view. However, in some replication environments, different users perform these operations on a particular materialized view. The privileges required to perform these operations depend on whether the same user performs them or different users perform them. The following sections explain the privileges requirements in detail.

Note:

The following sections do not cover the requirements necessary to create materialized views with query rewrite enabled. See the *Oracle9i SQL Reference* for information.

See Also:

The following sections discuss database links. See the *Oracle9i Database Administrator's Guide* for more information about using database links.

Creator Is Owner

If the creator of a materialized view also owns the materialized view, this user must have the following privileges to create a materialized view, granted either explicitly or through a role:

- `CREATE MATERIALIZED VIEW` or `CREATE ANY MATERIALIZED VIEW`
- `CREATE TABLE` or `CREATE ANY TABLE`
- `CREATE VIEW` or `CREATE ANY VIEW` if the compatibility level of the database is lower than 8.1.0
- `SELECT` object privilege on the master and the master's materialized view log or `SELECT ANY TABLE` system privilege. If the master site or

master materialized view site is remote, then the SELECT object privilege must be granted to the user at the master site or master materialized view site to which the user at the materialized view site connects through a database link.

If the owner of materialized view at the materialized view site has a private database link to the master site or master materialized view site, then the database link connects to the owner of the master at the master site or master materialized view site. Otherwise, the normal rules for connections through database links apply.

Creator Is Not Owner

If the creator of a materialized view is not the owner, certain privileges must be granted to the creator and to the owner to create a materialized view. The creator's privileges can be granted explicitly or through a role, but the owner's privileges must be granted explicitly. That is, the privileges granted to the owner cannot be granted through a role.

Table 3-1 shows the required privileges when the creator of the materialized view is not the owner.

Table 3-1 Required Privileges for Creating Materialized Views (Creator != Owner)

Creator	Owner
`CREATE ANY MATERIALIZED VIEW`	`CREATE TABLE` or `CREATE ANY TABLE` `CREATE VIEW` or `CREATE ANY VIEW` if the compatibility level of the database is lower than 8.1.0 `SELECT` object privilege on the master and the master's materialized view log or `SELECT ANY TABLE` system privilege. If the master site or master materialized view site is remote, then the `SELECT` object privilege must be granted to the user at the master site or master materialized view site to which the user at the materialized view site connects through a database link. If the owner of materialized view at the materialized view site has a private database link to the master site or master materialized view site, then the database link connects to the owner of the master at the master site or master materialized view site. Otherwise, the normal rules for connections through database links apply. **Note:** These privileges for the owner must be granted to the user explicitly, not through a role.

Refresher Is Owner

If the refresher of a materialized view also owns the materialized view, this user must have SELECT object privilege on the master and the master's materialized view log or SELECT ANY TABLE system privilege. If the master site or master materialized view site is remote, then the SELECT object privilege must be granted to the user at the master site or master materialized view site to which the user at the materialized view site connects through a database link**.** This privilege can be granted either explicitly or through a role.

If the owner of materialized view at the materialized view site has a private database link to the master site or master materialized view site, then the database link connects to the owner of the master at the master site or master materialized view site. Otherwise, the normal rules for connections through database links apply.

Refresher Is Not Owner

If the refresher of a materialized view is not the owner, certain privileges must be granted to the refresher and to the owner. These privileges can be granted either explicitly or through a role.

Table 3-2 shows the required privileges when the refresher of the materialized view is not the owner.

Table 3-2 Required Privileges for Refreshing Materialized Views (Refresher != Owner)

Refresher	Owner
`ALTER ANY MATERIALIZED VIEW`	If the master site or master materialized view site is local, then `SELECT` object privilege on the master and master's materialized view log or `SELECT ANY TABLE` system privilege. If the master site or master materialized view site is remote, then the SELECT object privilege must be granted to the user at the master site or master materialized view site to which the user at the materialized view site connects through a database link. If the owner of materialized view at the materialized view site has a private database link to the master site or master materialized view site, then the database link connects to the owner of the master at the master site or master materialized view site. Otherwise, the normal rules for connections through database links apply.

Data Subsetting with Materialized Views

In certain situations, you may want your materialized view to reflect a subset of the data in the master table or master materialized view. Row subsetting enables you to include only the rows that are needed from the master in the materialized views by using a `WHERE` clause. Column subsetting enables you to include only the columns that are needed from the master in the materialized views. You do this by specifying certain select columns in the `SELECT` statement during materialized view creation. If you use deployment templates to build your materialized views, then you can define column subsets on updatable materialized views.

Some reasons to use data subsetting are to:

- **Reduce Network Traffic**: In a column-subsetted materialized view, only changes that satisfy the `WHERE` clause of the materialized view's defining query are propagated to the materialized view site, thereby reducing the amount of data transferred and reducing network traffic.

- **Secure Sensitive Data**: Users can only view data that satisfies the defining query for the materialized view.
- **Reduce Resource Requirements**: If the materialized view is located on a laptop, then hard disks are generally significantly smaller than the hard disks on a corporate server. Subsetted materialized views may require significantly less storage space.
- **Improve Refresh Times**: Because less data is propagated to the materialized view site, the refresh process is faster, which is essential for those who need to refresh materialized views using a dial up network connection from a laptop.

For example, the following statement creates a materialized view based on the `oe.orders@orc1.world` master table and includes only the rows for the sales representative with a `sales_rep_id` number of `173`:

```
CREATE MATERIALIZED VIEW oe.orders REFRESH FAST AS
 SELECT * FROM oe.orders@orc1.world
 WHERE sales_rep_id = 173;
```

Rows of the orders table with a `sales_rep_id` number other than `173` are excluded from this materialized view.

Note:

The following sections discuss row subsetting through the use of subqueries. For more information about column subsetting, see "Column Subsetting with Deployment Templates".

Materialized Views with Subqueries

The previous example works well for individual materialized views that do not have any referential constraints to other materialized views. But, if you want to replicate data based on the information in more than one table, then maintaining and defining these materialized views may be difficult. The following sections provide examples of situations where a subquery is useful.

Many to One Subqueries

Consider a scenario where you have the `customers` table and `orders` table in the `oe` schema, and you want to create a materialized view of the `orders` table based on data in both the `orders` table and the `customers` table. For example, suppose a salesperson wants to see all of the orders for the customers with a credit limit greater than $10,000. In this case, the `CREATE MATERIALIZED VIEW` statement that creates the `orders` materialized view has a subquery with a many to one relationship, because there can be many orders for each customer.

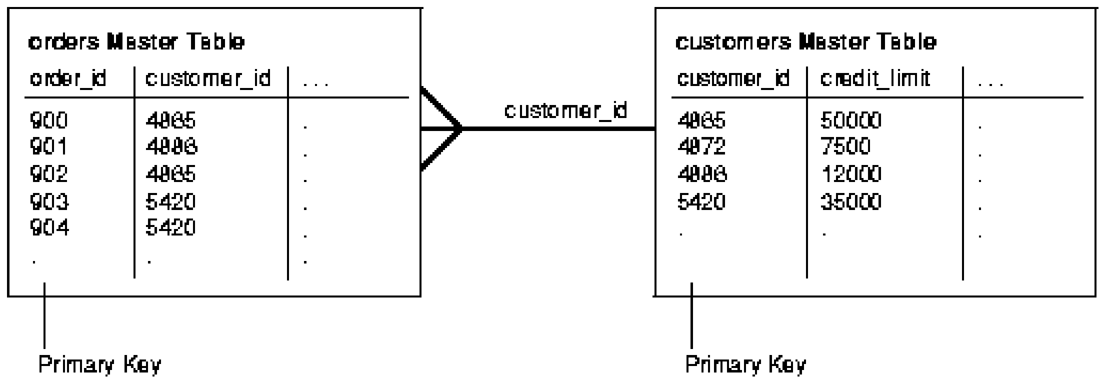
Look at the relationships in Figure 3-3, and notice that the `customers` and `orders` tables are related through the `customer_id` column. The following statement satisfies the original goal of the salesperson. That is, the following statement creates a materialized view that contains orders for customers whose credit limit is greater than $10,000:

```
CREATE MATERIALIZED VIEW oe.orders REFRESH FAST FOR UPDATE AS
  SELECT * FROM oe.orders@orc1.world o
  WHERE EXISTS
    (SELECT * FROM oe.customers@orc1.world c
     WHERE o.customer_id = c.customer_id AND c.credit_limit > 10000);
```

Note:

To create this `oe.orders` materialized view, `credit_limit` must be logged in the master's materialized view log. See "Logging Columns in the Materialized View Log" for more information.

Figure 3-3 Row Subsetting with Many to One Subqueries



Text description of the illustration rep81088.gif

As you can see, the materialized view created by this statement is fast refreshable and updatable. If new customers are identified that have a credit limit greater than $10,000, then the new data will be propagated to the materialized view site during the subsequent refresh process. Similarly, if a customer's credit limit drops to less than $10,000, then the customer's data will be removed from the materialized view during the subsequent refresh process.

One to Many Subqueries

Consider a scenario where you have the `customers` table and `orders` table in the `oe` schema, and you want to create a materialized view of the `customers` table based on data in both the `customers` table and the `orders` table. For example, suppose a salesperson wants to see all of the customers who have an order with an order total greater than $20,000, then the most efficient method is to create a materialized view with a one to many subquery in the defining query of a materialized view.

Here, the defining query in the `CREATE MATERIALIZED VIEW` statement on the `customers` table has a subquery with a one to many relationship. That is, one customer can have many orders.

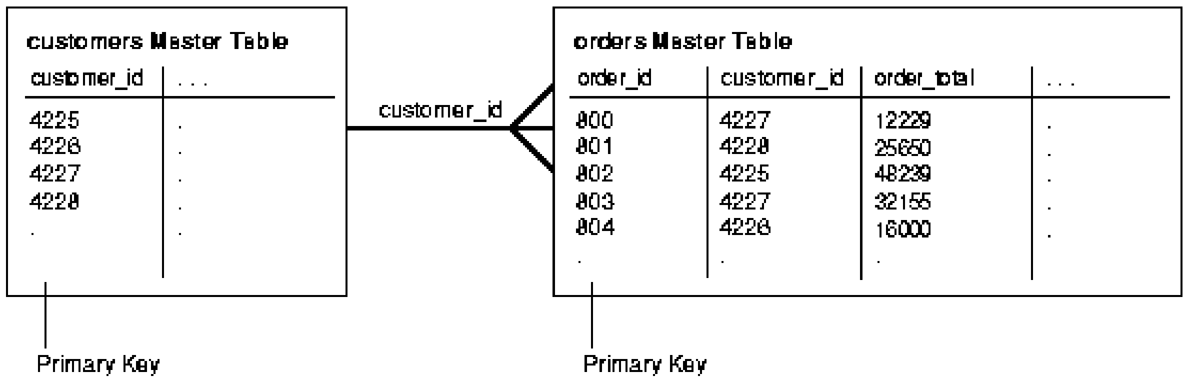
Look at the relationships in Figure 3-4, and notice that the `orders` table and `customers` table are related through the `customer_id` column. The following statement satisfies the original goal of the salesperson. That is, this statement creates a materialized view that contains customers who have an order with an order total greater than $20,000:

```
CREATE MATERIALIZED VIEW oe.customers REFRESH FAST FOR UPDATE AS
  SELECT * FROM oe.customers@orc1.world c
  WHERE EXISTS
    (SELECT * FROM oe.orders@orc1.world o
     WHERE c.customer_id = o.customer_id AND o.order_total > 20000);
```

Note:

To create this `oe.customers` materialized view, `customer_id` and `order_total` must be logged in the materialized view log for the `orders` table. See "Logging Columns in the Materialized View Log" for more information.

Figure 3-4 Row Subsetting with One to Many Subqueries



Text description of the illustration rep81087.gif

The materialized view created by this statement is fast refreshable and updatable. If new customers are identified that have an order total greater than $20,000, then the new data will be propagated to the materialized view site during the subsequent refresh process. Similarly, if a customer cancels an order with an order total greater than $20,000 and has no other order totals greater than $20,000, then the customer's data will be removed from the materialized view during the subsequent refresh process.

Note:

The materialized view site must have a compatibility level of 9.0.1 or higher because fast refresh of materialized views with one to many subqueries was not supported prior to release 9.0.1 of Oracle. The compatibility level is controlled by the `COMPATIBLE` initialization parameter.

Many to Many Subqueries

Consider a scenario where you have the `order_items` table and `inventories` table in the `oe` schema, and you want to create a materialized view of the `inventories` table based on data in both the `inventories` table and the `order_items` table. For example, suppose a salesperson wants to see all of the inventories with a quantity on hand greater than 0 (zero) for each product whose `product_id` is in the `order_items` table. In other words, the salesperson wants to see the inventories that are greater than zero for all of the products that customers have ordered. Here, an inventory is a certain quantity of a product at a particular warehouse. So, a certain product can be in many order items and in many inventories.

