Improving Performance using Query Rewrite in Oracle Database 10g

An Oracle White Paper
December 2003

Improving Performance using Query Rewrite in Oracle Database 10g

Improving Performance using Query Rewrite in Oracle Database 10g

EXECUTIVE OVERVIEW

Give an end-user or DBA three wishes for the database, and its highly likely that one of them will be to improve the performance. But what if you have already done all the standard tuning, is there anything else that you could try? Well significant performance gains can often be achieved using query rewrite and materialized views, and this white paper will show how these performance gains can be achieved without the need to change the application.

INTRODUCTION

Query Rewrite and Materialized Views were first introduced in Oracle 8i. In each subsequent release they have been enhanced with additional functionality and the lifting of certain restrictions.

A materialized view is a pre-computed set of results, which usually includes aggregation and joins. There can be any number of materialized views that may be defined and they can be maintained automatically or refreshed on user-demand.

The query rewrite mechanism is completely transparent to the application. Queries are constructed as per normal, written to access the tables in the database. Then, the optimizer will automatically determines whether it is cheaper to use the rewritten query or to access the tables as originally specified in the SQL statement and pick the best solution. If the rewritten query is deemed to be the best solution, then the query is automatically rewritten without any user intervention and in Oracle Database 10g, query rewrite is enabled by default.

WHY USE QUERY REWRITE

Query Rewrite is a very effective method of improving query response times in a database, especially when it is used against data that is not constantly changing, because it doesn't require any application changes. Certain results can then be pre-computed or joined and then used to satisfy any number of queries. In practice what this means is that a query which may have taken say 2hrs to run, can now return its results in say under 10 minutes, and an ad-hoc query that may have taken say 30 minutes, will now run in under 2 minutes and a query which use to take one or two minutes now returns its results virtually instantly. However, it should be stated that the performance improvement that you may see will be very dependent on your application and data.

There are many different types of query rewrite that are possible, which will be discussed later in this paper. Fortunately the user does not need to be concerned with this level of detail, however, a basic understanding of what is possible will help the DBA design materialized views that can satisfy as many queries as possible. If you are not sure which materialized views to create then the SQLAccess Advisor can recommend an initial set for you. Further information can be found in the White Paper, Performance Tuning using the SQLAccess Advisor.

OPTIMIZING THE MATERIALIZED VIEW TO ACHIEVE MAXIMUM QUERY REWRITE

Some people prefer to define their own materialized views, rather than use a tool like the SQLAccess Advisor to create them. If you prefer this approach, then there are three procedures available that you may also find useful:

- EXPLAIN_MVIEW, which types of query rewrite are possible
- EXPLAIN_REWRITE which can confirm that query rewrite will occur using the materialized view just created
- TUNE_MVIEW whether the materialized view could be optimized further

Using EXPLAIN_MVIEW to see the types of query rewrite

The DBMS_MVIEW.EXPLAIN_MVIEW procedure has been available since Oracle 9i and it will tell you whether fast refresh is possible on a materialized view and what types of query rewrite may occur. When a materialized view is first defined, this procedure should be run as part of your verification process so that you can check that the materialized view will behave as expected.

In the example shown below, the query that will be used to define the contents of the materialzed view is passed into the EXPLAIN_MVIEW procedure

```
execute DBMS_MVIEW.EXPLAIN_MVIEW ('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');
```

The results from EXPLAIN_MVIEW are placed into the table, MV_CAPABILITIES_TABLE and by querying this table we can find out what is possible with this materialized view. For this example, we can see that all the types of query rewrite are possible, but fast refresh is not because the materialized view logs are not present.

