Materialized Views in Action

By Jeff Maresh

Materialized views are a powerful feature that has been part of the Oracle RDBMS since version 8.1. Based upon the snapshot replication feature, they have evolved considerably into a capable tool for significantly improving performance on queries at higher levels of hierarchical organization structures. When they are effectively implemented across an entire data warehouse, the total number of logical reads can be reduced by well over 90%. Although materialized views are considered to be a data warehouse feature, they can also be employed in other environments, including Operational Data Stores (ODS), data marts, and reporting tables in OLTP environments, where end-users will perform rollup queries on the schema.

Like other mature Oracle RDBMS features that have been designed to satisfy many requirements, there are many ways to implement materialized views. Some of these methods are more effective and efficient than others. In this article, the materialized views will be designed for a real-world dimensional model. By example, the reader will gain understanding of some of the practical issues associated with designing and implementing materialized views. This article specifically covers versions 8.1.7 through 9.2.0 of the RDBMS, but most of the concepts can also be applied to version 10.

A Quick Primer

A materialized view is a special type of summary table that is constructed by aggregating one or more columns of data from a single table, or a series of tables that are joined together. When queries are executed at an aggregation level satisfied by a materialized view, the cost-based optimizer automatically rewrites the query to take advantage of the most appropriate materialized view. Ralph Kimball referred to this feature as the *aggregate navigator* (Kimball, 1996, p. 205). When implemented properly, materialized views can dramatically improve query performance, and significantly decrease the load on the system. This is because materialized views require fewer logical reads to satisfy the query than the same query running against the base tables.

There are many good resources available that cover details about materialized views that will not be covered here. Ralph Kimball provides a good foundation for understanding aggregates and many of the practical issues one must be aware of to properly implement them (Kimball, 1996, ch. 13). Gary Dodge's and Tim Gorman's Oracle data warehouse book contains a good section on the practicalities of materialized views (Dodge & Gorman, 2000, ch. 7). And last of all, the Oracle documentation is a good resource for many of the details, as well as statement syntax and limitations of the various subfeatures (Oracle Corp., 2002, Ch 8, DW Guide). Once the reader is familiar with the basic concepts of materialized views, the case study presented in this article shows in detail how they have been successfully implemented on a data warehouse. It provides the reader with some insights into the subjective decisions that must be made during the implementation process.

An Example Dimensional Schema

The following dimensional schema was used in the article entitled *Supercharging Star Transformations* by the author. This article was published in the 4Q 2004 issue of *SELECT Journal.* It is also available in the download section at www.evdbt.com. If materialized views will be created on star transformations, all of the recommendations contained in that paper should be followed before deploying materialized views. The same dimensional schema will be used to illustrate how to build a series of materialized views to significantly improve query performance.

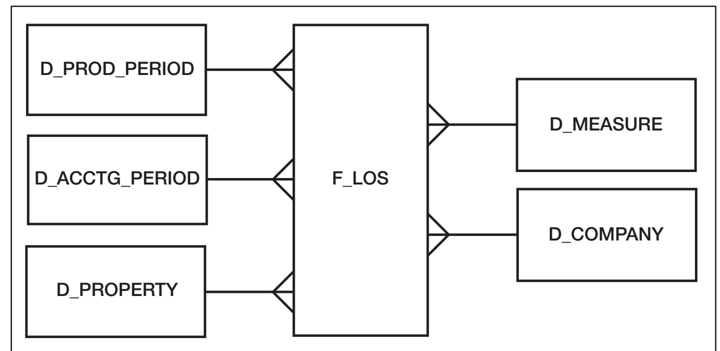

Figure 1: Example Dimensional Schema

The schema is used by an energy company to analyze oil and gas well performance, and to generate a variety of reports used for decision support. The d_prod_period dimension contains production dates. The d_acctg_period dimension contains accounting dates. The difference between the two is that production dates are when the business event occurs while accounting dates are when the event is booked for accounting purposes. The d_property dimension contains information about all of the oil and gas wells at a property level. This dimension also contains a collapsed location hierarchy that enables users to query at any level of the organization, for example, oil field, district, region, or country. Information about well ownership is contained in the d_company table.