To accomplish the salesperson's goal, you can create a materialized view with a subquery on the many to many relationship between the `order_items` table and the `inventories` table.

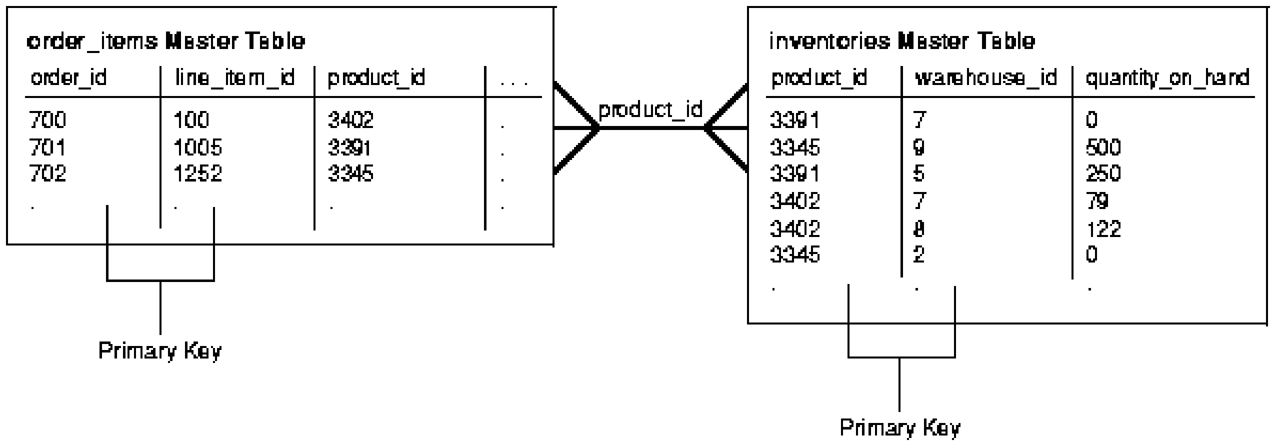
When you create the `inventories` materialized view, you want to retrieve the inventories with the quantity on hand greater than zero for the products that appear in the `order_items` table. Look at the relationships in Figure 3-5, and note that the `inventories` table and `order_items` table are related through the `product_id` column. The following statement creates the materialized view:

```
CREATE MATERIALIZED VIEW oe.inventories REFRESH FAST FOR UPDATE AS
  SELECT * FROM oe.inventories@orc1.world i
  WHERE i.quantity_on_hand > 0 AND EXISTS
    (SELECT * FROM oe.order_items@orc1.world o
     WHERE i.product_id = o.product_id);
```

Note:

To create this `oe.inventories` materialized view, the `product_id` column in the `order_items` table must be logged in the master's materialized view log. See "Logging Columns in the Materialized View Log" for more information.

Figure 3-5 Row Subsetting with Many to Many Subqueries



Text description of the illustration rep81089.gif

The materialized view created by this statement is fast refreshable and updatable. If new inventories that are greater than zero are identified for products in the `order_items` table, then the new data will be propagated to the materialized view site during the subsequent refresh process. Similarly, if a customer cancels an order for a product and there are no other orders for the product in the `order_items` table, then the inventories for the product will be removed from the materialized view during the subsequent refresh process.

Note:

The materialized view site must have a compatibility level of 9.0.1 or higher because fast refresh of materialized views with many to many subqueries was not supported prior to release 9.0.1 of Oracle. The compatibility level is controlled by the `COMPATIBLE` initialization parameter.

Materialized Views with Subqueries and Unions

In situations where you want a single materialized view to contain data that matches the complete results of two or more different queries, you can use the `UNION` operator. When you use the `UNION` operator to create a materialized view, you have two `SELECT` statements around each `UNION` operator, one is above it and one is below it. The resulting materialized view contains rows selected by either query.

You can use the `UNION` operator as a way to create fast refreshable materialized views that satisfy "or" conditions without using the `OR` expression in the `WHERE` clause of a subquery. Under some conditions, using an `OR` expression in the `WHERE` clause of a subquery causes the resulting materialized view to be complex, and therefore not fast refreshable.

See Also:

"Restrictions for Materialized Views with Subqueries" for more information about the `OR` expressions in subqueries

For example, suppose a salesperson wants the product information for the products in a particular `category_id` that are *either* in a warehouse in California *or* contain the word "Rouge" in their translated product descriptions (for the French translation). The following statement uses the `UNION` operator and subqueries to capture this data in a materialized view for products in `category_id` 29:

```
CREATE MATERIALIZED VIEW oe.product_information REFRESH FAST FOR UPDATE AS
 SELECT * FROM product_information@orc1.world pi
 WHERE pi.category_id = 29 AND EXISTS
  (SELECT * FROM product_descriptions@orc1.world pd
  WHERE pi.product_id = pd.product_id AND translated_description LIKE '%Rouge%')
UNION
 SELECT * FROM product_information@orc1.world pi
```

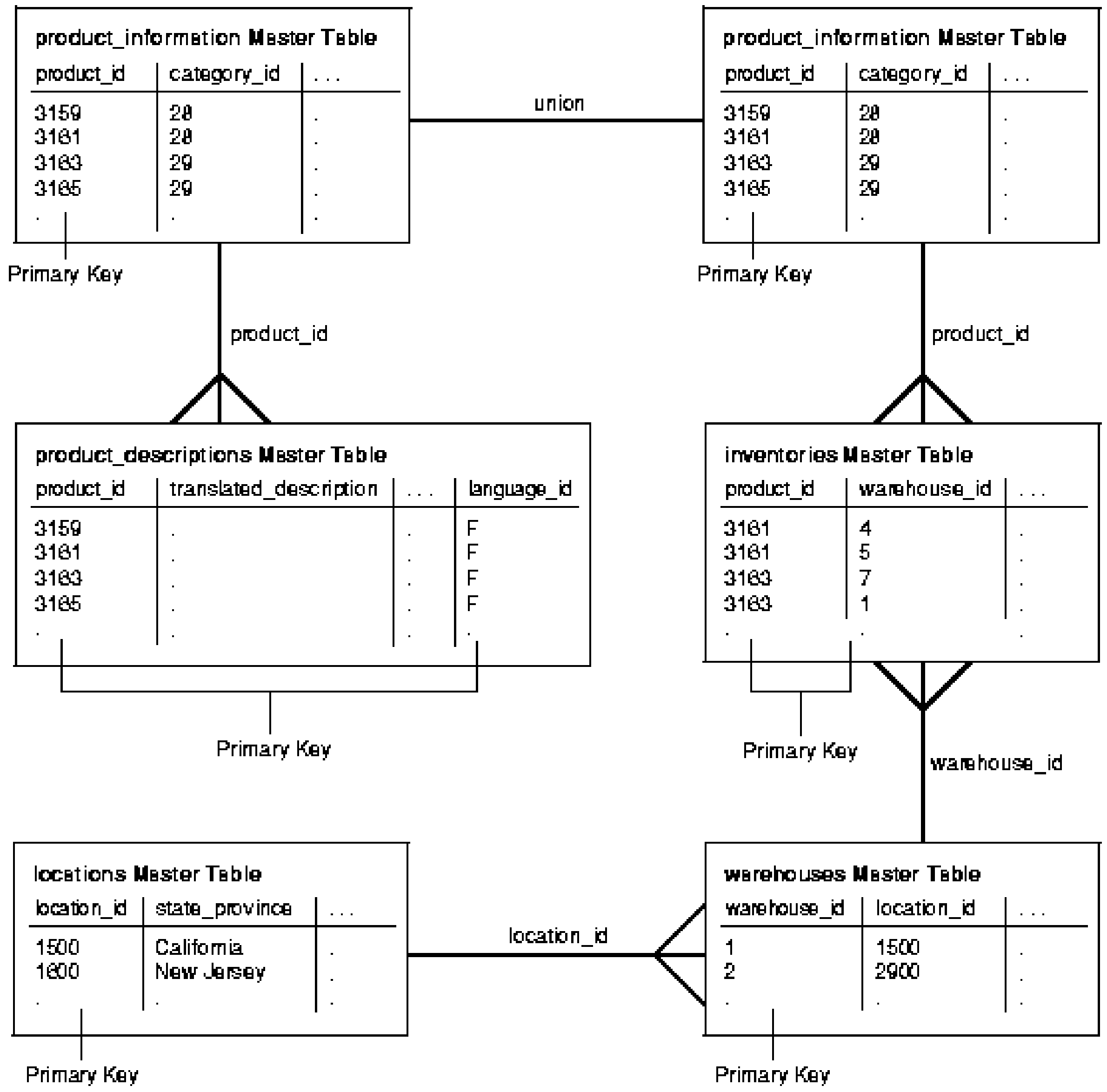
```
WHERE pi.category_id = 29 AND EXISTS
 (SELECT * FROM oe.inventories@orc1.world i
 WHERE pi.product_id = i.product_id AND EXISTS
   (SELECT * FROM oe.warehouses@orc1.world w
   WHERE i.warehouse_id = w.warehouse_id AND EXISTS
     (SELECT * FROM hr.locations@orc1.world l
      WHERE w.location_id = l.location_id
      AND l.state_province = 'California')));
```

Note:

To create the `oe.product_information` materialized view, `translated_description` in the `product_descriptions` table and `state_province` in the `locations` table must be logged in each master's materialized view log. See "Logging Columns in the Materialized View Log" for more information.

Figure 3-6 shows the relationships of the master tables involved in this statement.

Figure 3-6 Row Subsetting with Subqueries and Unions


product_information Master Table
product_id | category_id | . . .
3159 28
3161 28
3163 29
3165 29
Primary Key
union
product_id
product_descriptions Master Table
product_id | translated_description | . . . | language_id
3159 F
3161 F
3163 F
3165 F
inventories Master Table
product_id | warehouse_id | . . .
3161 4
3161 5
3163 7
3163 1
warehouse_id
locations Master Table
location_id | state_province | . . .
1500 California
1600 New Jersey
location_id
warehouses Master Table
warehouse_id | location_id | . . .
1 1500
2 2900
Primary Key

[Text description of the illustration rep81090.gif]()

In addition to the `UNION` operation, this statement contains the following subqueries:

- A subquery referencing the `product_information` table and the `product_descriptions` table. This subquery is one to many because one product can have multiple product descriptions (for different languages).

- A subquery referencing the `product_information` table and the `inventories` table. This subquery is one to many because a product can be in many inventories.
- A subquery referencing the `inventories` table and the `warehouses` table. This subquery is many to one because many inventories can be stored in one warehouse.
- A subquery referencing the `warehouses` table and the `locations` table. This subquery is many to one because many warehouses can be in one location.

The materialized view created by this statement is fast refreshable and updatable. If a new product is added that is stored in a warehouse in California or that has the string "Rouge" in the translated product description, then the new data will be propagated to the `product_information` materialized view during the subsequent refresh process.

Note:

The materialized view site must have a compatibility level of 9.0.1 or higher because fast refresh of materialized views with a `UNION` operator was not supported prior to release 9.0.1 of Oracle. Also, fast refresh of materialized views with many to one subqueries requires 9.0.1 or higher compatibility. The compatibility level is controlled by the `COMPATIBLE` initialization parameter.

Restrictions for Materialized Views with Subqueries

The defining query of a materialized view with a subquery is subject to several restrictions to preserve the materialized view's fast refresh capability.

The following are restrictions for fast refresh materialized views with subqueries:

- Materialized views must be primary key materialized views.
- The master's materialized view log must include certain columns referenced in the subquery. For information about which columns must be included, see "Logging Columns in the Materialized View Log".
- If the subquery is many to many or one to many, join columns that are not part of a primary key must be included in the materialized view log of the master. This restriction does not apply to many to one subqueries.
- The subquery must be a positive subquery. For example, you can use `EXISTS`, but not `NOT EXISTS`.
- The subquery must use `EXISTS` to connect each nested level (`IN` is not allowed).
- Each table can be in only one `EXISTS` expression.

- The join expression must use exact match or equality comparisons (that is, equi-joins).
- Each table can be joined only once within the subquery.
- A primary key must exist for each table at each nested level.
- Each nested level can only reference the table in the level above it.
- Subqueries can include `AND` operators, but each `OR` operator may only reference columns contained within one row. Multiple `OR` operators within a subquery can be connected with an `AND` operator.
- All tables referenced in a subquery must reside in the same master site or master materialized view site.

Note:

If the `CREATE MATERIALIZED VIEW` statement includes an `ON PREBUILT TABLE` clause and a subquery, then the subquery is treated as many to many. Therefore, in this case, the join columns must be recorded in the materialized view log. See the *Oracle9i SQL Reference* for more information about the `ON PREBUILT TABLE` clause in the `CREATE MATERIALIZED VIEW` statement.

