Oracle Materialized Views & Query Rewrite

An Oracle White Paper
May 2005

Oracle Materialized Views & Query Rewrite

EXECUTIVE OVERVIEW

Databases today, irrespective of whether they are data warehouses, data marts or OLTP systems, contain a wealth of information waiting to be discovered and understood. However, finding and presenting this information in a timely fashion can be a major issue, especially when vast amounts of data have to be searched. Materialized Views help solve this problem, by providing a means to access and report on this data very quickly.

INTRODUCTION

Materialized views were first introduced in Oracle8*i* and they are part of a component known as Summary Management. Your organization may already be using a materialized view, but may know it by another name, such as a summary or aggregate table. Here we will discuss how materialized views are created and managed and how the query rewrite capability will transparently rewrite your SQL query to use a materialized view to improve query response time. This allows users of the database to be completely unaware of which materialized views exist.

The materialized view should be thought of as a special kind of view, which physically exists inside the database, it can contain joins and or aggregates and exists to improve query execution time by pre-calculating expensive joins and aggregation operations prior to execution.

Today, organizations using their own summaries waste a significant amount of time manually creating summaries, identifying which ones to create, indexing the summaries, updating them and advising their users on which ones to use.

Now the DBA will only have to initially create the materialized view, it can then be automatically updated whenever changes occur to its data source. There is also a SQL Access Advisor component, which will recommend to the DBA which materialized views to create, delete and retain.

One of the biggest benefits of using materialized views will be seen by the users of the data warehouse or database. No longer will they have to be told by the DBA which materialized views exist. Instead, they can write their query against the tables or views in the database. Then the query re-write mechanism in the Oracle server will automatically re-write the SQL query to use the materialized views. Thus resulting in a significant improvement in query response time and eliminating the need for the end-user to be *'summary aware'*.

WHY USE SUMMARY MANAGEMENT

Ask any end-user of the Data Warehouse what they want from it, and they will most likely reply, fast and accurate information. But this presents a major problem for the warehouse designer because in order to answer the question ' how many of product x did we sell at location y' , a fast route to the data is required if we are going to avoid reading every row in the table.

One of the most common solutions used to solve this problem is to create summary tables or as Oracle calls them, a materialized view. This involves first understanding the typical workload and then creating materialized views which are much smaller in size and may contain, joins and or aggregates of the information required. For example, to answer the previous question, a materialized view would contain one row for every product, by region with the quantity sold. Therefore if a company sold 2000 products in 5 locations, the maximum number of rows to be read would always be 10 000, irrespective of how many items had been sold.

Obviously the materialized view must be kept accurate, but this technique means that the end-user now reads so few rows that they always receive results quickly. As databases grow to terabytes of data, it becomes increasingly important to use methods like this to improve query response time.

Many sites today create their own summary tables, therefore the extra benefit that would accrue by using Oracle Summary Management would be:

- the query rewrite mechanism in the Oracle server is transparent and will use a materialized view, even if the materialized view only partially fulfills the query requirements
- sophisticated query rewrite so that one materialized view can be used to report at different levels of aggregation such as at the week, month and year
- automatic mechanism to refresh materialized views and a single request will refresh all materialized views
- the DBA will not have to spend time trying to find out which materialized view should be created. Instead they will be provided with information on which summaries are needed based on previous queries to the database or data warehouse.

COMPONENTS OF SUMMARY MANAGEMENT

There are five components, which comprise Summary Management:

- Dimensions
- Materialized Views
- Refresh
- Query Rewrite
- SQL Access Advisor

They do not all have to be used, but maximum advantage will be achieved when more components that are selected. We will now look at these components in more detail.

Schema Requirements

There is no restriction on the type or design of schema that may be used with materialized views. Therefore in a data warehousing environment, the schema could be a snowflake design but this is not a requirement.

For the database designer who is familiar with database design techniques in production systems, different rules and techniques must be used in a data warehouse. For example, production databases are usually normalized, therefore in this instance the representation for the time dimension is most likely to result in three tables, one for *date*, *month*, and *year*. There would be join conditions that connect each *date* row to one and only one *month* row, and each *month* row to one and only one *year* row. The data warehouse implementation would typically result in a fully denormalized *time* dimension table, where the *date*, *month*, and *year* columns are all in the same table. However, you can use materialized views whether your design uses normalized or denormalized tables.

DIMENSIONS

Before creating a materialized views, the first step is to review the schema and identify the dimensions. A dimension is an Oracle object which defines a hierarchical (parent/child) relationships between columns, where all the columns do not have to come from the same table. It is highly recommended that dimensions on your data are defined because they help query rewrite and the summary advisor make better decisions.

Another issue for the database designer is that frequently queries will not involve the dimension column directly but refer to a column which is related to the dimension. e.g. the query refers to Tuesday rather than a specific date. Therefore when dimensions are defined, these relationships between the dimension columns and other columns in the table must also be described.

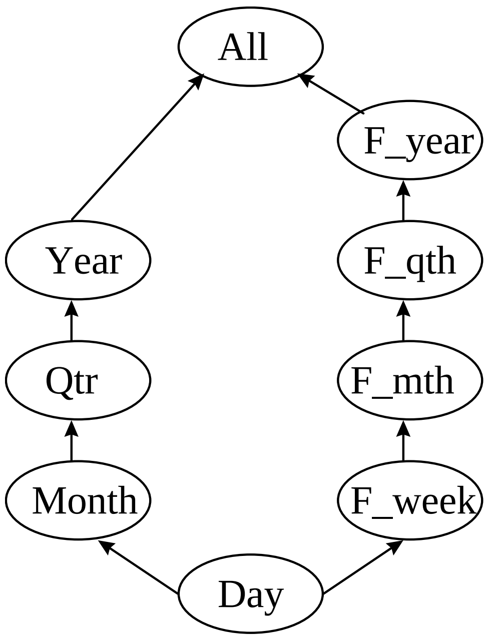
Figure 1 illustrates a time dimension, which contains two hierarchies. From a given date, one hierarchy tells us to which fiscal week or month or year this date refers, and the other hierarchy defines the relationship between a day, month, quarter and year.