The d_measure table is a normalizing dimension that is used to identify the operational measures contained in the fact table. Examples of measures are oil and gas production, and oil sales. In this particular case, the fact table contains only one measure on each row, rather than storing all facts for a

continued on page 34

particular set of dimension keys on a single row. This design was chosen because data is sparse, and to eliminate schema and code modifications when new measures are added.

Each of the dimension tables has a numeric primary key comprised of a sequence-generated number, otherwise known as a *surrogate* key. The fact table columns include the surrogate key for each of the dimension tables, and a column to hold the measure value. The row counts in the dimension tables range from several hundred to 20,000.

The fact table holds 8 years of data, and has 38 million rows. It is range partitioned on the production period dimension key column, prod_period_id. Since the granularity of the data is at the monthly level, the table has a total of 96 partitions. The fact table has the following column definitions.

```
Name                                      Null?    Type
----------------------------------------- -------- ------------
PROD_PERIOD_ID                            NOT NULL NUMBER(12)
ACCTG_PERIOD_ID                           NOT NULL NUMBER(12)
PROPERTY_ID                               NOT NULL NUMBER(12)
MEASURE_ID                                NOT NULL NUMBER(12)
COMPANY_ID                                NOT NULL NUMBER(12)
LOS_NMBR                                  NOT NULL NUMBER(24,8)
```

This is a small schema by data warehouse standards. Yet because of the nature of the analyses, the goal is for the median query to return in 3 seconds, a substantially higher response time requirement than an often-desirable target of 30 seconds to three minutes in data warehouse environments. This has been achieved by a careful schema design, implementation of the most useful Oracle data warehousing features including table and index partitioning, materialized views, and star transformations. Even though a table can be properly partitioned, and star transformations fully optimized, poor performance may still result if materialized views are not used, or if they are not properly implemented.

Schema Queries

Several types of queries run against the above schema. Drill down queries are those that have limiting conditions such that only few rows are retrieved from the fact table. The row counts may be on the order of several hundred to several thousand rows. These can be satisfied very efficiently by running a star transformation against all of the tables in the schema. Aggregation, if any, occurs at a relatively low level, so materialized views are not necessary to achieve great performance. A query of this type might answer the question:

"How much gas was produced in the Salt Flat #488 property during February 2005?"

Other types of queries aggregate data to higher levels. For most enterprises, two common aggregation dimensions are time and the organizational hierarchy. Time-based aggregations roll up data to months, quarters, and years, for example. The organizational hierarchy of the above schema is found in the d_property dimension and has the following levels, from lowest to highest.

Level	Distinct values
Property	20,000
Field	3,500
Route	625
Foreman	90
Superintendent	33
District	20
Region	8
Country	5
Company	1

Table 1: Organizational hierarchy

Here, property, field, district, region, and country represent physical locations in the hierarchy. Route, foreman, superintendent, and company represent other nonlocation entities, but they are nonetheless logical elements in the hierarchy. *Property* is the lowest level of the hierarchy while *company* is the highest. An aggregation query at the region level might answer the following question:

"How much gas was produced in the Northwestern region during February 2005?"

Because there are only eight regions in the company, an average of 12.5% of the rows in the d_property table belong to this region. In the real world where data skew is always present, the value likely ranges from 5% to 25%. Its easy to see that queries that request data at the region level of the schema would likely benefit from a region level materialized view. While a materialized view at this level may be obvious, what about materialized views at other levels? Would a materialized view at the field level benefit field level queries? At first blush, it appears that a field level materialized view would be beneficial simply because one field contains an average of almost six properties. But as we'll see in later sections, there are more things to consider than simply the row counts at a particular level in the hierarchy. The remainder of the article will show how a series of materialized views for the organizational hierarchy were developed. The organizational hierarchy was chosen because it is exposes more of the difficult issues than one typically encounters with the time hierarchy.

Designing Materialized Views

One method of implementing materialized views is to use the Summary Advisor tool that is documented in the *Oracle 9*i *Data Warehousing Guide.* The Summary Advisor is a comprehensive tool for designing and managing materialized views. While it has a certain degree of complexity to configure and use, it may be a good choice for many applications. One of the benefits of Summary Advisor is that it can be managed seamlessly from within Oracle Enterprise Manager. Summary Advisor is a topic in and of itself and won't be discussed further in this article.