See Also:

"Primary Key Materialized Views" for more information about primary key materialized views

Restrictions for Materialized Views with Unions Containing Subqueries

The following are restrictions for fast refresh materialized views with unions containing subqueries:

- All of the restrictions described in the previous section, "Restrictions for Materialized Views with Subqueries", apply to the subqueries in each union block.
- All join columns must be included in the materialized view log of the master, even if the subquery is many to one.
- All of the restrictions described in the previous section, "Complex Materialized Views", for clauses with `UNIONS`.

Examples of Materialized Views with Unions Containing Subqueries

The following statement creates the `oe.orders` materialized view. This materialized view is fast refreshable because the subquery in each union block satisfies the restrictions for subqueries described in "Restrictions for Materialized Views with Subqueries".

```
CREATE MATERIALIZED VIEW oe.orders REFRESH FAST AS
```

```
  SELECT * FROM oe.orders@orc1.world o
  WHERE EXISTS
    (SELECT * FROM oe.customers@orc1.world c
     WHERE o.customer_id = c.customer_id
     AND c.credit_limit > 50)
UNION
  SELECT *
  FROM oe.orders@orc1.world o
  WHERE EXISTS
    (SELECT * FROM oe.customers@orc1.world c
     WHERE o.customer_id = c.customer_id
     AND c.account_mgr_id = 30);
```

Notice that one of the restrictions for subqueries states that each table can be in only one `EXISTS` expression. Here, the `customers` table appears in two `EXISTS` expressions, but the `EXISTS` expressions are in separate `UNION` blocks. Because the restrictions described in "Restrictions for Materialized Views with Subqueries" only apply to each `UNION` block, not to the entire `CREATE MATERIALIZED VIEW` statement, the materialized view is fast refreshable.

In contrast, the materialized view created with the following statement cannot be fast refreshed because the `orders` table is referenced in two different `EXISTS` expressions within the same `UNION` block:

```
CREATE MATERIALIZED VIEW oe.orders AS
  SELECT * FROM oe.orders@orc1.world o
  WHERE EXISTS
    (SELECT * FROM oe.customers@orc1.world c
     WHERE o.customer_id = c.customer_id  -- first reference to orders table
     AND c.credit_limit > 50
     AND EXISTS
        (SELECT * FROM oe.orders@orc1.world o
         WHERE order_total > 5000
         AND o.customer_id = c.customer_id)) -- second reference to orders table
UNION
  SELECT *
  FROM oe.orders@orc1.world o
  WHERE EXISTS
    (SELECT * FROM oe.customers@orc1.world c
     WHERE o.customer_id = c.customer_id
     AND c.account_mgr_id = 30);
```

Determining the Fast Refresh Capabilities of a Materialized View

To determine whether a materialized view's subquery satisfies the restrictions detailed in the previous section, create the materialized view with fast refresh. Oracle returns errors if the materialized view violates any restrictions for subquery materialized views. If you specify force refresh, then you may not receive any errors because, when a force refresh is requested, Oracle automatically performs a complete refresh if it cannot perform a fast refresh.

You can also use the `EXPLAIN_MVIEW` procedure in the `DBMS_MVIEW` package to determine the following information about an existing materialized view or a proposed materialized view that does not yet exist:

- The capabilities of a materialized view
- Whether each capability is possible
- If a capability is not possible, why it is not possible

This information can be stored in a varray or in the `MV_CAPABILITIES_TABLE`. If you want to store the information in the table, then, before you run the `EXPLAIN_MVIEW` procedure, you must build this table by running the `utlxmv.sql` script in the *Oracle_home*`/rdbms/admin` directory.

For example, to determine the capabilities of the `oe.orders` materialized view, enter:

```
EXECUTE DBMS_MVIEW.EXPLAIN_MVIEW ('oe.orders');
```

Or, if the materialized view does not yet exist, then you can supply the query that you want to use to create it:

```
BEGIN
  DBMS_MVIEW.EXPLAIN_MVIEW ('SELECT * FROM oe.orders@orc1.world o
    WHERE EXISTS (SELECT * FROM oe.customers@orc1.world c
    WHERE o.customer_id = c.customer_id AND c.credit_limit > 500)');
END;
/
```

Query the `MV_CAPABILITIES_TABLE` to see the results.

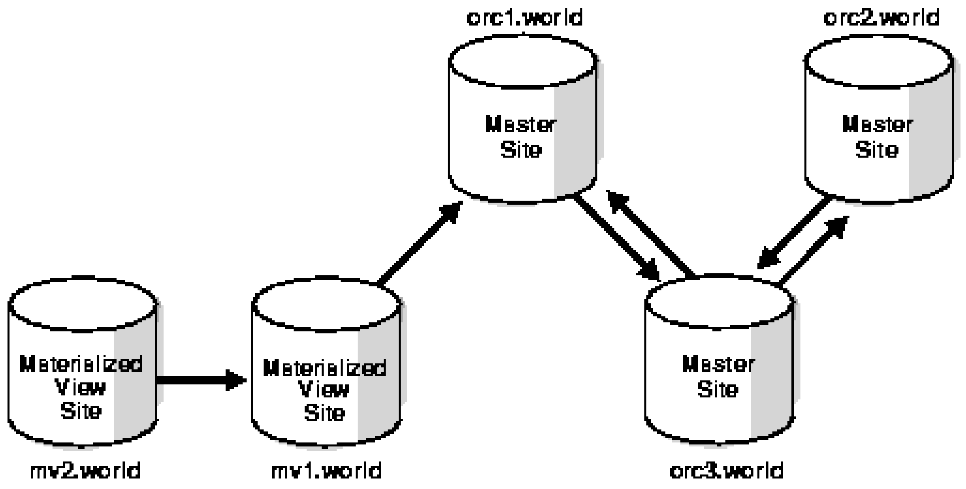
See Also:

Oracle9i Data Warehousing Guide for more information about the `EXPLAIN_MVIEW` procedure

Multitier Materialized Views

The ability to create materialized views that are based on other materialized views enables you to create **multitier materialized views**. Materialized views that are based on other materialized views can be read-only or updatable. The arrows in Figure 3-7 represent database links.

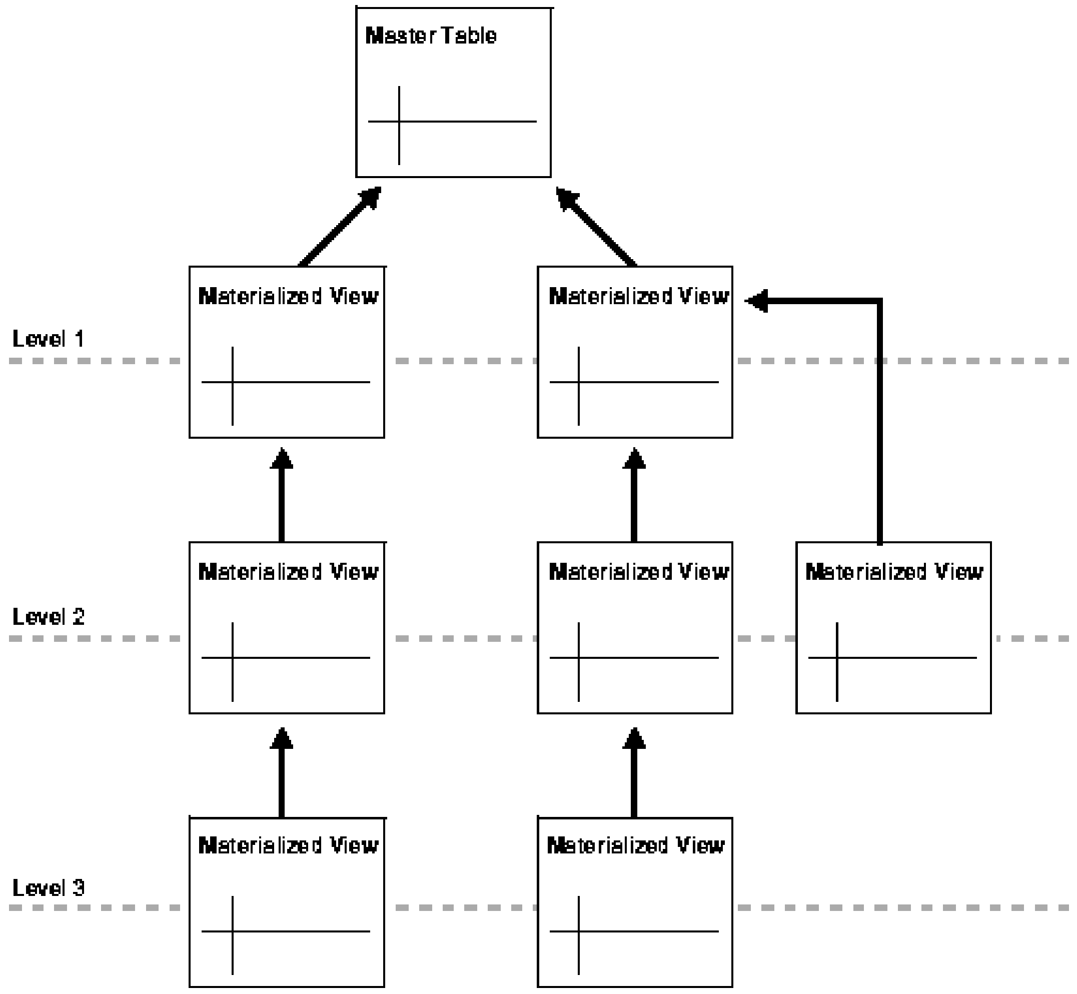
Figure 3-7 Multitier Materialized Views



Text description of the illustration rep81085.gif

When you are using multitier materialized views, the materialized view based on a master table is called a level 1 materialized view. Then, a materialized view based on the level 1 materialized view is called a level 2 materialized view. Next is level 3 and so on. Figure 3-8 shows these levels.

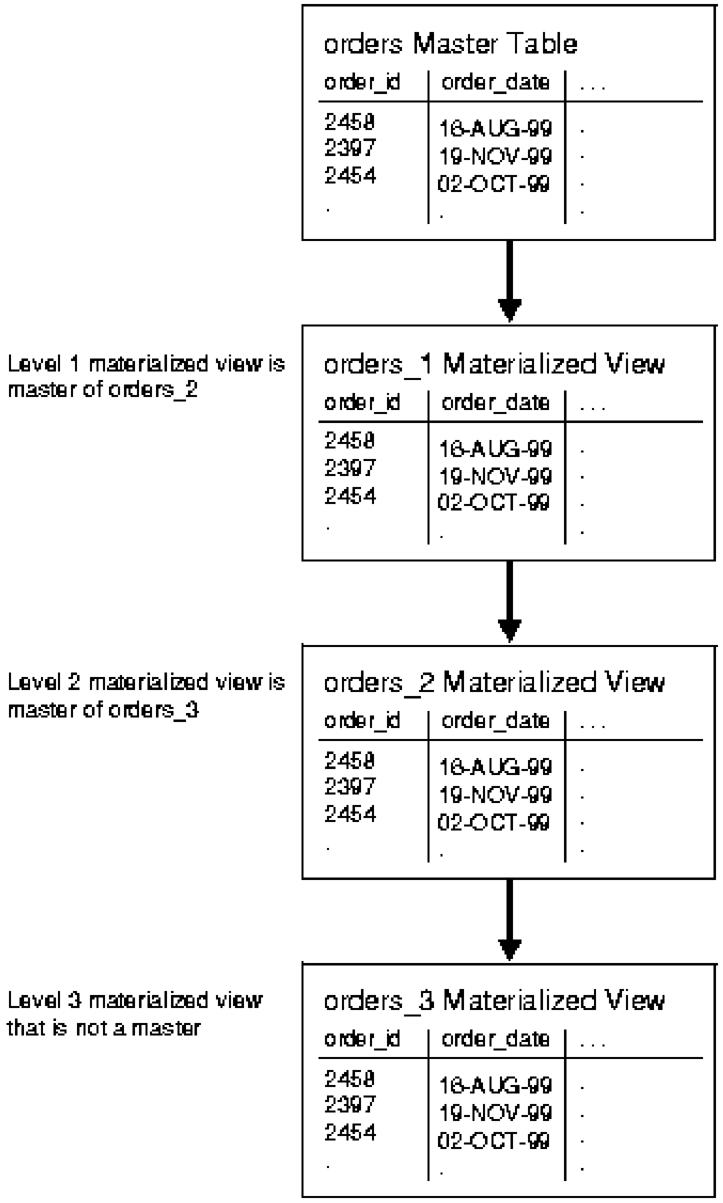
Figure 3-8 Levels of Materialized Views



Text description of the illustration rep81095.gif

A materialized view that is acting as the master for another materialized view is called a **master materialized view**. A materialized view at any level can be a master materialized view, and, as you can see in Figure 3-8, a master materialized view can have more than one materialized view based on it. In Figure 3-8, two level 2 materialized views are based on one level 1 materialized view. Figure 3-9 illustrates an example that shows a master materialized view at level 1 (`orders_1`) and level 2 (`orders_2`).