When defining a hierarchy, multiple columns may be specified to describe the hierarchy. e.g. if *City* is unique within each *State* but is not unique across *States*, then a Geography hierarchy might need to be specified as (*Country, State, <State, City>*) to satisfy the strict 1:n hierarchical relationship from the *City* level to the *State* level.

A dimension can be created using one of two methods:

- SQL CREATE DIMENSION statement as illustrated in Figure 2
- Dimension Wizard in Oracle Enterprise Manager

Figure 1 Illustrates the Time Dimension



Drawing the dimension as per figure 1 may assist the DBA in the definition process. Each bubble represents a *level* in the dimension and is declared using the LEVEL clause. The dimension *hierarchy* is declared using the HIERARCHY clause. Summary management will also rely on the DBA defining constraints to ensure that the columns of each level in the hierarchy level are non-null.

In Figure 2 we can see the SQL statement, which will create this dimension. The level name corresponds to columns in the dimension tables. Then each hierarchy is described using those level names. Finally the ATTRIBUTE clause is used to define those items which have a direct relationship. Therefore attribute calendar_month_name has a relationship with the level *month*.

The 1:n join relationships among the dimensions are declared using the JOIN KEY clause and between a fact table and a dimension table, they are represented with FOREIGN KEY and NOT NULL constraints on the fact table.

In Oracle Database 10*g* Release 2 it is also possible when defining a dimension to specify that a level in the hierarchy is to be ignored.

Hints on Defining Dimensions

To help you create dimensions, here are a few simple steps to follow.

1. Identify all dimensions and dimension tables in the schema. If the dimensions are normalised, that is, stored in more than one table then check that a join between the dimension tables guarantees that each child-side row joins with one and only one parent-side row. In the case of denormalized dimensions, check that the child-side columns uniquely determine the parent-side (or attribute) columns. Failure to abide by these rules may result in incorrect results being returned from queries. For example, a hierarchy of calendar_week CHILD_OF calendar_month, could return incorrect results because a calendar_week could span two calendar months.
2. Identify the hierarchies within each dimension. For example, *day* is a child of *month* (we can aggregate *day* level data up to *month)*, and *quarter* is a child of *year.*
3. Identify the attribute dependencies within each level of the hierarchy. For example, identify that *calendar_month_name* is an attribute of *month.*
4. Identify joins from the fact table in a data warehouse, to each dimension, and check that each join can guarantee that each fact row joins with one and only one dimension row. This condition must be declared, and optionally enforced, by adding FOREIGN KEY and NOT NULL constraints on the fact key columns and PRIMARY KEY constraints on the parent-side join keys. These constraints can be enabled using the NOVALIDATE option to avoid the time required to validate that every row in the table conforms to the constraints. The RELY clause is also required for all nonvalidated constraints to make them eligible for use in query rewrite.

Figure 2 SQL Statement to Create Time Dimension

```
CREATE DIMENSION    times_dim
LEVEL day            IS TIMES.TIME_ID
LEVEL month          IS TIMES.CALENDAR_MONTH_DESC
LEVEL quarter        IS TIMES.CALENDAR_QUARTER_DESC
LEVEL year           IS TIMES.CALENDAR_YEAR
LEVEL fis_week       IS TIMES.WEEK_ENDING_DAY
LEVEL fis_month      IS TIMES.FISCAL_MONTH_DESC
LEVEL fis_quarter    IS TIMES.FISCAL_QUARTER_DESC
LEVEL fis_year       IS TIMES.FISCAL_YEAR
HIERARCHY cal_rollup
(day CHILD OF   month CHILD OF quarter CHILD OF  year

HIERARCHY fis_rollup (day CHILD OF fis_week CHILD
OF fis_month CHILD OF fis_quarter CHILD OF fis_year)

ATTRIBUTE day  DETERMINES  (day_number_in_week,
day_name, day_number_in_month,  calendar_week_number)
```

```
ATTRIBUTE month DETERMINES (calendar_month_desc,
calendar_month_number, calendar_month_name,
days_in_cal_month, end_of_cal_month)
ATTRIBUTE quarter DETERMINES(calendar_quarter_desc,
calendar_quarter_number, days_in_cal_quarter,
end_of_cal_quarter)
ATTRIBUTE year DETERMINES (calendar_year,
days_in_cal_year, end_of_cal_year)
ATTRIBUTE fis_week DETERMINES
(week_ending_day,  fiscal_week_number)  ;
```

MATERIALIZED VIEWS

Once the dimensions have been defined, the materialized views can be created. For the moment we will concentrate on what a materialized view is, but later we will see how the advisory functions will recommend which materialized views to create.

A *materialized view* definition can include aggregation, such as SUM MIN, MAX, AVG, COUNT(*), COUNT(x), COUNT(DISTINCT), VARIANCE or STDDEV, one or more tables joined together and a GROUP BY. It may be indexed and partitioned and basic DDL operations such as CREATE, ALTER, and DROP may be applied.

