EXAM OBJECTIVE: WRITE MERGE STATEMENTS WITH THE NEW CONDITIONS AND EXTENSIONS

The merge statement is a statement introduced in Oracle 9i that was used to conditionally insert or update rows in a table. The MERGE statement could be typically written in the following manner:

MERGE INTO table_name table_alias
USING (table|view|sub_query) alias
ON (join condition)
WHEN MATCHED THEN
UPDATE SET
col1 = col_val1,
col2 = col2_val
WHEN NOT MATCHED THEN
INSERT (column_list)
VALUES (column_values);

In Oracle 10g, there are two major improvements to the MERGE statement.
- New conditional clauses and extensions to the standard MERGE making it easier to use and faster to execute.
    - You can optionally specify the UPDATE or INSERT clauses.
    - Perform a conditional UPDATE or a MERGE
    - Compile time recognition and evaluation of ON clause predicates.
- An optional DELETE clause to the MERGE statement can be specified.

OPTION TO OMIT UPDATE OR INSERT CLAUSES

You may now write the MERGE statement with the option of omitting either the INSERT or UPDATE clause.
In the example below, an update is being made to the ORDERS tables based on ORDER_CHANGE table. If there are 50 rows in the ORDER_CHANGE table that match the ORDERS table per order_id, then 50 rows are updated.
MERGE
    INTO ORDERS O
    USING ORDER_CHANGE OC
ON (o.order_id = oc.order_id)
WHEN MATCHED THEN
UPDATE
    SET o.ostatus = oc.ostatus;

CONDITIONAL UPDATE

You can also specify a WHERE clause in the UPDATE operation to test a certain condition when it is desirable to skip the UPDATE operation. In the example below the UPDATE is done only on rows where the Order_Date is before '10-JUN-2004'. This is a conditional update.

MERGE
    INTO ORDERS O
    USING ORDER_CHANGE OC
ON (o.order_id = oc.order_id)
WHEN MATCHED THEN
UPDATE
    SET oc.ostatus =o.ostatus
    WHERE O.order_date < '10-Jun-04' ;

CONDITIONAL INSERT

A WHERE clause maybe added to the INSERT clause of the MERGE statement in Oracle 10g. This results in the INSERT clause being skipped if a certain condition is not satisfied.

MERGE
    INTO ORDERS O
    USING ORDER_CHANGE OC
ON (o.order_id = oc.order_id)
WHEN MATCHED THEN
    UPDATE
            SET oc.ostatus = o.ostatus
            WHERE O.order_date < '10-Jun-04'
WHEN NOT MATCHED THEN
    INSERT (oc.ostatus) VALUES (o.ostatus)
            WHERE o.order_date < '10-Jun-04';

ON PREDICATE

This clause can be used to insert all the rows of one table into another without joining the source and target tables.
The ON clause takes the form:
ON (condition)
Consider the example given below:
The NEW_ORDERS table is a copy of the ORDERS table. It is created from the ORDERS table and is initially empty. The column order_status is renamed as new_order_status in the NEW_ORDERS table.

CREATE TABLE new_orders
AS
SELECT * FROM ORDERS
WHERE 1=2;

```
ALTER TABLE new_orders
RENAME COLUMN order_status TO new_order_status;
```

This merge copies all the rows from ORDERS to NEW_ORDERS without joining the two tables using the ON clause.

```
MERGE
INTO new_orders no
USING orders o
ON  (1=0)
WHEN NOT MATCHED THEN
INSERT (no.order_id, no.new_order_status)
VALUES (o.order_id, o.order_status);
```

OPTIONAL DELETE CLAUSE

To cleanup tables after populating or updating them the DELETE clause can be added in the MERGE statement. In the example below, after all the rows have been updated, the rows where order_date is less than '10-June-2004' is deleted.

```
MERGE
    INTO ORDERS O
    USING ORDER_CHANGE  OC
ON (o.order_id = oc.order_id)
WHEN MATCHED THEN
    UPDATE
            SET oc.ostatus = o.ostatus
            WHERE O.order_date < '10-Jun-04'
DELETE WHERE ( o.order_date < '10-Jun-04' )
WHEN NOT MATCHED THEN
    INSERT (oc.ostatus) VALUES (o.ostatus)
            WHERE o.order_date < '10-Jun-04';
```

EXAM OBJECTIVE: USE PARTITIONED OUTER JOIN SYNTAX FOR DENSIFICATION

Before we delve into this topic let us understand some basic terminology commonly used in datawarehousing.