Figure 3-9 Master Materialized Views


orders Master Table
order_id | order_date | . . .
2458 2397 2454 . | 18-AUG-99 19-NOV-99 02-OCT-99 .
Level 1 materialized view is master of orders_2
orders_1 Materialized View
Level 2 materialized view is master of orders_3
orders_2 Materialized View
Level 3 materialized view that is not a master
orders_3 Materialized View

Text description of the illustration rep81097.gif

The master for the level 1 materialized view `orders_1` is the master table `orders` at the master site, but, starting with level 2, each materialized view has a master materialized view at the level above it. For example, the master for the level 2 materialized view `orders_2` is the level 1 materialized view `orders_1`.

A master materialized view functions the same way a master table does at a master site. That is, changes pushed from a level 2 materialized view to a level 1 materialized view are handled in exactly the same way that changes pushed from a level 1 materialized view to a master table are handled.

A receiver must be registered at a master materialized view site. The receiver is responsible for receiving and applying the deferred transactions from the propagator at multitier materialized view sites that are based on the master materialized view.

See Also:

"Receiver"

Multitier materialized views offer greater flexibility in the design of a replication environment. Some materialized view sites may not need to replicate all of the data in master tables, and, in fact, these sites may not have the storage capacity for all of the data. In addition, replicating less data means that there is less activity on the network.

Multitier materialized views are ideal for organizations that are structured on three or more levels or constrained by limited network resources. For example, consider a company with international, national, and local offices. This company has many computers at both the national and local level that replicate data. Here, the replication environment can be configured with the master site at the international headquarters and with materialized views at the national level. These materialized views at the national level only replicate the subset of data from the master tables that apply to their respective countries. Now, using multitier materialized views, another level of materialized views at the local level can be based on the materialized views at the national level. The materialized views at the local level contain the subset of data from the level 1 materialized views that apply to their local customers.

Scenario for Using Multitier Materialized Views

Consider a multinational company that maintains all employee information at headquarters, which is in the in the United States. The company uses the tables in the `hr` schema to maintain the employee information. This company has one main office in 14 countries and many regional offices for cities in these countries.

For example, the company has one main office for all of the United Kingdom, but it also has an office in the city of London. The United Kingdom office maintains employee information for all of the employees in the United Kingdom, while the London office only maintains employee information for the employees at the London office. In this scenario, the `hr.employees` master table is at headquarters in the United States and each regional office has a an `hr.employees` materialized view that only contains the necessary employee information.

The following statement creates the `hr.employees` materialized view for the United Kingdom office. The statement queries the master table in the database at headquarters, which is `orc1.world`. Notice that the statement uses subqueries so that the materialized view only contains employees whose `country_id` is `UK`.

```
CREATE MATERIALIZED VIEW hr.employees REFRESH FAST FOR UPDATE AS
```

```
  SELECT * FROM hr.employees@orc1.world e
    WHERE EXISTS
      (SELECT * FROM hr.departments@orc1.world d
       WHERE e.department_id = d.department_id
       AND EXISTS
         (SELECT * FROM hr.locations@orc1.world l
          WHERE l.country_id = 'UK'
          AND d.location_id = l.location_id));
```

Note:

To create this `hr.employees` materialized view, the following columns must be logged:

- The `department_id` column must be logged in the materialized view log for the `hr.employees` master table at `orc1.world`.
- The `country_id` must be logged in the materialized view log for the `hr.locations` master table at `orc1.world`.

See "Logging Columns in the Materialized View Log" for more information.

The following statement creates the `hr.employees` materialized view for the London office based on the level 1 materialized view at the United Kingdom office. The statement queries the materialized view in the database at the United Kingdom office, which is `reg_uk.world`. Notice that the statement uses subqueries so that the materialized view only contains employees whose `city` is `London`.

```
CREATE MATERIALIZED VIEW hr.employees REFRESH FAST FOR UPDATE AS
  SELECT * FROM hr.employees@reg_uk.world e
    WHERE EXISTS
      (SELECT * FROM hr.departments@reg_uk.world d
       WHERE e.department_id = d.department_id
       AND EXISTS
         (SELECT * FROM hr.locations@reg_uk.world l
          WHERE l.city = 'London'
          AND d.location_id = l.location_id));
```

Note:

To create this `hr.employees` materialized view, the following columns must be logged:

- The `department_id` column must be logged in the materialized view log for the `hr.employees` master materialized view at `reg_uk.world`.
- The `country_id` must be logged in the materialized view log for the `hr.locations` master materialized view at `reg_uk.world`.

See "Logging Columns in the Materialized View Log" for more information.

Restrictions for Using Multitier Materialized Views

Both master materialized views and materialized views based on materialized views must:

- Be primary key materialized views
- Reside in a database that is at 9.0.1 or higher compatibility level

Note:

The `COMPATIBLE` initialization parameter controls a database's compatibility level.

Additional Restrictions for Master Materialized Views

The following types of materialized views cannot be masters for updatable materialized views:

- `ROWID` materialized views
- Complex materialized views
- Read-only materialized views

However, these types of materialized views can be masters for read-only materialized views.

Additional Restrictions for Updatable Materialized Views Based on Materialized Views

Updatable materialized views based on materialized views must:

- Belong to a materialized view group that has the same name as the materialized view group at its master materialized view site
- Reside in a different database than the materialized view group at its master materialized view site
- Be based on another updatable materialized view or other updatable materialized views, not on a read-only materialized view
- Be based on a materialized view in a materialized view group that is owned by `PUBLIC` at the master materialized view site.

Materialized Views with User-Defined Types

Oracle **object types** are user-defined datatypes that make it possible to model complex real-world entities such as customers and orders as single entities, called **objects**, in the database. You create object types using the `CREATE TYPE ... AS OBJECT` statement. You can replicate object types and objects between master sites and materialized view sites in a replication environment.

An Oracle object that occupies a single column in a table is called a **column object**. Typically, tables that contain column objects also contain other columns, which may be built-in datatypes, such as `VARCHAR2` and `NUMBER`. An **object table** is a special kind of table in which each row represents an object. Each row in an object table is a **row object**.

You can also replicate **collections**. Collections are user-defined datatypes that are based on `VARRAY` and nested table datatypes. You create varrays with the `CREATE TYPE ... AS VARRAY` statement, and you create nested tables with the `CREATE TYPE ... AS TABLE` statement.

Note:

- Both the master site and the materialized view site must have a compatibility level of 9.0.1 or higher to replicate user-defined types and any objects on which they are based. The compatibility level is controlled by the `COMPATIBLE` initialization parameter.
- You cannot create refresh-on-commit materialized views based on a master with user-defined types. Refresh-on-commit materialized views are those created using the `ON COMMIT REFRESH` clause in the `CREATE MATERIALIZED VIEW` statement.
- Advanced Replication does not support type inheritance.

See Also:

Oracle9i Application Developer's Guide - Object-Relational Features for detailed information about user-defined types, Oracle objects, and collections. This section assumes a basic understanding of the information in that book.

Type Agreement at Replication Sites

User-defined types include all types created using the `CREATE TYPE` statement, including object, nested table, `VARRAY`, and indextype. To replicate schema objects based on user-defined types, the user-defined types themselves must exist, and must be exactly the same, at all replication sites. In addition, Oracle Corporation recommends that you add a user-defined type to the replication group in which it is used, but doing so is not required.

When replicating user-defined types and the schema objects on which they are based, the following conditions apply:

- The user-defined types replicated at the master site and materialized view site must be created at the materialized view site before you create any materialized views that depend on these types.
- All of the masters on which a materialized view is based must be at the same master site to create a materialized view with user-defined types.
- A user-defined type must be exactly the same at all replication sites:
    - All replication sites must have the same object identifier (OID), schema owner, and type name for each replicated user-defined type.
    - If the user-defined type is an object type, then all replication sites must agree on the order and datatype of the attributes in the object type. You establish the order and datatypes of the attributes when you create the object type. For example, consider the following object type:

      ```
      CREATE TYPE cust_address_typ AS OBJECT
           (street_address     VARCHAR2(40),
            postal_code        VARCHAR2(10),
            city               VARCHAR2(30),
            state_province     VARCHAR2(10),
            country_id         CHAR(2));
      /
      ```

      At all replication sites, `street_address` must be the first attribute for this type and must be `VARCHAR2(40)`, `postal_code` must be the second attribute and must be `VARCHAR2(10)`, `city` must be the third attribute and must be `VARCHAR2(30)`, and so on.
    - All replication sites must agree on the hashcode of the user-defined type. Oracle examines a user-defined type and assigns the hashcode. This examination includes the type attributes, order of attributes, and type name. When all of these items are the same for two or more types, the types have the same hashcode. You can view the hashcode for a type by querying the `DBA_TYPE_VERSIONS` data dictionary view.

To ensure that a user-defined type is exactly the same at all replication sites, you must create the user-defined type at the materialized view site in one of the following ways:

- Use the Replication Management API

- Use a CREATE TYPE Statement

Use the Replication Management API

Oracle Corporation recommends that you use the replication management API to create, modify, or drop any replicated object at a materialized view site, including user-defined types. If you do not use the replication management API for these actions, then replication errors may result.

Specifically, to create a user-defined type that is exactly the same at the master site and the materialized view site, use the `CREATE_MVIEW_REPOBJECT` procedure in the `DBMS_REPCAT` package. This procedure creates the type and adds it to a materialized view group. To drop a user-defined type from the materialized view site, use the `DROP_MVIEW_REPOBJECT` procedure in the `DBMS_REPCAT` package.

See Also:

Oracle9i Replication Management API Reference

Use a CREATE TYPE Statement

You can use a `CREATE TYPE` statement at the materialized view site to create the type. It may be necessary to do this if you want to create a read-only materialized view that uses the type, and you do not want to add the read-only materialized view to a materialized view group.

If you choose this option, then you must ensure the following:

- The type is in the same schema at both the materialized view site and the master site.
- The type has exactly the same attributes in exactly the same order at both the materialized view site and the master site.
- The type has exactly the same datatype for each attribute at both the materialized view site and the master site.
- The type has the same object identifier at both the materialized view site and the master site.

You can find the object identifier for a type by querying the `DBA_TYPES` data dictionary view. For example, to find the object identifier (OID) for the `cust_address_typ`, enter the following query:

```
SELECT TYPE_OID FROM DBA_TYPES WHERE TYPE_NAME = 'CUST_ADDRESS_TYP';

TYPE_OID
--------------------------------
6F9BC33653681B7CE03400400B40A607
```

Now that you know the OID for the type at the master site, complete the following steps to create the type at the materialized view site:

1. Log in to the materialized view site as the user who owns the type at the master site. If this user does not exist at the materialized view site, then create the user.
2. Issue the `CREATE TYPE` statement and specify the OID:

```
CREATE TYPE oe.cust_address_typ OID '6F9BC33653681B7CE03400400B40A607'
     AS OBJECT (
     street_address     VARCHAR2(40),
     postal_code        VARCHAR2(10),
     city               VARCHAR2(30),
     state_province     VARCHAR2(10),
     country_id         CHAR(2));
/
```

The type is now ready for use at the materialized view site.

Column Subsetting of Masters with Column Objects

A read-only materialized view can replicate specific attributes of a column object without replicating other attributes. For example, using the `cust_address_typ` user-defined datatype described in the previous section, suppose a `customers` master table is created at master site `orc1.world`:

```
CREATE TABLE oe.customers (
          customer_id        NUMBER(6),
      cust_first_name    VARCHAR2(20),
      cust_last_name     VARCHAR2(20),
      cust_address       cust_address_typ);
```

You can create the following read-only materialized view at a remote materialized view site:

```
CREATE MATERIALIZED VIEW oe.customers_mv1 AS
   SELECT customer_id, cust_last_name, c.cust_address.postal_code
   FROM oe.customers@orc1.world c;
```

Notice that the `postal_code` attribute is specified in the `cust_address` column object.

An updatable materialized view must replicate the entire column object. It cannot replicate some attributes of a column object but not others.

The following statement is valid because it specifies the entire `cust_address` column object:

```
CREATE MATERIALIZED VIEW oe.customers_mv1 FOR UPDATE AS
   SELECT customer_id, cust_last_name, cust_address
   FROM oe.customers@orc1.world;
```

See Also:

"Column Subsetting with Deployment Templates" for more information about column subsetting with deployment templates. Column subsetting is supported only through the use of deployment templates.