Figure 1 Output from EXPLAIN_MVIEW

```
CAPABILITY         POSSIBLE          MESSAGE TEXT
--------------------------------------------------------
PCT                Y
REFRESH_COMPLETE   Y
REFRESH_FAST       Y
REWRITE            Y
PCT_TABLE          Y COSTS
PCT_TABLE          N PRODUCTS   relation is not a
partitioned table

REFRESH_FAST_AFTER_INSERT      N SH.COSTS     the detail
table does not have a materialized view log
REFRESH_FAST_AFTER_INSERT      N SH.PRODUCTS the detail
table does not have a materialized view log
REFRESH_FAST_AFTER_ONETAB_DML  N              see the
reason why REFRESH_FAST_AFTER_INSERT is disabled
REFRESH_FAST_AFTER_ANY_DML     N              see the
reason why REFRESH_FAST_AFTER_ONETAB_DML is disabled
REFRESH_FAST_PCT               Y

REWRITE_FULL_TEXT_MATCH        Y
REWRITE_PARTIAL_TEXT_MATCH     Y
REWRITE_GENERAL                Y
REWRITE_PCT                    Y
PCT_TABLE_REWRITE              Y COSTS
PCT_TABLE_REWRITE              N PRODUCTS      relation
is not a partitioned table
```

If when you review the output from EXPLAIN_MVIEW you are unsure of how to fix the problems described, then you should run the procedure TUNE_MVIEW, which will provide you with a script to resolve the problems reported by EXPLAIN_MVIEW.

Using TUNE_MVIEW to optimize the materialized view

Although it is easy to create a materialized view, some people may be unsure as to whether they have designed the most optimal materialized view. They could experiment with different materialized views and measure for themselves the impact, or they could run a new procedure in Oracle Database 10g, DBMS_ADVISOR.TUNE_MVIEW.

TUNE_MVIEW will analyze your materialized view statement and see if it can be improved to obtain fast refresh and as many types of query rewrite as possible. This procedure takes as its input the CREATE MATERIALIZED VIEW statement and returns a SQL script containing its recommendations.

In the example shown below, we first provide TUNE_MVIEW with the complete CREATE MATERIALIZED VIEW statement

```
variable mv1 VARCHAR2 (30);
execute :mv1 := 'TM1';
execute DBMS_ADVISOR.TUNE_MVIEW ( :mv1,'CREATE
MATERIALIZED VIEW costs_mv ENABLE QUERY REWRITE AS
SELECT time_id,prod_name,SUM(unit_cost)
,COUNT(unit_cost) ,COUNT(*) FROM costs c,products p
WHERE c.prod_id = p.prod_id GROUP BY time_id,
prod_name');
```

Prior to using this procedure the location of where the scripts is to be stored is defined using the CREATE DIRECTORY statement.

```
CREATE DIRECTORY loc '/tunemview/results';
GRANT READ ON DIRECTORY loc TO PUBLIC;
GRANT WRITE ON DIRECTORY loc TO PUBLIC ;
```

Next the script, which we have called tmv.sql, containing the recommendations from TUNE_MVIEW can be generated.

```
execute DBMS_ADVISOR.CREATE_FILE(
DBMS_ADVISOR.GET_TASK_SCRIPT ( :MV1 ) ,'LOC', tmv.sql');
```

When we review the recommendation script we will see that it also includes recommendations for materialized view log statements, which are needed to make this materialized view fast refreshable. .

Figure 2 Extract of Output from TUNE_MVIEW

```
……….
CREATE MATERIALIZED VIEW LOG ON  "SH"."PRODUCTS"
    WITH ROWID, SEQUENCE("PROD_ID","PROD_NAME")
    INCLUDING NEW VALUES;
………………..
CREATE MATERIALIZED VIEW SH.COSTS_MV
    REFRESH FAST WITH ROWID
    ENABLE QUERY REWRITE
    AS
SELECT SH.COSTS.TIME_ID C1, SH.PRODUCTS.PROD_NAME C2,
SUM("SH"."COSTS"."UNIT_COST") M1,
COUNT("SH"."COSTS"."UNIT_COST") M2,
COUNT(*) M3 FROM SH.COSTS, SH.PRODUCTS
 WHERE SH.PRODUCTS.PROD_ID = SH.COSTS.PROD_ID
  GROUP BY SH.COSTS.TIME_ID, SH.PRODUCTS.PROD_NAME;
```

Also don't be surprised if you give TUNE_MVIEW a single materialized view and it returns multiple materialized views, because in some cases, this is the only way that fast refresh may be possible.

