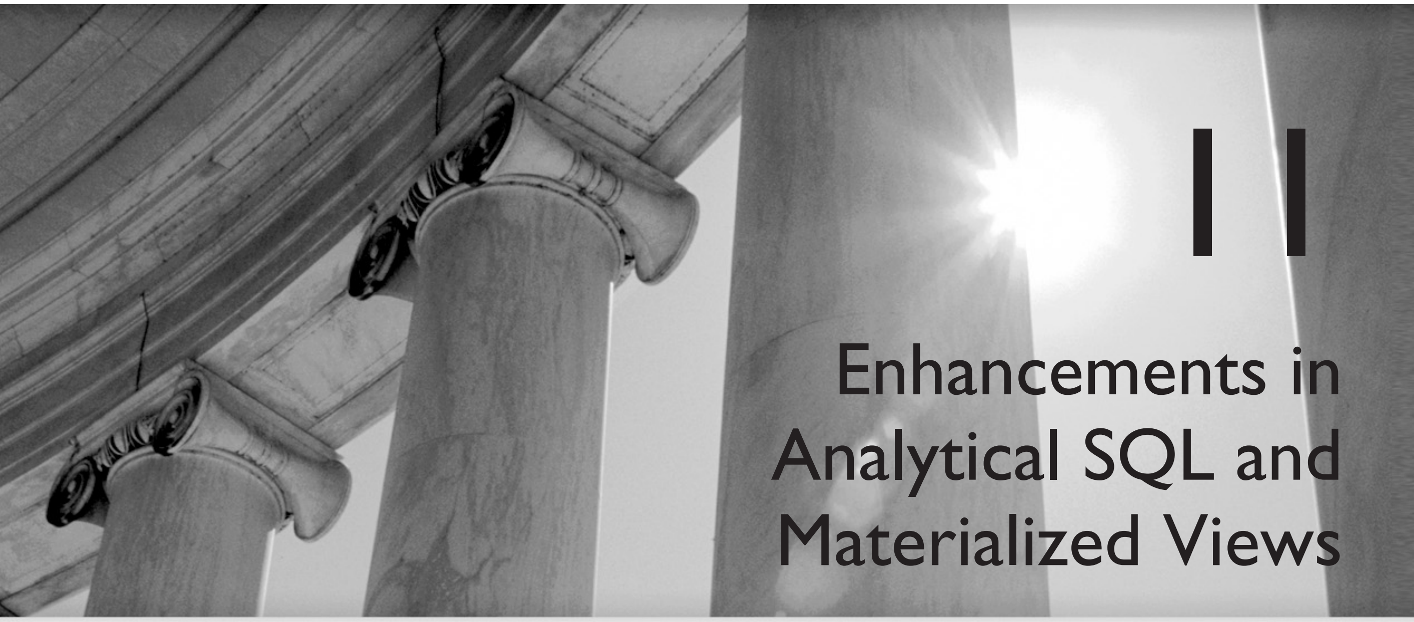

11 Enhancements in Analytical SQL and Materialized Views

CERTIFICATION OBJECTIVES

11.01	Enhancements in the MERGE Statement	11.04	Materialized View Enhancements
11.02	Using Partitioned Outer Joins	✓	Two-Minute Drill
11.03	Using the SQL MODEL Clause	Q&A	Self Test

This chapter deals with analytical SQL enhancements to support data warehousing applications and enhancements to several materialized view features. You'll start off with a review of the analytical SQL enhancements in Oracle Database 10*g*. The `MERGE` statement is now enhanced, with support for conditional statements. The new partitioned outer joins help in the densification of data, which will yield better calculation performance for several analytical queries, especially those dealing with time-series data. The brand-new `MODEL` SQL clause provides spreadsheet-like array computations that are highly scalable, directly from within your Oracle database.

Oracle Database 10*g* presents several improvements to the management of materialized views, which are critical in data warehousing environments. Oracle supports fast refresh for materialized join views under even more conditions now. You can use the new procedure `TUNE_MVIEW` of the `DBMS_ADVISOR` package to enhance the fast refresh and query rewrite capabilities of your materialized views. You'll also review several new materialized view enhancements like improvements to the partition change tracking feature and materialized view refresh when you're using trusted, instead of enforced, constraints.

Let's start the chapter with a discussion of the enhancements to the powerful `MERGE` statement.

CERTIFICATION OBJECTIVE 11.01

Enhancements in the MERGE Statement

The `MERGE` statement is primarily of value when you're moving vast amounts of data in a data warehousing application. When you are extracting data from a source system into a data warehouse, not all of the data will be completely new. You may find that some of the table rows are new, while others are modifications of existing data. Therefore, you'll need to insert some of the new data and also update some existing data with the new data from the source. For example, if a sales transaction is completely new, you'll insert that row into the data warehouse table. If that particular transaction already exists, you'll merely update the necessary columns.

Oracle9*i* made available the highly useful `MERGE` statement, which enables you conveniently perform *both inserts and updates* in a single SQL statement. You can perform an `UPDATE-ELSE-INSERT` operation using a `MERGE` statement. Oracle has enhanced the MERGE statement in Oracle Database 10*g*. Let's review the MERGE statement prior to Oracle Database 10*g*, and then cover its new capabilities in the following sections.

The Basic MERGE Statement

The basic MERGE statement has the following structure:

```
merge <hint> into <table_name>
using <table_view_or_query>
on (<condition>)
when matched then <update_clause>
when not matched then <insert_clause>;
```

Here's a simple example of a basic MERGE statement, as of the Oracle9*i* version:

```
SQL> merge into  sales s using new_sales n
     on (s.sales_transaction_id = n.sales_transaction_id)
when matched then update
s_quantity = s_quantity + n_quantity, s_dollar = s_dollar + n_dollar
     when not matched then insert (sales_quantity_sold, sales_dollar_amount)
     values (n.sales_quantity_sold, n.sales_dollar_amount);
```

The ON condition (s.sales_transaction_id = n.sales_transaction_id) determines if an update or an insert operation will take place. The previous statement will update a row in the new_sales table, if that row already exists (the sales_transaction_id column identifies the row). If there is no such row, Oracle will insert a new row with values for the sales_quantity_sold and sales_dollar_amount columns.

Conditional UPDATE and INSERT Statements

Rather than an unconditional insertion or updating of all the table rows, you may want to insert or update data only when certain conditions are met. In Oracle Database 10*g*, the MERGE statement has been enhanced to allow you to *conditionally* insert or delete data. Now, Oracle allows you to use a WHERE clause in a MERGE statement's UPDATE or INSERT clause to conditionally update or insert data.

Here's an example that shows how you can conditionally insert and update data using a MERGE statement (note the USING clause in the MERGE statement):

```
SQL> merge using product_cChanges s         -- Source table
     into products p                       -- Destination table
     on (p.prod_id = s.prod_id)           -- Search/Join condition
     when matched then update             -- Update if join
     set p.prod_list_price = s.prod_new_price
     where p.prod_status <> “EXPIRED”    -- Conditional update
     when not matched then
     insert                               -- Insert if not join
     set p.prod_list_price = s.prod_new_price
     where s.prod_status <> “EXPIRED”    -- conditional insertNote that Oracle
will skip the insert or update operation if the statement doesn’t satisfy the
WHERE condition. Both the insert and the update operations would occur only if
the product is not an expired item. (where s.prod_status <> “EXPIRED”).
```

The DELETE Clause with the MERGE Statement

You can now use the MERGE statement with an optional DELETE clause in a MERGE statement. However, you can’t use the DELETE clause independently in a MERGE statement, as with the UPDATE or INSERT clause. You must embed the DELETE statement inside the UPDATE statement. This means that the DELETE statement isn’t a “global” clause, but rather works in the confines of the data affected by the UPDATE clause of the MERGE statement. The following example shows how the DELETE clause is embedded within the UPDATE clause.

```
SQL> merge using  product_changes s
     into products p     on (d.prod_id = s.prod_id)
     when matched then
     update set d.prod_list_price = s.prod_new_price,
     d.prod_status = s.prod_new_status
     delete where (d.prod_status = “OLD_ITEM”)
     when not matched then
     insert (prod_id, prod_list_price, prod_status)
     values (s.prod_id, s.prod_new_price, s.prod_new_status);
```

The preceding MERGE statement will first update the prod_list_price and the prod_status columns of the product table wherever the join condition is true. The join condition (d..prod_id = s.prod_id) joins the two tables, product (the source table) and product_changes (the destination table).

Here are a couple of considerations when using the DELETE statement:

- The DELETE clause affects only the rows that were updated by the MERGE statement.
- The MERGE statement will delete only the rows included in the join condition specified by the ON clause.

exam watch

The `DELETE` clause in a `MERGE` operation will evaluate only the updated values (values updated by the `UPDATE` clause) and not the original values that were evaluated by the `UPDATE` clause.

When you use this `MERGE` statement, the `UPDATE` clause fires first, and it may set some of the prod_new_status values to expired. The `DELETE` clause will then remove all the rows whose prod_new_status value was set to expired by the `UPDATE` clause.The `DELETE` clause will not remove any other rows with the expired status, unless they are part of the join defined in the `ON` clause.

CERTIFICATION OBJECTIVE 11.02

Using Partitioned Outer Joins

Oracle provides a rich set of analytical functions to help you in business reporting. Using these functions, you can avoid the need to program tedious user-defined functions and formulas. Prior to Oracle Database 10g, you had access to the following analytical functions:

- Ranking and percentile functions include cumulative distributions, percentile ranks, and N-tiles.
- Moving window calculations provide the capacity to compute sums and averages.
- Lag/lead functions help you compute period-to-period changes.
- First/last functions help you figure out the first and last values in an ordered group.
- Linear-regression functions help you calculate linear regression and other related statistics.