Materialized Views Based on Object Tables

If a materialized view is based on an object table and is created using the `OF` *type* clause, then the materialized view is called an **object materialized view**. An object materialized view is structured in the same way as an object table. That is, an object materialized view is composed of row objects. If a materialized view that is based on an object table is created without using the `OF` *type* clause, then the materialized view is read-only and is not an object materialized view. That is, such a materialized view has regular rows, not row objects.

To create a materialized view based on an object table, the types on which the materialized view depends must exist at the materialized view site, and each type must have the same object identifier as it does at the master site.

Creation of Object Materialized Views Using the OF *type* Clause

After the required types are created at the materialized view site, you can create an object materialized view by specifying the `OF` *type* clause.

For example, suppose the following SQL statements create the `categories_tab` object table at the `orc1.world` master site:

```
CREATE TYPE category_typ AS OBJECT
   (category_name           VARCHAR2(50),
    category_description    VARCHAR2(1000),
    category_id             NUMBER(2))
NOT FINAL;
/

CREATE TABLE categories_tab OF category_typ
    (category_id    PRIMARY KEY);
```

If you want to create materialized views that can be fast refreshed based on the `categories_tab` master table, then create a materialized view log for this table:

```
CREATE MATERIALIZED VIEW LOG ON categories_tab WITH OBJECT ID;
```

The `WITH OBJECT ID` clause is required when you create a materialized view log on an object table.

After you create the `category_typ` type at the materialized view site, you can create an object materialized view based on the `categories_tab` object table using the `OF` *type* clause, as in the following SQL statement:

```
CREATE MATERIALIZED VIEW categories_objmv OF category_typ
   REFRESH FAST FOR UPDATE
   AS SELECT * FROM categories_tab@orc1.world;
```

Here, *type* is `category_typ`.

Note:

The types must be exactly the same at the materialized view site and master site. See "Type Agreement at Replication Sites" for more information.

Materialized Views Based on Object Tables Created Without Using the OF *type* Clause

If you create a materialized view based on an object table without using the `OF` *type* clause, then the materialized view is read-only, and it loses the object properties of the object table on which it is based. That is, the resulting read-only materialized view contains one or more of the columns of the master, but each row functions as a row in a relational table. The rows are not row objects.

For example, you can create a materialized view base on the `categories_tab` master by using the following SQL statement:

```
CREATE MATERIALIZED VIEW categories_relmv
   AS SELECT * FROM categories_tab@orc1.world;
```

In this case, the `categories_relmv` materialized view must be read-only, and the rows in this materialized view function in the same way as rows in a relational table.

OID Preservation in Object Materialized Views

An object materialized view inherits the object identifier (OID) specifications of its master. If the master has a primary key-based OID, then the OIDs of row objects in the materialized view are primary key-based. If the master has a system generated OID, then the OIDs of row objects in the materialized view are system generated. Also, the OID of each row in the object materialized view matches the OID of the same row in the master, and the OIDs are preserved during refresh of the materialized view. Consequently, `REF`s to the rows in the object table remain valid at the materialized view site.

Materialized Views with Collection Columns

Collection columns are columns based on varray and nested table datatypes. Oracle supports the creation of materialized views with collection columns.

If the collection column is a nested table, then you can optionally specify the *nested_table_storage_clause* during materialized view creation. The *nested_table_storage_clause* lets you specify the name of the storage table for the nested table in the materialized view. For example, suppose you create the master table `people_reltab` at the master site `orc1.world` that contains the nested table `phones_ntab`:

```
CREATE TYPE phone_typ AS OBJECT (
   location    VARCHAR2(15),
   num         VARCHAR2(14));
/

CREATE TYPE phone_ntabtyp AS TABLE OF phone_typ;
/

CREATE TABLE people_reltab (
   id               NUMBER(4) CONSTRAINT pk_people_reltab PRIMARY KEY,
   first_name       VARCHAR2(20),
   last_name        VARCHAR2(20),
   phones_ntab      phone_ntabtyp)
   NESTED TABLE phones_ntab STORE AS phone_store_ntab
   ((PRIMARY KEY (NESTED_TABLE_ID, location)));
```

Notice the `PRIMARY KEY` specification in the last line of the preceding SQL statement. You must specify a primary key for the storage table if you plan to create materialized views based on its parent table. In this case, the storage table is `phone_store_ntab` and the parent table is `people_reltab`.

If you want to create materialized views that can be fast refreshed, then create a materialized view log on both the parent table and the storage table, specifying the nested table column as a filter column for the parent table's materialized view log:

```
CREATE MATERIALIZED VIEW LOG ON people_reltab;
```

```
ALTER MATERIALIZED VIEW LOG ON people_reltab ADD(phones_ntab);

CREATE MATERIALIZED VIEW LOG ON phone_store_ntab WITH PRIMARY KEY;
```

At the materialized view site, create the required types, ensuring that the object identifier for each type is the same as the object identifier at the master site. Then, you can create a materialized view based on `people_reltab` and specify its storage table using the following statement:

```
CREATE MATERIALIZED VIEW people_reltab_mv
   NESTED TABLE phones_ntab STORE AS phone_store_ntab_mv
   REFRESH FAST AS SELECT * FROM people_reltab@orc1.world;
```

In this case, the *nested_table_storage_clause* is the line that begins with "`NESTED TABLE`" in the previous example, and it specifies that the storage table's name is `phone_store_ntab_mv`. The *nested_table_storage_clause* is optional. If you do not specify this clause, Oracle automatically names the storage table. To view the name of a storage table, query the `DBA_NESTED_TABLES` data dictionary table.

The storage table:

- Is a separate, secondary materialized view
- Is refreshed automatically when you refresh the materialized view containing the nested table
- Is dropped automatically when you drop the materialized view containing the nested table
- Inherits the primary key constraint of the master's storage table

Because the storage table inherits the primary key constraint of the master's storage table, do not specify `PRIMARY KEY` in the `STORE AS` clause.

The following actions are not allowed directly on the storage table of a nested table in a materialized view:

- Refreshing the storage table
- Adding the storage table to a replication group
- Altering the storage table
- Dropping the storage table
- Generating replication support on the storage table

These actions can occur indirectly when they are performed on the materialized view that contains the nested table. In addition, you cannot replicate a subset of the columns in a storage table.

See Also:

Oracle9i SQL Reference for more information about the *nested_table_storage_clause*, which is fully documented in the CREATE TABLE statement

Restrictions for Materialized Views with Collection Columns

The following restrictions apply to materialized views with collection columns:

- Row subsetting of collection columns is not allowed. However, you can use row subsetting on the parent table of a nested table and doing so can result in a subset of the nested tables in the materialized view.
- Column subsetting of collection columns is not allowed.
- A nested table's storage table must have a primary key.
- For the parent table of a nested table to be fast refreshed, both the parent table and the nested table's storage table must have a materialized view log.

Materialized Views with REF Columns

You can create materialized views with REF columns. A REF is an Oracle built-in datatype that is a logical "pointer" to a row object in an object table. A scoped REF is a REF that can contain references only to a specified object table, while an unscoped REF can contain references to any object table in the database that is based on the corresponding object type. A scoped REF requires less storage space and provides more efficient access than an unscoped REF.

As you will see in the following section, you can rescope a REF column to a local materialized view or table at the materialized view site during creation of the materialized view. If you do not rescope the REF column, then they continue to point to the remote master. Unscoped REF columns always continue to point to the master. When a REF column at a materialized view site points to a remote master, the REFs are considered dangling. In SQL, dereferencing a dangling REF returns a NULL. Also, PL/SQL only supports dereferencing REFs by using the UTL_OBJECT package and raises an exception for dangling REFs.

Scoped REF Columns

If you are creating a materialized view based on a master that has a scoped REF column, then you can rescope the REF to a different object table or object materialized view at the materialized view site. Typically, you would rescope the REF column to the local object materialized view instead of the original remote object table. To rescope a materialized view, you can either use the SCOPE FOR clause in the CREATE MATERIALIZED VIEW statement, or you can use the ALTER MATERIALIZED VIEW statement after creating the materialized view. If you do not rescope the REF column, then the materialized view retains the REF scope of the master.

For example, suppose you create the customers_with_ref master table at the orc1.world master site using the following statements:

```
-- Create the user-defined datatype cust_address_typ.
CREATE TYPE cust_address_typ AS OBJECT
   (street_address     VARCHAR2(40),
    postal_code        VARCHAR2(10),
    city               VARCHAR2(30),
    state_province     VARCHAR2(10),
    country_id         CHAR(2));
/

-- Create the object table cust_address_objtab.
CREATE TABLE cust_address_objtab OF cust_address_typ;

-- Create table with REF to cust_address_typ.
CREATE TABLE customers_with_ref (
         customer_id        NUMBER(6) PRIMARY KEY,
     cust_first_name    VARCHAR2(20),
     cust_last_name     VARCHAR2(20),
     cust_address       REF cust_address_typ SCOPE IS cust_address_objtab);
```

Assuming the `cust_address_typ` exists at the materialized view site, you can create a `cust_address_objtab_mv` object materialized view using the following statement:

```
CREATE MATERIALIZED VIEW cust_address_objtab_mv OF cust_address_typ AS
   SELECT * FROM oe.cust_address_objtab@orc1.world;
```

Now, you can create a materialized view of the `customers_with_ref` master table and rescope the `REF` to the `cust_address_objtab_mv` materialized view using the following statement:

```
CREATE MATERIALIZED VIEW customers_with_ref_mv
   (SCOPE FOR (cust_address) IS oe.cust_address_objtab_mv)
   AS SELECT * FROM oe.customers_with_ref@orc1.world;
```

If you want to use the `SCOPE FOR` clause when you create a materialized view, then remember to create the materialized view or table specified in the `SCOPE FOR` clause first. Otherwise, you cannot specify the `SCOPE FOR` clause during materialized view creation. For example, if you had created the `customers_with_ref_mv` materialized view before you created the `cust_address_objtab_mv` materialized view, then you could not use the `SCOPE FOR` clause when you created the `customers_with_ref_mv` materialized view. In this case, the `REF`s are considered dangling because they point back to the object table at the remote master site.

However, even if you do not use the `SCOPE FOR` clause when you are creating a materialized view, you can alter the materialized view to specify a `SCOPE FOR` clause. For example, you can alter the `customers_with_ref_mv` materialized view with the following statement:

```
ALTER MATERIALIZED VIEW customers_with_ref_mv
   MODIFY SCOPE FOR (cust_address) IS oe.cust_address_objtab_mv;
```

Unscoped REF Columns

If you create a materialized view based on a remote master with an unscoped `REF` column, then the `REF` column is created in the materialized view, but the `REF`s are considered dangling because they point to a remote database.

Logging REF Columns in the Materialized View Log

If necessary, you can log REF columns in the materialized view log.

See Also:

"Logging Columns in the Materialized View Log" for more information

REFs Created Using the WITH ROWID Clause

If the `WITH ROWID` clause is specified for a `REF` column, then Oracle maintains the rowid of the object referenced in the `REF`. Oracle can find the object referenced directly using the rowid contained in the `REF`, without the need to fetch the rowid from the OID index. Therefore, you use the `WITH ROWID` clause to specify a rowid hint. The `WITH ROWID` clause is not supported for scoped `REF`s.

Replicating a `REF` created using the `WITH ROWID` clause results in an incorrect rowid hint at each replication site except the site where the `REF` was first created or modified. The `ROWID` information in the `REF` is meaningless at the other sites, and Oracle does not correct the rowid hint automatically. Invalid rowid hints can cause performance problems. In this case, you must use the `ANALYZE` statement to correct rowid hints at each replication site where they are incorrect.

See Also:

Oracle9i SQL Reference for more information about the `ANALYZE` statement

Materialized View Registration at a Master Site or Master Materialized View Site

At the master site and master materialized view site, an Oracle database automatically registers information about a materialized view based on its master table(s) or master materialized view(s). The following sections explain more about Oracle's materialized view registration mechanism.

Viewing Information about Registered Materialized Views

A level 1 materialized view or materialized view group is registered at its master site. A level 2 or higher multitier materialized view or materialized view group is registered at its master materialized view site, not at the master site. You can query the `DBA_REGISTERED_MVIEWS` data dictionary view at a master site or master materialized view site to list the following information about a remote materialized view:

- The owner, name, and database that contains the materialized view
- The materialized view's defining query
- Other materialized view characteristics, such as its refresh method

You can also query the `DBA_MVIEW_REFRESH_TIMES` view at a master site or master materialized view site to obtain the last refresh times for each materialized view. Administrators can use this information to monitor materialized view activity and coordinate changes to materialized view sites if a master table or master materialized view needs to be dropped, altered, or relocated.

