

## *Query Rewrite*

In Chapter 7, we described materialized views, which can be used to precompute and store results of frequently used queries. A major benefit of doing this is that you can use query rewrite to transparently rewrite queries to use these materialized views, thus significantly reducing the query response time. With this feature, queries that used to take hours to return results can now return them in minutes or even instantly.

As with indexes, materialized views and query rewrite should be considered an essential part of query tuning in a data warehouse. Just as the query optimizer considers all available indexes when determining the fastest way to answer the query, it also considers any available materialized views that may have already precomputed part or all of the answer to the query. This means that no application changes are needed in order to use materialized views. If the optimizer determines that the materialized view is insufficient to answer the query, it uses the detail data. Therefore, end users do not have to be aware of the existence of materialized views and hence they can be created and modified without impacting users.

In this chapter, we will describe various techniques used by Oracle to rewrite queries using materialized views.

### **9.1 Setting up Query Rewrite**

There are three steps that must be followed to enable queries to be rewritten to use materialized views.

- The materialized views must be created with the `ENABLE QUERY REWRITE` clause, discussed in Chapter 7.

- The initialization parameter `QUERY_REWRITE_ENABLED` must be set to `TRUE`. This is the default in Oracle Database 10g.

```
-- enable query rewrite
ALTER SESSION SET QUERY_REWRITE_ENABLED=TRUE;
```

- The initialization parameter `QUERY_REWRITE_INTEGRITY` must be set to an appropriate level for the application. This parameter indicates to the optimizer what type of metadata information (e.g., dimensions) may be used to rewrite queries. We will discuss this parameter in detail later in this chapter.

Query rewrite uses the cost-based optimizer, which automatically compares the cost of the query execution plan with and without query rewrite and uses the one with the lower cost. To ensure that the optimizer makes the correct choices, you need to collect statistics both on the detail tables involved in the query and on the materialized views using the `DBMS_STATS` package.

Occasionally, your application may require that the query must only use the materialized view and must never use the base tables. In such cases, you can override the optimizer's cost-based decision making by setting the `QUERY_REWRITE_ENABLED` parameter to `FORCE`. In this mode, if there is a materialized view that satisfies the query, the optimizer will use it without comparing the cost of the plan with and without rewrite. You can disable query rewrite by setting this parameter to `FALSE`.

### 9.1.1 How Can We Tell If a Query Was Rewritten?

To determine if the query was rewritten, we use the `EXPLAIN PLAN` utility (described in Chapter 6) to look at the query execution plan. Specifically, if the query was rewritten, the output of `EXPLAIN PLAN` will include the special operation, `MAT_VIEW REWRITE`, and the name of the materialized view used to rewrite the query. If you find that query rewrite is not occurring as expected, you should use the `DBMS_MVIEW.EXPLAIN_REWRITE`, utility discussed later in this chapter, to diagnose the problem.

---

## 9.2 Types of Query Rewrite

Oracle supports several types of query rewrite transformations, allowing a single materialized view to be used to answer several queries. We will illustrate several of these in the following sections, including:

- SQL text match
- Aggregate rollup
- Join-back
- Computing aggregates from other aggregates
- Filtered data
- Rewrite using dimensions and constraints

Our first few examples will use the following materialized view, which computes the sum of sales and total sales for products by month.

```
CREATE MATERIALIZED VIEW monthly_sales_mv
ENABLE QUERY REWRITE
AS
SELECT t.year, t.month, p.product_id,
       SUM (ps.purchase_price) as sum_of_sales,
       COUNT (ps.purchase_price) as total_sales
FROM time t, product p, purchases ps
WHERE t.time_key = ps.time_key AND
      ps.product_id = p.product_id
GROUP BY t.year, t.month, p.product_id;
```

### 9.2.1 SQL Text Match

The simplest type of query rewrite is when the SQL text of the materialized view's defining query exactly matches that of the incoming query. The text match is not case-sensitive and ignores any comments and whitespace differences.

The execution plan that follows shows that the optimizer chose to access the materialized view MONTHLY\_SALES\_MV via a full table scan.

```
-- exact text match
EXPLAIN PLAN FOR
SELECT t.year, t.month, p.product_id,
```

```

        sum (ps.purchase_price) as sum_of_sales,
        count (ps.purchase_price) as total_sales
FROM time t, product p, purchases ps
WHERE t.time_key = ps.time_key AND
      ps.product_id = p.product_id
GROUP BY t.year, t.month, p.product_id;

```

PLAN\_TABLE\_OUTPUT

```

-----
|Id| Operation                               |Name                               | Rows | Cost |
-----
| 0| SELECT STATEMENT                         |                                   | 3522 | 7    |
| 1| MAT_VIEW REWRITE ACCESS FULL|MONTHLY_SALES_MV | 3522 | 7    |
-----

```

Oracle will also try a text match starting from the FROM keyword of the query. This allows for differences in column ordering in the SELECT list and computation of expressions. In the following example, SUM(purchase\_price) and COUNT(purchase\_price) have been used to compute the average, and, also, the order of columns in the SELECT list is changed. You can see from the execution plan that the materialized view has been used to rewrite this query.

```

EXPLAIN PLAN FOR
SELECT t.month, p.product_id, t.year,
       AVG(ps.purchase_price) avg_of_sales
FROM time t, product p, purchases ps
WHERE t.time_key = ps.time_key AND
      ps.product_id = p.product_id
GROUP BY t.year, t.month, p.product_id;

```

PLAN\_TABLE\_OUTPUT

```

-----
|Id| Operation                               |Name                               | Rows | Cost |
-----
| 0| SELECT STATEMENT                         |                                   | 3522 | 7    |
| 1| MAT_VIEW REWRITE ACCESS FULL|MONTHLY_SALES_MV | 3522 | 7    |
-----

```

If the text of the query and materialized view does not match, Oracle will then compare the join conditions, GROUP BY clauses, and aggregates in the query and materialized view to determine if the query can be rewritten using the materialized view. We will illustrate these rules in the following sections.

### 9.2.2 Aggregate Rollup

An **aggregate rollup** occurs when the aggregates in the materialized view can be further aggregated to supply the aggregates requested by the query. A simple example of this is when the query contains only some of the grouping columns from the materialized view. For instance, the following query asks for the sum of sales and total number of sales by product and year. The materialized view contains the sum of sales by product by year and month. During query rewrite, the monthly sales are added together to compute the yearly totals.

```
-- rollup over month column
EXPLAIN PLAN FOR
SELECT t.year, p.product_id,
       SUM (ps.purchase_price) as sum_of_sales,
       COUNT (ps.purchase_price) as total_sales
FROM time t, product p, purchases ps
WHERE t.time_key = ps.time_key AND
      ps.product_id = p.product_id
GROUP BY t.year, p.product_id;
```

PLAN\_TABLE\_OUTPUT

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		232	9
1	SORT GROUP BY		232	9
2	MAT_VIEW REWRITE ACCESS FULL	MONTHLY_SALES_MV	3522	7

When a rollup occurs, the rewritten query will contain a GROUP BY clause, as shown in the previous execution plan.

A more interesting case of rollup is when your data has a hierarchy described by a dimension object. In this case, query rewrite can roll up data from a lower level to a higher level in the hierarchy. We will explain this in more detail in section 9.2.7.

### 9.2.3 Join-back

For query rewrite to occur, all the columns in the query must either appear in the materialized view or must be derivable from some column in the materialized view. In the latter case, the materialized view must be joined to the base table to obtain the required column. This is called a **join-back**. In

the simple case, for a join-back to occur, the materialized view must contain either the primary key or the rowid of the detail table.

For instance, suppose, in addition to the sum of sales by product id, we would also like to see the product name. The column `PRODUCT_ID` is the primary key of the `PRODUCT` table; therefore, the following query, asking for `PRODUCT_NAME`, can be answered using the `MONTHLY_SALES_MV` materialized view and using a join-back. The optimizer's plan shows that the query has been rewritten to use the materialized view `MONTHLY_SALES_MV` with a join to the `PRODUCT` table. The predicate information printed by `EXPLAIN PLAN` can be used to see that the join-back is done using the `PRODUCT_ID` column in the materialized view.

```
-- join-back to product table using primary key constraint
EXPLAIN PLAN FOR
SELECT t.year, t.month, p.product_name,
       SUM (ps.purchase_price) as sum_of_sales
FROM time t, product p, purchases ps
WHERE t.time_key = ps.time_key AND
      ps.product_id = p.product_id
GROUP BY t.year, t.month, p.product_name;
```

PLAN\_TABLE\_OUTPUT

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		3522	4
1	SORT GROUP BY		3522	4
*2	HASH JOIN		3522	1
3	TABLE ACCESS FULL	PRODUCT	164	
4	MAT_VIEW REWRITE ACCESS FULL	MONTHLY_SALES_MV	3522	

Predicate Information (identified by operation id):

```
2 - access("P"."PRODUCT_ID"="MONTHLY_SALES_MV"."PRODUCT_ID")
```

The advantage of using the materialized view with a join-back, even though there is an extra join, is that this join is likely to be based on a smaller number of records and so will be usually much faster than using the detail data.