Oracle Database 10*g* provides an extension of the SQL outer join concept to improve the performance of analytical functions when they encounter data with missing values for some combinations of dimension values. In the following sections, I'll introduce you to some basic data warehousing concepts and analytical processing features, before explaining how to use partitioned outer joins to handle problems caused by sparse data (data with missing values).

Fact Tables, Measures, and Dimensions

Fact tables contain the business data of an organization. Sales and inventory items are two common examples of the type of data captured in a fact table. Of course, fact tables contain facts, but the data is also referred to as a *measure*. Thus, sales in a sales fact table and inventory in an inventory fact table are the measures. Literally, a measure denotes what it is that you are measuring.

Fact tables often contain links to several entities like time, product, and market region. For example, the fact table might tell you what a firm's total sales are for the year 2005. However, you are more likely to want the data to answer more meaningful questions like, "What are our sales for dish detergents in the New York area during the first quarter of 2005?" To answer questions like this, you use the concept of dimensions.

A *dimension* is a means of dividing your data into meaningful categories. Using dimensions, you can turn your raw facts into meaningful data. For example, for the sales fact table, the correct dimensions may be time (year and quarters), product, and region. It is customary in data warehousing applications to create small tables, called *dimension tables*, to describe the various dimensions. These dimension tables serve as the *reference*, or lookup, tables. The combination of several dimension values helps you answer detailed analytical questions. For example, using the time, region, and product dimension values, you can easily answer complex business questions like the question posed in the previous paragraph.

The use of a central fact table and a number of dimension tables linked to it through foreign keys is called a *star schema* in data warehousing circles. The primary key of the fact table is usually a composite key that is made up of all of its foreign keys.

Dimension tables are usually organized along a hierarchical basis. Thus, the detailed data in a dimension, which is normally collected at the lowest possible level, is aggregated into more useful aggregates. When you move up a hierarchy, it is called *rolling up*. Conversely, when you move down a hierarchy of a dimension, it is called *drilling down*. For example, in the customers dimension, customers may roll up to a city. The cities may be rolled up into a division. The division may be rolled up into a region. Regions may, in turn, be rolled up into a country.

How Analytical Functions Process Data

Analytical SQL functions efficiently deliver complex summary, aggregation, and other analytical results. In order to produce these results, analytical functions follow a methodical processing order. Analytical processing usually follows a three-step sequence:

- **Grouping** In the preliminary grouping step, Oracle performs the various joins, `WHERE`, `GROUP BY`, and similar *grouping* operations.
- **Calculation (analysis)** In this middle stage of analytical processing, the result sets from the grouping step are divided into sets of rows called *partitions*. The result set from the previous step could be broken into one, a few, or many partitions. Oracle then hands the result set of the grouping process from the previous step to the analytical functions. The middle stage is at the heart of analytical function usage, since this where the analytical functions process the rows of each of the partitions.
- **Output ordering** Finally, Oracle hands you the output in the precise order specified by any `ORDER BY` clause that you may specify at the end of your query.

Data Densification

The concept of* data densification *has to do with the fact that you may view your data in two different forms:

- *Dense data* is when you have rows for all possible combinations of dimension values, even when you don't have any data (facts) for certain combinations.
- You have *sparse data* when you don't show any values for certain combinations of dimension values, if you don't have any data (facts) for those combinations. In real life, data is usually sparse.

To understand why dense data is better, let's say that you are dealing with time-series data. If you have sparse data, you'll have the problem of an inconsistent number of rows for groups of dimensions. This makes it harder for you to use some SQL analytical functions such as the lag and lead functions, which help you compute period-to-period changes. These functions perform slowly, and the report formatting would be uneven. Performance takes a big hit when you don't have a row for each combination of the dimensions.

exam watch

The partitioned outer join is ideal for time dimensions, but it can be used for any kind of dimensions.

Partitioned outer joins help turn sparse data into dense data, thus helping you avoid the drawbacks of using sparse data for analytical computations. When you use a partitioned outer join, Oracle replaces the missing values along any dimensions. You thus have faster performance and a better reporting format when you use partitioned outer joins. We'll look at how partitioned outer joins work after a quick review of Oracle's join methods.

A Review of Oracle Join Methods

Let's quickly review Oracle's join methods before plunging into the partitioned outer join concepts. Join queries combine rows from two or more tables, views, or materialized views. Oracle performs a join whenever multiple tables appear in the query's `FROM` clause. The query's `SELECT` list can select any columns from any of these tables. The `WHERE` clause, also called the *join condition*, compares two columns, each from a different table. To execute the join, Oracle combines pairs of rows, each containing one row from each table for which the join condition evaluates to `TRUE`.

Oracle joins are of the following types:

- **Natural joins** A natural join is based on all columns that have the same name in the two tables. It selects rows from the two tables that have equal values in the relevant columns.
- **Inner joins** An inner join (also called a *simple join*) is a join of two or more tables that returns only those rows that satisfy the join condition. An inner join is the default join type for a join operation.
- **Outer joins** An outer join extends the result of an inner join. An outer join returns all rows that satisfy the join condition and also returns some or all of those rows from one table for which *no rows* from the other satisfy the join condition.
  - A left outer join performs an outer join of tables A and B and returns all rows from table A. For all rows in table A that have no matching rows in table B, Oracle returns NULL for any `SELECT` list expressions containing columns of table B.
  - A right outer join performs an outer join of tables A and B and returns all rows from table B. For all rows in table B that have no matching rows in table A, Oracle returns NULL for any `SELECT` list expressions containing columns of table A.
  - A full outer join performs an outer join and returns all rows from A and B, extended with NULLs if they do not satisfy the join condition.

Partitioned Outer Joins

A *partitioned outer join* is nothing but an extension of an Oracle outer join. You use partitioned outer joins to fill the gaps in sparse data. In order to use a partitioned outer join, you add the `PARTITION BY` clause to the outer join clause. The `PARTITION BY` clause partitions the rows in your query output on the basis of the expression you provide within the clause.

Here's the syntax of a partitioned outer join:

```
select .....
from table_reference
partition by (expr [, expr ]... )
right outer join table_reference
and
select .....
from table_reference
left outer join table_reference
partition by {expr [,expr ]...)
```

For example, suppose your `SELECT` list consists of three columns: product, time_ id, and quantity. The logical partitioning can be done on the basis of the following condition:

```
partition by  product order by time_id
```

The query output will be partitioned into groups by the product column. If there were two products—bottles and cans—there would be two partitions. Once Oracle logically partitions the query output, it applies the outer join to each of the logical partitions. You can thus view the output of a partitioned outer join as a `UNION` of several outer joins, consisiting of a join of each of the logical partitions with the other table in the join. In the example, the bottle and can partitions are joined to the other table by using the time_id column.

Sparse Data

Let's look at a typical set of sparse data by using the following example, which shows the weekly and year-to-date sales for the same set of 11 weeks in two years (2004 and 2005).

```
Select substr(p.prod_Name,1,15) product_name, t.calendar_year year,
t.calendar_week_number week, SUM(amount_sold) sales
from sales s, times t, products p
```

```
where s.time_id = t.time_id and s.prod_id = p.prod_id AND
 p.prod_name in ('Bounce') and
 t.calendar_year in (2004,2005) and
 t.calendar_week_number between 20 and 30
group by p.prod_name, t.calendar_year, t.calendar_week_number;

PRODUCT_NAME          YEAR       WEEK      SALES
--------------- ---------- ---------- ----------
Bounce                2004         20        801
Bounce                2004         21    4062.24
Bounce                2004         22    2043.16
Bounce                2004         23    2731.14
Bounce                2004         24    4419.36
Bounce                2004         27    2297.29
Bounce                2004         28    1443.13
Bounce                2004         29    1927.38
Bounce                2004         30    1927.38
Bounce                2005         20     1483.3
Bounce                2005         21    4184.49
Bounce                2005         22    2609.19
Bounce                2005         23    1416.95
Bounce                2005         24    3149.62
Bounce                2005         25    2645.98
Bounce                2005         27    2125.12
Bounce                2005         29    2467.92
Bounce                2005         30    2620.17
```

We should normally have a total of 22 rows (11 weeks for each year) of sales data. However, we have a set of sparse data, with only 18 rows. Four rows are missing, because we have no data for weeks 25 and 26 in the year 2004 and weeks 26 and 28 in the year 2005.