Since a materialized view is an object in the database then in many ways, a materialized view behaves like an index because:

- the purpose of the materialized view is to increase query execution performance
- the existence of a materialized view is transparent to SQL applications, so a DBA can create or drop materialized views at any time without affecting the SQL applications
- a materialized view consumes storage space and must be updated when the underlying detail tables are modified

Many sites already have a data warehouse where they have defined their own summaries. Therefore existing summaries may be registered for use by query rewrite, rather than forcing the user to regenerate their summary table from scratch.

Creating a Materialized View

A materialized view is created using the *CREATE MATERIALIZED VIEW* statement. Figure 3 illustrates the creation of a materialized view called *costs_mv* that computes the sum of *costs* by *time* and *prod_nam*.

When a materialized view is defined a few simple guidelines should be followed. The SELECT list should contain all of the GROUP BY columns and the GROUP BY columns should be simple columns. The expression to be aggregated can be any

SQL value expression that does not contain a subquery or nested aggregate function.

The materialized view can have its own storage specification so that you can specify in which tablespace it is to be stored and the size of its extents. You can also include the partition clause so that the contents of the materialized view can be stored in many tablespaces.

Both tables and views can be used in a materialized view definition. Therefore, referring to the previous example, costs could be a table and product could be a view. Any view can be used provided it does not have user-varying data through the use of functions like SYSDATE and USER.

Figure 3 SQL Statement to Create Materialized View

```
CREATE MATERIALIZED VIEW costs_mv
PCTFREE 0 STORAGE (initial 8k next 8k pctincrease 0)
BUILD IMMEDIATE
REFRESH FAST ON DEMAND
ENABLE QUERY REWRITE
AS SELECT
time_id, prod_name, SUM( unit_cost) AS sum_units,
COUNT(unit_cost) AS count_units, COUNT(*) AS cnt
FROM costs c, products p
WHERE c.prod_id = p.prod_id
GROUP BY time_id, prod_name;
```

Using your own pre-built materialized views

Sites that already have a Data Warehouse may already have summaries and procedures to maintain them. Therefore, they will want to take advantage of query rewrite without recreating their summaries.

Pre-existing tables can be registered as materialized views by using the CREATE MATERIALIZED VIEW statement with the ON PREBUILT TABLE clause. The name of the materialized view, must be the same as the table name and the SELECT clause describing the query that creates this table must also be provided. It may not always be possible to ensure that the precision of the query matches the precision of the table. To overcome this problem, the clause WITH REDUCED PRECISION is included in the specification.

Index selection for Materialized Views

Depending on the number of rows in the materialized view it may be necessary to create indexes on the materialized views. Therefore, consideration should be given to first creating a unique, local index which contains all of the materialized view keys. Other indexes could include single-column bitmap indexes on each materialized view key column. The index required for fast refresh is automatically created when the materialized view is first defined. When creating indexes, don't forget to consider the storage space requirements of each index and the impact those indexes will have on the refresh time. If you are unsure which indexes to create then consider running the SQL Access Advisor to see what it recommends.

What can this Materialized View Do?

Prior to creating a materialized view or once it is created, the DBA may wonder what is possible with this materialized view, such as it is fast refreshable and if not, why not. The procedure DBMS_MVIEW.EXPLAIN_MVIEW can provide this information.

Referring to our materialized view which we created in Figure 3, if we remove the COUNT(*) from the definition and then call the procedure DBMS_MVIEW.EXPLAIN_MVIEW as illustrated in Figure 4. It will tell us that Partition Change Tracking (PCT) is available because the costs table is partitioned and all types of query rewrite are possible However, Fast Refresh is not possible after DML because COUNT(*) is missing from the materialized view.

Figure 4 Explain Materialized View Example

```
TRUNCATE TABLE  mv_capabilities_table;


EXEC DBMS_MVIEW.EXPLAIN_MVIEW (‘ SELECT  time_id,
prod_name, SUM( unit_cost) AS sum_units,
COUNT(unit_cost) AS count_units, FROM costs c,
products p WHERE c.prod_id = p.prod_id
GROUP BY time_id, prod_name’);

SELECT capability_name, possible, related_text, msgtxt
FROM  mv_capabilities_table;


PCT_TABLE: Y           COSTS:

PCT_TABLE: N           PRODUCTS:   relation is not a
partitioned table

REFRESH_COMPLETE: Y

REFRESH_FAST: Y

REFRESH_FAST_AFTER_ANY_DML: N    see the reason why

REFRESH_FAST_AFTER_ONETAB_DML is disabled

REFRESH_FAST_AFTER_INSERT: Y

REFRESH_FAST_AFTER_ONETAB_DML: N    COUNT(*) is not
present in the select list

REFRESH_FAST_PCT: Y
```

```
REWRITE: Y

REWRITE_FULL_TEXT_MATCH: Y

REWRITE_GENERAL: Y

REWRITE_PARTIAL_TEXT_MATCH: Y

REWRITE_PCT: Y
```

Tuning a Materialized View

The EXPLAIN_MVIEW procedure advises what is possible with a materialized view, but it doesn't show you how to fully optimize the materialized view to its full potential. This is when the procedure TUNE_MVIEW should be used. Given the definition of a materialized view, it will show you how to create your materialized view so that it is fast refreshable and can take advantage of as many types of query rewrite as possible.

Referring once again to the materialized view in Figure 3. We know from EXPLAIN_MVIEW that fast refresh is not possible if the COUNT(*) is not present. If this materialized view is given as input to TUNE_MVIEW, as illustrated in Figure 5, it will generate a new materialized view definition, which is shown in Figure 6.