The other alternative is to design and manage materialized views manually. There are several cases where this method is practical. One case is that only one or two tables or table joins require materialized views so one doesn't want to endure the effort of implementing Summary Advisor. Another instance is where one wants to achieve the highest performance possible from materialized views. Once you've had some experience designing and building materialized views, it's likely that you'll achieve better performance designing and building them yourself. This may result in considerable

performance improvements, hence cost savings, particularly on large databases. It is admittedly a lot of effort to learn the process, but the results are worth it. Even if you choose to use Summary Advisor, learning the manual design process will help gain insights into how to better use the tool.

Sampling Candidate Queries

The first step is to pick a table or table join on which to create materialized views. Then gather queries that run against table or table join. This can be accomplished by sampling the shared pool directly using SQL, through GUI based shared pool sampling tools, or by asking developers for their queries. The author prefers either of the first two methods because one can sample what is actually running on the tables. The number for queries you collect depends upon the complexity of the table or table join, and diversity of the user group. Higher-level users such as managers and Vice Presidents are likely to look at different data, and data at different organizational levels than lower level employees. It's also wise to sample queries over a period of time to capture query variants that don't necessarily occur on a daily basis. For example, certain aggregate reports might only run on a weekly, monthly, or quarterly basis. On a single table or simple schema, one might capture between 10 and 20 queries that access a unique set of columns. On a more complex schema, one might capture between 40 and 60 queries.

Using these queries, a SQL script file should be created. This script will be used to evaluate the performance of the queries in the absence of the materialized views, and the performance with various materialized view configurations. All of the queries should be wrapped in a SELECT COUNT(*) FROM (…) block so that the log files don't grow unnecessarily large. The presence of the wrapper will have negligible affect on query performance.

Choosing Aggregation Levels and Columns

The next step it to determine at what levels in the hierarchy that materialized views will be built. This is accomplished by building a spreadsheet as follows. Create one row for each level in the hierarchy. Increasing row numbers should move higher in the hierarchy. Create one column in the spreadsheet for each column referenced in the query. Starting at the left side of the spreadsheet, list all of the dimensional attribute columns referenced in the queries that are not in the hierarchy. Also, list any columns referenced in limiting conditions in the query. Next, list the dimensional attributes of the hierarchy starting with the highest levels of the hierarchy first, and ending with the lowest levels of the hierarchy. The next step is to classify each of the queries in the regression set at the appropriate level in the hierarchy so that their counts can be recorded in the spreadsheet. The hierarchy level of the query is the lowest hierarchy level of any column referenced in the query, or any join condition. For example, consider the following query.

```
SELECT pp.prod_dt, f.measure_id, p.operate, p.region_nm,
    p.district_nm, SUM(f.los_nmbr)
  FROM f_los f, d_prod_period pp, d_property p
  WHERE f.prod_period_id = pp.prod_period_id
  AND f.property_id = p.property_id
  AND pp.prod_dt BETWEEN '01-JUL-2004' AND '31-JUL-2004'
  AND f.measure_id = 144
  AND p.region_nm = 'USA SOUTH'
  GROUP BY pp.prod_dt, f.measure_id, p.operate, p.region_nm,
    p.district_nm;
```

Note that the query references both region and district names, but because district is at a lower level in the hierarchy, it is classified as a district level query. So on the district row of the spreadsheet, we record a tick mark for each column referenced in the query. This process is repeated for each query in the regression set. To complete the spreadsheet, add up the tick marks in each cell and replace them with the number of occurrences as shown in the spreadsheet below. The spreadsheet will have been completed properly if there are progressively more zeros in the rightmost columns of the spreadsheet as the level of the hierarchy increases. For example, notice that at the foreman level, the last four columns contain zeros, while the rightmost ten columns at the company level contain zeros.