### 9.2.4 Computing Other Aggregates in the Query

Aggregates in the query can be computed from different aggregates in the materialized view. We have already seen a simple example of this in the SQL text match section, where SUM and COUNT were used to compute the AVG. However, the power of query rewrite comes from the fact that many different transformations can be combined together. For instance, in the following query, we want to know the average purchase price of each item by year. The materialized view has the sum and count of the purchase price at the monthly level. The average can be computed by first doing a rollup of months to years and then dividing the sum by the count of the purchase price. The query is therefore rewritten to use the MONTHLY\_SALES\_MV materialized view.

```
-- aggregate computability
EXPLAIN PLAN FOR
SELECT t.year, p.product_id, AVG(ps.purchase_price) as ave_sales
FROM time t, product p, purchases ps
WHERE t.time_key = ps.time_key AND
ps.product_id = p.product_id
GROUP BY t.year, p.product_id;
```

PLAN\_TABLE\_OUTPUT

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		232	9
1	SORT GROUP BY		232	9
2	MAT_VIEW REWRITE ACCESS FULL	MONTHLY_SALES_MV	3522	7

To compute aggregates in the query with a rollup, the materialized view may need to contain additional aggregates. For instance, to roll up AVG, the materialized view must have SUM and COUNT or AVG and COUNT.

### 9.2.5 Filtered Data

The materialized view MONTHLY\_SALES\_MV defined previously contained data for all products for each month and year. Sometimes you may only want to summarize data for a certain product or year or have separate materialized views for each region. In this case, the materialized view will only contain a subset of data indicated by a selection condition in the WHERE clause of its query. Sophisticated query rewrites are possible with

one or more such materialized views. Oracle will determine if the data in the query can be answered by a materialized view by analyzing and comparing the WHERE clauses of the materialized view and the query.

The following materialized view contains sum of sales and the total number of sales for the electronics category for the months from January 2003 through June 2003.

```
CREATE MATERIALIZED VIEW sales_elec_1_6_2003_mv
ENABLE QUERY REWRITE
AS
SELECT t.month, t.year, p.product_id,
       SUM(ps.purchase_price) as sum_of_sales,
       COUNT (ps.purchase_price) as total_sales
FROM time t, product p, purchases ps
WHERE t.time_key = ps.time_key AND
      ps.product_id = p.product_id AND
      p.category = 'ELEC' AND
      t.month >= 200301 AND t.month <= 200306
GROUP BY t.month, t.year, p.product_id;
```

This materialized view can be used to answer the following query, which requests the sum of sales and number of sales for the Electronics category for May 2003. The predicate information section, which is output by EXPLAIN PLAN shows the predicates applied during each step of the execution plan. From this, we can see that the MV data is filtered to select only the row for May 2003.

```
EXPLAIN PLAN FOR
SELECT t.month, p.product_id,
       SUM(ps.purchase_price) as sum_of_sales,
       COUNT (ps.purchase_price) as total_sales
FROM time t, product p, purchases ps
WHERE t.time_key = ps.time_key AND
      ps.product_id = p.product_id AND
      p.category = 'ELEC' AND t.month = 200305
GROUP BY t.month, t.year, p.product_id;
```

PLAN\_TABLE\_OUTPUT

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		100	2
*1	MAT_VIEW REWRITE ACCESS FULL	SALES_ELEC_1_6_2003_MV	100	2

Predicate Information (identified by operation id):

```
1 - filter("SALES_ELEC_1_6_2003_MV"."MONTH"=200305)
```

A query can have additional conditions not mentioned in the materialized view. For instance, in the following query, we are looking for monthly sales of digital camera products in the electronics category for Jan 2003. The materialized view has all products within this category and we can determine the PRODUCT\_NAME from PRODUCT\_ID using a join-back. Hence, the query will be rewritten to use the materialized view with a join to the product table.

```
EXPLAIN PLAN FOR
SELECT t.month, SUM(ps.purchase_price) as sum_of_sales,
       COUNT (ps.purchase_price) as total_sales
FROM time t, product p, purchases ps
WHERE t.time_key = ps.time_key AND
      ps.product_id = p.product_id AND p.category = 'ELEC' AND
      t.month = 200301 AND product_name = 'Digital Camera'
GROUP BY t.month;
```

PLAN\_TABLE\_OUTPUT

Id	Operation	Name	Row	Cst
0	SELECT STATEMENT		1	5
1	SORT GROUP BY NOSORT		1	5
*2	HASH JOIN		1	5
*3	TABLE ACCESS FULL	PRODUCT	1	2
*4	MAT_VIEW REWRITE ACCESS FULL	SALES_ELEC_1_6_2003_MV	96	2

Predicate Information (identified by operation id):

```
-----
2 -
access("P"."PRODUCT_ID"="SALES_ELEC_1_6_2003_MV"."PRODUCT_ID")
3 - filter("PRODUCT_NAME"='Digital Camera')
4 - filter("SALES_ELEC_1_6_2003_MV"."MONTH"=200301)
```

This example illustrates how two rewrite mechanisms can be applied together—namely, join-back and filtered data.

### 9.2.6 Rewrite Using Materialized Views with No Aggregation

The examples in the previous sections all involve materialized views with aggregation. Materialized views are sometimes used to precompute expensive joins and may not involve any aggregation. Query rewrite can use

such materialized views to rewrite queries, which may or may not contain aggregation.

Consider the following materialized view, which stores the information about purchases made by customers, including the customer gender and occupation. This materialized view has a join between the CUSTOMER and PURCHASES table but no aggregation.

```
CREATE MATERIALIZED VIEW customer_purchases_mv
ENABLE QUERY REWRITE
AS
SELECT c.gender, c.occupation, f.purchase_price
FROM purchases f, customer c
WHERE f.customer_id = c.customer_id;
```

The following query, which asks for the purchases made by doctors, can be answered using this materialized view.

```
EXPLAIN PLAN FOR
SELECT c.gender, f.purchase_price
FROM purchases f, customer c
WHERE f.customer_id = c.customer_id
      AND c.occupation = 'Doctor';
```

PLAN\_TABLE\_OUTPUT

Id	Operation	Name	Cost
0	SELECT STATEMENT		54
1	MAT_VIEW REWRITE ACCESS FULL	CUSTOMER_PURCHASES_MV	

Prdicate Information (identified by operation id):

```
1 - filter("CUSTOMER_PURCHASES_MV"."OCCUPATION"='Doctor')
```

The same materialized view can also be used to answer the following query, which asks for the total purchases made by women.

```
EXPLAIN PLAN FOR
SELECT c.occupation, SUM(f.purchase_price)
FROM purchases f, customer c
WHERE f.customer_id = c.customer_id
      AND c.gender = 'F'
GROUP BY c.occupation;
```

```

PLAN_TABLE_OUTPUT
-----
| Id | Operation                               | Name                               | Cost |
-----
| 0 | SELECT STATEMENT                         |                                     | 62   |
| 1 |  SORT GROUP BY                           |                                     | 62   |
| 2 |    MAT_VIEW REWRITE ACCESS FULL|CUSTOMER_PURCHASES_MV|53   |
-----

```

```

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

```

```

 2 - filter("CUSTOMER_PURCHASES_MV"."GENDER"='F')

```

Note that even though the materialized view does not contain aggregation, it can still be used to answer a query with aggregation.

All the query rewrites we have seen so far have not required any additional information from the user. However, to get the most out of query rewrite, you must inform query rewrite about relationships between data columns using constraints and dimensions.

### 9.2.7 Rewrite Using Dimensions

One of the powerful features of query rewrite is the ability for a single materialized view to be used to satisfy a wide range of queries. The Dimension object, discussed in Chapter 8, is extremely useful in this respect. By allowing you to declare relationships within columns of dimension tables, it provides query rewrite with information to roll up from a lower to a higher level of a hierarchy. For example, suppose your users want to know the sum of sales by day, month, or year. You could create three materialized views to answer these queries, or you could create one at the level of day and then define a dimension object that contains a hierarchy to show the relationship between time, month and year. Now, when the query asks for data at the month level, the materialized view at the daily level will be used to roll up the data to the monthly level.

#### Using the *HIERARCHY* Clause

Consider the following definition for a TIME dimension. The HIERARCHY clause tells query rewrite that the TIME\_KEY rolls up into WEEK\_NUMBER, which, in turns, rolls up into QUARTER. This means that if we knew the TIME\_KEY value for some row in the TIME table, we know which week it belonged to. Similarly, if we knew the MONTH (say January 2004), we know which YEAR it belonged to: 2004.

```

CREATE DIMENSION time
  LEVEL time_key          is time.time_key
  LEVEL month             is time.month
  LEVEL quarter           is time.quarter
  LEVEL year              is time.year
  LEVEL week_number      is time.week_number
  HIERARCHY fiscal_rollup (
    time_key              CHILD OF
    week_number           CHILD OF
    quarter )
  HIERARCHY calendar_rollup(
    time_key              CHILD OF
    month                 CHILD OF
    year);

```

Suppose our materialized view MONTHLY\_SALES\_MV was defined to report the total sales by product and month, as shown in the following code. Note that we have **not** included the year column in the materialized view.

```

CREATE MATERIALIZED VIEW monthly_sales_mv
  ENABLE QUERY REWRITE
  AS
  SELECT t.month, p.product_id,
         SUM(ps.purchase_price) as sum_of_sales,
         COUNT (ps.purchase_price) as total_sales
  FROM time t, product p, purchases ps
  WHERE t.time_key = ps.time_key AND
         ps.product_id = p.product_id
  GROUP BY t.month, p.product_id;

```

In the following query, we want to know the total sales by product by year. Since we have a materialized view with the total sales by product by month, and months can be rolled up into years, as specified in the calendar\_rollup hierarchy in the time dimension, the optimizer will rewrite the query to use the materialized view, MONTHLY\_SALES\_MV. Note that in order to determine the YEAR value for the MONTH, a join-back is done from the materialized view to the TIME table.

```

-- rollup to higher LEVEL in the HIERARCHY
EXPLAIN PLAN FOR
SELECT t.year, p.product_id,
       COUNT (ps.purchase_price) as total_sales
FROM time t, product p, purchases ps
WHERE t.time_key = ps.time_key AND
       ps.product_id = p.product_id

```

```
GROUP BY t.year, p.product_id;
PLAN_TABLE_OUTPUT
```

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		2	11
1	SORT GROUP BY		2	11
*2	HASH JOIN		4990	9
3	VIEW		34	4
4	SORT UNIQUE		34	4
5	TABLE ACCESS FULL	TIME	731	3
6	MAT_VIEW REWRITE ACCESS FULL	MONTHLY_SALES_MV	3522	4

### Using the ATTRIBUTE Clause

When defining a data warehouse, the dimension object is often overlooked, because its value to query rewrite is not fully appreciated. However, a dimension object gives you tremendous query rewrite power at no extra cost. We have already seen how query rewrite can take advantage of the HIERARCHY clause to rewrite several queries with one materialized view. Query rewrite can also make use of the ATTRIBUTE clause of dimension. In the following example, we want to know the sum of sales by customer based on gender and occupation.

```
SELECT c.gender, c.occupation,
       SUM(ps.purchase_price) as sum_of_sales
FROM purchases ps, customer c
WHERE c.customer_id = ps.customer_id
GROUP BY c.gender, c.occupation;
```

We could have put the columns GENDER and OCCUPATION into a materialized view. But we know that given the CUSTOMER\_ID, we can find information such as the customer's name, gender, and occupation. Such relationships within a table that are not hierarchical in nature are defined by the ATTRIBUTE clause in a dimension.