In order to use TUNE_MVIEW a directory path must be specified using the CREATE DIRECTORY command, which specifies where the results will be stored. The full CREATE MATERIALIZED VIEW statement is given as input to TUNE_MVIEW and the results are stored in a unique task.

Figure 5 Tune Materialized View Example

```
CREATE DIRECTORY TUNE_RESULTS AS '/tuning/';

GRANT READ, WRITE ON DIRECTORY TUNE_RESULTS TO PUBLIC;


VARIABLE task_mv VARCHAR2(30);

VARIABLE create_mv_ddl VARCHAR2(4000);


EXECUTE :task_mv := 'cust_mv';

EXECUTE :create_mv_ddl := ' -

CREATE MATERIALIZED VIEW cust_mv –

REFRESH FAST   ENABLE QUERY REWRITE AS -

SELECT  time_id, prod_name, SUM( unit_cost) AS
sum_units, COUNT(unit_cost) AS count_units -
```

```
FROM costs c, products p GROUP BY time_id, prod_name';
WHERE c.prod_id = p.prod_id GROUP BY time_id,
prod_name');
```

```
EXECUTE DBMS_ADVISOR.TUNE_MVIEW(:task_mv,
:create_mv_ddl);
```

The recommendations from TUNE_MVIEW are stored in an advisor task. They can easily be retrieved, by calling the procedure GET_TASK_SCRIPT, and placing them in a file, using the procedure CREATE_FILE, as illustrated below.

```
EXECUTE DBMS_ADVISOR.CREATE_FILE -

(DBMS_ADVISOR.GET_TASK_SCRIPT(:task_mv),-

'TUNE_RESULTS', 'mv_create.sql');
```

Figure 6 shows the complete output from TUNE_MVIEW, which includes the new materialized view statement and the required materialized view logs.

Figure 6 Recommendations from TUNE_MVIEW

```
CREATE MATERIALIZED VIEW LOG ON "SH"."COSTS" WITH ROWID,
SEQUENCE("TIME_ID","UNIT_COST") INCLUDING NEW VALUES;

ALTER MATERIALIZED VIEW LOG FORCE ON "SH"."COSTS"
ADD ROWID, SEQUENCE("TIME_ID","UNIT_COST")
INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW LOG ON "SH"."PRODUCTS"
WITH ROWID, SEQUENCE("PROD_NAME")INCLUDING NEW VALUES;

ALTER MATERIALIZED VIEW LOG FORCE ON "SH"."PRODUCTS"
ADD ROWID, SEQUENCE("PROD_NAME") INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW SH.CUST_MV
REFRESH FAST WITH ROWID
ENABLE QUERY REWRITE
AS SELECT SH.PRODUCTS.PROD_NAME C1, SH.COSTS.TIME_ID C2,
SUM("SH"."COSTS"."UNIT_COST")M1,
COUNT("SH"."COSTS"."UNIT_COST") M2, COUNT(*) M3 FROM
SH.PRODUCTS, SH.COSTS
GROUP BY SH.PRODUCTS.PROD_NAME, SH.COSTS.TIME_ID;
```

Materialized View Invalidation

Materialized views are constantly being monitored to ensure that the data they contain is fresh. The purpose of invalidating the materialized view is to ensure that invalid data is not returned. A materialized view will be marked as stale whenever an object on which it is based is changed.

The state of a materialized view can be determined by querying the table USER_MVIEWS. If any column in this table has the value NEEDS_COMPILE, issue the command, ALTER MATERIALIZED VIEW COMPILE to obtain its current status.

Security Implications

Some information in the database may have restricted access and query rewrite could be viewed as a mechanism for bypassing security. However, since all security verification is performed by the Oracle Database, far greater protection of the data and materialized views is provided. To prevent unauthorized access to either materialized views or detail tables, using CREATE MATERIALIZED VIEW requires the CREATE MATERIALIZED VIEW privilege, SELECT WITH GRANT privilege on the detail tables and SELECT WITH GRANT and INSERT privileges on the materialized view container object. In addition, if a user has access to the tables in a request and one or more materialized view have been defined on those tables, then the user will be given access to the materialized views regardless of the privileges attached to the materialized view container table. Therefore no matter where the query originates from, access to the data is only possible if you pass the security checks.

LOADING & REFRESHING THE MATERIALIZED VIEW

Historically, one of the problems of using summary tables has been the initial loading and subsequent updating of the summary. These issues are now addressed because summary management provides mechanisms to:

- fully refresh the data
- perform a fast refresh, that is add/merge only the changes
- automatically update a materialized view when ever changes are made

Therefore the DBA must consider how much time is required to create and maintain each materialized view and balance that against the performance gains achieved by using this materialized view.

Oracle Database 10*g* provides the following refresh methods:

- complete
- fast (only the changes are applied)
- force, do a fast if possible, otherwise perform a complete refresh
- partition change tracking (fast refresh rows in changed partitions only)

These operations may be performed:

- on demand refreshing by:
    - specific materialized views (DBMS_MVIEW.REFRESH)

    - those materialized views dependent on a table (DBMS_MVIEW.REFRESH_DEPENDENT)
    - all materialized views (DBMS_MVIEW.REFRESH_ALL_MVIEWS)
- on commit, whenever the tables on which the materialized view is defined are changed

The ON DEMAND refresh is achieved by calling one of the procedures listed above, thus giving the DBA total control over when a materialized view is update.