Making the Data Dense

Using the query with the partitioned outer join produces the following output, which gets rid of the sparse data we had in the earlier query output. Instead of blanks, we now have zero values. In the following query, let's call our original query `v`, as we select data from the table times, which we'll refer to as `t`. Note that all 22 rows are retrieved this time, leaving no gaps in our time series.

```
select  product_name, t.year, t.week,

NVL(sales,0) dense_sales from
(select substr(p.prod_name,1,15) product_name,
t.calendar_year year, t.calendar_week_number week,
SUM(amount_sold) sales
from  sales s, times t, products p
where s.time_id = t.time_id and s.prod_id = p.prod_id and
p.prod_name in ('Bounce') and
t.calendar_year in (2004,2005) and
t.calendar_week_number between 20 and 30
group by p.prod_name, t.calendar_year, t.calendar_week_number) v
partition by (v.product_name)
right outer join
 (select distinct calendar_week_number week, calendar_year year
  from times
  where calendar_year IN (2004, 2005) and
   calendar_week_number between 20 AND 30) t
on (v.week = t.week AND v.Year = t.Year)
order by t.year, t.week;
PRODUCT_NAME           YEAR       WEEK DENSE_SALES
--------------- ---------- ---------- -----------
Bounce                 2004         20         801
Bounce                 2004         21     4062.24
Bounce                 2004         22     2043.16
Bounce                 2004         23     2731.14
Bounce                 2004         24     4419.36
Bounce                 2004         25           0
Bounce                 2004         26           0
Bounce                 2004         27     2297.29
Bounce                 2004         28     1443.13
Bounce                 2004         29     1927.38
Bounce                 2004         30     1927.38
Bounce                 2005         20      1483.3
Bounce                 2005         21     4184.49
Bounce                 2005         22     2609.19
Bounce                 2005         23     1416.95
Bounce                 2005         24     3149.62
Bounce                 2005         25     2645.98
Bounce                 2005         26           0
Bounce                 2005         27     2125.12
Bounce                 2005         28           0
Bounce                 2005         29     2467.92
Bounce                 2005         30     2620.17
```

For the four added rows that had no sales data, the `NVL` function transformed the NULL values to 0. This is how partitioned outer joins convert sparse data into a dense form. You may also choose to replace the NULL values with the most recent non-NULL values. To do this, you can add the `IGNORE NULLS` clause to the Oracle `LAST_VALUE` and `FIRST_VALUE` functions.

CERTIFICATION OBJECTIVE 11.03

Using the SQL MODEL Clause

It is common for Oracle users to process data using third-party tools, since Oracle SQL has traditionally lacked sophisticated modeling capabilities to produce complex reports. A basic example is the use of spreadsheets, which apply formulas to transform data into new forms. In previous versions of Oracle, in order to produce these spreadsheet-like reports, you needed to either download data into spreadsheet programs like Microsoft Excel or use dedicated multidimensional online analytical processing (OLAP) servers such as Oracle Express. For example, you might use Excel to convert your business data into rule-based business models, with the help of various macros. But third-party spreadsheet tools are cumbersome to use, and you need to expend considerable effort and time to constantly import updated Oracle data into the spreadsheet programs.

Oracle Database 10*g* offers the extremely powerful `MODEL` clause, which enables the use of SQL statements to categorize data and apply sophisticated formulas to produce fancy reports directly from within the database itself. You can now produce highly useful Oracle analytical queries, overcoming several drawbacks of Oracle SQL. With the new `MODEL` clause, you can use normal SQL statements to create multidimensional arrays and conduct complex interrow and interarray calculations on the array cells. Here, you'll learn how the `MODEL` clause produces its powerful results.

How the MODEL Clause Works

Oracle professionals commonly make heavy use of multiple table joins and unions when dealing with complex data warehousing data. These techniques help you peform very complex computations, but they are usually slow and computationally expensive. The `MODEL` enhancement enables you to perform complex enterprise-level computations.

The `MODEL` clause provide interrow calculation functionality by enabling you to create multidimensional arrays of your query data and letting you randomly access the cells within the arrays. The way the `MODEL` clause addresses individual cells is called *symbolic cell addressing*. The `MODEL` clause also performs *symbolic array computation*, by transforming the individual cells using formulas, which it calls *rules*.

The `MODEL` clause enables you to apply business models to your existing data. When you use the `MODEL` clause as part of a query, Oracle feeds the data retrieved by the query to the `MODEL` clause. The `MODEL` clause rearranges the data into a multidimensional array and applies your business rules to the individual elements of the array. From the application of various user-specified business rules, Oracle derives *updated* as well as newly *created* data. However, you won't actually *see* an array as your final form of the output, since Oracle will format the new and updated data into a row format when it delivers the `MODEL` clause's output to you.

The first step in a `MODEL`-based query is the creation of the multidimensional array. The following section explains the basis of the arrays created by the `MODEL` clause.

Creating the Multidimensional Arrays

The `MODEL` clause creates the multidimensional arrays that are at the heart of its functionality by mapping all the columns of the query that contains a `MODEL` clause into the following three groups.

- **Partitions** These are similar to the analytical function partitions described earlier in this chapter. Basically, a partition is a result handed to the `MODEL` clause by previous grouping operations. The `MODEL` clause is always separately applied to the data within each partition.
- **Dimensions** These are the same dimensions that you saw earlier in this chapter; for example, they might be time, region, and product.
- **Measures** Measures are the fact table data on which you are modeling your report, such as sales or inventories. You can look at the aggregate measure as consisting of a bunch of measure *cells*, with each of the cells identified by a unique combination of dimensions. For example, if sales is your measure, then the sales of detergents for the third quarter of 2004 in the New York region is one cell of the measure, since you can have only one such unique combination of your three dimensions: product (detergents), time (third quarter of 2004), and region (New York region).

The next section looks at how the `MODEL` feature uses rules to modify your multidimensional array data.

Transforming Array Data with Rules

A *rule* in the context of the `MODEL` clause is any business rule or formula you want to apply to the array data created by the `MODEL` clause. You may, for example, use a formula to forecast next year's sales on the basis of the preceding two years' sales data. You create a simple forecasting formula that expresses your business reasoning, and then pass it along to the `MODEL` clause as a rule.

You use the keyword RULES to indicate that you are specifying the rules that the MODEL clause must apply to its multidimensional array data. For example, you could specify a simple rule as follows:

```
MODEL
…
RULES
…
(sales['Kleenex', 2005] = sales['Kleenex', 2003] + sales['Kleenex', 2004]
…
```

This rule specifies that the sales of Kleenex for the year 2005 would be the sum of the sales of Kleenex in the years 2003 and 2004.

When you specify the RULES keyword, you may also want to indicate whether the rules you are specifying will be transforming existing data or inserting new rows of data. By default, the RULES keyword operates with the UPSERT specification. That is, if the measure cell on the left hand of a rule exists, Oracle will update it. Otherwise, Oracle will create a new row with the measure cell values. Here's an example:

```
MODEL
…
RULES UPSERT
sales ('Kleenex, 2005) = sales ('Kleenex, 2003') + sales ('Kleenex, 2004)
…
(MORE RULES HERE)
```

In this rules specification, if there is already a table or view row that shows the sales for Kleenex in the year 2005, Oracle will update that row with the values derived from applying the rule formula. If there is no such row, Oracle will create a new row to show the forecasted sales of Kleenex for the year 2005.

If you don't want Oracle to insert any new rows, but just update the existing rows, you can change the default behavior of the RULES clause by specifying the UPDATE option for all the rules, as shown here:

```
MODEL
…
RULES UPDATE
Sales ('Kleenex, 2005) = sales ('Kleenex, 2003') + sales ('Kleenex, 2004)
…
(MORE RULES HERE)
```

The previous two examples demonstrated how to apply different rule options at the MODEL clause level. You may also specify rule options at the individual rule level, as shown here:

```
RULES
(UPDATE sales ('Kleenex, 2005) = sales ('Kleenex, 2003') + sales ('Kleenex, 2004)
```

When you specify a rule option at the individual rule level as shown in this example, the use of the RULES keyword is optional.

exam watch

If you specify a rule option at the rule level, it will overrirde the RULES specification at the MODEL clause level. If you don't specify a rule option at the rule level, the MODEL level option applies to all the rules. If you don't specify an option at the MODEL level, the default UPSERT option will prevail.

You can specify that Oracle should evaluate the rules in either of the following two ways:

- `sequential order` Oracle will evaluate a rule in the order it appears in the MODEL clause.
- `automatic order` Rather than evaluating a rule based on its order of appearance in a list of several rules, Oracle will evaluate the rule on the basis of the dependencies between the various rules in the MODEL clause. If rule A depends on rule B, Oracle will evaluate rule B first, even though rule A appears before rule B in the list of rules under the RULES keyword.

on the job

Sequential order is the default order of processing rules in a MODEL clause.

Producing the Final Output

As its output, the MODEL clause will give the results of applying your rules to the multidimensional arrays it created from your table data. A MODEL-based SQL analytical query typically uses an ORDER BY clause at the very end of the query to precisely order its output.

You can use the optional RETURN UPDATED ROWS clause after the MODEL keyword to specify that only the new values created by the MODEL statement should be returned. These new values may either be updated values of a column or newly created rows.

exam watch

When I say that the MODEL clause will create or update rows, I strictly mean that the changes are shown in the MODEL clause output. The MODEL clause doesn't update or insert rows into the table or views. To change the base table data, you must use the traditional INSERT, UPDATE, or MERGE statements.

A MODEL Clause Example

Let's look at a simple SQL example that demonstrates the capabilities of the MODEL clause. Here's the query:

```
SQL> select country, product, year, sales
     from sales_view
 where country in ('Mexico', 'Canada')
 MODEL
 partition by (country) DIMENSION BY (product, year)
 measures (sale sales)
 rules
    (sales['Kleenex', 2005]     = sales['Kleenex', 2004] + sales['Kleenex',2003],
    sales['Pampers', 2005]      = sales['Pampers', 2004],
    sales['All_Products', 2005] = sales['Kleenex', 2005] + sales['Pampers',2005])
 order by country, product, year;
```

Sales units are the measure in this example. The query partitions the data by country and form the measure cells consists of product and year combinations. The three rules specify the following:

- Total sales of Kleenex in 2005 are forecast as the sum of Kleenex sales in the years 2003 and 2004.
- Total sales of Pampers in the year 2005 are forecast to be the same as the sales in 2004.
- Total product sales in 2005 are computed as the sum of the Kleenex and Pampers sales in 2005.