Suppose we have the following dimension, which defines the relationships within the customer table.

```
CREATE DIMENSION customer_dim
  LEVEL customer IS customer.customer_id
  LEVEL city     IS customer.city
  LEVEL state    IS customer.state
HIERARCHY customer_zone
  ( customer CHILD OF
```

```

    city      CHILD OF
    state )
ATTRIBUTE customer DETERMINES (customer.gender,
                                customer.occupation);

```

Now that we have this dimension object, we only need to include the CUSTOMER\_ID in the materialized view.

```

CREATE MATERIALIZED VIEW cust_sales_mv
ENABLE QUERY REWRITE
AS
SELECT c.customer_id, SUM(ps.purchase_price) as sum_of_sales,
       COUNT (ps.purchase_price) as total_sales
FROM customer c, purchases ps
WHERE c.customer_id = ps.customer_id
GROUP BY c.customer_id;

```

The execution plan of the query shows that the query was rewritten to use the materialized view. Note that a join-back was done to the customer table to retrieve the values of the OCCUPATION and GENDER columns.

PLAN\_TABLE\_OUTPUT

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		9	7
1	SORT GROUP BY		9	7
*2	HASH JOIN		500	6
3	MAT_VIEW REWRITE ACCESS FULL	CUST_SALES_MV	500	2
4	TABLE ACCESS FULL	CUSTOMER	500	3

When designing your data warehouse, you should try to identify relationships between your dimension tables and define dimensions, wherever possible. This will lead to significant space savings and increase query rewrite opportunities, thereby improving your query performance.

Recall that the relationships declared by dimensions are not validated automatically by Oracle. Hence, to take advantage of dimensions, you must set the QUERY\_REWRITE\_INTEGRITY parameter to TRUSTED or STALE\_TOLERATED, as we will discuss shortly in section 9.3.

### 9.2.8 Rewrite Using Constraints

In a data warehouse, constraints may be used to define the join relationships between the fact and dimension tables. Typically, a primary-key constraint is defined on the unique key column on each dimension table. A foreign-key constraint and a NOT NULL constraint are defined on each corresponding key in the fact table. For example, The EASYDW schema has primary-key constraints on each of the dimension tables: CUSTOMER, PRODUCT, and TIME. Also, there are foreign key and NOT NULL constraints on the foreign-key columns of the PURCHASES table that join to these dimension tables.

The relationship defined by these constraints indicates to query rewrite that a join between the PURCHASES table and, for example, the TIME table will produce exactly one row for every row in the PURCHASES table. Rows from the PURCHASES table cannot be lost during the join, because the NOT NULL and foreign key constraints mean that there **must** be a parent TIME record for every row in the PURCHASES table. Also, because of the primary-key constraint on TIME, each row in PURCHASES will join to a single parent TIME record and so no rows can be duplicated. Such a join is known as a **loss-less join**, because no rows in the PURCHASES table will be lost or duplicated by the join process.

The benefit of a loss-less join is that if a materialized view has more joins than the query, but the *extra* joins in the materialized view are *loss-less* joins, then the query can be rewritten using the materialized view. For instance, in the following example, the query does not have the TIME table. However, we can still rewrite the query with the MONTHLY\_SALES\_MV materialized view (which has tables PURCHASES, PRODUCT, and TIME). This is because the extra join in the materialized view, between tables PURCHASES and TIME, is a loss-less join.

```
EXPLAIN PLAN FOR
SELECT p.product_id, SUM(ps.purchase_price) as sum_of_sales
FROM product p, purchases ps
WHERE ps.product_id = p.product_id
GROUP BY p.product_id;
```

PLAN\_TABLE\_OUTPUT

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		164	6
1	SORT GROUP BY		164	6
2	MAT_VIEW REWRITE ACCESS FULL	MONTHLY_SALES_MV	3522	4

### **Using the NOVALIDATE and RELY Clauses on Constraints**

As described in Chapter 2, when a constraint is enabled, you can choose to have Oracle validate the integrity of the data. If you are concerned about the overhead of maintaining the constraint, or if you have already validated the data during the ETL process, you could use the NOVALIDATE clause to tell Oracle that the data has already been validated.

```
ALTER TABLE purchases ENABLE NOVALIDATE CONSTRAINT fk_customer_id;
```

An additional RELY clause should be used to tell Oracle that it can rely on the constraint being correct and can use it in query rewrite even when the constraint has not been validated. It allows the Database Administrator (DBA) to say: “Trust me. I’ve already checked the data validity. Query rewrite can rely on the relationship being correct.”

```
ALTER TABLE purchases MODIFY CONSTRAINT fk_customer_id RELY;
```

---

---

**Hint:** Use constraints to define the relationship between your fact and dimension tables. Use the dimension object to declare the relationships within your dimension tables, such as a time or a region hierarchy.

---

---

As with dimensions, in order to use RELY constraints, you must set the QUERY\_REWRITE\_INTEGRITY parameter to TRUSTED. The next section explains this parameter in detail.

## **9.3 Query Rewrite Integrity Modes**

We have mentioned the QUERY\_REWRITE\_INTEGRITY parameter several times in our examples. This parameter indicates to Oracle the extent to which the rewritten queries must reflect the data in the detail tables and what metadata can be used to rewrite queries. This parameter can take three values:

- **ENFORCED:** In this mode (which is the default), Oracle will guarantee that the rewritten query will return the same results as the original query when executed without query rewrite.

- **TRUSTED:** In this mode, Oracle will use data and relationships that have been “blessed” by the DBA—namely, dimension objects, RELY constraints, and materialized views created from PREBUILT tables. Oracle does not validate that the relationships declared by the dimension are indeed valid in the data or that a prebuilt table is the same as the materialized view’s query.
- **STALE\_TOLERATED:** In this mode, Oracle will use stale materialized views, which may not contain the very latest data, because they have not yet been refreshed. This is appropriate if the business users do not need to have the most up-to-date data in order to perform their analyses.

You must decide what query rewrite integrity level is appropriate for your application. In the following sections, we will discuss the differences between the three modes and the motivation behind using one versus the other.

### 9.3.1 Comparing ENFORCED and TRUSTED Modes

Setting the parameter to ENFORCED guarantees that you will see the same results from using the materialized view or querying the detail tables. You are probably thinking that this is the best mode to use! However, the problem is that this mode also requires that all defined relationships, such as constraints, be validated. This can be a huge overhead in a data warehouse. Another issue with the ENFORCED mode is that dimension objects are not used. This greatly limits the power of query rewrite, and you may need a large number of materialized views to answer all your queries. For example, the query rewrites with dimensions shown in section 9.2.7 will never occur in ENFORCED mode. Also, you cannot use materialized views defined using the PREBUILT TABLE clause, unless they have been completely rebuilt.

Unlike ENFORCED mode, in TRUSTED mode constraints are used even though they are not validated, provided they have the RELY clause and dimensions are also considered. Therefore, in a warehouse you will more likely use the TRUSTED mode.

A note of caution: Query rewrite in TRUSTED mode depends on the integrity of your dimension and constraint definitions. Does each product in the product table roll up to one and only one category, as specified in your dimension definition? Does each product in the PURCHASES table

have a corresponding `product_id` in the `products` table, as specified by your referential integrity constraints? If your data does not reflect the relationships defined by the constraint or dimension, then you may get unexpected results. The same holds for materialized views on prebuilt tables: If the prebuilt table does not reflect the materialized view's query accurately, then results can be unexpected.

The last mode is `STALE_TOLERATED`, and it is even more relaxed than the `TRUSTED` mode, as discussed in the next section

### 9.3.2 Comparing `TRUSTED` and `STALE_TOLERATED` Modes

The `STALE_TOLERATED` mode also allows use of trusted relationships like the `TRUSTED` mode. However, the key difference with the `STALE_TOLERATED` mode is that it allows use of materialized views even if they are stale. In both `TRUSTED` and `ENFORCED` modes, the optimizer will use the detail table if necessary but will never return stale data.

---

---

**Hint:** You can determine if a materialized view is `FRESH` or `STALE` by using the `STALENESS` column of the catalog view `USER_MVIEWS`.

---

---

Most of the time, you would like to get the result the fastest way possible, rewriting your queries to use materialized views. However, if your materialized views have become stale and no longer represent the summarization of all your detail data, depending on your application, you may prefer to get the results from the detail tables until you can perform your next refresh. If so, then the `TRUSTED` mode is the right choice.