If the ON COMMIT refresh method is chosen, whenever a materialized view is affected by changes made to the source data, the materialized view will automatically be updated to reflect this data. However, it should be remembered that this update to the materialized view occurs as part of the commit processing in the transaction where the changes to the base table are made. Therefore, the commit will take slightly longer, because changes are being made to both the original table and then any materialized view whose definition includes that table.

Complete Refresh

When a complete refresh of a materialized view occurs, all data is removed and then it is reloaded. Depending on the size of the materialized view, this could be a time consuming operation. Complete refresh is a good technique to use when:

- the number of new rows to be inserted is more than 50% of the cardinality of the tables on which the materialized view is based
- there is no index on the materialized view that is usable for merging
- the time required to perform a fast refresh is longer than a complete refresh

Fast Refresh

Some materialized views could be very large and the time required to regularly perform a complete refresh may not be available. The alternative is a fast refresh where only the changes to the detail table are applied against the materialized view. New data loaded into any table in the warehouse is identified and any materialized view referencing that table is automatically updated with the new data.

In order to perform a fast refresh operation, changes made to the data must be recorded and this is achieved in one of two ways. If your data is only ever inserted into the database using SQL*Loader direct path, then the refresh mechanism will detect this and identify the new data to be loaded. However, many data changes will occur via the SQL commands, INSERT, UPDATE and DELETE. In this case a MATERIALIZED VIEW LOG is required on each of the tables on which the materialized views are based.

Only one log is required per table and the materialized view log is on the table, not the materialized view. Therefore if you only had 6 tables in your database that were changing, you would only need 6 materialized view logs. But you could have any number of materialized views using those logs.

It should be noted that not all materialized views are fast refreshable and confirmation of whether it is possible can be obtained by calling the procedure DBMS_MVIEW.EXPLAIN_MVIEW. This procedure will also advise what needs to be done to the materialized view to make it fast refreshable. The procedure TUNE_MVIEW can then be called to generate a script, which will show, if it is possible, how to make the materialized view fast refreshable.

Partition Change Tracking Refresh

Oracle Database 10*g* provides a component known as Partition Change Tracking (PCT) which transparently detects when changes to partitions occur and then determines whether the operation has made the data in the materialized view inconsistent. For example, a merge partition or add partition operation will not affect the materialized view and can be performed without causing the materialized view to be marked as stale.

Partition Change Tracking can also be used to identify which materialized view rows are affected by partition operations. For example, if a detail table partition is truncated or dropped, the fast refresh procedure can use PCT to identify the affected rows in the materialized view and delete them. If the tables on which the materialized view is based are partitioned, and a fast refresh is required, a Partition Change Tracking (PCT) refresh may be performed if it is consider to be better than the materialized view log based fast refresh. In Oracle Database 10*g* Release 2, the restriction has been lifted which requires a materialized view log to be present on the PCT table when fast refresh is required.

When a PCT refresh is performed, the partitions where data has changed are identified and only the contents of those partitions are recomputed. Therefore, if the table has many partitions and only one or a few partitions have changed, this can be a very fast method for updating the materialized view with the very latest data. The procedure DBMS_MVIEW.EXPLAIN_MVIEW will advise whether a materialized view can use PCT.

Refresh & Constraints

It was stated earlier that ideally constraints, especially foreign key ones, should be defined on the fact table to ensure that a row in the fact table can be matched with a dimension. At the very mention of the word constraints, some DBA's may throw their hands in the air and declare that there will be no constraints in this database because of a possible performance overhead. However, the DBA can rest assured that by using the clause

ALTER TABLE <table name> ENABLE NOVALIDATE CONSTRAINT <name>

constraints can be enabled immediately without checking the data. If data is loaded into the fact table using SQL*Loader direct path, then by default all constraints are disabled. After the load of the fact table, issuing the enable NOVALIDATE statement, will immediately enable the constraints without checking the data. Therefore, there is no impact on data load time and no time required to enable the constraint. However, since no validation of the data loaded is performed, it is very important to ensure that all loaded data will not violate any integrity constraints and that the RELY clause is included so the constraint will be used by Summary Management.

Data Availability

Whilst refreshing the data, materialized views are still available for atomic and fast log based refresh. Query rewrite will ignore any materialized view that is being refreshed.

QUERY REWRITE

One of the major benefits of using summary management, which the end-user will really appreciate, is the query rewrite capability. It is a query optimization technique that transforms a user query written in terms of tables and views, to execute faster by fetching data from materialized views. It is completely transparent to the end user, requiring no intervention or hints in the SQL application because the Oracle Database will automatically rewrite any appropriate SQL application to use the materialized views. Although all the references in this document will refer to the SQL SELECT clause, query rewrite will also apply on INSERT and CREATE TABLE statements that include the SELECT clause.

Query rewrite can be used against a wide variety of queries. It should be noted that the relationships declared in dimension objects are not required to be enforced, but they are assumed to be true, when running in QUERY_REWRITE_INTEGRITY = TRUSTED or ENFORCED mode. If the declaration of a relationship doesn't match with the actual relationship that exists in the table data, then when query rewrite makes use of the flawed relationship declaration to rewrite a query, the rewritten query will most probably produce an incorrect result. However, by defining the relationships and using constraints, so that the system guarantees correctness of the data,, the reports generated can be relied upon to contain the correct results. Fast, accurate query results are significant benefits arising from the minimal effort and overhead required when enforcing system integrity.

The composition of the query does not have to exactly match the definition of the materialized view because this would require that the DBA knew in advance what queries would be executed against the data. This is of course impossible, especially with respect to data warehouses where one of the main benefits to an organization is to suddenly execute a new query. Therefore, query rewrite will still occur even if using the materialized view can satisfy only part of the query.