Internal Mechanisms

Oracle automatically registers a materialized view at its master site or master materialized view site when you create the materialized view, and unregisters the materialized view when you drop it. The same applies to materialized view groups.

When you drop a master materialized view, Oracle does not automatically drop the materialized views based on it. You must drop these materialized views manually. If you do not drop such a materialized view and the materialized view tries to refresh to a master materialized view that has been dropped, Oracle returns an error.

For example, suppose a materialized view named `orders_lev1` is based on the `oe.orders` master table, and a materialized view named `orders_lev2` is based on `orders_lev1`. If you drop `orders_lev1`, `orders_lev2` remains intact. However, if you try to refresh `orders_lev2`, Oracle returns an error because `orders_lev1` no longer exists.

Caution:

Oracle cannot guarantee the registration or unregistration of a materialized view at its master site or master materialized view site during the creation or drop of the materialized view, respectively. If Oracle cannot successfully register a materialized view during creation, then you must complete the registration manually using the `REGISTER_MVIEW` procedure in the `DBMS_MVIEW` package. If Oracle cannot successfully unregister a materialized view when you drop the materialized view, then the registration information for the materialized view persists in the master site or master materialized view site until it is manually unregistered. It is possible that complex materialized views may not be registered.

Note:

Oracle7 master sites cannot register materialized views.

Manual Materialized View Registration

If necessary, you can maintain registration manually. Use the `REGISTER_MVIEW` and `UNREGISTER_MVIEW` procedures of the `DBMS_MVIEW` package at the master site or master materialized view site to add, modify, or remove materialized view registration information.

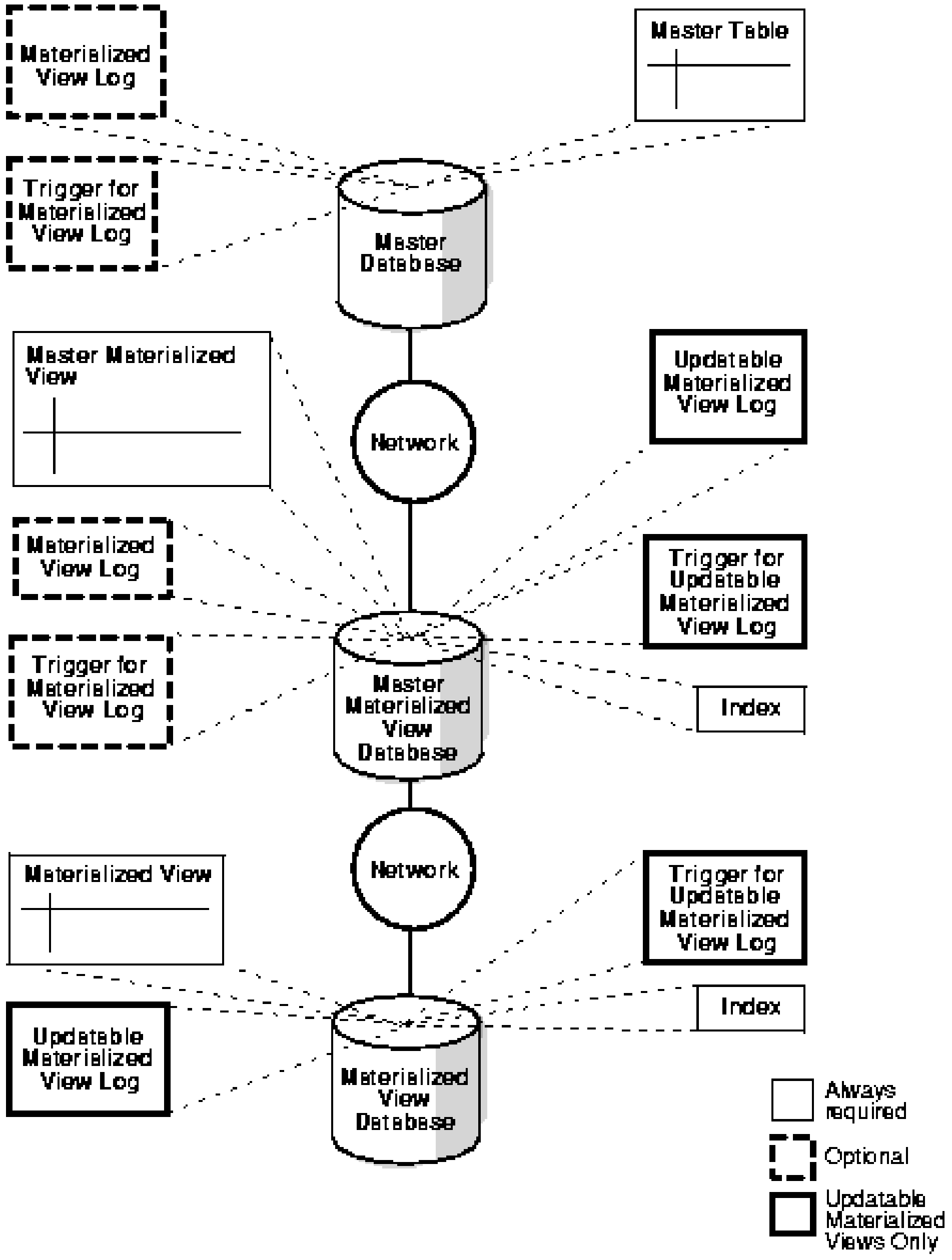
See Also:

The `REGISTER_MVIEW` and `UNREGISTER_MVIEW` procedures are described in the *Oracle9i Replication Management API Reference*

Materialized View Architecture

The objects used in materialized view replication are depicted in Figure 3-10. Some of these objects are optional and are used only as needed to support the created materialized view environment. For example, if you have a read-only materialized view, then you do not have an updatable materialized view log nor an internal trigger at the materialized view site. Also, if you have a complex materialized view that cannot be fast refreshed, then you may not have a materialized view log at the master site.

Figure 3-10 Materialized View Replication Objects


Materialized View Log
Master Table
Trigger for Materialized View Log
Master Database
Master Materialized View
Network
Updatable Materialized View Log
Materialized View Log
Trigger for Updatable Materialized View Log
Trigger for Materialized View Log
Master Materialized View Database
Index
Network
Materialized View
Trigger for Updatable Materialized View Log
Index
Updatable Materialized View Log
Materialized View Database
Always required
Optional
Updatable Materialized Views Only

Text description of the illustration rep81096.gif

Notice that a master materialized view may have both a materialized view log and an updatable materialized view log. Make sure you account for the extra space required by these logs when you are planning for your master materialized view site.

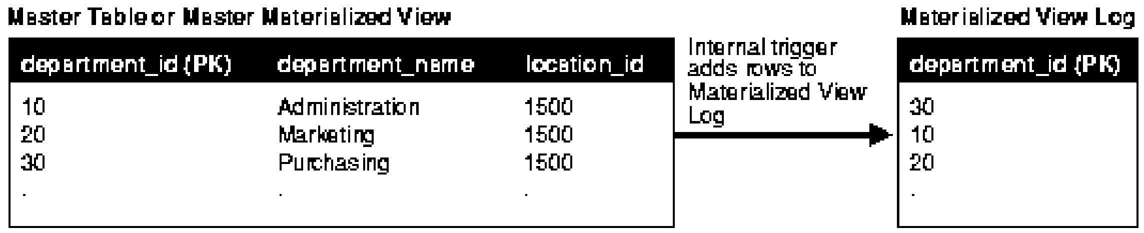
Master Site and Master Materialized View Site Mechanisms

The three mechanisms displayed in Figure 3-11 are required at a master site and at a master materialized view site to support fast refreshing of materialized views.

Note:

Master materialized views contain the mechanisms described in "Materialized View Site Mechanisms" in addition to the mechanisms described in this section.

Figure 3-11 Master Site and Master Materialized View Site Objects



Text description of the illustration rep81071.gif

Master Table or Master Materialized View

The master table or master materialized view is the basis for the materialized view. A master table is located at the target master site while a master materialized view is located at a master materialized view site. If the master is a master table, then this table may be involved in both materialized view replication and multimaster replication. Remember that a materialized view points to only one master site or master materialized view site.

Changes made to the master table or master materialized view, as recorded by the materialized view log, are propagated to the materialized view during the refresh process.

Note:

Fast refreshable materialized views can be created based on master tables and master materialized views only. Materialized views based on a synonym or a view must be complete refreshed.

Internal Trigger for the Materialized View Log

When changes are made to the master table or master materialized view using DML, an internal trigger records information about the affected rows in the materialized view log. This information includes the values of the primary key, rowid, or object id, or both, as well as the values of the other columns logged in the materialized view log. This is an internal `AFTER ROW` trigger that is automatically activated when you create a materialized view log for the target master table or master materialized view. It inserts a row into the materialized view log whenever an `INSERT`, `UPDATE`, or `DELETE` statement modifies the table's data. This trigger is always the last trigger to fire.

Note:

When the materialized view contains a subquery, you may need to log columns referenced in a subquery. See "Data Subsetting with Materialized Views" for information on subquery materialized views and "Logging Columns in the Materialized View Log" for more information about the columns that must be logged.

Materialized View Log

A materialized view log is required on a master if you want to fast refresh materialized views based on the master. When you create a materialized view log for a master table or master materialized view, Oracle creates an underlying table as the materialized view log. A materialized view log can hold the primary keys, rowids, or object identifiers of rows, or both, that have been updated in the master table or master materialized view. A materialized view log can also contain other columns to support fast refreshes of materialized views with subqueries.

The name of a materialized view log's table is `MLOG$`_*master_name*. The materialized view log is created in the same schema as the target master. One materialized view log can support multiple materialized views on its master table or master materialized view. As described in the previous section, the internal trigger adds change information to the materialized view log whenever a DML transaction has taken place on the target master.

Following are the types of materialized view logs:

- **Primary Key**: The materialized view records changes to the master table or master materialized view based on the primary key of the affected rows.

- **Row ID**: The materialized view records changes to the master table or master materialized view based on the rowid of the affected rows.
- **Object ID**: The materialized view records changes to the master object table or master object materialized view based on the object identifier of the affected row objects.
- **Combination**: The materialized view records changes to the master table or master materialized view based any combination of the three options. It is possible to record changes based on the primary key, the `ROWID`, and the object identifier of the affected rows. Such a materialized view log supports primary key, `ROWID`, and object materialized views, which is helpful for environments that have all three types of materialized views based on a master.

A combination materialized view log works in the same manner as a materialized view log that tracks only one type of value, except that more than one type of value is recorded. For example, a combination materialized view log can track both the primary key and the rowid of the affected row are recorded.

Though the difference between materialized view logs based on primary keys and rowids is small (one records affected rows using the primary key, while the other records affected rows using the physical rowid), the practical impact is large. Using rowid materialized views and materialized view logs makes reorganizing and truncating your master tables difficult because it prevents your `ROWID` materialized views from being fast refreshed. If you reorganize or truncate your master table, then your rowid materialized view must be `COMPLETE` refreshed because the rowids of the master table have changed.

Note:

- You use the `BEGIN_TABLE_REORGANIZATION` and `END_TABLE_REORGANIZATION` procedures in the `DBMS_MVIEW` package to reorganize a master table. See the *Oracle9i Replication Management API Reference* for more information.
- Online redefinition of tables is another possible way to reorganize master tables, but online redefinition is not allowed on master tables with materialized view logs, master materialized views, and materialized views. Online redefinition is allowed on master tables that do not have materialized view logs. See the *Oracle9i Database Administrator's Guide* for more information about online redefinition of tables.

Materialized View Logs on Object Tables

You can create materialized view logs on object tables. For example, the following SQL statement creates the `categories_typ` user-defined type:

```
CREATE TYPE category_typ AS OBJECT
```

```
   (category_name           VARCHAR2(50),
    category_description    VARCHAR2(1000),
    category_id             NUMBER(2))
NOT FINAL;
/
```

When you create an object table based on this type, you can either specify that the object identifier should be system-generated or primary key-based:

```
CREATE TABLE categories_tab_sys OF category_typ
    (category_id    PRIMARY KEY)
    OBJECT ID SYSTEM GENERATED;

CREATE TABLE categories_tab_pkbased OF category_typ
    (category_id    PRIMARY KEY)
    OBJECT ID PRIMARY KEY;
```

When you create a materialized view log on an object table, you must log the object identifier by specifying the `WITH OBJECT ID` clause, but you can also specify that the primary key is logged if the object identifier is primary key-based.