On the other hand, if the results obtained from a materialized view are “close enough” for your application, you may want to use the materialized view even if it is stale. For example, to determine the month-over-month growth rate of on-line sales, you do not need every single sales transaction in the materialized view. As long as the data is reasonably recent, you could still get an answer that was close enough. Or, if the application knew that the missing data was beyond the scope of the query, it may still want to use the materialized view. For instance, if the missing data is for the last month but your query does not need it, you can use the materialized view. Or, it may be appropriate to use the materialized view when the fact table is stale, but not when a dimension is updated. The decision to use stale data or not should be made after consulting business users who would be using the data for analysis.

---

If you would like the optimizer to use the materialized view even if it is stale, set the `QUERY_REWRITE_INTEGRITY` parameter to `STALE_TOLERATED`.

---

**Hint:** When first testing a materialized view to see if query rewrite will occur, set `QUERY_REWRITE_INTEGRITY` to `STALE_TOLERATED`, because if the query does not rewrite in this mode, it will not rewrite in any other mode. Once you know it works, you can try setting the parameter mode to your desired level.

---

The following example shows the difference between the two integrity modes `STALE_TOLERATED` and `TRUSTED` (or `ENFORCED`) with regard to stale data. Suppose we introduced a new product code, `SP1300` and inserted two new rows into the `purchases` fact table corresponding to it.

```
INSERT INTO product VALUES ('SP1300', 'XYZ', 'ELEC', '75.0',
                             '100.0', 15, 4.50, 'ABC', 'UVW');
COMMIT;

INSERT INTO purchases VALUES ('SP1300', '1-FEB-2003',
                              'AB123456', '1-FEB-2003', 28.01, 4.50, 'Y');

INSERT INTO purchases VALUES ('SP1300', '2-FEB-2003',
                              'AB123457', '1-FEB-2003', 28.01, 4.50, 'Y');
COMMIT;
```

The `MONTHLY_SALES_MV` materialized view is now stale, which means all the data in the detail table is not reflected in the materialized view.

```
SELECT staleness FROM user_mviews
WHERE mview_name = 'MONTHLY_SALES_MV';

STALENESS
-----
STALE
```

Suppose we had the following query, which requests sales by month for product `SP1300`. If you set `QUERY_REWRITE_INTEGRITY` to `STALE_TOLERATED`, then we see that no rows are returned in the result. This is because the materialized view was created before the new rows were inserted and so the data about `SP1300` is not in the materialized view.

```

ALTER SESSION SET QUERY_REWRITE_INTEGRITY=STALE_TOLERATED;

SELECT t.month, p.product_id,
       SUM(ps.purchase_price) as sum_of_sales,
       COUNT(ps.purchase_price) as total_sales
FROM time t, product p, purchases ps
WHERE t.time_key = ps.time_key AND
      ps.product_id = p.product_id
GROUP BY t.month, p.product_id
HAVING p.product_id = 'SP1300';

no rows selected

```

The execution plan shows that the materialized view was indeed used for this query.

PLAN\_TABLE\_OUTPUT

```

-----
|Id|Operation                               | Name                | Rows | Cost |
-----
| 0 | SELECT STATEMENT                       |                     |    21 |    4 |
| 1 | MAT_VIEW REWRITE ACCESS FULL| MONTHLY_SALES_MV   |    21 |    4 |
-----

```

On the other hand, if you set the QUERY\_REWRITE\_INTEGRITY to TRUSTED, Oracle will use the detail tables, PURCHASES, PRODUCT, and TIME, rather than the materialized view and the sales numbers include the new rows we just inserted.

```

ALTER SESSION SET QUERY_REWRITE_INTEGRITY=TRUSTED;

SELECT t.month, p.product_id,
       SUM(ps.purchase_price) as sum_of_sales,
       COUNT(ps.purchase_price) as total_sales
FROM time t, product p, purchases ps
WHERE t.time_key = ps.time_key AND
      ps.product_id = p.product_id
GROUP BY t.month, p.product_id
HAVING p.product_id = 'SP1300';

      MONTH PRODUCT_ SUM_OF_SALES TOTAL_SALES
-----
      200302 SP1300          56.02          2

```

The execution plan indicates that the materialized view was **not** used for this query.

PLAN\_TABLE\_OUTPUT

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		1	143
1	FILTER			
2	SORT GROUP BY		1	143
3	HASH JOIN		81169	121
4	TABLE ACCESS FULL	TIME	731	3
5	HASH JOIN		81169	115
6	INDEX FULL SCAN	PRODUCT_PK_INDEX	164	1
7	PARTITION RANGE ALL		81169	111
8	TABLE ACCESS FULL	PURCHASES	81169	111

To summarize, in a data warehouse it is recommended to use either TRUSTED or STALE\_TOLERATED modes. Use the TRUSTED mode if your applications require up-to-date data at all times. If you can tolerate a materialized view that does not contain all the latest data, use the STALE\_TOLERATED mode instead, to get the most benefit. Note that when using either of these modes, you must ensure that all the data satisfies the relationships declared in the RELY constraints and dimensions; otherwise, you may get unexpected results.

## 9.4 Query Rewrite and Partition Change Tracking

In Chapter 7, we discussed the Partition Change Tracking (PCT) feature, which allows materialized views to be fast refreshed after partition maintenance operations. PCT also increases the query rewrite capabilities of the materialized view. As discussed in the previous section, when a detail table is updated, the materialized view becomes stale and cannot be used by query rewrite in ENFORCED or TRUSTED integrity levels. However, if the detail table is partitioned and the materialized view supports PCT on that table, Oracle can determine which portion of the materialized view is fresh and which is not. Now, if a query can be answered by using only the fresh portion of the materialized view, query rewrite will use the materialized view. For example, in Figure 7.7, in Chapter 7, the data for Feb 2002 was updated, and a new partition with data for Apr 2002 was added. The fresh portion of the materialized view corresponds to the Jan 2002 and Mar 2002 data. If a query only required data for these partitions, the material-

ized view can be used. The materialized view cannot be used for Feb 2002 (updated partition) or Apr 2002 (new partition).

Query rewrite is supported when PCT is enabled using either the partition key or partition marker techniques described in Chapter 7. Query rewrite currently does not take advantage of PCT using the join dependency technique.

### 9.4.1 Query Rewrite with PCT Using Partition Key

Consider the following materialized view containing sales data for products. The PURCHASES table is partitioned by TIME\_KEY, which is included in the materialized view.

```
CREATE MATERIALIZED VIEW product_category_sales_mv
  ENABLE QUERY REWRITE
  AS
  SELECT ps.time_key, p.category,
         SUM(ps.purchase_price) as sum_of_sales
  FROM   product p, purchases ps
  WHERE  ps.product_id = p.product_id
  GROUP BY ps.time_key, p.category;
```

If we query the view USER\_MVIEWS, we will see that the materialized view is FRESH.

```
SELECT staleness FROM user_mvviews
WHERE mview_name = 'PRODUCT_CATEGORY_SALES_MV';
```

```
STALENESS
-----
FRESH
```

The PURCHASES table has data through Dec 2004, so the materialized view only contains data through Dec 2004. Now, suppose we added a new partition to the PURCHASES table and loaded data for Jan 2005.

```
ALTER TABLE purchases ADD PARTITION purchases_jan2005
  values less than (TO_DATE('01-02-2005', 'DD-MM-YYYY'));

INSERT INTO purchases VALUES ( 'SP1063', '2-JAN-2005',
  'AB123457', '7-JAN-2005', 28.01, 4.50, 'N');

INSERT INTO purchases VALUES ( 'SP1064', '2-JAN-2005',
  'AB123457', '8-JAN-2005', 28.01, 4.50, 'N');

COMMIT;
```

If we query the view `user_mvviews` now, we will see that the materialized view is `STALE`.

```
SELECT staleness FROM user_mvviews
WHERE mview_name = 'PRODUCT_CATEGORY_SALES_MV';
```

```
STALENESS
-----
STALE
```

Now consider the following query, which asks for the sum of sales for the last quarter of 2004: October through December 2004. The optimizer will determine that the query only needs to access partitions for Oct, Nov, and Dec 2004 of the `PURCHASES` table. Since the materialized view is enabled for partition change tracking for this table, Oracle will track that the materialized view is fresh with respect to these partitions. Hence, it can rewrite with the materialized view, as shown in the following execution plan.

```
EXPLAIN PLAN FOR
SELECT ps.time_key, p.category,
       SUM(ps.purchase_price) as sum_of_sales
FROM   product p, purchases ps
WHERE  ps.product_id = p.product_id and
       ps.time_key BETWEEN TO_DATE('01-10-2004', 'DD-MM-YYYY')
                          AND TO_DATE('31-12-2004', 'DD-MM-YYYY')
GROUP BY ps.time_key, p.category;
```

```
PLAN_TABLE_OUTPUT
```

```
-----
|Id|Operation                                |Name                                |Cost|
-----
| 0|SELECT STATEMENT                          |                                     |    3|
| 1|MAT_VIEW REWRITE ACCESS FULL|PRODUCT_CATEGORY_SALES_MV|    3|
-----
```

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

```
-----
1 - filter("PRODUCT_CATEGORY_SALES_MV"."TIME_KEY">=
           TO_DATE('2004-10-01 00:00:00','yyyy-mm-ddhh24:mi:ss')
           AND "PRODUCT_CATEGORY_SALES_MV"."TIME_KEY"<=
           TO_DATE('2004-12-31 00:00:00','yyyy-mm-dd hh24:mi:ss'))
```

On the other hand, a query that requests the sum of sales for Oct 2004 through Mar 2005 cannot be answered using the materialized view.