Here's the output generated by using the preceding SQL statement with the MODEL clause (the new data created by the MODEL clause is shown in boldface here):

```
COUNTRY              PRODUCT                YEAR     SALES
-------------------- --------------- ---------- ----------
Mexico                Kleenex               2002    2474.78
Mexico                Kleenex               2003    4333.69
Mexico                Kleenex               2004     4846.3
Mexico                Kleenex               2005    9179.99
Mexico                Pampers               2002   15215.16
Mexico                Pampers               2003   29322.89
Mexico                Pampers               2004   81207.55
Mexico                Pampers               2005   81207.55
Mexico                All_Products          2005   90387.54
```

```
Canada               Kleenex              2002     2961.3
Canada               Kleenex              2003    5133.53
Canada               Kleenex              2004     6303.6
Canada               Kleenex              2005   11437.13
Canada               Pampers              2002   22161.91
Canada               Pampers              2003   45690.66
Canada               Pampers              2004   89634.83
Canada               Pampers              2005   89634.83
Canada               All_Products         2005  101071.96
```

The SELECT clause first retrieves the product, year, and sales data for the two countries (Mexico and Canada) and feeds it into the MODEL clause. The MODEL clause takes this raw data and rearranges it into a multidimensional array, based on the values of the PARTITION BY (country) and DIMENSION BY (product and year) clauses. After the MODEL clause creates the array, it applies the three formulas listed under the RULES clause to the data. It finally produces the resulting row data, after ordering it by country, product, and year.

Note that the MODEL clause shows the original table or view data, as well as the new data that the MODEL clause has calculated from the three rules supplied in the MODEL clause. The MODEL clause applies the rules within each partition of data.

CERTIFICATION OBJECTIVE 11.04

Materialized View Enhancements

Materialized views have storage structures like regular Oracle tables, and they are used to hold aggregate or summary data. One of the biggest advantages of materialized views is that you can use them to precompute joins on commonly used tables, called the *detail tables*. Expensive joins and aggregates are precomputed and stored by Oracle in materialized views, also referred to as *summaries*. These materialized views are transparent to the end users, who still address the detailed base tables in their queries. The Oracle optimizer knows when a materialized view would offer superior results compared with addressing the base tables directly. Oracle uses the query-rewriting mechanism behind the scenes to automatically rewrite a user's query if it thinks that using a materialized view would give faster results.

A materialized view can include aggregations like SUM, COUNT(*), MAX, MIN, and any number of joins. You may index materialized views as well. If you aren't sure which materialized view to create, you can use the SQL Access Advisor to help you design and evaluate materialized views.

Once you create a materialized view, you have two main concerns:

- Refreshing the materialized views so they contain the latest data
- Ensuring that your query rewrite mechanism will use the materialized view to rewrite queries

In the next two sections, we'll look at the fast refresh feature and query rewriting mechanisms, and then look at the new Oracle Database 10*g* procedure that helps you optimize your materialized views.

Materialized View Fast Refresh Feature

Data in the base (detail) tables of a materialized view changes over time, due to various DML operations. Thus, a materialized view should be refreshed frequently to keep up with the changes in the underlying tables. There are several ways to refresh a materialized view. One of the best ways is to use the fast refresh method, which applies incremental changes to refresh materialized views. The fast refresh method of updating materialized views relies on the use of *materialized view logs*. Materialized view logs are created on the underlying base tables, not on the materialized views themselves.

Here is a basic materialized view log creation statement:

```
SQL> CREATE MATERIALIZED VIEW LOG ON sales WITH ROWID
    (prod_id, cust_id, time_id, channel_id, promo_id, quantity_sold, amount_sold)
    INCLUDING NEW VALUES;
```

exam watch

For fast refresh of materialized views, the definition of the materialized view logs must normally specify the `ROWID` clause.

The following are some of the important restrictions on using the fast refresh method (note that some restrictions are general; others are specific to the type of materialized view—whether it is based on aggregates or joins):

- The materialized view must not use SYSDATE, ROWNUM, RAW, or LONG datatypes.
- A materialized view can't have `GROUP BY` clauses.
- You must include the ROWIDs of all the tables in the `FROM` list in the `SELECT` list of the query.
- You must have materialized view logs with ROWIDs for all the base tables.

The Query Rewrite Feature

Automatic query rewriting is the key feature that makes materialized views a faster means of processing complex data warehousing-type queries compared to the direct use of the base tables. Oracle takes your query against the base tables and rewrites it to use the underlying materialized views, if Oracle's query optimizer decides it's a faster way to return the query results. However, query rewriting is not guaranteed, and Oracle fails to rewrite queries on occasion. When this happens, Oracle can't use the underlying materialized views.

In order to ensure a query rewrite, a query must meet the following conditions:

- QUERY_REWRITE_ENABLED = TRUE (the default value in Oracle Database 10g)
- The underlying materialized views must be enabled for query rewrite, by using the `ENABLE QUERY REWRITE` clause. You can specify this clause either with the `ALTER MATERIALIZED VIEW` statement or when you create the materialized view.
- You must set the query rewrite integrity level appripriately by specifying the relevant value for the `QUERY_REWRITE_INTEGRITY` parameter. For example, if a materialized view is not fresh you set query rewrite integrity to `ENFORCED,` then Oracle won't use the materialized view. To enable query rewrite in this situation and cases where you have constraints that haven't been validated, you need to set the integrity level to a less restrictiv level of granularity such as `TRUSTED` or `STALE_TOLERATED`.
- The database must be able to drive either all or part of the results requested by the query from the precomputed result stored in the materialized view.

exam watch

In Oracle Database 10g, the `ENABLE_QUERY_REWRITE` parameter is `TRUE` by default. You must, however, ensure that the `OPTIMIZER_FEATURES_ENABLE` initialization parameter is set to 10.0.0 or higher. The `QUERY_REWRITE_INTEGRITY` initialization parameter still has the same default value (`ENFORCED`).

Here's a brief list of the important restrictions on using the query rewrite feature:

- You can't refer to any RAW or LONG RAW datatypes and object REFs.
- You can't use any nonrepeatable expressions like SYSDATE and ROWNUM.
- If you include a column or expression in the `GROUP BY` clause, it must also be a part of the `SELECT` list.

The SQL Access Advisor can help you by suggesting ideal materialized views for the detail tables in a query, and this advisor can also help you create the materialized views. Once you create the materialized views, you can use various procedures of the `DBMS_MVIEW` package to optimize your materialized views. Here is a summary of the two key procedures of the `DBMS_MVIEW` package that help in understanding the capabilities of materialized views and potential materilaized views, especially concerning rewrite availability:

- **`EXPLAIN_MVIEW`** This procedure tells you what kinds of query rewrites are posible. It will also tell you why a certain materialized view isn't fast refreshable.
- **`EXPLAIN_REWRITE`** This procedure tells you why a query failed to rewrite. If the query rewrites, the procedure will tell you which materialized views will be used

exam watch

The `DBMS_ADVISOR.TUNE_MVIEW` procedure recommends materialized views with optimized defining queries, decomposition of nonrefreshable materialized views, and fixes for materialized view log problems.

The `DBMS_ADVISOR` package offers you the new procedure `TUNE_MVIEW`, which you can use in the SQL Access Advisor. ?This is a new Oracle Database 10g procedure that helps you alter a materialized view to ensure query rewriting wherever it is possible. The procedure will let you decompose a materialized view into two or more materialized views or to restate the materialized view in a way that is more conducive to a fast refresh and query rewrite.

The `DBMS_ADVISOR.TUNE_MVIEW` procedure will optimize the materialized view in such a way that it can use several types of query rewrites. This procedure will also provide you with the necessary statements to ensure a fast refresh. Let's look at the `DBMS_ADVISOR.TUNE_MVIEW` procedure in detail in the following section.

Materialized View Optimization with the TUNE MVIEW Procedure

The new Oracle Database 10*g* `TUNE_MVIEW` procedure of the `DBMS_ADVISOR` package helps you in fixing problematic materialized views, where either a fast refresh or a query rewrite is not happening as you would like. The procedure takes a `CREATE`

MATERIALIZED VIEW as its input and performs the following materialized view tuning functions:

- Redefine materialized views so they refresh fast as well as use query rewrite, if the materialized views are currently not using these features for some reason.
- Fix materialized view log problems that may be keeping the view from using the fast refresh mechanism. These problems include verifying that a materialized view log exists in the first place. If the materialized view log exists, it may have problems, like missing columns, which prevent its use by a materialized view.
- If a materialized view turns out to be nonrefreshable, break it up into submaterialized views that are eligible for a fast refresh. The parent materialized view can then reference the submaterialized views, thus getting around its inability to refresh fast.

If a materialized view isn't fast refreshable, the data in the materialized view will become stale, thus making your queries progressively worthless. When you find that a materialized view isn't fast refreshable, it's usually because one or more of the fast refresh restrictions aren't satisfied. The DBMS_ADVISOR.TUNE_MVIEW procedure provides you with the necessary SQL statements that you need to implement to ensure that you can fast refresh your materialized views.

on the job

The DBMS_MVIEW.EXPLAIN_MVIEW procedure tells you why you can't refresh a materialized view. The DBMS_ADVISOR.TUNE_MVIEW procedure tells you how to make the materialized view eligible for a fast refresh. In addition, the DBMS_ADVISOR.TUNE_MVIEW procedure also makes recommendations to enable a query rewrite.