For example, the following statement creates a materialized view log for the `categories_tab_sys` object table and specifies that the object identifier column be logged:

```
CREATE MATERIALIZED VIEW LOG ON categories_tab_sys
   WITH OBJECT ID;
```

The following statement creates a materialized view log for the `categories_tab_pkbased` object table and specifies that the primary key column be logged along with the object identifier column:

```
CREATE MATERIALIZED VIEW LOG ON categories_tab_pkbased
   WITH OBJECT ID, PRIMARY KEY;
```

Restriction on Import of Materialized Views and Materialized View Logs to a Different Schema

Materialized views and materialized view logs are exported with the schema name explicitly given in the DDL statements. Therefore, materialized views and materialized view logs cannot be imported into a schema that is different than the schema from which they were exported. If you attempt to use the `FROM USER` and `TO USER` import options to import an export dump file that contains materialized views or materialized view logs, then an error will be written to the import log file and the items will not be imported.

Materialized View Site Mechanisms

When a materialized view is created, several additional mechanisms are created at the materialized view site to support the materialized view. Specifically, a base table, at least one index, and possibly a view are created. If you create an updatable materialized view, then an internal trigger and a local log (the updatable materialized view log) are also created at the materialized view site.

Note:

If the materialized view site is a master materialized view site, then it contains the mechanisms described in the previous section in addition to the mechanisms described in this section. See "Master Site and Master Materialized View Site Mechanisms".

See Also:

Figure 3-10, "Materialized View Replication Objects"

Base Table

The way materialized view base tables function depends on the compatibility level of your materialized view database. The compatibility setting is defined by the `COMPATIBLE` initialization parameter in the initialization parameter file.

Base Table with Compatibility at 8.1.0 or Higher

If the compatibility setting is 8.1.0 or higher, the following applies:

- The base table is the actual materialized view (an additional view is not required).
- The size limit for a materialized view name is 30 bytes. If you try to create a materialized view with a name larger than 30 bytes, Oracle returns an error.
- The materialized view has the name that you specified during materialized view creation.

Base Table with Compatibility Lower Than 8.1.0

If the compatibility setting is lower than 8.1.0, the following applies:

- The base table is the underlying support object for a view, and the view is the materialized view.

- Any indexes generated when you create the materialized view are created on the base table.
- The size limit for a materialized view name is 30 bytes. If you try to create a materialized view with a name larger than 30 bytes, Oracle returns an error.
- The materialized view has the name that you specified during materialized view creation.
- When the materialized view name is less than or equal to 19 bytes, the base table is named `SNAP$_`*materialized_view_name*.
- When the materialized view name is between 20 and 30 bytes, the base table name is truncated to 20 bytes, prefixed by `SNAP$_`, and possibly postfixed by a number. If no other base table at the materialized view site has the same name as the truncated name, then nothing is added to the truncated name. If another base table at the materialized view site has the same name as the truncated name, then the first number postfixed is 1, the second postfixed is 2, and so on up to 9999. Therefore, the name of the base table is `SNAP$_`, followed by the first 20 bytes of the materialized view name, followed by nothing or a one to four digit number.

  For example, a materialized view named `abcdefghijklmnopqrstuvwxyz` has a base table named `SNAP$_abcdefghijklmnopqrst`, assuming no other base table has the same name. A subsequently created materialized view at the same materialized view site named `abcdefghijklmnopqrstuvwxy` has a base table named `SNAP$_abcdefghijklmnopqrst1`.

Note:

The compatibility setting for Oracle release 8.0 databases must be lower than 8.1.0.

See Also:

"Initialization Parameters" and *Oracle9i Database Migration* for more information about the `COMPATIBLE` parameter, and see "View" for more information about the view that is created in support of materialized views with a compatibility level lower than 8.1.0.

View

A view is created only to support materialized view replication with Oracle release 8.0 and earlier, or if a release 8.1 or higher materialized view site's compatibility setting is less than 8.1.0. If a view is created, then the view has the same name specified in the `CREATE MATERIALIZED VIEW` statement. For example, a `CREATE MATERIALIZED VIEW sales.mview_customer AS` .... statement creates a view named `mview_customer`.

Index

At least one index is created at the remote materialized view site for each primary key materialized view. This index corresponds to the primary key of the target master table or master materialized view and has the name `I_SNAP$_`*materialized view_name*. Additional indexes may be created by Oracle at the remote materialized view site to support fast refreshing of materialized views with subqueries.

Note:

When `I_SNAP$`*_materialized view_name* exceeds the 32 character limit, the table name is truncated and a sequence number is appended.

Updatable Materialized View Log

An updatable materialized view log (`USLOG$`*_materialized_view_name*) is used to determine which rows must be overwritten or removed from a materialized view during a fast refresh. A read-only materialized view does not create this log, and Oracle does not use this log during a complete refresh because, in this case, the entire materialized view is replaced.

If there is a conflict between an updatable materialized view and a master, then, during a refresh, the conflict may result in an entry in the updatable materialized view log that is not in the materialized view log at the master site or master materialized view site. In this case, Oracle uses the updatable materialized view log to remove or overwrite the row in the materialized view.

The updatable materialized view log is also used when you fast refresh a writeable materialized view, as illustrated in the following scenario:

1. A user inserts a row into a writeable materialized view that has a remote master. Because the materialized view is writeable and not updatable, the transaction is not stored in the deferred transaction queue at the materialized view site.
2. Oracle logs information about this insert in the updatable materialized view log.
3. The user fast refreshes the materialized view.
4. Oracle uses the information in the updatable materialized view log and deletes the inserted row. A materialized view must be an exact copy of the master when the fast refresh is complete. Therefore, Oracle must delete the inserted row.

Internal Trigger for the Updatable Materialized View Log

Like the internal trigger at the master site or master materialized view site, the internal trigger at the materialized view site records DML changes applied to an updatable materialized view in the `USLOG$`*_materialized_view_name* log. A read-only materialized view does not create this trigger.

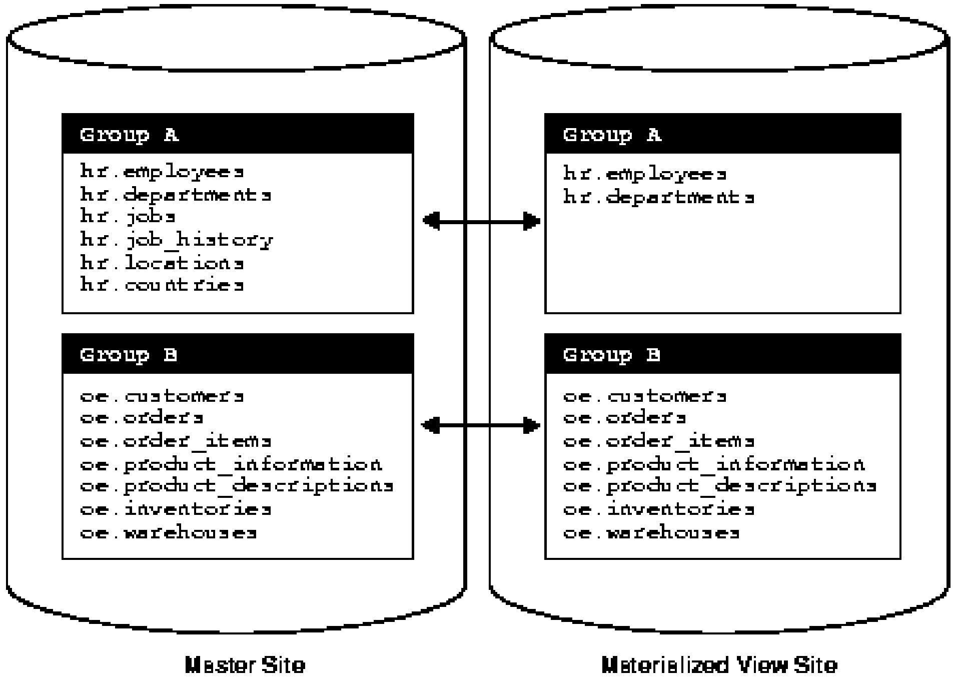
Organizational Mechanisms

In addition to the materialized view mechanisms described in the previous section, several other mechanisms organize the materialized views at the materialized view site. These mechanisms maintain organizational consistency between the materialized view site and its master site or master materialized view site, as well as transactional (read) consistency with the target replication group. These mechanisms are materialized view groups and refresh groups.

Materialized View Groups

A **materialized view group** in a replication system maintains a partial or complete copy of the objects at the target replication group at its master site or master materialized view site. Materialized view groups cannot span the boundaries of the replication group at the master site or master materialized view site. Figure 3-12 displays the correlation between Groups A and B at the master site and Groups A and B at the materialized view site.

Figure 3-12 Materialized View Groups Correspond with Master Groups



Text description of the illustration rep81074.gif

Group A at the materialized view site (see Figure 3-12) contains only some of the objects in the corresponding Group A at the master site. Group B at the materialized view site contains all objects in Group B at the master site. Under no circumstances, however, could Group B at the materialized view site contain objects from Group A at the master site. As illustrated in Figure 3-12, a materialized view group has the same name as the master group on which the materialized view group is based. For example, a materialized view group based on a `personnel` master group is also named `personnel`.

In addition to maintaining organizational consistency between materialized view sites and their master sites or master materialized view sites, materialized view groups are required for supporting updatable materialized views. If a materialized view does not belong to a materialized view group, then it must be a read-only or writeable materialized view.

Materialized View Group Owners

A materialized view group owner enables you to have multiple materialized view groups based on a single replication group at a master site or master materialized view site. For example, if you need to support multiple users within the same database at a materialized view site, then you may want to create multiple materialized view groups for a target master group. Doing so enables you to define different subqueries for your materialized view definitions in each materialized view group, and allows each user to access only his or her subset of the data.

Defining multiple materialized view groups gives you the ability to control data sets at a group level. For example, if you create different materialized view groups named `hr`, `personnel`, and `manufacturing` for these departments, then you can administer each department as a group, instead of as individual objects. For example, you can drop the objects as a group.

To accommodate multiple materialized view groups at the same materialized view site that are based on a single replication group at the master site or master materialized view site, you can specify a group owner as an additional identifier when defining your materialized view group.

After you have defined your materialized view group with the addition of a group owner, you add your materialized view objects to the target materialized view group by defining the same group owner. When using a group owner, remember that each materialized view object must have a unique name. If a single materialized view site has multiple materialized view groups based on the same replication group at the master site or master materialized view site, then a materialized view group's object names cannot have the same name as materialized view objects in another materialized view group. To avoid conflicting names, you may want to append the group owner name to the end of your object name. For example, if you have group owners `hr` and `ac`, then you might name the `employees` materialized view object as `employees_hr` and `employees_ac`, respectively.

Additionally, all materialized view groups that are based on the same replication group at a single materialized view site must "point" to the same master site or master materialized view site. For example, if the `hr_repg` materialized view group owned by `hr` is based on the associated master group at the `orc1.world` master site, then the `hr_repg` materialized view group owned by `personnel` must also be based on the associated master group at `orc1.world`, assuming that the `hr` and `personnel` owned groups are at the same materialized view site.

See Also:

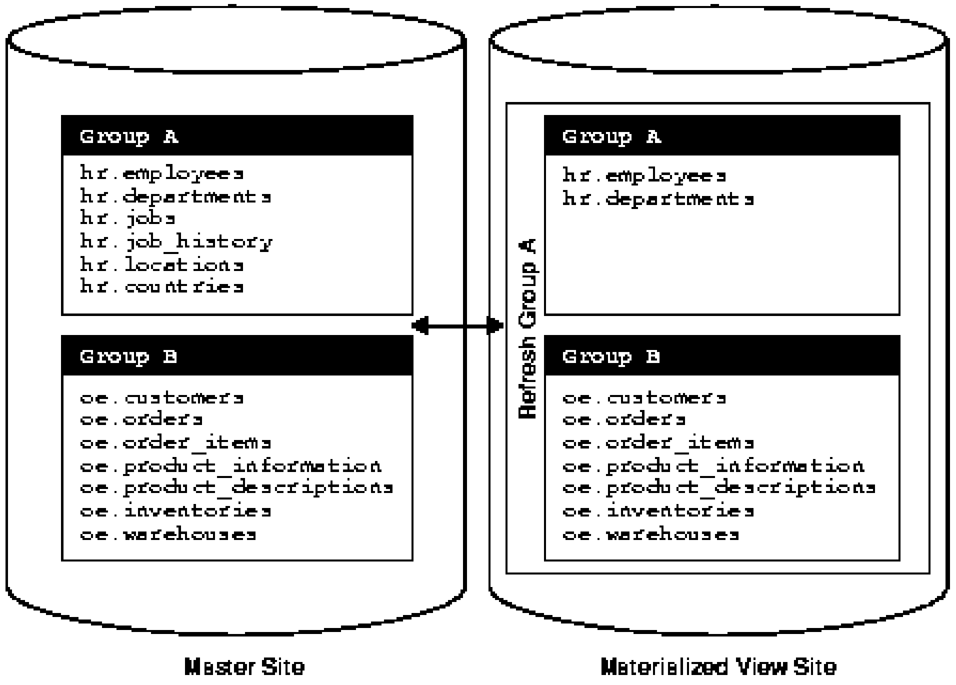
Oracle9i Replication Management API Reference for more information on defining a group owner using the replication management API

Refresh Groups

To preserve referential integrity and transactional (read) consistency among multiple materialized views, Oracle has the ability to refresh individual materialized views as part of a refresh group. After refreshing all of the materialized views in a refresh group, the data of all materialized views in the group correspond to the same transactionally consistent point in time.