Did the Query Rewrite

Because query rewrite is transparent, it is not always easy to determine if query rewrite occurred. If the query use to run for a very long time and now it is quick, then query rewrite probably happened, but how can you be certain. Well there are two techniques you can use to verify if rewrite will occur:

• explain_plan

• explain_rewrite

Explain Plan and Query Rewrite

In Oracle Database 10g, EXPLAIN PLAN now shows if a materialized view is used and whether it was accessed directly or as a result of a query rewrite. For example, if we create the materialized view that was suggested by TUNE_MVIEW and now run EXPLAIN PLAN using that query

```
Explain Plan for
SELECT SH.COSTS.TIME_ID C1, SH.PRODUCTS.PROD_NAME C2,
SUM("SH"."COSTS"."UNIT_COST")    M1,
COUNT("SH"."COSTS"."UNIT_COST") M2,
COUNT(*) M3 FROM SH.COSTS, SH.PRODUCTS
WHERE SH.PRODUCTS.PROD_ID = SH.COSTS.PROD_ID
GROUP BY SH.COSTS.TIME_ID, SH.PRODUCTS.PROD_NAME;
```

We can see that EXPLAIN_PLAN advises us that MAT_VIEW REWRITE will occur so we know that query rewrite will happen. If you see MAT_VIEW without rewrite, then that means that the materialized view was queried directly.

Figure 3   Output from EXPLAIN PLAN

```
Query Plan
--------------------------------------------------------
Operation                        | Table Name
--------------------------------------------------------
SELECT STATEMENT                 |
 MAT_VIEW REWRITE ACCESS FULL    | COSTS_MV
--------------------------------------------------------
```

EXPLAIN_REWRITE advises why Query Rewrite didn't occur

Query Rewrite may not always occur and trying to determine why may prove a little daunting if this is the first time that you have used this feature. Therefore, to help you quickly determine why a query didn't rewrite, use the EXPLAIN_REWRITE procedure, which takes as your input the query, then using all the materialized views, it advises whether query rewrite will occur.

Since query statements can be quite long, its easier to pass the statement into EXPLAIN_REWRITE as a variable as in the example shown here.

```
DECLARE
 qrytxt VARCHAR2(1000) := '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';

BEGIN
 dbms_mview.Explain_Rewrite(qrytxt,'SH.COSTS_MV','ID1');
END;
```

In this first example, EXPLAIN_REWRITE advises that query rewrite will occur and it will use the materialized view COSTS_MV

Figure 4 Output from EXPLAIN REWRITE when rewrite is possible

```
MESSAGE
-------------------------------------------------------
QSM-01033:
query rewritten with materialized view, COSTS_MV
```

Now lets look at another example using a different query.

```
DECLARE
  qrytxt VARCHAR2(3000) := 'SELECT  prod_name,
SUM(unit_cost) , SUM(quantity_sold), COUNT(*) FROM costs
c, products p , sales s WHERE c.prod_id = p.prod_id  AND
c.prod_id = s.prod_id GROUP BY prod_name';

BEGIN
 dbms_mview.Explain_Rewrite(qrytxt, SH.COSTS_MV','ID1');
END;
```

In the next example, query rewrite is not possible so you will be advised why it cannot occur.

Figure 5 Output from EXPLAIN REWRITE when rewrite not possible

```
MESSAGE
------------------------------------------------------
QSM-01082: Joining materialized view, COSTS_MV, with
table, COSTS, not possible

QSM-01102: materialized view, COSTS_MV, requires join
back to table, COSTS, on column, PROD_ID
```

Now you can fix the materialized view yourself, or alternatively you could use the procedure QUICK_TUNE to show you how to create a materialized view to support this query.