Enabling/Disabling Query Rewrite

For Query Rewrite to occur it must be enabled on both the materialized view and for the session, although it is on by default in Oracle Database 10*g*. It can be enabled on the materialized view using the ENABLE QUERY REWRITE clause in the CREATE or ALTER MATERIALIZED VIEW statement.

You can disable query rewrite in your session with the

ALTER SESSION SET QUERY_REWRITE_ENABLED = FALSE

or for all sessions with

ALTER SYSTEM SET QUERY_REWRITE_ENABLED = FALSE

or using the DISABLE QUERY REWRITE clause on a specific materialized view.

Types of Query Rewrite

There are various types of query rewrite which are possible in Oracle Database 10*g* and the following examples illustrate some of what is possible using the materialized view shown in Figure 7.

Figure 7 Materialized Views for Query Rewrite Examples

```
CREATE MATERIALIZED VIEW all_cust_sales_mv
BUILD IMMEDIATE
REFRESH COMPLETE
ENABLE QUERY REWRITE
AS  SELECT   c.cust_id, sum(s.amount_sold) AS dollars,
p.prod_id,  sum(s.quantity_sold) as quantity
FROM  sales s , customers c, products p
WHERE c.cust_id = s.cust_id  AND s.prod_id = p.prod_id
GROUP BY  c.cust_id, p.prod_id;
```

Exact Match

The simplest kind of query rewrite takes place when a materialized view definition exactly matches a query definition. That is, the tables in the FROM clause, joins in the WHERE clause and the keys in the GROUP BY clause match exactly between the query and the materialized view. For example, given the following query:

```
SELECT c.cust_id,  sum(s.quantity_sold) as quantity
FROM   sales s , customers c, products p
WHERE  c.cust_id = s.cust_id AND s.prod_id = p.prod_id
GROUP BY  c.cust_id, p.prod_id;
```

it is rewritten by Oracle Database 10*g* to use the materialized view all_cust_sales_mv

JoinBack

Some times a query may contain reference to a column which is not stored in the materialized view, but it can be obtained by joining back the materialized view to the appropriate dimension table. For example, consider the previous query, but instead of reporting on customer id, the report uses the customers name.

```
SELECT c.cust_last_name,
sum(s.quantity_sold) as quantity
FROM  sales s , customers c, products p
WHERE c.cust_id = s.cust_id AND s.prod_id = p.prod_id
GROUP BY  c.cust_last_name, p.prod_id;
```

This query references the column c.cust_last_name which is not in the materialized view *all_cust_sales_mv*, but c.cust_last_name is functionally dependent on c.cust_id because of the hierarchical relationship between them. This means this query can be rewritten in term of *all_cust_sales_mv*, which is joined back to the customers table in order to obtain c.cust_last_name column.

Rollup & Aggregate Rollup

When a query requests aggregates such as SUM(sales) at a higher level in a hierarchy than the level at which the aggregates in a materialized view are stored, then the query can be rewritten by using the materialized view and rolling up its aggregates to the desired level.

For example, our materialized view *all_cust_sales_mv*, groups data at the customer level, but we would like to report data at the state level. A customer dimension has been created which describes the relationship between customer and state. Therefore the following query will use our materialized view *all_cust_sales_mv* to produce the report where it will aggregate together all the data for a customer and then roll it up to the state level.

```
SELECT c.cust_state_province,
sum(s.quantity_sold) as quantity
FROM sales s , customers c, products p
WHERE c.cust_id = s.cust_id AND s.prod_id = p.prod_id
GROUP BY  c.cust_state_province;
```

Data Subsets

When a materialized view contains all of the data, this can result in a very large materialized view. Therefore materialized view can be defined which only contain a subset of the data, as shown in Figure 8, where we only have data for the cities Dublin, Galway, Hamburg and Istanbul.

Figure 8 Materialized Views containing a subset of data

```
CREATE MATERIALIZED VIEW some_cust_sales_mv
BUILD IMMEDIATE
REFRESH COMPLETE
ENABLE QUERY REWRITE
AS
SELECT   c.cust_id, sum(s.amount_sold) AS dollars,
p.prod_id, sum(s.quantity_sold) as quantity
FROM  sales s , customers c, products p
WHERE c.cust_id = s.cust_id AND s.prod_id = p.prod_id
AND   c.cust_state_province IN
('Dublin','Galway','Hamburg','Istanbul')
GROUP BY  c.cust_id, p.prod_id;
```

This materialized view can now be used to satisfy queries which contains ranges, IN and BETWEEN clauses such as the one shown below.

```
SELECT c.cust_state_province,
sum(s.quantity_sold) as quantity
FROM  sales s , customers c, products p
WHERE c.cust_id = s.cust_id AND s.prod_id = p.prod_id
AND      c.cust_state_province IN ('Dublin','Galway')
GROUP BY  c.cust_state_province;
```

Using Multiple Materialized Views

Sometimes there may not be a single materialized view that can be used to obtain the results for a query, but query rewrite would be possible if the results from several materialized views could be joined together. This type of query rewrite is possible from Oracle Database 10*g* Release 2.

Suppose there are three materialized views, which record sales by the regions, EMEA, APAC and Americas. In Figure 9 there is a query, which uses two of those materialized views, so that the sales for the UK and the US can be reported.