### 9.4.2 Query Rewrite Using PCT with Partition Marker

Query rewrite can also take advantage of partition change tracking using the partition marker. The following query illustrates how rewrite can be used if the `PRODUCT_CATEGORY_SALES_MV` had a partition marker for the `PURCHASES` table instead of the partition key (`TIME_KEY`).

```
CREATE MATERIALIZED VIEW product_category_sales_mv
ENABLE QUERY REWRITE
AS
SELECT DBMS_MVIEW.PMARKER(ps.rowid) pmarker, p.category,
       SUM(ps.purchase_price) as sum_of_sales
FROM   product p, purchases ps
WHERE  ps.product_id = p.product_id
GROUP BY DBMS_MVIEW.PMARKER(ps.rowid), p.category;
```

The following query asks for data for the last quarter of 2004. Note that for rewrite to work, the bounds specified by the filter condition must match exactly with partition boundaries.

```
EXPLAIN PLAN FOR
SELECT p.category,
       SUM(ps.purchase_price) as sum_of_sales
FROM   product p, purchases ps
WHERE  ps.product_id = p.product_id and
       ps.time_key >= TO_DATE('01-10-2004', 'DD-MM-YYYY')
       AND ps.time_key < TO_DATE('01-01-2005', 'DD-MM-YYYY')
GROUP BY p.category;
```

PLAN\_TABLE\_OUTPUT

Id	Operation	Name	Cost
0	SELECT STATEMENT		3
1	SORT GROUP BY		3
*2	MAT_VIEW REWRITE ACCESS FULL	PRODUCT_CATEGORY_SALES_MV	2

Predicate Information (identified by operation id):

```
2 - filter("PRODUCT_CATEGORY_SALES_MV"."PMARKER"=52520 OR
          "PRODUCT_CATEGORY_SALES_MV"."PMARKER"=52521 OR
          "PRODUCT_CATEGORY_SALES_MV"."PMARKER"=52522)
```

As we can see, partition change tracking is a very useful technique—not only to speed up refresh of your materialized views but also to improve the ability of the optimizer to rewrite queries with those materialized views.

Oracle query rewrite is a very powerful feature but with this power comes some complexity. The next section explains how you can identify and fix common problems in query rewrite.

## 9.5 Troubleshooting Query Rewrite with EXPLAIN\_REWRITE

In the examples in this chapter, we have used EXPLAIN PLAN to see if a query was rewritten to use a materialized view. However, sometimes you may find that the query did not rewrite with the materialized view as you had expected. In some cases, the reason is extremely trivial, such as the parameter QUERY\_REWRITE\_ENABLED not being set to TRUE. In other cases, the reason could be more subtle, such as a constraint that was not present or validated or some column required by the query not being present in the materialized view. The rules governing query rewrite can be extremely complex and the reasons for not using a materialized view may not be obvious. To diagnose the reasons for such missed rewrites, you should use the PL/SQL procedure DBMS\_MVIEW.EXPLAIN\_REWRITE.

To use EXPLAIN\_REWRITE, you provide the query and, optionally, the materialized view it is supposed to rewrite with. The procedure will tell you if the query will use that materialized view and, if not, then the reason for not doing the rewrite. Prior to using the procedure, you must create a table named REWRITE\_TABLE in your schema, using the script utlxrw.sql in the rdbms/admin directory. The results of EXPLAIN\_REWRITE are placed in this table. There is also a varray interface, which allows you to access the results through a PL/SQL program instead.

We will now illustrate how to use this utility. In the first example, the user forgot to set the QUERY\_REWRITE\_ENABLED parameter to TRUE. To diagnose the problem you issue EXPLAIN\_REWRITE and select the results from the REWRITE\_TABLE. The message column in the REWRITE\_TABLE indicates the reason why query rewrite did not happen with the materialized view, specified in the MV\_NAME column.

```

BEGIN
dbms_mview.explain_rewrite('
SELECT t.month, t.year, p.product_id,
       SUM (ps.purchase_price) as sum_of_sales,
       COUNT (ps.purchase_price) as total_sales
FROM time t, product p, purchases ps
WHERE t.time_key = ps.time_key AND
ps.product_id = p.product_id
GROUP BY t.month, t.year, p.product_id', 'MONTHLY_SALES_MV');
END;
/

SELECT mv_name, message FROM rewrite_table;

MV_NAME                                MESSAGE
-----                                -
MONTHLY_SALES_MV                        QSM-01001: query rewrite not enabled

```

EXPLAIN\_REWRITE can be used to check why a specific materialized view was not used to rewrite the query. For example, consider the following query, which is asking for the total sales by quarter, which is not possible to compute using the MONTHLY\_SALES\_MV because the QUARTER column is not in the materialized view.

```

BEGIN
dbms_mview.explain_rewrite('
SELECT p.product_id, t.quarter,
       SUM (ps.purchase_price) as sum_of_sales,
       COUNT(ps.purchase_price) as total_sales
FROM product p, purchases ps, time t
WHERE p.product_id = ps.product_id
      AND t.time_key = ps.time_key
GROUP BY p.product_id, t.quarter', 'MONTHLY_SALES_MV');
END;
/

SELECT mv_name, message FROM rewrite_table;

MV_NAME                                MESSAGE
-----                                -
MONTHLY_SALES_MV                        QSM-01082: Joining materialized
view, MONTHLY_SALES_MV, with
table, TIME, not possible

MONTHLY_SALES_MV                        QSM-01102: materialized view,
MONTHLY_SALES_MV, requires join
back to table, TIME, on column,
QUARTER

```

The EXPLAIN\_REWRITE output clearly indicates that it is not possible to do the rewrite because of the missing quarter column.

Sometimes query rewrite may be possible with the requested materialized view; however, there may be a more optimal materialized view that can be used. Suppose we create a materialized view, PRODUCT\_SALES\_EXACT\_MATCH, for the following query, matching its text exactly. Query rewrite now uses this materialized view instead, since it is more optimal. EXPLAIN\_REWRITE will tell you that this is the case.

```
BEGIN
dbms_mview.explain_rewrite('
SELECT p.product_id,
       SUM(ps.purchase_price) as sum_of_sales,
       COUNT(ps.purchase_price) as total_sales
FROM   product p, purchases ps, time t
WHERE  p.product_id = ps.product_id
       AND t.time_key = ps.time_key
GROUP BY p.product_id', 'MONTHLY_SALES_MV');
END;
/

SELECT mv_name, message FROM rewrite_table;

MV_NAME                                MESSAGE
-----                                -
MONTHLY_SALES_MV                       QSM-01009: materialized view,
                                         PRODUCT_SALES_EXACT_MATCH, matched
                                         query text
```

EXPLAIN\_REWRITE can also be used with very large queries by declaring them using a character large object (CLOB) data type.

## 9.6 Advanced Query Rewrite Techniques

In the preceding sections, we have discussed the most commonly used types of query rewrites. In this section, we will discuss some advanced topics in query rewrite. If you are just getting familiar with query rewrite, the preceding sections may be enough to get you started and you can come back to the remainder of this chapter as you get more familiar with using it.

### 9.6.1 Optimizer Hints for Query Rewrite

Ordinarily, the query optimizer will automatically decide whether or not to rewrite a query, and if there are several materialized views that are eligible to

rewrite the query, it will pick the best one. You can, however, influence this behavior using the following optimizer hints:

- **REWRITE(mv)** hint request the optimizer to use a specific materialized view.
- **NO\_REWRITE** hint to not use query rewrite for the query.
- **REWRITE\_OR\_ERROR** to throw an error when it is not possible to rewrite.

For instance, suppose we had two eligible materialized views: `MONTHLY_SALES_MV`, which computes sum of sales by month, and `YEARLY_SALES_MV`, which computes the sum of sales by year. If we wanted to know the sum of sales by year, as shown in the following query, you would expect query rewrite to pick the latter, since it would read less data.

```
SELECT t.year, p.product_id, SUM(ps.purchase_price) sum_of_sales
FROM time t, product p, purchases ps
WHERE t.time_key = ps.time_key AND
      ps.product_id = p.product_id
GROUP BY t.year, p.product_id;
```

PLAN\_TABLE\_OUTPUT

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		329	2
1	MAT_VIEW REWRITE ACCESS FULL	YEARLY_SALES_MV	329	2

You could, however force query rewrite to use `MONTHLY_SALES_MV` with a hint.

```
SELECT /*+ REWRITE(monthly_sales_mv) */ t.year, p.product_id,
      SUM(ps.purchase_price) as sum_of_sales
FROM time t, product p, purchases ps
WHERE t.time_key = ps.time_key AND
      ps.product_id = p.product_id
GROUP BY t.year, p.product_id;
```

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		2	11
1	SORT GROUP BY		2	11

*2	HASH JOIN			4990	9
3	VIEW			34	4
4	SORT UNIQUE			34	4
5	TABLE ACCESS FULL	TIME		731	3
6	MAT_VIEW REWRITE ACCESS FULL	MONTHLY_SALES_MV		3522	4

### Forcing an Error When Query Rewrite Is Not Possible

For some applications, query rewrite is critical to achieve good performance; it is preferable for a query to fail rather than execute against the detail data, because it may take too long to complete. In Oracle Database 10g, you can specify the `REWRITE_OR_ERROR` hint to force the query to fail if query rewrite is not possible. In the following example, the query asking for the sum of sales by day cannot rewrite against the available monthly or yearly summaries and hence will fail.

```
SELECT /*+ REWRITE_OR_ERROR */ t.time_key,
       SUM(ps.purchase_price) as sum_of_sales
FROM time t, product p, purchases ps
WHERE t.time_key = ps.time_key AND
       ps.product_id = p.product_id
GROUP BY t.time_key, p.product_id;
```

ORA-30393: a query block in the statement did not rewrite

### 9.6.2 Query Rewrite and Bind Variables

You may use bind variables in your queries to allow the query plan to be shared by multiple invocations of the query. However, in certain cases, use of bind variables can prohibit query rewrite. First of all, when the optimizer makes its decisions to rewrite a query, the bind variable values are generally not available. Further, the bind values can change for subsequent executions, without again going through the query rewrite process. Therefore, if the value of the bind variable could influence the correctness of query rewrite, then the query will not be rewritten.