Level	num queries	los_nmbr	prod_dt	prod_month	prod_year	measure_id	operate_cd	company_nm	country_cd	country_nm	region_nm	district_nm	foreman_nm	foreman_id	route_nm	route_id	field_nm	field_cd
Field	8	8	8	4	5	7	4	2	3	2	3	2	1	2	2	1	7	8
Route	9	9	8	2	2	9	3	1	1	0	2	3	2	2	8	7	0	0
Foreman	4	4	4	1	1	3	2	0	1	1	3	2	4	2	0	0	0	0
Superintendent	2	2	2	0	0	2	0	0	1	1	1	0	0	0	0	0	0	0
District	9	9	8	3	2	7	5	1	2	3	3	8	0	0	0	0	0	0
Region	6	6	6	3	3	5	4	4	3	3	4	0	0	0	0	0	0	0
Country	3	3	3	1	1	3	3	0	3	3	0	0	0	0	0	0	0	0
Company	2	2	2	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0

Table 2: Materialized view column candidates

Next, we must figure out at what levels of the hierarchy materialized views will be created. There is nothing that prevents one from creating materialized views at all levels. The only problem with doing so is that more disk storage space will be used and more time will be required to build them than if only a few are created. Having additional materialized views will also result in more time being spent maintaining them. In most cases, building materialized views at a few critical levels will produce acceptable performance gains without incurring the overhead associated with creating them at all levels.

At the lower levels of the hierarchy, a 10:1 difference in distinct values between the base table and the first materialized view is a good starting rule of thumb to use if the materialized view has less than 30 columns. The reasoning is as follows. A materialized view is going to be generated from the join between the fact table and all of the dimension tables. While the fact table has few columns, the materialized view will have considerably more. Even though the joins are eliminated when the materialized view is accessed, as the size of materialized view approaches that of the base fact table, the performance benefits diminish progressively. So the factor of ten to one is usually a good first cut towards achieving a materialized view size that will provide significant performance gains.

From Table 1, the ratio of the distinct values between the property and field levels is only 6:1. This doesn't meet our minimum criteria of 10:1. The ratio between and property and route levels is 32:1 making the route level a good candidate for the first level materialized view. The second level of the materialized view that meets the 10:1 ratio would be at the superintendent level. The ratio between this level and the route level is 19:1. At this level, there is more flexibility in choosing levels because neither disk storage nor build time will be much of a consideration because the number of rows and columns in the materialized view will be considerably smaller than the volume at the lower levels. Based upon this fact, one could argue that materialized views could be built at the top five levels of the hierarchy. But this added complexity should be balanced against the performance gains that will be achieved. Reviewing Table 2, we see that there is considerable query

continued on page 36

Materialized Views in Action continued from page 35

activity at the district and region levels. Therefore, these are two good levels for materialized views. The few queries that run at the superintendent and foreman levels with use the materialized view at the route level. Queries at the country and company level will use the materialized view at the region level.

Next, the columns that will appear in the materialized view will be chosen. The number and types of columns directly affect the performance of the materialized view because it directly affects the overall size of the materialized view. Some discretion must be exercised when choosing columns because all columns do not have the same affect on the size of the materialized view. A *break* column is one that adds a new break level to the materialized view, hence the number of rows will increase when such a column is added. Obviously, the introduction of any column from any lower level of the hierarchy that has no other corresponding column in the materialized view is a break column. Another type of break column would be a code value that is currently not represented by some other description or code column in the materialized view. Break columns can cause materialized view row counts to explode, particularly when the column has many values. An *attribute* column is a column that is added whose value corresponds directly to some other column currently in the materialized view. Attribute columns cause the width of the row to increase, but not the number of rows. To better understand this concept, consider two field level columns that have a one to one correspondence: the field code and the field name. If the field code is a break column already in the materialized view, then field name is an attribute column because there is a one to one correspondence between the two. Attribute columns can usually be added to materialized views with less of an impact on their performance than adding a new break column.

In Oracle 9*i*, if the number of attribute columns becomes too large, and the column is not required in any higher level materialized views, eliminate the columns that are accessed few times. One of the improvements to the query rewrite capabilities of the optimizer in Oracle 9*i* is that access paths can now be generated that join a materialized view with one or more dimension tables when the requested dimensional attribute columns are not present in the materialized view. This join back capability adds significant capability to materialized views.

Physical Design

There are several physical design issues that should be understood to achieve the highest level of performance from materialized views. If the base fact table is partitioned, the lower levels of the materialized view should also be partitioned. To be more specific, the lower level materialized views should be partitioned so that they match the partitioning strategy of the underlying fact table. There should be one to one correspondence between fact table and materialized view partitions. Materialized views at intermediate levels of the hierarchy may also be partitioned. In many cases, rather than using bitmap indexes, the optimizer will choose to simply scan a materialized view partition. Materialized views at high levels in the hierarchy can be nonpartitioned. In the above example, the route and district materialized views are partitioned, while the region level materialized view is not.