As illustrated in Figure 3-13, a refresh group can contain materialized views from more than one materialized view group to maintain transactional (read) consistency across replication group boundaries.

Figure 3-13 Refresh Groups May Contain Objects from Multiple Materialized View Groups



Text description of the illustration rep81075.gif

While you may want to define a single refresh group for each materialized view group, it may be more efficient to use one large refresh group that contains objects from multiple materialized view groups. Such a configuration reduces the amount of "overhead" needed to refresh your materialized views. A refresh group can contain up to 400 materialized views.

One configuration that you want to avoid is using multiple refresh groups to refresh the contents of a single materialized view group. Using multiple refresh groups to refresh the contents of a single materialized view group may introduce inconsistencies in the materialized view data, which may cause referential integrity problems at the materialized view site. Only use this type of configuration when you have in-depth knowledge of the database environment and can prevent any referential integrity problems.

Refresh Group Size

There are a few trade-offs to consider when you are deciding on the size of your refresh groups. Oracle is optimized for large refresh groups. So, large refresh groups refresh faster than an equal number of materialized views in small refresh groups, assuming that the materialized views in the groups are similar. For example, refreshing a refresh group with 100 materialized views is faster than refreshing five refresh groups with 20 materialized views each. Also, large refresh groups enable you to refresh a greater number of materialized views with only one call to the replication management API.

During the refresh of a refresh group, each materialized view in the group is locked at the materialized view site for the amount of time required to refresh all of the materialized views in the refresh group. This locking is required to prevent users from updating the materialized views during the refresh operation, because updates may make the data inconsistent. Therefore, having smaller refresh groups means that the materialized views are locked for less time when you perform a refresh.

Network connectivity must be maintained while performing a refresh. If the connectivity is lost or interrupted during the refresh, then all changes are rolled back so that the database remains consistent. Therefore, in cases where the network connectivity is difficult to maintain, consider using smaller refresh groups.

Advanced Replication includes an optimization for null refresh. That is, if there were no changes to the master tables or master materialized views since the last refresh for a particular materialized view, then almost no extra time is required for the materialized view during materialized view group refresh. However, for materialized views in a database prior to release 8.1, consider separating materialized views of master tables that are not updated often into a separate refresh group of their own. Doing so shortens the refresh time required for other materialized view groups that contain materialized views of master tables that are updated frequently.

Table 3-3 summarizes the advantages of large and small refresh groups.

Table 3-3 Large and Small Refresh Groups

Advantages of Large Refresh Groups	Advantages of Small Refresh Groups
• Refreshes faster than an equal number of materialized views in multiple refresh groups	• Materialized views locked for shorter periods of time

• Refreshes with single replication management API call	• Rollback of refresh changes due to loss of connectivity is less likely

Refresh Process

A materialized view's data does not necessarily match the current data of its master table or master materialized view at all times. A materialized view is a transactionally (read) consistent reflection of its master as the data existed at a specific point in time (that is, at creation or when a refresh occurs). To keep a materialized view's data relatively current with the data of its master, the materialized view must be refreshed periodically. A **materialized view refresh** is an efficient batch operation that makes a materialized view reflect a more current state of its master table or master materialized view.

A refresh of an updatable materialized view first pushes the deferred transactions at the materialized view site to its master site or master materialized view site. Then, the data at the master site or master materialized view site is pulled down and applied to the materialized view.

A row in a master table may be updated many times between refreshes of a materialized view, but the refresh updates the row in the materialized view only once with the current data. For example, a row in a master table may be updated 10 times since the last refresh of a materialized view, but the result is still only one update of the corresponding row in the materialized view during the next refresh.

Decide how and when to refresh each materialized view to make it more current. For example, materialized views based on masters that applications update often may require frequent refreshes. In contrast, materialized views based on relatively static masters usually require infrequent refreshes. In summary, analyze application characteristics and requirements to determine appropriate materialized view refresh intervals.

To refresh materialized views, Oracle supports several refresh types and methods of initiating a refresh.

Refresh Types

Oracle can refresh a materialized view using either a fast, complete, or force refresh.

Complete Refresh

To perform a **complete refresh** of a materialized view, the server that manages the materialized view executes the materialized view's defining query, which essentially recreates the materialized view. To refresh the materialized view, the result set of the query replaces the existing materialized view data. Oracle can perform a complete refresh for any materialized view. Depending on the amount of data that satisfies the defining query, a complete refresh can take a substantially longer amount of time to perform than a fast refresh.

If you perform a complete refresh of a master materialized view, then the next refresh performed on any materialized views based on this master materialized view must be a complete refresh. If a fast refresh is attempted for such a materialized view after its master materialized view has performed a complete refresh, then Oracle returns the following error:

```
ORA-12034 mview log is younger than last refresh
```

Note:

If a materialized view is complete refreshed, then set its `PCTFREE` to `0` and `PCTUSED` to `99` for maximum efficiency.

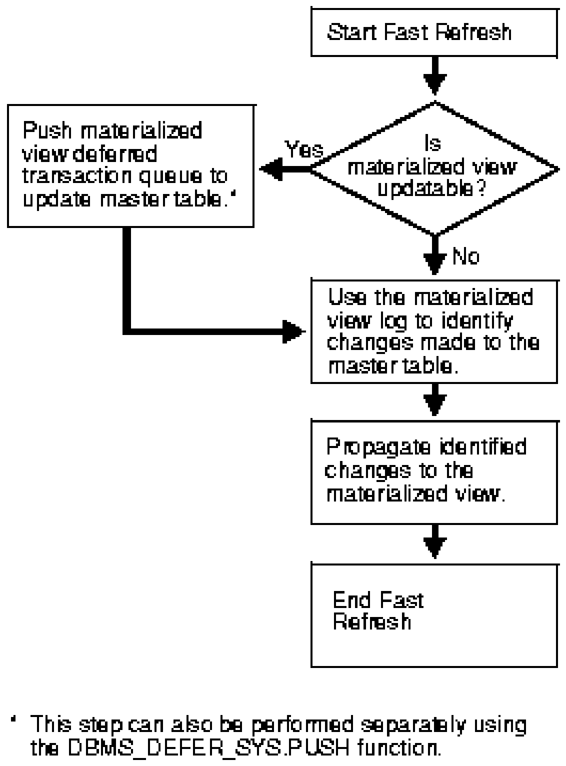
Fast Refresh

To perform a **fast refresh**, the master that manages the materialized view first identifies the changes that occurred in the master since the most recent refresh of the materialized view and then applies these changes to the materialized view. Fast refreshes are more efficient than complete refreshes when there are few changes to the master because the participating server and network replicate a smaller amount of data.

You can perform fast refreshes of materialized views only when the master table or master materialized view has a materialized view log. Also, for fast refreshes to be faster than complete refreshes, each join column in the `CREATE MATERIALIZED VIEW` statement must have an index on it.

After a direct path load on a master table or master materialized view using SQL*Loader, a fast refresh does not apply the changes that occurred during the direct path load. Also, fast refresh does not apply changes that result from other types of bulk load operations on masters. Examples of these operations include some `INSERT` statements with an `APPEND` hint and some `INSERT ... SELECT * FROM` statements.

Figure 3-14 Fast Refresh of a Materialized View


Start Fast Refresh
Is materialized view updatable?
Yes
No
Push materialized view deferred transaction queue to update master table.*
Use the materialized view log to identify changes made to the master table.
Propagate identified changes to the materialized view.
End Fast Refresh
* This step can also be performed separately using the DBMS_DEFER_SYS.PUSH function.

Text description of the illustration rep81077.gif

If you have updatable multitier materialized views, then DML changes made to the multitier materialized view may be pulled back to this materialized view multiple times to ensure data consistency after each refresh of a materialized view. This behavior is best illustrated through an example. Consider a replication environment with the following characteristics:

- Master site `orc1.world` has the `oe.customers` table.
- Level 1 materialized view site `ca.us` has the `oe.customers_region` updatable materialized view based on the `oe.customers` table at `orc1.world`.
- Level 2 updatable materialized view site `sf.ca` has the `oe.customers_sf` updatable materialized view based on the `oe.customers_region` materialized view at `ca.us`.

Given these characteristics, the following scenario may follow:

1. The `credit_limit` for a customer is changed from `3000` to `5000` in the `oe.customers_sf` updatable materialized view at `sf.ca`
2. Oracle enters the change in the deferred transaction queue at `sf.ca`.
3. A fast refresh of the level 2 materialized view `oe.customers_sf` pushes the new value for the `credit_limit` to `oe.customers_region` materialized view at `ca.us`.
4. The change is applied to the `oe.customers_region` materialized view at `ca.us`.
5. The update for the `credit_limit` at the `ca.us` site is recorded in both the deferred transaction queue and the materialized view log a this level 1 materialized view site.
6. A fast refresh of the level 2 materialized view `oe.customers_sf` pulls the `credit_limit` value of `5000` back down to this materialized view at `sf.ca`.
7. A fast refresh of the level 1 materialized view `oe.customers_region` pushes the new value for the `credit_limit` to `oe.customers` master table at `orc1.world`.
8. The change is applied to the `oe.customers` master table at `orc1.world`.
9. The update for the `credit_limit` at the `orc1.world` site is recorded in both the deferred transaction queue and the materialized view log a this master site.
10. A new fast refresh of the level 1 materialized view `oe.customers_region` pulls the `credit_limit` value of `5000` back down to this materialized view at `ca.us`.
11. The update for the `credit_limit` at the `ca.us` site is recorded in the materialized view log a this level 1 materialized view site.
12. A fast refresh of the level 2 materialized view `oe.customers_sf` pulls the `credit_limit` value of `5000` back down to this materialized view at `sf.ca` for a second time.

Force Refresh

To perform a **force refresh** of a materialized view, the server that manages the materialized view attempts to perform a fast refresh. If a fast refresh is not possible, then Oracle performs a complete refresh. Use the force setting when you want a materialized view to refresh if a fast refresh is not possible.

Initiating a Refresh

When creating a refresh group, you can configure the group so that Oracle automatically refreshes the group's materialized views at scheduled intervals. Conversely, you can omit scheduling information so that the refresh group needs to be refreshed manually or "on-demand." Manual refresh is an ideal solution when the refresh is performed with a dial-up network connection.

Scheduled Refresh

When you create a refresh group for automatic refreshing, you must specify a scheduled refresh interval for the group during the creation process. When setting a group's refresh interval, consider the following characteristics:

- The dates or date expressions specifying the refresh interval must evaluate to a future point in time.
- The refresh interval must be greater than the length of time necessary to perform a refresh.
- Relative date expressions evaluate to a point in time relative to the most recent refresh date. If a network or system failure interferes with a scheduled group refresh, then the evaluation of a relative date expression could change accordingly.
- Explicit date expressions evaluate to specific points in time, regardless of the most recent refresh date.
- Consider your environment's tolerance for stale data: if there is a low tolerance, then refresh often; whereas if there is a high tolerance, then refresh less often.

The following are examples of simple date expressions that you can use to specify an interval:

- An interval of one hour is specifies as:

  ```
  SYSDATE + 1/24
  ```

- An interval of seven days is specifies as:

  ```
  SYSDATE + 7
  ```

See Also:

Oracle9i Database Administrator's Guide and *Oracle9i SQL Reference* for more information about date arithmetic

On-Demand Refresh

Scheduled materialized view refreshes may not always be the appropriate solution for your environment. For example, immediately following a bulk data load into a master table, dependent materialized views no longer represent the master table's data. Rather than wait for the next scheduled automatic group refreshes, you can manually refresh dependent materialized view groups to immediately propagate the new rows of the master table to associated materialized views.

You may also want to refresh your materialized views on-demand when your materialized views are integrated with a sales force automation system located on a disconnected laptop. Developers designing the sales force automation software can create an application control, such as a button, that a salesperson can use to refresh the materialized views when they are ready to transfer the day's orders to the server after establishing a dial-up network connection.

The following example illustrates an on-demand refresh of the `hr_refg` refresh group:

```
execute DBMS_REFRESH.REFRESH('hr_refg');
```

Previous Next

Home Book List Contents Index Master Index Feedback