You use the DBMS_ADVISOR.TUNE_MVIEW procedure as follows, either before creating a new materialized view or when you are tuning an existing materialized view:

```
begin
dbms_advisor.tune_mview (:task_name,
'CREATE MATERIALIZED VIEW test_mv
REFRESH FAST WITH ROWID ENABLE QUERY REWRITE
AS SELECT DISTINCT prod_name, prod_type
From products');
end;
```

The preceding code will populate the new DBA_TUNE_MVIEW view, which you must query to see how you can change your materialized view to make it fast refreshable as well as capable of using query rewrite. The DBA_TUNE_MVIEW view has the following structure:

```
SQL> desc dba_tune_mview
 Name                                      Null?    Type
 ----------------------------------------- -------- -------------
 OWNER                                              VARCHAR2(30)
 TASK_NAME                                          VARCHAR2(30)
 ACTION_ID                                 NOT NULL NUMBER
 SCRIPT_TYPE                                        VARCHAR2(14)
 STATEMENT                                          CLOB
SQL>
```

You use the TASK_NAME column value to identify and query a particular TUNE_MVIEW recommendation. (Make sure you provide a value for the TASK_NAME variable that I highlighted in the previous PL/SQL code block.) The ACTION_ID column shows the command order number. The SCRIPT_TYPE column can take values of CREATE or DROP (or UNKNOWN). The CREATE value is for the new materialized view recommendation. The DROP value shows the materialized view that the TUNE_MVIEW procedure wants you to drop. The STATEMENT column of the view shows the recommended materialized view changes that make your materialized view eligible for a fast refresh and a query rewrite. If you wish, you can use the DBMS_ADVISOR.GET_TASK_SCRIPT procedure to output the recommendations to a text file.

Here's the basic syntax of a query on the DBA_TUNE_MVIEW view:

```
SQL> select statement
     from dba_tune_mview
     where task_name = :task_name
     order by script_type, action_id;
```

If the original materialized view statement isn't eligible for fast refresh, the DBMS_ADVISOR.TUNE_MVIEW procedure suggests an alternate way of defining your materialized view. In this case, the DBMS_ADVISOR.TUNE_MVIEW procedure might recommend the following changes to make your materialized view eligible for a fast refresh of its materialized view logs. (Note that the new materialized view recommendation replaces the DISTINCT clause in the original materialized view with the COUNT(*) and GROUP BY clauses.)

```
SQL> create materialized view test_mv
     refresh fast with rowid enable query rewrite
     as select prod_type t,
                    prod_name  p, count(*)
     from products
     group by prod_type, prod_name
```

Let's look at a couple of examples that illustrate how you can use the DBMS_ ADVISOR.TUNE_MVIEW procedure to enable the fast refresh of a recalcitrant materialized view.

Creating Materialized View Logs

As you know, one of the restrictions on the fast refresh feature is that you must include the ROWIDs of all tables that are in the FROM list in your SELECT list. Thus, if a certain statement fails due to noninclusion of the ROWIDs of the tables in the FROM list, the DBMS_ADVISOR.TUNE_MVIEW procedure will suggest the inclusion of the ROWIDs, as shown in the following example.

```
SQL> create materialized view test_mv
     build immediate refresh fast enable query rewrite
     as select  e.ROWID r1, d.ROWID r2,
     e.first_name, d.department_name
     from departments d, employees e
     where e.department_id = d.department_id;
```

The third line shows how you can use the ROWIDs to modify the materialized view. This materialized view will now be eligible for query rewrite, as long as you make sure that you create the following pair of materialized view logs, one for each of the tables in the materialized view.

```
SQL> create materialized view log onemployees
     with sequence, rowid
     including new values;
SQL> create materialized view log on  departments
     with sequence, rowid    including new values
```

Decomposing Materialized Views

Sometimes, a materialized view isn't fast refreshable because it violates one of the restrictions for a fast refresh, like having an unflattenable *inline view*. In cases like this, the TUNE_MVIEW procedure helps you by making recommendations for the decomposition of the materialized view into two nested submaterialized views.

The parent materialized view will refer to the submaterialized view that you create. Again, you must create materialized view logs on each of the tables in the materialized view in order to make it eligible for the fast refresh feature. The following types of situations call for a materialized view decomposition:

- A subquery in the `WHERE` clause
- Use of set operations like `UNION`, `UNION ALL`, `INTERSECT`, and `MINUS`
- Use of inline views

Partition Change Tracking Enhancements

You generally use materialized views in a data warehouse setting. Thus, it's no surpise that many materialized views will have partitioned base tables, since data warehouse tables are large, and hence, usually partitioned. Oracle's partition change tracking (PCT) feature lets you figure out which rows of a materialized view are affected by a change in a base table's partitioning. Why is this ability important? Well, Oracle doesn't perform a query rewrite on a materialized view it considers stale, and it considers a materialized view stale if the base table partitioning has changed.

The PCT feature, by maintaining links to table partitions and materialized view rows, helps the materialized views handle the aftermath of a partitioning change in the base tables. By enabling Oracle to consider only a part of a materialized view as stale, PCT will enable the use of the query rewrite feature by letting it use those rows that are still fresh, provided you use the `QUERY_REWRITE_INTEGRITY=ENFORCED` or `TRUSTED` mode.

on the job

Any time you change a base table's partition scheme, the relevant materialized view rows become stale.

exam watch

A PCT-based materialized view refresh will minimize the amout of refreshes and maximize the use of the query rewrite feature.

In Oracle Database 10g, there are several enhancements with regard to the PCT feature. Let's briefly review these enhancements in the following sections.

List-Partitioning Scheme

In previous versions, you could use the PCT feature only for partitioned base tables using the range and range-hash partitioning schemes. Now, you can also use it for detail tables using the list-partitioning scheme.

ROWID Columns as Partition Markers

Materialized view joins sometimes make ROWID references in their defining queries You can now use a ROWID column as a PCT column in order to help identify table partitions during a PCT refresh.

exam watch

Oracle Database 10g extends the use of PCT to list-partitioned tables, enables the use of ROWID columns as partition markers, and lets you use a PCT refresh if a materialized view contains a join-dependent expression.

Join Dependency

You can now use a PCT-based refresh if your MV contains a *join-dependent expression* of one of its tables. A table is a join-dependent table if you equijoin it with a partitioned base table on its partioning key column. An expression consisting of columns from the resulting equijoin is a join-dependent expression.

Truncating Materialized View Partitions

In previous versions, PCT used `DELETE` statements to remove materialized views. In Oracle Database 10g, the database truncates materialized view partitions, as long as the following conditions apply:

- You are limited to range partitioning only (for both the base tables and the materialized view), and the partitioning bounds must be the same for the materialized view and its base tables.
- There should be a one-to-one relationship between the two sets of partitions (base table paritions and the materialized view partitions).
- You must partition the materialized view on its single PCT key column.
- You shouldn't refresh on the basis of atomic transactions.

Forcing a Refresh

The PCT feature automatically refreshes your materialized view when there are partition maintenance operations in the underlying base tables. However, you may sometimes wish to manually use a PCT-based refresh, even in the absence of any base table partitioning scheme modifications. The `DBMS_MVIEW.REFRESH` procedure has a new option, `P`, to indicate a *forced* PCT-based refresh. Here's the syntax:

```
execute dbms_mview.refresh(mview_name, method =>'P')
```

Other Materialized View Enhancements

In addition to the introduction of the TUNE_MVIEW procedure to help with the fast refresh and query rewrite features, and improvements to the PCT feature, Oracle Database 10*g* provides several other enhancements related to materialized views. I summarize these enhancements in the following sections.

Materialized View Execution Plans

In Oracle Database 10*g*, both the plan tableused by the explain plan feature and the V$SQL_PLAN view will show you if a particular query is using a materialized view. In Oracle Database 10*g*, you can find out if a materialized view was accessed directly or if Oracle rewrote the query in order to use the materialized view. The new feature here is that now you can clearly tell if Oracle is using a materialized view as a result of a query rewrite or because the programmer specified it.

exam watch

The explain plan feature shows you whether a materialized view is being accessed as a result of a query rewrite or because you specified direct materialized view access.

Here's an example that shows an execution plan that indicates how a materialized view is being used as result of the query rewrite. If you don't see the keyword REWRITE, it means that the materialized view was accessed directly.

```
Query Plan
SELECT STATEMENT
 SORT ORDER BY
  MATERIALIZED VIEW REWRITE ACCESS FULL EMP_INFO
```

exam watch

The REWRITE_OR_ERROR hint stops the execution of a SQL statement if a query doesn't rewrite.

The REWRITE_OR_ERROR Hint

If a planned query rewrite doesn't take place, Oracle will end up running the original query. Since the whole point of rewriting the query was to make a slow query fast, you may not want this to happen. Oracle Database 10*g* contains a new optimizer hint called REWRITE_OR_ERROR, which forces a query to error out if it can't rewrite the query:

```
select  /*+ REWRITE_OR_ERROR */
```

Instead of running the original query, the hint throws the following error and stops the execution of the SQL statement.

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

New Columns in the REWRITE_TABLE

In the previous section, you saw how a `REWRITE_OR_ERROR` hint will stop the execution of a query that failed to rewrite. In cases like this, you can use the `DBMS_MVIEW.EXPLAIN_REWRITE` procedure to find out why the query failed to rewrite (the procedure also tells you which materialized view will be used if the query does rewrite). Using the output from the execution of this procedure, you can find out what you need to do in order to make the query rewrite, if that is at all possible.