Bitmap indexes should be created on each of the columns that are expected to be limiting conditions in the queries. For the wide variety of queries that typically run on either dimensional schemas or data mart tables, bitmap indexes enable the optimizer to use multiple indexes whenever possible. On partitioned tables, local indexes should be employed instead of global indexes. Local indexes have a flatter structure than corresponding global indexes and are likely to produce better query performance. The optimizer is also more likely to pick the most efficient access path if the index partition structure corresponds to the table partition structure.

When building the materialized views for a particular dimensional schema, only the first one will be constructed from the base tables. The remaining materialized views are constructed by querying the materialized view at the next lowest level. This is referred to as *nesting* materialized views. In the above example, the route level materialized view is created from a join on the base tables. The district level materialized view is created from the route level one, and the region level materialized view is created from the district level one. To ensure that nesting is possible, an exercise similar to the one shown in Table 2 should be performed. Make sure that the lower level materialized views always contain columns required in the upper level ones.

As stated earlier, a good indication that the materialized view will consistently perform better than queries against the base tables is when the size of the materialized view is smaller than the size of the base fact table. Table 3 shows the various sizes after the materialized views were constructed with the columns at the various levels shown in Table 2.

Table	Size (MB)
f_los base fact table	1,300
Route level materialized view	600
District level materialized view	107
Region level materialized view	52

Table 3: Storage space comparison

Build Strategies

There are many options for building materialized views. The first decision is to determine how materialized views will be maintained. When the data in the base tables changes, the corresponding rows in the materialized views must change accordingly so that queries will return accurate results. Two methods are available for this operation. The *fast refresh* method uses a log table to record any changes that occur using a trigger on the base table. Periodically, a command can be executed to refresh the materialized views so that they are synchronized with the base tables. Alternatively, the materialized views can be refreshed each time a COMMIT is issued on the base tables. This is usually an acceptable method for building and maintaining materialized views if several critical conditions are met. First, *fast refresh* can only be used for materialized views that aggregate data on a single table. The feature cannot be used for materialized views that join multiple tables and aggregate data using the GROUP BY clause. Second, if a substantial number of rows change on the underlying table, slower performance of the DML operations on the base table will result. Consult the Oracle documentation for the complete list of limitations on the *fast refresh* method. The *fast refresh* method is the only method available for synchronizing materialized views if DML operations occur frequently on the base tables while they are available to end-users, as would occur in an OLTP environment.

In the data warehouse setting, there is typically a maintenance window that occurs when new data are loaded each day. This is a period when the database is unavailable to end-users. In this environment, a practical method for maintaining materialized views is to simply drop and recreate them after the new data are loaded. This method supports materialized views with joins containing aggregation operations. It is therefore the only method available for maintaining the types of aggregation materialized views that are most useful for improving performance on dimensional schemas. With the current enterprise class of high performance general-purpose computers that commonly host data warehouses, it is practical to parallelize the entire

materialized view rebuild operation using the Oracle Parallel Execution (PX) to reduce the overall build time.

Once the materialized views have been built, gather optimizer statistics on all of the underlying tables and indexes. Having spent a considerable amount of time testing the ANALYZE command and the DBMS_STATS package, statistics generated by the latter usually produce better execution plans, hence better performance. Some good initial settings to use for DBMS_STATS are a 10% sample size with 10-bucket histograms on indexed columns.

Once materialized views have been built, verify that they are being used, and that the goal of improved performance has been consistently achieved on each query. To verify that the materialized views are being used, inspect the execution plan and verify that the expected materialized view is referenced. Finally, run the queries in the regression set to verify that performance is as expected. Compare performance without materialized views against performance with materialized views. It is beneficial to not only compare response time, but also the amount of I/O required to satisfy each query. One effective method for this exercise is to turn on tracing at the session level and run the entire regression set of queries. Then use the *tkprof* or *trace analyzer* utility to interpret the trace files. Here, Summary Advisor can also be used to monitor materialized view performance and usage.