For example, consider the following query, which asks for total sales for a specific product by month. This query has a bind variable on `PRODUCT_ID`. Now, if we had a materialized view with all product values, such as `MONTHLY_SALES_MV` defined earlier, the optimizer could safely use query rewrite for this query, regardless of the actual value of the bind variable, as shown in the execution plan.

```

EXPLAIN PLAN FOR
SELECT t.month, p.product_id,
       SUM(ps.purchase_price) as sum_of_sales,
       COUNT (ps.purchase_price) as total_sales
FROM time t, product p, purchases ps
WHERE t.time_key = ps.time_key AND
      ps.product_id = p.product_id AND
      p.product_id = :1
GROUP BY t.month, p.product_id;

```

PLAN\_TABLE\_OUTPUT

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		35	4
1	MAT_VIEW REWRITE ACCESS FULL	MONTHLY_SALES_MV	35	4

Predicate Information (identified by operation id):

```

-----
1 - filter("MONTHLY_SALES_MV"."PRODUCT_ID"=:1)

```

However, recall the materialized view SALES\_ELEC\_1\_6\_2003\_MV, defined earlier in section 9.2.5, which only has data for the ELEC category for the months January through June 2003. Suppose we had the following query instead; with a bind variable for the CATEGORY value, the optimizer cannot safely determine if your query can be answered using the materialized view. For example, if the bind variable :1 had the value MUSC, then the materialized view would not contain the data and query rewrite is not possible. However, if the value were ELEC, query rewrite would be possible. Because the actual value of the bind variable is not available when the decision to rewrite the query is made, the optimizer is unable to use the materialized view, as indicated by the execution plan output.

```

EXPLAIN PLAN FOR
SELECT t.month, t.year, p.product_id,
       SUM(ps.purchase_price) as sum_of_sales,
       COUNT (ps.purchase_price) as total_sales
FROM time t, product p, purchases ps
WHERE t.time_key = ps.time_key AND
      ps.product_id = p.product_id AND
      p.category = :1 AND
      t.month >= 200301 and t.month <= 200306
GROUP BY t.month, t.year, p.product_id;

```

```
PLAN_TABLE_OUTPUT
```

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		6628	207
1	SORT GROUP BY		6628	207
* 2	HASH JOIN		6628	122
* 3	TABLE ACCESS FULL	PRODUCT	55	2
* 4	HASH JOIN		20004	119
* 5	TABLE ACCESS FULL	TIME	181	3
6	PARTITION RANGE ITERATOR		80790	113
7	TABLE ACCESS FULL	PURCHASES	80790	113

Instead of bind variables, if the previous query had literal values, such as ELEC, Oracle may internally replace these values with bind variables known as **internal bind variables**. This allows queries differing only in these literal values to reuse or share compiled execution plans. This reuse of execution plans is called **cursor sharing** in Oracle. (To enable this feature the initialization parameter `CURSOR_SHARING` must be set to a value other than `EXACT`.) Just as with user-specified bind variables, the decision to rewrite the query may depend on the value of the internal bind variable. The difference here is that unlike user-specified bind variables, Oracle knows the values of internal bind variables at the time of query rewrite and so the query will be rewritten as expected. However, because the query plan now depends on the value of the literal, you may not see the expected amount of cursor sharing you may have otherwise seen.

To summarize, if your application would like to use query rewrite, you must carefully design the use of bind variables in your queries, otherwise you may not be able to take full advantage of query rewrite.

### 9.6.3 Query Rewrite with Complex SQL Constructs

The examples discussed so far used the more common constructs in SQL, such as joins, selections, and aggregation operators. However, query rewrite can also work in the presence of complex SQL expressions, including set operators, subqueries in the `FROM` clause, analytical functions, and `GROUPING SETS`.

#### Set Operators

A set operator is a SQL operation such as `UNION ALL`, `UNION`, `MINUS`, and `INTERSECT`. If a query has multiple subqueries, such as in a `UNION ALL`, the optimizer will try to rewrite each branch of the `UNION`

ALL individually using a simple materialized view. In addition, Oracle Database 10g supports query rewrite using a materialized view with the set operators. If the query has a UNION ALL, then query rewrite will try to match each branch in the query with appropriate branches in the materialized view. This is best illustrated with an example.

The following materialized view has a UNION ALL with two branches. Each branch also has a special column known as a **subselect marker**, which is required by query rewrite to identify rows for each branch. The marker can be any constant column (numeric or string) with a distinct value for each branch of the UNION ALL operation. In this example, the marker column has been aliased as *um*.

```
CREATE MATERIALIZED VIEW muscelec_mv
ENABLE QUERY REWRITE
AS
SELECT 'M' um, p.product_id, p.manufacturer,
       SUM(ps.purchase_price)
FROM purchases ps, product p
WHERE ps.product_id = p.product_id AND p.category = 'MUSC'
GROUP BY p.product_id, p.manufacturer
UNION ALL
SELECT 'E' um, p.product_id, p.manufacturer,
       SUM(ps.purchase_price)
FROM purchases ps, product p
WHERE ps.product_id = p.product_id AND p.category = 'ELEC'
GROUP BY p.product_id, p.manufacturer;
```

The following query can now be rewritten with this materialized view. Note that the order of the branches for ELEC and MUSC in the query has been reversed and one of the branches has an additional selection.

```
EXPLAIN PLAN FOR
SELECT p.product_id, SUM(ps.purchase_price)
FROM purchases ps, product p
WHERE ps.product_id = p.product_id AND p.category = 'ELEC'
GROUP BY p.product_id
UNION ALL
SELECT p.product_id, SUM(ps.purchase_price)
FROM purchases ps, product p
WHERE ps.product_id = p.product_id AND p.category = 'MUSC' AND
       p.manufacturer = 'ABC'
GROUP BY p.product_id;
```

## PLAN\_TABLE\_OUTPUT

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		112	4
1	UNION-ALL			
* 2	MAT_VIEW REWRITE ACCESS FULL	MUSCELEC_MV	111	2
* 3	MAT_VIEW REWRITE ACCESS FULL	MUSCELEC_MV	1	2

Unlike UNION ALL, other set operators, UNION, MINUS, and INTERSECT, are not commutative (cannot be reordered) and do not preserve duplicates in the query. So for a query to be rewritten, it must match the materialized view exactly.

### Subqueries in the FROM Clause

Applications often need to use subqueries in the FROM clause due to security reasons (e.g., they do not want to expose table names) or because the query is dynamically generated by a tool. If a query has sub-queries in the FROM clause, the optimizer can replace subqueries with the underlying tables and then query rewrite can take place as usual. In addition, in Oracle Database 10g, query rewrite will look for materialized views that have the identical subquery in the FROM clause (i.e., the text of the subquery in the query and materialized view match exactly). If any such matching materialized views are found, all the normal rules of query rewrite will then be checked. The subquery can be arbitrarily complex.

For example, suppose we have a materialized view, as follows, that contains the total sales by category for the manufacturer ABC:

```
CREATE MATERIALIZED VIEW prodcat_sales_mv
ENABLE QUERY REWRITE
AS
SELECT v.category, SUM(ps.purchase_price) as sum_of_sales
FROM (SELECT * FROM product p
WHERE p.manufacturer = 'ABC') v, purchases ps
WHERE ps.product_id = v.product_id
GROUP BY v.category;
```

Now, the following query, which is asking for the total sales for the ELEC category for the manufacturer ABC, can be rewritten to use this materialized view.

```

EXPLAIN PLAN FOR
SELECT v.category, SUM(ps.purchase_price) as sum_of_sales
FROM (SELECT * FROM product p
WHERE p.manufacturer = 'ABC') v, purchases ps
WHERE ps.product_id = v.product_id and v.category = 'ELEC'
GROUP BY v.category;

```

PLAN\_TABLE\_OUTPUT

```

-----
|Id|Operation                                |Name                |Rows |Cost|
-----
| 0|SELECT STATEMENT                        |                    |    1|    2|
| 1| MAT_VIEW REWRITE ACCESS FULL|PRODCAT_SALES_MV   |    1|    2|
-----

```

### **Multiple Occurrences of a Table in the FROM Clause**

Occasionally, queries may need to include the same table multiple times in the FROM clause. For instance, in the following query, we are finding the total monthly sales for orders that took at most one week to ship. We are using two date columns from the PURCHASES table: TIME\_KEY and SHIP\_DATE. So in order to determine any auxiliary information for that date, such as WEEK\_NUMBER, MONTH, from the TIME dimension table, you will need to join with TIME separately for each of these columns.

```

EXPLAIN PLAN FOR
SELECT ot.month, SUM(ps.purchase_price) as sum_of_sales
FROM purchases ps, time ot, time st
WHERE ps.time_key = ot.time_key AND ps.ship_date = st.time_key
AND st.week_number - ot.week_number <= 1
GROUP BY ot.month;

```

In Oracle Database 10g, query rewrite can automatically analyze the joins in the query and correctly match multiple instances of tables with their corresponding instances in a materialized view. So the preceding query can rewrite with the following materialized view (we have deliberately changed table aliases to rule out any simple text match rewrite).

```

CREATE MATERIALIZED VIEW sameweek_sales_mv
ENABLE QUERY REWRITE
AS
SELECT od.month ord_mon, sd.month ship_mon,
       od.week_number ord_week, sd.week_number ship_week,
       SUM(ps.purchase_price) as sum_of_sales
FROM purchases ps, time od, time sd

```

```
WHERE ps.time_key = od.time_key AND ps.ship_date = sd.time_key
GROUP BY od.month, sd.month, sd.week_number, od.week_number;
```

The following execution plan shows the query rewritten:

```
PLAN_TABLE_OUTPUT
-----
| Id | Operation                                | Name                | Cost |
-----|-----|-----|-----|
| 0 | SELECT STATEMENT                          |                      |      |
| 1 | SORT GROUP BY                              |                      |      |
|* 2 | MAT_VIEW REWRITE ACCESS FULL| SAMEWEEK_SALES_MV | 2 |
-----
```

Predicate Information (identified by operation id):

```
-----
2 - filter("SAMEWEEK_SALES_MV"."SHIP_WEEK"-
           "SAMEWEEK_SALES_MV"."ORD_WEEK"<=1)
```

So for all practical purposes, query rewrite will work as if the multiple occurrences were different tables.