Following is the syntax for using the `EXPLAIN_REWRITE` procedure. Note that this version is for when you want to create a *table* to hold the procedure's output. You can also use a VARRAY instead of a table if you want to access the procedure's output directly.

```
dbms_mview.explain_rewrite (
    query           IN [VARCHAR2 | CLOB],
    mv              IN VARCHAR2,
    statement_id    IN VARCHAR2;
```

To obtain the output into a table, you must run the utlxrw.sql script (located in the $ORACLE_HOME/rdbms/admin directory) before calling `EXPLAIN_REWRITE`. This script creates a table named REWRITE_TABLE in the current schema.

In order to view the results of the `EXPLAIN_REWRITE` procedure, first create the REWRITE_TABLE table, using the utlxrw.sql script, as shown here:

```
SQL> @c:\oracle\product\10.1.0\Db_1\RDBMS\ADMIN\utlxrw.sql
Table created.
SQL>
```

Here's the structure of the REWRITE_TABLE:

```
SQL> desc rewrite_table
 Name                                      Null?    Type
 ----------------------------------------- -------- -----------------------
 STATEMENT_ID                                       VARCHAR2(30)
 MV_OWNER                                           VARCHAR2(30)
 MV_NAME                                            VARCHAR2(30)
 SEQUENCE                                           NUMBER(38)
 QUERY                                              VARCHAR2(2003)
 MESSAGE                                            VARCHAR2(512)
 PASS                                               VARCHAR2(3)
 MV_IN_MSG                                          VARCHAR2(30)
 MEASURE_IN_MSG                                     VARCHAR2(30)
 JOIN_BACK_TBL                                      VARCHAR2(30)
 JOIN_BACK_COL                                      VARCHAR2(30)
```

```
ORIGINAL_COST                                     NUMBER(38)
REWRITTEN_COST                                    NUMBER(38)
FLAGS                                             NUMBER(38)
RESERVED1                                         NUMBER(38)
RESERVED2                                         VARCHAR2(10)
```

Four REWRITE_TABLE columns are *new* in Oracle Database 10*g*:

- JOIN_BACK_TBL provides the name for the table with which a join back operation was performed in the materialized view.
- JOIN_BACK_COL provides the name of the column involved in the join back operation.
- ORIGINAL_COST shows the cost of the prematerialized view query.
- REWRITTEN_COST is the cost of the rewritten query, if there was one. If not, this column will be zero.

The MESSAGE column shows the `EXPLAIN_REWRITE` procedure error message. If it contains multiple materialized views, you'll see a separate row for each materialized view.

exam watch

The REWRITTEN_COST column of the REWRITE_TABLE shows the cost for any materialized view that was used in a query rewrite. The column will show a zero if there was no rewrite of a query or if a diffent materialized view was used, even if there was a query rewrite.

Materialized Join View Enhancements

Oracle Database 10*g* contains enhancements to *materialized join views* (MJVs), which contain only joins (and not aggregates). In Oracle Database 10*g*, you can now conduct a fast refresh of a materialized join view, under the following conditions:

- If the materialized join view contains a self join in its `FROM` clause, you must include the ROWID columns for each instance in the `SELECT` list. The materialized join view log must contain all the ROWID coumns as well.
- If the materialized join view has an inline view (or a named view), the database must be able to perform complete view merging. Once the view merging is done, the `SELECT` list must have the ROWID columns for all the tables in the `FROM` clause.

- If you are dealing with remote tables in materialized join views, make sure that all of the tables are on the same site. The SELECT list must have the ROWID columns for all the tables in the FROM clause.

exam watch

For a fast refresh of materialized join views—whether they use self joins, inline views, or remote tables—you must create materialized view logs on each of the base tables. The materialized view logs must also contain the ROWID column.

Partition Maintenance Operations

In previous versions of Oracle, when dealing with partitioned materialized views, you needed to perform partition maintenance operations by using the ALTER TABLE commands against the materialized view container tables, rather than the materialized views themselves. In Oracle Database 10g, you can issue commands that truncate, exchange, or drop partitions by using the ALTER MATERIALIZE VIEW statement. For example, you can drop a partition from a partitioned materialized view in the following manner:

```
alter materialized view <mv_name>
drop partition <partition_name>
```

Materialized View Refresh Using Trusted Constraints

Oracle doen't enforce functional dependencies in dimensions. Similarly, it doesn't enforce primary key and foreign key relationships that are RELY constraints. As a result of this behavior, when you refresh a materialized view, you may end up with inconsistent results.

When you create a materialized view, you can specify the policy that Oracle should adopt when it encounters constraints during a materialized view refresh. If you use the ENFORCED option, Oracle won't refresh any materialized view with constraint violations. If you use the TRUSTED constraints option, on the other hand, Oracle will perform a materialized view refresh. However, Oracle will set the *new column* UNKNOWN_TRUSTED_FD in the DBA_MVIEWS view to a value of Y following the refresh. This indicates that the materialized view is now in an unknown state, because it used trusted functional dependencies that were not enforced by Oracle during the refresh process.

exam watch

If you use the TRUSTED option, the resulting materialized views are in an unknown state, and you can use them for a query rewite in a TRUSTED or a STALE_TOLERATED mode only.

INSIDE THE EXAM

The exam will test your knowledge of the enhancements to the DBMS_ADVISOR package. You must understand how the new TUNE_MVIEW procedure helps fast refresh and query rewrite when you use materialized views. You must remember the specific ways in which the TUNE_MVIEW procedure can help in tuning materialized views (decomposition of nonwritable materialized views, for example). Under what conditions is it good to decompose a materialized view? How does the TUNE_MVIEW procedure contributein promoting a fast refresh of a query? You must also be conversant with the important procedures of the DBMS_MVIEW package. What does the EXPLAIN_MVIEW procedure help you do?

The exam will test your knowledge of the MERGE statement enhancements. Exactly what rows will the DELETE clause delete in a MERGE statement?

You most likely are going to encounter a question on the new partitioned outer join feature. Know that it is simply an extension of the outer join feature in previous versions.

The MODEL clause is too important not to be touched by the exam. You must thoroughly understand the concepts of partitions, measures, and dimensions. What role do rules play in a MODEL clause? What are the default semantics of rules in a MODEL clause (UPSERT)? What is the difference between sequential order and automatic order when you are evaluating a set of rules?

Expect a question about the PCT feature on the exam. What enhancements has Oracle introduced for PCT in Oracle Database 10*g*? What are the preconditions for using the TRUNCATE PARTITION command when you are performing PCT operations?

You must know how to enable the automatic query rewriting feature in Oracle Database 10*g* (hint: it is automatic!). What hint will stop query execution if the query fails to rewrite? Expect a query on materialized join views as well. What are the conditions under which Oracle supports fast refresh for materialized join views?

CHAPTER SUMMARY

This chapter started by reviewing the two new enhancements to the MERGE command in a SQL statement. You then learned about the difference between dense and sparse data, and how to densify data using the new partitioned outer join enhancement.

The SQL MODEL clause offers you tremendous analytical capabilities when dealing with interrow calculations in a data warehouse setting. This chapter provided a basic review of the MODEL clause enhancement, after first reviewing some essential data warehousing concepts.

There are several important Oracle Database 10*g* enhancements pertaining to materialized views. Important among these is the new `TUNE_MVIEW` procedure of the `DBMS_ADVISOR` package. You saw how you can ensure (almost!) a fast refresh of materialized views, as well as increase query rewrites using the `DBMS_ADVISOR.TUNE_MVIEW` procedure. You also reviewed materialized join view enhancements, materialized view refreshing using trusted constraints, and PCT enhancements.

✓ TWO-MINUTE DRILL

Enhancements in the MERGE Statement

- ❑ The MERGE statement enables you to perform conditional update and insert operations when loading data from a source table into another table.
- ❑ In Oracle Database 10*g*, you can use a WHERE clause in a MERGE statement's UPDATE or INSERT clause to conditionally update or insert data.
- ❑ In addition to the INSERT and UPDATE clauses, you can now use a DELETE clause in a MERGE statement.
- ❑ The DELETE clause in a MERGE statement will delete only rows included in join condition (specified by the ON clause).
- ❑ The DELETE clause of the MERGE statement evaluates only the post-updated values of rows.

Using Partitioned Outer Joins

- ❑ The partitioned outer join is an extension of the Oracle SQL outer join concept.
- ❑ Fact tables contain the data of an organization, and dimension tables contain descriptions of the dimensions you use to categorize data into meaningful entities.
- ❑ Dimension tables are organized on a hierarchical basis, enabling you to perform roll up and drill down operations.
- ❑ The analytical process follows a processing order consisting of the grouping, calculation (analysis), and output ordering steps.
- ❑ Dense data doesn't contain gaps in any of the dimensional cells.
- ❑ Sparse data is data that contains no values for some combinations of dimension values.
- ❑ Sparse data presents analytical and report formatting problems.
- ❑ Dense data leads to better-performing analytical queries, especially when you're dealing with time-series data.
- ❑ Partitioned outer joins help turn sparse data into dense data by replacing missing values in the data.
- ❑ You create a partitioned outer join by adding the PARTITION BY clause to the outer join clause.