The DDL statements used to create the materialized views are too voluminous to be presented in *SELECT Journal.* However, the complete set of annotated DDL scripts are posted at www.selectonline.org.

Troubleshooting

Any number of things can go wrong when working with materialized views. One of the most common problems encountered is that the optimizer won't rewrite the query to use materialized views. First, make sure that the two query rewrite parameters are set properly. Set QUERY_REWRITE_ENABLED=TRUE, otherwise, query rewrite will be disabled because the default value is FALSE. Next, make sure that the QUERY_REWRITE_INTEGRITY parameter is properly set. A value of ENFORCED means that the materialized view will not be eligible for rewrite unless all of the base tables are synchronized with the materialized view. This value should be used if materialized views are built with the *fast refresh* method. A value other than FRESH in the staleness column of dba_mviews for the materialized view in question will prevent query rewrite from occurring. A value of STALE_TOLERATED can be used if Oracle is to assume that the materialized views are synchronized with the base tables. This is a more permissive value that can be used if materialized views are built during a maintenance window when it is assured that none of the base tables will change when the database is available to end users. This value must also be used if the materialized view is defined on a prebuilt table. Both of these parameters can also be set at the session level using the ALTER SESSION statement.

The next step in troubleshooting is to use the DBMS_MVIEW.EXPLAIN_REWRITE diagnostic procedure that is available in Oracle 9*i*. This procedure can be called from the command line, but its more useful to modify the PL/SQL anonymous block found in $ORACLE_HOME/rdbms/demo/smxrw.sql which calls the procedure and displays the results. This procedure will pinpoint a whole host of problems associated with the query rewrite process. Messages will indicate why query rewrite did not occur. Most of these are related to structural issues associated with the materialized views, for example, missing columns or tables. The QSM series of diagnostic messages can be found in the Oracle error message documentation. Here is an example of how the anonymous block can be used to diagnose problems.

```
DECLARE
  rewrite_array SYS.RewriteArrayType := sys.RewriteArrayType();
  row_count NUMBER;  -- Number of rows in rewrite_array
  current_row NUMBER;  -- Current row in rewrite_array
  mview_name VARCHAR2(80) := 'MV_F_LOS_PROD_ROUTE';  -- MV to test

  -- Define statement text
  sql_statement VARCHAR2(4000) := 'SELECT pp.prod_dt,
  f.measure_id, SUM(f.los_nmbr)
  FROM f_los f, d_prod_period pp, d_property p
  WHERE f.prod_period_id = pp.prod_period_id
  AND f.property_id = p.property_id
  AND pp.prod_dt BETWEEN ''01-JUL-2004'' AND ''31-JUL-2004''
  GROUP BY pp.prod_dt, f.measure_id';

  query_text VARCHAR2(2000);
BEGIN
  query_text := substr(sql_statement,1,240);
    DBMS_OUTPUT.PUT_LINE(query_text);

  -- Call the query rewrite procedure
  dbms_mview.Explain_Rewrite(sql_statement, mview_name, rewrite_array);

  -- Get the row count from the array
  row_count := rewrite_array.count;

  -- Loop on all rows in the array
  FOR current_row IN 1..row_count LOOP
    DBMS_OUTPUT.PUT_LINE(rewrite_array(current_row).message);
  END LOOP;
END;
/
```

The query can successfully be rewritten to use the materialized view as indicated by the following QSM message output.

```
QSM-01033: query rewritten with materialized view, MV_F_LOS_PROD_REGION
QSM-01033: query rewritten with materialized view, MV_F_LOS_PROD_DISTRICT
QSM-01033: query rewritten with materialized view, MV_F_LOS_PROD_ROUTE
QSM-01101: rollup(s) took place on mv, MV_F_LOS_PROD_ROUTE
```

In the example output shown below, the query was modified to return the measure_dsc column from the d_measure table. This column does not exist in any of the three materialized views on the dimensional schema. This example demonstrates the enhanced Oracle 9*i* query rewrite *join back* capability to *join* the materialized view *back* to the dimension table to return the missing attribute, as indicated by the last message.

```
QSM-01033: query rewritten with materialized view, MV_F_LOS_PROD_REGION
QSM-01033: query rewritten with materialized view, MV_F_LOS_PROD_DISTRICT
QSM-01033: query rewritten with materialized view, MV_F_LOS_PROD_ROUTE
QSM-01101: rollup(s) took place on mv, MV_F_LOS_PROD_ROUTE
QSM-01102: materialized view, MV_F_LOS_PROD_ROUTE, requires join back to table,
D_MEASURE, on column, MEASURE_DSC
```

Bear in mind that these diagnostics simply indicate whether or not it's feasible to rewrite the query on the specified materialized view. Whether or not the optimizer chooses an access path using the materialized view ultimately depends upon the query cost compared with other access paths.