### Grouping Sets

The SQL aggregation operators, CUBE, ROLLUP, and GROUPING SETS, described in Chapter 6, provide a mechanism to compute multiple levels of aggregation in a single query. You can create a materialized view using a query with these operators to store multiple levels of aggregation, instead of separate materialized views for each level. Query rewrite can be used to rewrite a query that asks for any of these levels of aggregation.

The following example shows a materialized view with grouping sets that computes the sum of sales for the 3 groupings: (category, time\_key), (category, time\_key, state), and (time\_key, country). Note that the materialized view must have a GROUPING\_ID or GROUPING function on the group by columns to distinguish rows that correspond to different groupings.

```
CREATE MATERIALIZED VIEW sales_mv
ENABLE QUERY REWRITE
AS
SELECT p.category, t.time_key, c.country, c.state,
       SUM(f.purchase_price) sales,
       GROUPING_ID(p.category, t.time_key, c.country, c.state) gid
FROM product p, purchases f, time t, customer c
WHERE p.product_id = f.product_id AND
       t.time_key = f.time_key AND
```

```

c.customer_id = f.customer_id
GROUP BY GROUPING SETS ((p.category, t.time_key),
                        (p.category, t.time_key, c.state),
                        (t.time_key, c.country));

```

This materialized view can be used to rewrite a query that asks for any grouping that is present in the materialized view or one that can be derived using a rollup. For example, the following query, which asks for total sales by category and time\_key, can be rewritten to use the SALES\_MV materialized view.

```

EXPLAIN PLAN FOR
SELECT p.category, t.time_key, SUM(f.purchase_price) sales
FROM product p, purchases f, time t, customer c
WHERE p.product_id = f.product_id AND
      t.time_key = f.time_key AND
      c.customer_id = f.customer_id
GROUP BY p.category, t.time_key;

```

From the predicate information in the EXPLAIN PLAN output, we see that rewrite was done by selecting rows for the grouping (p.category, t.time\_key), which corresponds to gid = 3.

PLAN\_TABLE\_OUTPUT

```

-----
| Id | Operation                               | Name      | Rows  | Cost |
-----
|  0 | SELECT STATEMENT                         |           | 2194  |  18  |
|*  1 |  MAT_VIEW REWRITE ACCESS FULL           | SALES_MV  | 2194  |  18  |
-----

```

Predicate Information (identified by operation id):

```

-----
1 - filter("SALES_MV"."GID"=3)

```

If the query itself has multiple groupings, Oracle will try to find a materialized view that satisfies all the groupings. If Oracle cannot find a single materialized view to answer a GROUPING SETS query, it will try to rewrite each grouping separately. (Note that a query with GROUPING SETS can be expressed using a UNION ALL of queries with the individual groupings.) As a result, several materialized views may get used to rewriting the query and some groupings may remain unrewritten and use the detail data.

The next query cannot be rewritten using SALES\_MV alone. The grouping (p.category, t.time\_key) is present in the SALES\_MV materialized view, and the grouping (t.time\_key, c.state) can be derived using a rollup of (p.category, t.time\_key, c.state). However, the grouping (p.category, c.country, t.year) is not present in this materialized view.

```
EXPLAIN PLAN FOR
SELECT p.category, t.time_key, c.country, t.year, c.state,
       SUM(f.purchase_price) sales
FROM product p, purchases f, time t, customer c
WHERE p.product_id = f.product_id AND
      t.time_key = f.time_key AND
      c.customer_id = f.customer_id
GROUP BY GROUPING SETS ((p.category, t.time_key),
                        (t.time_key, c.state),
                        (p.category, c.country, t.year));
```

PLAN\_TABLE\_OUTPUT

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		16052	189
1	VIEW		16052	189
2	UNION-ALL			
3	SORT GROUP BY		6	150
* 4	HASH JOIN		80569	129
5	TABLE ACCESS FULL	CUSTOMER	500	3
* 6	HASH JOIN		80569	122
7	TABLE ACCESS FULL	TIME	731	3
* 8	HASH JOIN		80679	116
9	TABLE ACCESS FULL	PRODUCT	164	2
10	PARTITION RANGE ALL		81171	111
11	TABLE ACCESS FULL	PURCHASES	81171	111
12	SORT GROUP BY		13852	22
* 13	MAT_VIEW REWRITE ACCESS FULL	SALES_MV	13852	18
* 14	MAT_VIEW REWRITE ACCESS FULL	SALES_MV	2194	18

Predicate Information (identified by operation id):

```
4 - access("C"."CUSTOMER_ID"="F"."CUSTOMER_ID")
6 - access("T"."TIME_KEY"="F"."TIME_KEY")
8 - access("P"."PRODUCT_ID"="F"."PRODUCT_ID")
13 - filter("SALES_MV"."GID"=2)
14 - filter("SALES_MV"."GID"=3)
```

The EXPLAIN PLAN output shows that rewrite was done using SALES\_MV for two groupings (gid = 3 and gid = 2) and using the detail tables for grouping (p.category, c.county, t.year).

If we had another simple materialized view, SALES\_MV2 (not shown here), that had the missing grouping (p.category, c.county, t.year), the optimizer would use it to rewrite the remaining grouping, as shown in the following execution plan.

PLAN\_TABLE\_OUTPUT

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		16059	42
1	VIEW		16059	42
2	UNION-ALL			
3	MAT_VIEW REWRITE ACCESS FULL	SALES_MV2	13	2
4	SORT GROUP BY		13852	22
* 5	MAT_VIEW REWRITE ACCESS FULL	SALES_MV	13852	18
* 6	MAT_VIEW REWRITE ACCESS FULL	SALES_MV	2194	18

Predicate Information (identified by operation id):

- 5 - filter("SALES\_MV"."GID"=2)
- 6 - filter("SALES\_MV"."GID"=3)

### Analytical Functions

Oracle Database 10g supports limited query rewrite with the analytical functions, which were discussed in Chapter 6. If an analytical function in the query matches exactly with one in the materialized view, and the query and the materialized view aggregate at the same level (i.e., there is no need for a rollup), then query rewrite can occur. For example, the following materialized view includes the RANK() function and contains the ranks for products, ordered by their total sales, with the worst-selling products first.

```
CREATE MATERIALIZED VIEW rank_mv
ENABLE QUERY REWRITE
AS
SELECT p.product_id p_id, SUM(f.purchase_price) as sales,
       RANK() over (ORDER BY SUM(f.purchase_price)) as rank
FROM purchases f, product p
WHERE f.product_id = p.product_id
GROUP BY p.product_id;
```

The following query, asking for the 10 worst-selling products, can now rewrite against this materialized view.

```
EXPLAIN PLAN FOR
SELECT * FROM
(SELECT p.product_id p_id,
       RANK() over (ORDER BY SUM(f.purchase_price)) as rank
FROM purchases f, product p
WHERE f.product_id = p.product_id
GROUP BY p.product_id)
WHERE rank < 10;
```

PLAN\_TABLE\_OUTPUT

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		9	2
* 1	MAT_VIEW REWRITE ACCESS FULL	RANK_MV	9	2

Predicate Information (identified by operation id):

```
1 - filter("RANK_MV"."RANK"<10)
```

If the analytical function is not present or does not match the one in the materialized view, but the underlying aggregate is present in the materialized view, then query rewrite can happen. In this case, the analytical function will be computed from the aggregate in the materialized view. This is indicated by a **window sort** operation in the execution plan. In the following example, the query computes the DENSE\_RANK() and also computes RANK() in descending order.

```
EXPLAIN PLAN FOR
SELECT p.product_id p_id,
       DENSE_RANK() over (ORDER BY SUM(f.purchase_price)) as drank,
       RANK() over (ORDER BY SUM(f.purchase_price) DESC) as
rev_rank
FROM purchases f, product p
WHERE f.product_id = p.product_id
GROUP BY p.product_id;
```

Since SUM(purchase\_price) is present in the materialized view, query rewrite takes place, as shown in the following execution plan.

```
PLAN_TABLE_OUTPUT
```

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		165	4
1	WINDOW SORT		165	4
2	WINDOW SORT		165	4
3	MAT_VIEW REWRITE ACCESS FULL	RANK_MV	165	2

### 9.6.4 Query Rewrite Using Nested Materialized Views

In Chapter 7, we discussed how nested materialized views could be used to share common joins across several materialized views or to materialize different levels in a hierarchy. After a query has been rewritten using a materialized view, the optimizer will check if it can be further rewritten using a nested materialized view. To illustrate this, consider the following query, which asks for total product sales.

```
EXPLAIN PLAN FOR
SELECT p.product_id, SUM(ps.purchase_price) as ave_sales
FROM product p, purchases ps
WHERE ps.product_id = p.product_id
GROUP BY p.product_id;
```

In section 9.2.8, we saw how this query can be rewritten using MONTHLY\_SALES\_MV. Now, suppose we had a nested materialized view on top of this materialized view, which computed the total sales by product\_id as follows:

```
CREATE MATERIALIZED VIEW YEARLY_PROD_SALES_MV
ENABLE QUERY REWRITE
AS
SELECT m.product_id, SUM(m.sum_of_sales) as yearly_sales
FROM monthly_sales_mv m
GROUP BY m.product_id;
```

After the query has been rewritten using the MONTHLY\_SALES\_MV materialized view, the optimizer will further rewrite this query to use YEARLY\_PROD\_SALES\_MV, as illustrated in the following execution plan output.

```
PLAN_TABLE_OUTPUT
```

Id	Operation	Name	Cost
0	SELECT STATEMENT		2
1	MAT_VIEW REWRITE ACCESS FULL	YEARLY_PROD_SALES_MV	2

Thus, the answer to the query is directly obtained from the smaller YEARLY\_PROD\_SALES\_MV nested materialized view without using the larger intermediate MONTHLY\_SALES\_MV materialized view.