- Dimension Table
  A table, typically in a data warehouse, that contains further information about an attribute in a fact table. For example, a SALES table can have the

following dimension tables TIME, PRODUCT, REGION, SALESPERSON, etc.

- Fact Table
  A table, typically in a data warehouse, that contains the measures and facts such as the table SALES.

Very often data is stored in a sparse form. This means that is a value does not exist for a given combination of dimensions, and then no row exists in the fact table. However some calculations and report formatting can be performed more easily if a row existed for each combination of dimensions. This is referred to as dense data. Dense data will return a consistent number of rows fro each group of dimensions, making it simpler to use SQL analytic functions.

The PARTITION OUTER JOIN clause can be used to make sparse data dense. The clause can be used to fill the gaps in a dimension. This joins extends the conventional outer join syntax by applying the outer join to each logical partition defined in a query. The Oracle database logically partitions the rows in your query based on the expression that you specify in the PARTITION BY clause. The result of a partitioned outer join is a UNION of the outer joins of each of the groups in the logically partitioned table with the table on the other side of the join.

PARTITION OUTER JOIN SYNTAX

There are two forms available to perform a partitioned outer join.

SELECT select_expression
FROM table_reference
PARTITION BY (expr [,expr]…)
RIGHT OUTER JOIN table_reference

SELECT select_expression
FROM table_reference
LEFT OUTER JOIN table_reference
PARTITION BY {expr [,expr]…)

EXAMPLE OF SPARSE DATA

If the query executed was requesting sales for a product in the weeks 10-20 of year 2000 and 2001, the output should have retrieved 22 rows however only 18 rows are displayed since no sales were made in the weeks (15, 16) in year 2003 and week (16, 18) of 2004.

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 ('Blank CDs') AND
      t.Calendar_Year IN (2003,2004) AND
      t.Calendar_Week_Number BETWEEN 10 AND 20
  GROUP BY p.Prod_Name, t.Calendar_Year, t.Calendar_Week_Number;
```

PRODUCT_NAME	YEAR	WEEK	SALES
Blank CDs	2003	10	45.00
Blank CDs	2003	11	200.00
Blank CDs	2003	12	25.00
Blank CDs	2003	13	33.00
Blank CDs	2003	14	89.00
Blank CDs	2003	17	25.00
Blank CDs	2003	18	13.00
Blank CDs	2003	19	65.00
Blank CDs	2003	20	35.00
Blank CDs	2004	10	565.00
Blank CDs	2004	11	5.00
Blank CDs	2004	12	45.00
Blank CDs	2004	13	33.00
Blank CDs	2004	14	45.00
Blank CDs	2004	15	28.00
Blank CDs	2004	17	80.00
Blank CDs	2004	19	26.00
Blank CDs	2004	20	145.00

To fill in the gaps you can use the partitioned outer join shown below.