Using the SQL MODEL Clause

- ❑ The SQL MODEL clause provides you with spreadsheet-like output capabilities.
- ❑ The MODEL clause provides interrow and interarray analytical capabilities.
- ❑ The MODEL clause enables you to perform symbolic cell addressing and symbolic array computations.
- ❑ The MODEL clause doesn't change table data, unless you explicitly use a separate UPDATE, INSERT, or MERGE statement to modify table data with the MODEL clause's output.
- ❑ The MODEL clause first creates multidimensional arrays from the raw data derived from using the initial SELECT statement.
- ❑ Oracle uses partitions, measures, and dimensions to derive the multidimensional data arrays from table data.
- ❑ The MODEL clause then applies simple or complex business rules to the array data, using rules that you can specify.
- ❑ A rule is any business rule or formula that you apply to the array data derived by the MODEL clause.
- ❑ A rule can update or insert data. The default rule semantics at the MODEL level use the UPSERT operation.
- ❑ You can use UPDATE instead of the default UPSERT rule semantics.
- ❑ If you specify a rule level option, it will override an option specified at the rules level.
- ❑ You can specify sequential order or automatic order for evaluating multiple rules in a MODEL clause.
- ❑ Sequential order is the default rule processing order.
- ❑ The RETURN UPDATED ROWS clause will return only any new values created by the MODEL clause.
- ❑ The MODEL clause output shows both the original data and the data that the MODEL clause has inserted or updated.

Materialized View Enhancements

- ❑ Oracle Database 10g enables query rewriting by default.
- ❑ The DBMS_MVIEW.EXPLAIN_REWRITE procedure tells you why Oracle isn't rewriting a query to take advantage of an existing materialized view.

- ❑ The `DBMS_ADVISOR.TUNE_MVIEW` procedure helps fast refresh a materialized view, as well as enhance query rewriting.
- ❑ The `DBMS_ADVISOR.TUNE_MVIEW` procedure helps in fixing problems with materialized view logs.
- ❑ The `DBMS_ADVISOR.TUNE_MVIEW` procedure also can suggest breaking up a materialized view into submaterialized views to ensure a fast refresh of the parent materialized view.
- ❑ The `DBA_TUNE_MVIEW` view contains the results of executing the `TUNE_MVIEW` procedure.
- ❑ The STATEMENT column of the `DBA_TUNE_MVIEW` view shows the recommended materialized view changes.
- ❑ The `DBMS_ADVISOR.TUNE_MVIEW` procedure can suggest that you include the ROWIDs of all tables in the `SELECT` list of a materialized view.
- ❑ A materialized view decomposition into parent and submaterialized views may be called for when you have subqueries in the `WHERE` clause or use an inline view which you can't flatten.
- ❑ The `EXPLAIN PLAN` statement will now show you if the materialized view use is because of a query rewrite or developer specification.
- ❑ The new `REWRITE_OR_ERROR` hint stops the execution of a SQL statement if query rewriting doesn't take place.
- ❑ There are four new columns in the REWRITE_TABLE to help you figure out the cost of an original and rewritten query.
- ❑ Once you create materialized view logs with a ROWID column, you can now conduct a fast refresh of a materialized view join that contains self joins, inline views, or remote tables.
- ❑ You can directly use the `ALTER MATERIALIZED VIEW` statement to perform partitioning maintenance operations on partitioned materialized views.
- ❑ Oracle will conduct a fast refresh when dealing with trusted constraints, but it will put the materialized view into an *unknown state*.
- ❑ You must use the `TRUSTED` or `STALE_TOLERATED` mode when dealing with materialized views that are in an unknown state due to the use of trusted functional dependencies and constraints.

- ❑ The partition change tracking (PCT) feature is now applicable to list-partitioned tables or when your materialized view contains join-dependent expressions.
- ❑ PCT operations truncate data now, instead of deleting it.
- ❑ You can now manually use a PCT-based refresh, even in the absence of partition operations, by specifying the new option when executing the `DBMS_MVIEW.REFRESH` procedure.

SELF TEST

The following questions will help you measure your understanding of the material presented in this chapter. Read all the choices carefully because there might be more than one correct answer. Choose all correct answers for each question.

Enhancements in the MERGE Statement

1. What does the `MERGE` statement do if an identical row already exists in the table?

   A. It deletes the existing row first.

   B. It inserts the duplicate row.

   C. The `MERGE` operation fails with an error message.

   D. It performs an update, although there won't be any difference in the row ultimately.

2. When using the `MERGE` statement, what will the `DELETE` clause delete?

   A. All rows that satisfy the `DELETE` clause's `WHERE` condition

   B. All rows that satisfy the `DELETE` clause's `WHERE` condition, provided they have been updated by the `UPDATE` clause

   C. All rows that satisfy the `DELETE` clause's `WHERE` condition, provided they have been newly inserted by the `INSERT` clause

   D. All rows that fail to satisfy the `UPDATE` clause

3. Which one of the three clauses—`INSERT`, `DELETE`, and `UPDATE`—fires first in a `MERGE` statement if all three of them are present?

   A. `INSERT` clause

   B. `DELETE` clause

   C. `UPDATE` clause

   D. Depends on the order in which you specify the three operations

4. Which rows will the `DELETE` clause in a `MERGE` statement delete?

   A. Rows modified by the `UPDATE` clause

   B. Rows inserted by the `INSERT` clause

   C. Rows neither updated nor inserted by the `MERGE` statement

   D. Rows selected by the `WHERE` clause embedded inside the `INSERT` clause

Using Partitioned Outer Joins

5. The partitioned outer join is especially beneficial for which type of data?
   A. Time-series data
   B. Cross-section data
   C. Summary data
   D. Analytical data

6. What can the output of a partitioned outer join be considered as?
   A. `UNION` of several outer joins, each join being between a partition and the other table(s) in the join
   B. `UNION` of several equijoins, each join being between a partition and the other table(s) in the join
   C. `UNION` of several self joins, each join being between a partition and the other table(s) in the join
   D. `UNION` of several inner joins, each join being between a partition and the other table(s) in the join

7. Which of the following is true when you densify sparse data by using partitioned outer joins?
   A. The missing data must be filled in by zeros.
   B. You can use the `IGNORE NULLS` clause with the `LAST_VALUE` function to replace the missing values with the most recent non-NULL value in that column.
   C. The missing data must be filled in by NULLs.
   D. You must provide the missing data by updating the column with the missing values.

8. What will the `PARTITION BY` clause in a partitioned outer join statement do?
   A. Partition the underlying table, using Oracle's partitioning option
   B. Partition the output rows into equal segments
   C. Partition the table into equal subpartitions
   D. Partition the rows in the output based on the expression your provide within the clause

Using the SQL MODEL Clause

9. What is symbolic cell addressing?
   A. The way the `MODEL` clause makes statistical calculations
   B. The way the `MODEL` clause addresses statistical symbols

C. The way the MODEL clause addresses the individual cells of an array
D. The way the MODEL clause addresses the rules

10. What can you derive by using a MODEL clause?

A. Both updated and deleted data
B. Both updated and newly created data
C. Only updated data
D. Only newly changed data

11. By default, the RULES keyword operates with which specification?

A. UPDATE specification
B. INSERT specification
C. UPDATE and UPSERT specifications
D. UPSERT specification

12. What will the RETURN UPDATED ROWS keyword in a MODEL clause do?

A. Return only the new rows, not the updated rows
B. Return only the updated rows, not the new rows
C. Return both the updated and the new rows
D. Return all rows that are going to be deleted

Materialized View Enhancements

13. What does the fast refresh method of updating materialized views always use?

A. Incremental changes
B. Decomposed submaterialized views
C. Query rewrite
D. Materialized view logs

14. Under some conditions, for fast refresh to occur, which of the following must be true?

A. Materialized view logs must specify the ROWID clause.
B. The materialized view must use GROUP BY clauses.
C. The materialized view must use ROWIDs.
D. The materialized view must use ROWNUMs.

15. Which of the following will tell you why a certain materialized view isn't fast refreshable?

A. `EXPLAIN_REWRITE` procedure
B. `TUNE_MVIEW` procedure
C. `MVIEW_EXPLAIN` procedure
D. `EXPLAIN_MVIEW` procedure

16. What does the `TUNE_MVIEW` procedure do?

A. Automatically creates any necessary materialized view logs
B. Recommends the creation of any necessary materialized view logs
C. Automatically creates the materialized view
D. Automatically conducts a fast refresh of a materialized view

17. What does the `DBA_TUNE_MVIEW` view show?

A. The results of executing the `TUNE_MVIEW` procedure
B. The output of the `CREATE_MVIEW` procedure
C. The output of the `EXPLAIN_MVIEW` procedure
D. The output of both the `TUNE_MVIEW` and `EXPLAIN_MVIEW` procedures

18. Which of the following helps you identify your statement in the `DBA_TUNE_MVIEW` view?

A. The STATEMENT variable
B. The STATEMENT_ID column
C. The VIEW_ID column
D. The TASK_NAME variable

19. What does the `REWRITE_OR_ERROR` hint do?

A. Rewrites the query if it doesn't lead to any errors
B. Stops executing a query if it can't rewrite it
C. Sends out a report of all the rewrite errors
D. Enforces a query rewrite even if it leads to an error

20. In Oracle Database 10*g*, which of the following is true regarding the partition change tracking feature?

A. It has been extended to Oracle partitions.
B. It has been extended to rule partitioning.
C. It has been extended to hash partitioning.
D. It has been extended to list partitioning.

LAB QUESTION

Compare the results of a regular outer join with the new partitioned outer join. For this lab exercise, you need to use the SH schema in the sample schemas supplied by Oracle as part of your Oracle software.