Figure 9 Query Rewrite using Multiple Materialized Views

```
CREATE MATERIALIZED VIEW emea_sales_mv
BUILD IMMEDIATE
REFRESH COMPLETE
ENABLE QUERY REWRITE
AS
SELECT   c.cust_id, sum(s.amount_sold) AS dollars,
p.prod_id, sum(s.quantity_sold) as quantity
FROM  sales s , customers c, products p
WHERE c.cust_id = s.cust_id AND s.prod_id = p.prod_id
AND   c.country_id IN (52779, 52789,52770,52777)
GROUP BY  c.cust_id, p.prod_id;
```

```
CREATE MATERIALIZED VIEW americas_sales_mv
BUILD IMMEDIATE
REFRESH COMPLETE
ENABLE QUERY REWRITE
AS
SELECT   c.cust_id, sum(s.amount_sold) AS dollars,
p.prod_id, sum(s.quantity_sold) as quantity
FROM  sales s , customers c, products p
WHERE c.cust_id = s.cust_id AND s.prod_id = p.prod_id
AND   c.country_id IN (52790, 52772,52773)
GROUP BY  c.cust_id, p.prod_id;
```

Show sales by product for the US and the UK. This query will be rewritten to use the materialized views *americas-sales_mv* and *emea_sales_mv*.

```
SELECT   c.country_id, t.country_name,
sum(s.amount_sold) AS dollars, p.prod_id,
sum(s.quantity_sold) as quantity
FROM  sales s , customers c, products p , countries t
WHERE c.cust_id = s.cust_id AND s.prod_id = p.prod_id
AND   c.country_id IN (52790, 52789)     and
c.country_id=t.country_id
GROUP BY  c.country_id, p.prod_id, t.country_name;
```

Query Rewrite Integrity Modes

Different users may have different data quality requirements, for this reason, Oracle Database 10*g* supports three integrity levels, which are selected by the parameter QUERY_REWRITE_INTEGRITY

- STALE_TOLERATED
- TRUSTED
- ENFORCED (default)

The STALE_TOLERATED mode is the least restrictive. In this mode, the optimizer uses the same trusted relationships as TRUSTED mode, and in addition will use materialized views that are known to be stale.

In the TRUSTED mode, the optimizer trusts that unenforced relationships such as those declared in dimensions and RELY constraints are correct. In this mode, the optimizer will also use prebuilt materialized views, even though the optimizer cannot verify that the contents of the materialized view are in fact the same as the results returned by the materialized view's defining query.

The ENFORCED mode, which is the default, the optimizer will only use materialized views that it knows contain fresh data and it will only use validated relationships. Therefore you may find that query rewrite will not occur using this method, but it will occur using the less restrictive TRUSTED or STALE_TOLERATED modes.

Are the Results Correct

Whenever a SQL query uses a materialized view rather than the actual source of the data, there are instances when the results returned may be different.

1. A materialized view can be out of synchronization with the detail data. This generally happens because the refresh procedure is pending and STALE_TOLERATED integrity mode has been selected.
2. Join columns may violate referential integrity. In this case, some child-side rows are not rolled up into exactly one parent-side row. To avoid this situation, use system enforced integrity whose overheads are negligible and benefits are significant.
3. If a *rolling materialized view* is created which contains information about rows that no longer exist in the detail data. For example, the materialized view may contain 18 months worth of data, but the detail tables only contain the last 6 months. Therefore, if a query were ever to go against the base table rather than the materialized view then different results would be shown.

Explain Rewrite

When using Query Rewrite the most frequently asked questions are ' will this query rewrite?' or 'why didn't this query rewrite?' These questions can be answered by using the procedure DBMS_MVIEW.EXPLAIN_REWRITE and an example of its use is shown below in Figure 10. Therefore, this information can be known even before the query is ever run.

The query text is passed as a long string and the procedure stores the results of its findings in the table REWRITE_TABLE, which must be queried to see the results of the procedure.

The REWRITE_TABLE can be queried directly via SQL, but a demo file, smxrw.sql can format the output as illustrated in figure 10. In the example below we can see that the materialized view *all_cust_sales_mv* will be used for this query and it also shows the rewritten query and pre and post query costs.

Figure 10 Explain Rewrite Example

```
set serveroutput on

DECLARE

querytxt VARCHAR2(1500) := 'SELECT c.cust_id,
sum(s.quantity_sold) as quantity FROM   sales s ,
customers c, products p WHERE  c.cust_id = s.cust_id
```

```
AND s.prod_id = p.prod_id GROUP BY  c.cust_id,
p.prod_id';

BEGIN

  SYS.XRW('', 'QUERY_TXT, REWRITTEN_TXT,
QUERY_BLOCK_NO, COSTS', querytxt);

END;
```

```
======================================================
------------------------- ANALYSIS OF QUERY REWRITE -------------------------
>> QRY BLK #: 0
>> MESSAGE  : QSM-01209: query rewritten with materialized view,
ALL_CUST_SALES_MV, using text match algorithm
>> QUERY    : SELECT c.cust_id,  sum(s.quantity_sold) as quantity FROM
sales s , customers c, products p WHERE  c.cust_id = s.cust_id AND s.prod_id =
p.prod_id GROUP BY  c.cust_id, p.prod_id
>> RW QUERY : SELECT ALL_CUST_SALES_MV.CUST_ID,
ALL_CUST_SALES_MV.QUANTITY QUANTITY FROM
SH.ALL_CUST_SALES_MV
>> ORIG COST: 4203.03120092711
 RW COST: 164.901117031229
============================ END OF MESSAGES
```

SQL ACCESS ADVISOR

When the decision is first made to use materialized views an initial set has to be defined. Now this could be quite a challenge for the DBA, especially if they don't know the business very well, or the queries raised by the application are rather unpredictable. The same issue also arises around which indexes should be created in the database.