```
SELECT product_name, t.year, t.week, sales, 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 ('CD Blanks') AND
        t.Calendar_Year IN (2003,2004) AND
         t.Calendar_Week_Number BETWEEN 10 AND 20
        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 (2003,2004)
        AND Calendar_Week_number between 10 and 20) t
ON (v.week = t.week AND v.year = t.year)
ORDER BY t.year, t.week;
```

You take the sparse data of the query and do a partitioned outer join with a dense set of time data. In the example the original query has an alias of "v" and the data retrieved from the times tables has an alias of "t".

Also in the example, the WHERE condition for weeks between 10 and 20 is placed in the inline view for the time dimension so that the number of rows handled by the outer join is reduced.

The output of the modified query is given below:

PRODUCT_NAME	YEAR	WEEK	SALES	DENSE_SALES
Blank CDs	2003	10	45.00	45.00
Blank CDs	2003	11	200.00	200.00
Blank CDs	2003	12	25.00	25.00
Blank CDs	2003	13	33.00	33.00
Blank CDs	2003	14	89.00	89.00
Blank CDs	2003	15		0.00
Blank CDs	2003	16		0.00
Blank CDs	2003	17	25.00	25.00
Blank CDs	2003	18	13.00	13.00
Blank CDs	2003	19	65.00	65.00
Blank CDs	2003	20	35.00	35.00
Blank CDs	2004	10	565.00	565.00
Blank CDs	2004	11	5.00	5.00
Blank CDs	2004	12	45.00	45.00
Blank CDs	2004	13	33.00	33.00

Blank CDs	2004	14	45.00	45.00
Blank CDs	2004	15	28.00	28.00
Blank CDs	2003	16		0.00
Blank CDs	2004	17	80.00	80.00
Blank CDs	2003	18		0.00
Blank CDs	2004	19	26.00	26.00
Blank CDs	2004	20	145.00	145.00

Exam Objective: Use interrow calculations to enhance SQL for analytical capabilities.

In Oracle 10g, with the SQL MODEL clause, you can treat a query result as a multi-dimensional array and apply formulas on the array to calculate new values. The formulas can be sophisticated interdependent calculations.

By integrating advanced calculations into the database, performance, scalability and manageability are enhanced significantly when compared to external solutions.

Rather than copying data into separate applications such as spreadsheets to perform calculations and build complex models, users can now maintain the data within the Oracle environment. The Oracle server is now available to perform spreadsheet-like array computations into SQL. The main benefits in this method, is because an external tool can have a significant scalability problem when the number of formulas and data becomes large. Further desktop spreadsheets have no access to parallel processing abilities of advanced servers.

The MODEL clause is used to integrate interrow functionality into the Oracle Server.

The MODEL clause defines a multidimensional array by mapping the columns of a query into three groups, namely **partitioning**, **dimension** and **measure** columns.

Partitions are logical blocks of the result set in a way similar to partitions of analytical functions. Model formulas are applied to the cells of each partition. Measures are analogous to the measures of a fact table in a star schema. They typically contain numeric values such as sales units or cost. Each cell is accessed within its partitions by specifying its full combination of dimensions. Dimensions identify each measure cell within a partition. These columns identify characteristics such as data, region and product name.

To create formulas on these multidimensional arrays, you define computation rules expressed in terms of the dimension values. The rules are flexible and concise, and can use wild cards and FOR loops for maximum expressiveness.

In the example given below, you get a conceptual overview of the MODEL clause using a hypothetical SALES table. The table has columns for COUNTRY, PRODUCT, YEAR and SALES amount. The figure has three parts.
The top segment shows the concept of dividing the table into partitioning, dimension and measure columns. The middle segments shows two rules that calculate the value of PROD1 and PROD2 for the year 2004. Finally, the third part shows the output of the query appl ying the rules to a table with hypothetical data. The shaded portion of the output shows rows that are calculated from rules. Note that the rules are applied within each partition.

Mapping of columns to model entities

Country	Product	Year	Sales
Partition	Dimension	Dimension	Measure

Rules

Sales (PROD1,2004) = Sales (PROD1,2002)+Sales(PROD1,2003)
Sales(PROD2,2004) = Sales (PROD2,2002) + Sales(PROD2,2003)

Output of model clause:

Country	Product	Year	Sales
Partition	Dimension	Dimension	Measure
A	PROD1	2002	10
A	PROD1	2003	15
A	PROD2	2002	12
A	PROD2	2003	16
B	PROD1	2002	21
B	PROD1	2003	23
B	PROD2	2002	28
B	PROD2	2003	29
A	PROD1	2004	25
A	PROD2	2004	28
B	PROD1	2004	44
B	PROD2	2004	57

The MODEL clause does not update existing data in tables, nor does it insert new data into tables: to change values in a table, the MODEL results are supplied to an INSERT or UPDATE or MERGE statement.

SYNTAX OF MODEL clause:

<prior clauses of SELECT statement>

```
MODEL [main]
[reference models]
[PARTITION BY (<cols>)]
DIMENSION BY (<cols>)
MEASURES (<cols>)
[IGNORE NAV] | [KEEP NAV]
[RULES
[UPSERT | UPDATE]
[AUTOMATIC ORDER | SEQUENTIAL ORDER]
[ITERATE (n) [UNTIL <condition>] ]
( <cell_assignment> = <expression> ... )
```

To keep our example simple, let us create a view using the SALES HISTORY table in the SH schema (part of the sample schema set) provided with Oracle 10g. The view sales_view provides annual sums for product sales, in dollars and units, by country, aggregated across all channels. This view is built from a one million row fact table and is defined as given below:

```
CREATE VIEW sales_view AS
SELECT country_name country, prod_name prod,
calendar_year year,
SUM(amount_sold) sale, COUNT(amount_sold) cnt
FROM sales, times, customers, countries, products
WHERE sales.time_id = times.time_id AND
sales.prod_id = products.prod_id AND
sales.cust_id = customers.cust_id AND
customers.country_id = countries.country_id
GROUP BY country_name, prod_name, calendar_year;
```

The following example, using the MODEL clause calculates the sales values for two products and defines sales for a new product based on the other two products.

```
SELECT SUBSTR(country,1,20) country, SUBSTR(prod,1,15) prod,
year, sales
FROM sales_view
WHERE country IN ('Italy','Japan')
MODEL RETURN UPDATED ROWS
PARTITION BY (country)
DIMENSION BY (prod, year)
MEASURES (sale sales)
RULES (
sales['Bounce', 2004] = sales['Bounce', 2003] +sales['Bounce', 2002],
sales['Y Box', 2004] = sales['Y Box', 2003],
```

sales['2_Products', 2004] = sales['Bounce', 2004] +sales['Y Box', 2004])
ORDER BY country, prod, year;

The results are:

COUNTRY	PROD	YEAR	SALES
--------------------	--------------- -	---------	----------
Italy	2_Products	2004	92613.16
Italy	Bounce	2004	9299.08
Italy	Y Box	2004	83314.08
Japan	2_Products	2004	103816.6
Japan	Bounce	2004	11631.13
Japan	Y Box	2004	92185.47

This statement partitions data by country, so the formulas are applied to data of one country at a time.

Our sales fact data ends with 2003, so any rules defining values for 2004 or later will insert new cells. The first rule defines the sales of a video games called "Bounce" in 2004 as the sum of its sales in 2002 and 2003.

The second rule defines the sales for Y Box in 2004 to be the same value they were for 2003.

The third rule defines a product called "2_Products," which is simply the sum of the Bounce and Y Box values for 2004. Since the values for 2_Products are derived from the results of the two prior formulas, the rules for Bounce and Y Box must be executed before the 2_Products rule.

Note the following characteristics of the example above:

• The "RETURN UPDATED ROWS" clause following the keyword MODEL limits the results returned to just those rows that were created or updated in this query. Using this clause is a convenient way to limit result sets to just the newly calculated values.

• The keyword "RULES," shown in all our examples at the start of the formulas, is optional, but we include it for easier reading.

Exam Objective: Use new fast refresh capabilities for Materialized Join Views

A **materialized view** provides indirect access to table data by storing the results of a query in a separate schema object. Unlike an ordinary view, which does not take up any storage space or contain any data, a materialized view contains the rows resulting from a query against one or more base tables or views. Materialized views can be refreshed from their base tables either completely or incrementally using a fast refresh mechanism. Whether the fast refresh option is available depends upon the type of materialized view.

You can call the procedure DBMS_MVIEW.EXPLAIN_MVIEW to determine whether fast refresh is possible. This procedure can tell you whether a materialized view is fast refreshable or eligible for general query rewrite an the EXPLAIN_REWRITE will tell you whether query rewrite will occur at all. However, neither tells you how to achieve fast refresh or query rewrite.

In addition to the EXPLAIN_MVIEW procedure, in Oracle 10g, you can also use the DBMS_ADVISOR.TUNE_MVIEW procedure to optimize a CREATE MATERIALIZED VIEW statement to achieve REFRESH FAST and ENABLE QUERY REWRITE goals. With the TUNE_MVIEW procedure, a user no longer needs a detailed understanding of materialized views to create a materialized view in an application, since the materialized view and its required components (such as materialized view log) will be created correctly through the procedure.

The TUNE_MVIEW analyzes and processes the input statement and generates two sets of output results: one for the materialized view implementation and the other for undoing the CREATE MATERIALIZED VIEW operations. These two sets of output results can be accessed through Oracle views or be stored in external script files created by SQLAccess Advisor. The external script files are ready to execute to implement the materialized view.

The TUNE_MVIEW procedure can generate the output statements that correct the defining query by adding extra columns such as required aggregate columns or fix the materialized view logs to achieve the FAST REFRESH goal.

TUNE_MVIEW syntax

The syntax for TUNE_MVIEW is:
DBMS_ADVISOR.TUNE_MVIEW(task_name IN OUT VARCHAR2,
                                    Mv_create_stmt IN [CLOB|VARCHAR2])
where

task_name is a user-provided task identifier used to access the output results
mv_create_stmt is a complete CREATE MATERIALIZED VIEW statement that
needs to be tuned.

Output of TUNE_MVIEW

The TUNE_MVIEW procedure generates two sets of output results as executable
statements. One set of output (IMPLEMENTATION) is for implementing
materialized views and required components such as materialized view logs or
rewrite equivalences to achieve fast refreshability and query rewritability as much
as possible. The other set of output (UNDO) is for dropping the materialized views
and rewrite equivalences in case you decide they are not required.

There are two ways to access TUNE_MVIEW output results:
- Script generation using DBMS_ADVISOR.GET_TASK_SCRIPT function
  and the DBMS_ADVISOR.CREATE_FILE procedure.
- Use USER_TUNE_MVIEW or DBA_TUNE_MVIEW views, which get
  their output from the SQLAccess Advisor repository tables.

This example shows how TUNE_MVIEW changes the defining query to be fast
refreshable.

A MATERIALIZED VIEW CREATE statement is defined in variable
create_mv_ddl. This create statement has the REFRESH FAST clause specified.
Its defining query contains a single query block in which an aggregate column,
SUM(s.amount_sold), does not have the required aggregate columns to support
fast refresh.

If you execute the TUNE_MVIEW statement with this MATERIALIZED VIEW
CREATE statement, the output produced will be fast refreshable:

```
VARIABLE task_cust_mv VARCHAR2(30);
VARIABLE create_mv_ddl VARCHAR2(4000);
EXECUTE :task_cust_mv := 'cust_mv';

EXECUTE :create_mv_ddl := ' -
CREATE MATERIALIZED VIEW cust_mv -
REFRESH FAST -
DISABLE QUERY REWRITE AS -
SELECT s.prod_id, s.cust_id, SUM(s.amount_sold) sum_amount -
FROM sales s, customers cs -
WHERE s.cust_id = cs.cust_id -
GROUP BY s.prod_id, s.cust_id';
```

EXECUTE DBMS_ADVISOR.TUNE_MVIEW(:task_cust_mv, :create_mv_ddl);

The projected output of TUNE_MVIEW includes an optimized materialized view defining query as follows:

```
CREATE MATERIALIZED VIEW SH.CUST_MV
REFRESH FAST WITH ROWID
DISABLE QUERY REWRITE AS
SELECT SH.SALES.PROD_ID C1, SH.CUSTOMERS.CUST_ID C2,
      SUM("SH"."SALES"."AMOUNT_SOLD") M1,
      COUNT("SH"."SALES"."AMOUNT_SOLD") M2,
      COUNT(*) M3
   FROM SH.SALES, SH.CUSTOMERS
WHERE SH.CUSTOMERS.CUST_ID = SH.SALES.CUST_ID
   GROUP BY SH.SALES.PROD_ID, SH.CUSTOMERS.CUST_ID;
```

The UNDO output is as follows:

```
DROP MATERIALIZED VIEW SH.CUST_MV;
```

The original defining query of cust_mv has been modified by adding aggregate columns in order to be fast refreshable.

Example 17-4 Access IMPLEMENTATION Output Through USER_TUNE_MVIEW View

```
SELECT * FROM USER_TUNE_MVIEW
WHERE TASK_NAME= :task_cust_mv AND
SCRIPT_TYPE='IMPLEMENTATION';
```

Example 17-5 Save IMPLEMENTATION Output in a Script File

```
CREATE DIRECTORY TUNE_RESULTS AS '/myscript'
GRANT READ, WRITE ON DIRECTORY TUNE_RESULTS TO PUBLIC;

EXECUTE
DBMS_ADVISOR.CREATE_FILE(DBMS_ADVISOR.GET_TASK_SCRIPT(:ta
sk_cust_mv), -
'TUNE_RESULTS', 'mv_create.sql');
```