Another reason why query rewrite may not occur is that the query against the materialized view would actually produce worse performance than running the query on the base tables. This can be easily verified. Start by setting QUERY_REWRITE_ENABLED=FORCE at the session level. This setting causes the materialized view to be used by the optimizer regardless of cost, as long as the query can be rewritten. Then run the query and compare the

continued on page 38

Materialized Views in Action continued from page 37

response time and the number of logical reads against the query that runs against the base tables. If the query against the materialized view does indeed run faster, then the likely cause is inadequate optimizer statistics. Make sure that statistics are present on all base tables and materialized views and all related indexes. If statistics are present, increasing the sample size usually solves the problem. If the query against the materialized view is slower than the query against the base tables, then its likely that the materialized view has too many columns and/or rows. Verify that the size of the materialized view is smaller than the size of the base fact table. If the materialized view is larger then reduce the number of columns, or replace the materialized view with one at a higher level in the hierarchy. The above section on choosing aggregation levels and columns will be helpful in solving this problem. An alternative to setting QUERY_REWRITE_ENABLED=FORCE is to use the REWRITE query hint which is a more surgical approach since it can be applied to individual problem queries.

Summary

Materialized views are a powerful and valuable tool for building high performance data warehouses. There are two general methods of designing and deploying materialized views. The first method is to use the Summary Advisor utility provided by Oracle. The second method is to design them manually. The manual design method is indeed complex, but the results are usually worth it, particularly on large data warehouses. There are several steps required to design materialized views manually. The first one is to sample prospective queries. Next, a spreadsheet is built that classifies each query at the lowest level in the hierarchy, and documents what columns are accessed. This aids the designer in choosing the correct level of the hierarchies to create materialized views, and what columns should be included in each level. The third step is to choose how the materialized views will be built and maintained. The *fast refresh* method can be used if DML transactions on the base tables are low, and when the materialized views will be maintained in environments where there is no maintenance window to rebuild them. The second method is to simply drop and recreate them as part of the data loading cycle. Last of all, the queries should be regressed against the materialized view design to assure that the desired level of performance has been achieved. When performance is less than adequate, some troubleshooting should be conducted to correct any problems that may exist. Once these steps have been taken, the full potential of materialized views will be realized.

References

Dodge, G. & Gorman, T. (2000). *Essential Oracle 8i Data Warehousing.* New York, NY: John Wiley & Sons, Inc.

Kimball, R. (1996) *The Data Warehouse Toolkit.* New York, NY: John Wiley & Sons, Inc.

Oracle 9i Data Warehousing Guide Release 2 (9.2). (2002). Redwood City, CA: Oracle Corporation

Oracle 9i Database Error Messages Release 2 (9.2). (2002). Redwood City, CA: Oracle Corporation

Oracle 9i Database Reference Release 2 (9.2). 1996, 2002. (2002). Redwood City, CA: Oracle Corporation

About the Author

Jeff Maresh is an Oracle Database Architect, DBA, and application developer with over 20 years of consulting experience in the IT industry. His areas of specialty include data warehousing, process automation, and database performance tuning. Over the past several years, he has written and taught a number of training courses and provided mentoring on a number of Oracle related topics. Jeff has provided consulting services to large corporations primarily in the telecommunications, and oil and gas industries. He has been a speaker at a number of Oracle user group meetings and has written articles for a number of Oracle related publications. He is a Contributing Editor for IOUG-A *SELECT Journal,* and the recipient of the 2004 IOUG-A *SELECT Journal* Editor's Choice award. Aside from work, Jeff enjoys camping, hiking, mountain biking, and cross-country skiing in Colorado. Jeff can be reached at **jeff@maresh-usa.com**. Other papers and presentations are available at **www.evdbt.com.**