To help resolve this problem, Summary Management contains a component called the SQL Access Advisor which can either be invoked by calling one of the many procedures in the DBMS_ADVISOR or from Oracle Enterprise Manager and it can provide the following information:

- Recommend materialized views and indexes based on a collected or hypothetical workload
- Estimates the size of a materialized view
- Define filters to use against a workload
- Load and validate a workload

- Purge filters, workloads, and results
- Provides a script to implement the recommendations

Before using the SQL Access Advisor, the DBA should run the procedure DBMS_STATS, to gather cardinality information on the tables and materialized views in the database. This information is used as part of the prediction process.

Providing a Workload

Although the SQL Access Advisor can recommend materialized views without a workload, it performs best when it is has a workload, which in Oracle Database 10*g* can be provided in the form of:

- User-Defined (DBMS_ADVISOR.IMPORT_SQLWKLD_USER)
- SQL Tuning Set (DBMS_ADVISOR.IMPORT_SQLWKLD_STS)
- Current contents of the SQL Cache (DBMS_ADVISOR.IMPORT_SQLWKLD_SQLCACHE)
- Summary Advisor 9*i* workload (DBMS_ADVISOR.IMPORT_SQLWKLD_SUMADV)

A user-defined workload involves storing the queries in a table in the database. This will then be read by the SQL Access Advisor and taken as its workload.

Alternatively, the current queries in the SQL Cache can be made into a workload and used as input to the SQL Access Advisor.

Although only one workload can be used at a time as input to the recommendation procedure DBMS_ADVISOR.EXECUTE_TASK, multiple workloads may be stored in the database and then compared to see which one generates the best recommendations.

Filtering Workloads

A workload doesn't have to be considered in its entirety, it may be filtered using the SET_SQLWKLD_PARAMETER   A number of filters are available and these include application name, tables used in the queries, queries executed during a specific time, the query frequency and the table owners.

Recommending Materialized Views & Indexes

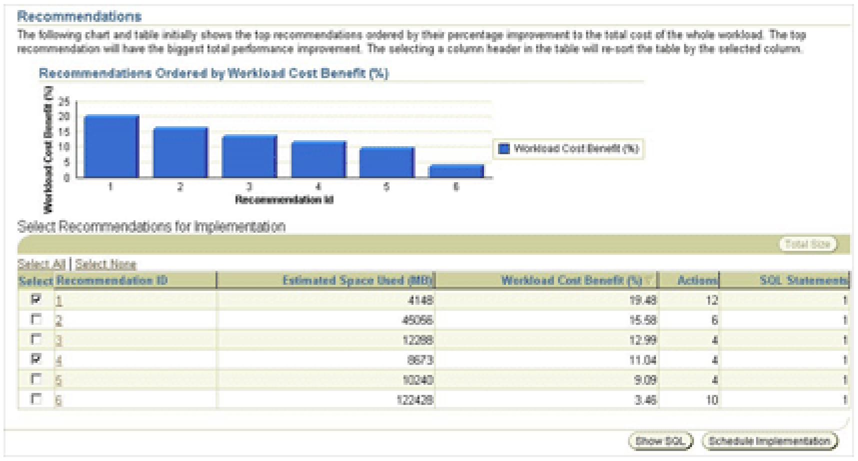
All the information that is required to generate a set of recommendations and the actual recommendations are stored in a task. These recommendations as to which materialized views and indexes to create, can be obtained ein two ways. One approach is to use the SQL Access Advisor Wizard in Oracle Enterprise Manager, which takes you step by step through the process of recommending materialized views and actually implements them.

Alternatively, the recommendations can be generated by calling the procedure EXECUTE_TASK. Irrespective of the method chosen, the SQL Access Advisor will recommend whether to drop or retain existing materialized views and indexes, and what if any to create.

Implementing Recommendations

One of the many advantages of using the SQL Access Advisor Wizard, shown in Figure 11, is that after reviewing the recommendations, you can select which ones to implement and Enterprise Manager will schedule a job to implement them.

Figure 11 SQL Access Advisor



Recommendations

The following chart and table initially shows the top recommendations ordered by their percentage improvement to the total cost of the whole workload. The top recommendation will have the biggest total performance improvement. The selecting a column header in the table will re-sort the table by the selected column.

Recommendations Ordered by Workload Cost Benefit (%)
Workload Cost Benefit (%)
25
20
15
10
5
0
1
2
3
4
5
6
Recommendation Id
Workload Cost Benefit (%)

Select Recommendations for Implementation

Select	Recommendation ID	Estimated Space Used (MB)	Workload Cost Benefit (%)	Actions	SQL Statements
✓	1	4148	19.48	12	1
☐	2	45056	15.58	6	1
☐	3	12288	12.99	4	1
✓	4	8673	11.04	4	1
☐	5	10240	9.09	4	1
☐	6	122428	3.46	10	1

Alternatively the procedure GET_TASK_SCRIPT in conjunction with CREATE_FILE can be called to created a SQL file containing the statements required to implement these recommendations.

CONCLUSION

Anyone who is looking to improve the performance of queries in their Data Warehouse or database should seriously consider implementing materialized views if they can pre-compute the results of some queries. There is only a minimal effort required to create materialized views, and the SQL Access Advisor will advise which ones to create and even provide a script to execute its recommendations. Once established, materialized views can be virtually self maintaining and end-users see a dramatic improvement in query response times without ever needing to change a line of SQL.

Oracle Materialized Views & Query Rewrite
May 2005
Author: Dr. Lilian Hobbs

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
oracle.com