### 9.6.5 Rewrite Equivalences

The optimizer can rewrite a query using a materialized view, provided it can determine that the answer is contained in the materialized view. However, to do so, it can only rely on available metadata in the database. Sometimes it may be not possible to rewrite the query in general, but with some specific application knowledge, it may indeed be possible to rewrite the query. Oracle Database 10g has a new feature called **rewrite equivalence**, which allows you to declare an alternative equivalent form of a given query. You can use this feature to do a user-defined query rewrite using your application knowledge. To use this feature you must use the procedure DBMS\_ADVANCED\_REWRITE.DECLARE\_REWRITE\_EQUIVALENCE to declare to Oracle that two statements are identical. We will illustrate this concept with an example.

Suppose we had the following materialized view, which computes a monthly sales forecast using the user-defined aggregate function, SalesForecast(), which was discussed in Chapter 6.

```
CREATE MATERIALIZED VIEW SALES_FORECAST_MV
ENABLE QUERY REWRITE
AS
SELECT t.month, t.year,
       SalesForecast(ps.purchase_price) sales_forecast
FROM time t, purchases ps
WHERE t.time_key = ps.time_key
GROUP BY t.month, t.year;
```

Some simple query rewrites are possible with user-defined aggregates—for example, you can use this materialized view to return the precomputed sales forecast by month. However, it is not possible to do a rollup of a user-defined aggregate—for instance, from monthly to yearly level. In other

words, if we wanted to calculate the sales forecast on a yearly basis, we would ordinarily have to use the detail data or create a separate materialized view for it.

Now, suppose that, because of the nature of this aggregate function, it is possible to roll up to a yearly level by simply doing a SUM over the monthly forecasts. Obviously, this is not generally true with all user-defined aggregates; however, in this specific case, we know this to be the case based on “insider” knowledge of its implementation. In this case, we can declare a rewrite equivalence, as follows:

```
BEGIN
SYS.DBMS_ADVANCED_REWRITE.DECLARE_REWRITE_EQUIVALENCE (
  'SALES_FORECAST_ROLLUP',
  'SELECT t.year, SALESFORECAST(ps.purchase_price) sales_forecast
   FROM time t, purchases ps
   WHERE t.time_key = ps.time_key
   GROUP BY t.year',
  'SELECT year, SUM(sales_forecast) yearly_forecast
   FROM sales_forecast_mv
   GROUP BY year'
);
END;
/
```

This procedure has three required parameters: a name (which can later be used to drop or edit the equivalence), a source statement, and a destination statement. With this declaration, if you now ask the query on the source statement, Oracle will automatically use the equivalent query specified by the destination statement.

---

**Hint:** The optimizer only uses rewrite equivalences provided the QUERY\_REWRITE\_INTEGRITY parameter is set to the TRUSTED or STALE\_TOLERATED modes.

---

In our example, if we issue the query on the yearly level, it will be transparently replaced with the query using the SALES\_FORECAST\_MV, as shown in the following execution plan. Thus, we have done a user-defined query rewrite!

```
EXPLAIN PLAN FOR
SELECT t.year, SALESFORECAST(ps.purchase_price) sales_forecast
FROM time t, purchases ps
```

---

```
WHERE t.time_key = ps.time_key
GROUP BY t.year;
```

```
PLAN_TABLE_OUTPUT
```

Id	Operation	Name	Cost
0	SELECT STATEMENT		3
1	SORT GROUP BY		3
2	MAT_VIEW ACCESS FULL	SALES_FORECAST_MV	2

---

**Hint:** If there is a choice between using a rewrite equivalence and a materialized view to rewrite a query, the optimizer will prefer the equivalence to the materialized view.

---

When declaring the equivalence, if you set the parameter *validate* to true, Oracle will execute both the source and destination queries and verify that they return the same results or otherwise give an error. However, as the underlying data changes, it is possible that the two statements may no longer return identical results. Oracle will not check the validity of the equivalence as the data changes. It is up to the user who created the equivalence to ensure that the two queries are equivalent for the application; otherwise, it may produce unexpected results. You can check the validity of the equivalence at any time by issuing the procedure `DBMS_ADVANCED_REWRITE.VALIDATE_REWRITE_EQUIVALENCE`.

As you can see, this is an extremely powerful feature; hence, this package is owned by SYS and its use is not enabled by default. The DBA needs to explicitly grant access to the package to trusted users who can create these equivalences, as follows:

```
GRANT EXECUTE ON DBMS_ADVANCED_REWRITE TO <user>;
```

You can disable a rewrite equivalence using the following procedure:

```
EXECUTE DBMS_ADVANCED_REWRITE.ALTER_REWRITE_EQUIVALENCE
('SALES_FORECAST_ROLLUP', mode=>'disabled');
```

Rewrite equivalences should only be used if absolutely necessary and should be made available only to the most advanced users of query rewrite.

Incorrect use of this powerful feature can wreak havoc, since users could get unexpected or bogus results.

### 9.6.6 Using Query Rewrite during Refresh

Another new feature in Oracle Database 10g is using query rewrite when populating or refreshing a materialized view. This means that to populate or refresh one materialized view, Oracle will try to reuse the precomputed data in another materialized view, using query rewrite. This can be much quicker than refreshing the materialized view directly from the detail data! For example, if you had a materialized view at a monthly grain and another one at a daily grain, Oracle can use the materialized view at the daily grain to refresh the monthly one.

Note that only fresh materialized views will be used for query rewrite during refresh so that the materialized view being refreshed always sees the most up-to-date data. In addition, by default query rewrite will be performed with the `QUERY_REWRITE_INTEGRITY` parameter setting of `ENFORCED`, which means trusted relationships such as dimensions will not be used by query rewrite. However, if you would like to have refresh use the `QUERY_REWRITE_INTEGRITY` setting of `TRUSTED`, you may specify the `USING TRUSTED CONSTRAINTS` clause on the `CREATE MATERIALIZED VIEW` statement, as shown in the following example.

```
CREATE MATERIALIZED VIEW product_category_sales_mv
REFRESH FORCE
USING TRUSTED CONSTRAINTS      <- constraints clause
  ENABLE QUERY REWRITE
AS
SELECT ps.time_key, p.category,
       SUM(ps.purchase_price) as sum_of_sales
FROM   product p, purchases ps
WHERE  ps.product_id = p.product_id
GROUP BY ps.time_key, p.category;
```

If you do not specify this clause, the default clause is `USING ENFORCED CONSTRAINTS`.

This allows refresh to take advantage of trusted information, such as `RELY` constraints, dimensions, and materialized views on prebuilt tables, to rewrite the internal queries issued during refresh. Note, however, that, as discussed in section 9.3, it is the DBA's responsibility to guarantee correctness of the trusted information; otherwise, your materialized view could have incorrect data.

---

---

---

**Hint:** To enable use of query rewrite during refresh, set the initialization parameter, `QUERY_REWRITE_INTEGRITY`, to `TRUE` in the session performing the refresh.

---

---

When refreshing materialized views, it is recommended that you refresh multiple materialized views simultaneously and enable query rewrite. This allows Oracle Database 10g to optimize the ordering of the refresh operations such that it can make best use of query rewrite during refresh. For example, if you have two materialized views, one at a monthly level and one at a daily level, Oracle will first refresh the materialized view at a daily grain and then use this materialized view to refresh the one at a monthly grain, using query rewrite.

Using query rewrite during refresh can significantly improve performance of refreshing your materialized views.

## 9.7 Summary

Summary management in Oracle provides a very powerful set of tools to improve query performance in your warehouse. With query rewrite, the queries will be transparently rewritten to use the materialized views. We have seen how you can use the same materialized view to rewrite a large class of queries, thereby reducing the space and maintenance resources required for materialized views. We have also seen how to troubleshoot problems with query rewrite and how to use query rewrite to improve performance of refreshing the materialized views.

This brings us to the question: How do you know which materialized views to create? Determining the optimal set of materialized views to create for a large number of queries can be tricky, and, if not done correctly, the disk space requirements and refresh overhead could soon get prohibitive. Fortunately, Oracle Database 10g provides a tool called the SQL Access Advisor, which is designed to choose the best set of materialized views and indexes for an application.

The next chapter discusses the SQL Access Advisor and various other query techniques and tools to tune query performance in a data warehouse.