1. Create a small table with sales data for various years for two products.
2. Create another table with just time data, so you can join the first table with this one.
3. Use a traditional outer join and check the results.
4. Use a partitioned outer join and compare its results with those from step 3.

SELF TEST ANSWERS

Enhancements in the MERGE Statement

1. ☑ **D.** The MERGE statement will perform the update, but the row will not change its values.
   ☒ **A** is wrong because the MERGE statement always performs an UPDATE operation first. **B** is wrong because the INSERT statement doesn't fire. **C** is wrong because the existence of an identical row will not lead to any errors.

2. ☑ **B.** The DELETE clause in a MERGE statement will delete all rows that meet the WHERE condition, subject to the important provision that these rows must have been updated by the UPDATE clause in the MERGE statement.
   ☒ **A** is wrong because the DELETE clause will not delete all rows that satisfy the WHERE clause, but only those that have been updated prior to the DELETE operation. **C** and **D** indicate the wrong rows.

3. ☑ **C.** In a MERGE statement, the UPDATE operation always takes place first.
   ☒ **A, B,** and **C** are wrong because the UPDATE operation is performed first.

4. ☑ **A.** The DELETE clause in a MERGE statement will delete only those rows that are modified by the UPDATE clause.
   ☒ B and **D** are wrong because the DELETE clause will not delete any rows that are inserted as a result of the INSERT operation. **C** is wrong because the DELETE clause will lead to the deletion of only those rows that have been updated because of the UPDATE operation.

Using Partitioned Outer Joins

5. ☑ **A**. Partitioned outer joins help you transform sparse data into dense data. Since having dense data is very important for time-series based data, this is the right answer.
   ☒ Although you can use partitioned outer joins with the data mentioned in **B, C,** and **D**, they are most useful for time-series data. Alternative **D** could confuse some, since you might argue that analytical data includes time-series data.

6. ☑ **A.** You could view the output of a partitioned outer join as a union of several smaller outer joins between each of the partitions and the other table(s) in the join operation.
   ☒ **B, C,** and **D** are wrong since they refer to equijoins, self joins, and inner joins, whereas the partitioned outer join performs only *outer* joins.

7. ☑ **B.** You can use the IGNORE NULLS clause with the LAST_VALUE function to replace the missing values with the most recent non-NULL value in that column.

☒ **A** and **C** are wrong because you are not required to use zeros or NULLs to fill in missing values. **D** is wrong because you don't provide the values for the missing data—you use the partitioned outer joins to do that job for you.

8. ☑ **D.** The PARTITION BY clause will partition the rows in the output based on the expression you provide inside the PARTITION BY clause.
☒ **A** and **C** are wrong because the PARTITION BY clause doesn't actually partition the tables. **B** is wrong because the output isn't partitioned into equal segments, but rather is partitioned on the basis of the expression you provide within the PARTITION BY clause.

Using the SQL MODEL Clause

9. ☑ **C.** Symbolic cell addressing is the way the MODEL clause handles the individual cells of a multidimensional array.
☒ **A, B,** and **D** are wrong because symbolic cell addressing deals only with addressing cells within the array.

10. ☑ **B.** Using a MODEL clause, you can derive both updated and new data.
☒ **A** is wrong since you don't see and deleted data in the output produced by the MODEL clause. **C** and **D** are wrong since you can derive *both* updated and new data when you use the MODEL clause.

11. ☑ **D.** By default, the RULES keyword operates with the UPSERT specification.
☒ **A, B,** and **C** offer the wrong specifications.

12. ☑ **C.** The RETURN UPDATED ROWS clause ensures that the MODEL clause outputs *both* the updated and the new rows.
☒ **A** and **B** are wrong since you drive *both* updated as well as new data. **D** is wrong since the MODEL clause doesn't delete data.

Materialized View Enhancements

13. ☑ **A** and **D. A** is correct since a fast refresh is accomplished by using incremental changes. **D** is also correct, since the materialized view logs hold the incremental change data.
☒ **B** is wrong because the fast refresh method doesn't depend on decomposing your materialized views. **C** is wrong since the fast refresh method is a method of freshening the data and doesn't have anything to with query rewriting.

14. ☑ **A** and **C.** Under some conditions, both the materialized views and the materialized view logs must use ROWIDs to ensure a fast refresh.
☒ **B** is wrong because a GROUP BY operation is something you must avoid in order to force a fast refresh. **D** is wrong because ROWNUM don't force a fast refresh either.

15. ☑ **D.** The EXPLAIN_MVIEW procedure tells you why a materialized view isn't fast refreshable.
☒ **A** is wrong because the EXPLAIN_REWRITE procedure tells you if a materialized view will or won't rewrite a query. **B** is wrong because the TUNE_MVIEW procedure helps you in making a materialized view fast refreshable. **C** is wrong because it refers to a nonexistent procedure.

16. ☑ **B.** The TUNE_MVIEW procedure only makes recommendations, including the creation of necessary materialized view logs, to make a query fast refreshable.
☒ **A, B,** and **D** are wrong since the TUNE_MVIEW procedure doesn't automatically create any views, logs, or a fast refresh of a materialized view. It's a purely advisory view. You can implement the changes recommended by the view.

17. ☑ **A.** The DBA_TUNE_MVIEW view holds the output of the TUNE_MVIEW procedure.
☒ **B, C,** and **D** are wrong since they refer to the wrong procedures as the source of the view.

18. ☑ **D.** The TASK_NAME variable helps you identify your query in the DBA_TUNE_MVIEW view (WHERE TASKNAME = :TASK_NAME).
☒ **A, B,** and **C** refer to the wrong variables or columns.

19. ☑ **B.** The REWRITE_OR_ERROR hint will stop any query that fails to rewrite and issues an automatic error.
☒ **A** is wrong because the hint doesn't rewrite the query if it doesn't have errors. **C** is wrong since the hint doesn't send a report of the errors—it merely sends a single error message when a query fails to rewrite. **D** is wrong since the hint does not force a query rewrite when there are errors—it terminates the execution of the query when the query fails to rewrite and issues an error.

20. ☑ **D.** In Oracle Database 10*g*, the PCT feature has been extended to materialized views based on list-partitioned tables.
☒ **A** is wrong because you could use the PCT feature with partitioned tables in earlier versions of Oracle. Similarly, **B** and **C** are wrong since you could use the PCT feature in both rule- and hash-partitioned tables.

Lab Answer

1. Create the tables to show sales data for two products and check the data in the new table, as in this example:

```
SQL> create table s1 as
  2  select distinct time_id, prod_id, quantity_sold
  3  from sales
  4  where time_id between '02-JAN-2005'
  5  and '05-JAN-2005'
  6* and prod_id < 15;
Table created.
```

```
SQL> select * from s1;
TIME_ID        PROD_ID QUANTITY_SOLD
----------- ---------- -------------
02_JAN-2005         13             1
02_JAN-2005         14             1
03_JAN-2005         14             1
04_JAN-2005         13             1
```

2. Create a second table with four rows in it, one for each day of January 2005, and check the data in the table after creating it, as in this example:

```
SQL> begin
  2  for i in 0..3 loop
  3  insert into t1 values (to_date('02-JAN-2005') + i);
  4  end loop;
  5* end;
PL/SQL procedure successfully completed.
SQL> select * from t1;
TIME_ID
-----------
02_JAN-2005
03_JAN-2005
04_JAN-2005
05_JAN-2005
```

3. Create a regular outer join between tables s1 and t1. The following example uses the Oracle function CUMULATIVE, to produce cumulative values for each day, from the quantity_sold column values. The regular outer join will show a day, even if there aren't any matching values for it in the s1 table. This query shows a row for the 05-Jan-2005 date, even though there are no product sales for that date (shown in table s1).

```
SQL> select prod_id, time_id, quantity_sold,
  2  sum(quantity_sold) over
  3  (partition by prod_id
  4  order by time_id)
  5  as cumulative
6  from s1
  7  right outer join t1
  8  using (time_id)
  9  order by prod_id, time_id;
   PROD_ID TIME_ID     QUANTITY_SOLD CUMULATIVE
---------- ----------- ------------- ----------
        13 02_JAN-2005             1          1
        13 04_JAN-2005             1          2
        14 02_JAN-2005             1          1
        14 03_JAN-2005             1          2
           05_JAN-2005
```

4. Finally, use a partitioned outer join, to see how you can improve over the results obtained with just a regular outer join. The partitioned outer join in this example partition-joins the data by the prod_id column, and shows rows for each day one of the products wasn't sold. Product ID 13 didn't sell on the third and fifth of January 2005. Product ID 14 didn't sell on the fourth and fifth. However, the partitioned outer join ensures that you see rows for both the products, for all the days.

```
SQL>  select prod_id, time_id, quantity_sold,
  2   sum(quantity_sold) over
  3   (partition by prod_id
  4   order by time_id)
  5   as cumulative
  6   from s1
  7   partition by (prod_id)
  8   right outer join t1
  9   using (time_id)
 10*  order by prod_id, time_id;
   PROD_ID TIME_ID     QUANTITY_SOLD CUMULATIVE
---------- ----------- ------------- ----------
        13 02_JAN-2005             1          1
        13 03_JAN-2005                        1
        13 04_JAN-2005             1          2
        13 05_JAN-2005                        2
        14 02_JAN-2005             1          1
        14 03_JAN-2005             1          2
        14 04_JAN-2005                        2
        14 05_JAN-2005                        2
8 rows selected
SQL>
```