Using QUICK_TUNE to recommend a materialized view for a query

If EXPLAIN_REWRITE advises that query rewrite is not possible, you may not be certain how to fix your materialized view to make query rewrite occur. The SQLAccess Advisor has been designed to tune a workload of SQL statements, and the Oracle Database 10g procedure QUICK_TUNE, is the single statement version of the SQLAccess Advisor, which recommends indexes and materialized views. It is very easy to use, simply supply a task name and the SQL statement and it generates recommendations as shown below.

```
variable tname varchar2(255);
variable sql_statement clob;

execute :tname := 'qtune';
execute :sql_statement := 'SELECT prod_name,
SUM(unit_cost) , SUM(quantity_sold), COUNT(*) FROM costs
c, products p , sales s WHERE c.prod_id = p.prod_id  AND
c.prod_id = s.prod_id GROUP BY prod_name';
```

```
execute DBMS_ADVISOR.QUICK_TUNE
(DBMS_ADVISOR.SQLACCESS_ADVISOR, :tname, :sql_statement,
NULL, NULL, DBMS_ADVISOR.SQLACCESS_WAREHOUSE);
```

The script is only generated when you run the following procedure.

```
execute DBMS_ADVISOR.CREATE_FILE(
DBMS_ADVISOR.GET_TASK_SCRIPT('qtune'),'LOC','qmv.sql') ;
```

The script generated by QUICK_TUNE will contain everything you need to obtain query rewrite and fast refresh and could comprise of recommendations for both indexes, materialized views and materialized view logs. In this example, the script only contained a materialized view recommendation which is shown below.

Figure 6   Recommendations from QUICK_TUNE

```
CREATE MATERIALIZED VIEW "SH"."LMV__0001"
    REFRESH FAST WITH ROWID
    ENABLE QUERY REWRITE
    AS SELECT SH.PRODUCTS.PROD_NAME C1,
SUM("SH"."COSTS"."UNIT_COST")  M1 ,
COUNT("SH"."COSTS"."UNIT_COST") M2,
SUM("SH"."SALES"."QUANTITY_SOLD") M3,
COUNT("SH"."SALES"."QUANTITY_SOLD") M4,
COUNT(*) M5 FROM SH.COSTS, SH.SALES, SH.PRODUCTS
WHERE SH.SALES.PROD_ID = SH.COSTS.PROD_ID AND
SH.PRODUCTS.PROD_ID = SH.COSTS.PROD_ID AND
SH.PRODUCTS.PROD_ID = SH.SALES.PROD_ID
GROUP BY SH.PRODUCTS.PROD_NAME;
```

As you can see from this example, QUICK_TUNE provides a quick and easy solution for creating materialized views and index recommendations, thus eliminating the need for a detailed understanding of how to define a materialized view.

How to stop a Query Executing if Query Rewrite is not possible

Once an application relies on query rewrite, if for some reason it didn't occur, then it could have a serious impact on application performance if it suddenly stopped working. This could happen for a variety of reasons, such as the latest data in not in the materialized view, and the rewrite integrity mode states that we must only use fresh data. In this instance, the query would execute, but it would revert to obtaining the information from the original tables.

In Oracle Database 10g, a hint /*+ REWRITE_OR_ERROR */ can be added to the SQL query so that if query rewrite is not possible, the query will fail, rather than gather the data from the original tables. Although using this approach does require an application change, it does avoid this performance problem. In the example shown below, the query is unable to use query rewrite so it will fail with the error ORA-30393

Figure 7 Stopping a run away query using a Hint

```
SELECT /*+ REWRITE_OR_ERROR */  prod_name,
SUM(unit_cost) , SUM(quantity_sold), COUNT(*)
 FROM costs c, products p , sales s
  WHERE c.prod_id=p.prod_id  AND c.prod_id = s.prod_id
   GROUP BY prod_name;

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

Query Rewrite Integrity Modes

When using query rewrite and materialized views, it is imperative to ensure that the correct data is returned at all times, since results are being determined by referencing the materialized view and not the original tables. To ensure that you do not see incorrect results, three integrity levels are available, which are selected by the parameter QUERY_REWRITE_INTEGRITY

- STALE_TOLERATED
- TRUSTED
- ENFORCED (default)

In STALE_TOLERATED mode, a materialized view will always be used even if it is doesn't contain the latest data. Most production systems probably wouldn't want to use this mode, but it can be useful when testing to see if query rewrite will occur. But if you do have a system where you don't have to see absolutely the latest view then it is safe to use it.

TRUSTED mode, is the one used in most production systems and here the optimizer uses the data in the materialized view provided it is fresh and that the relationships declared in dimensions and RELY constraints are correct. In this mode, the optimizer will also use prebuilt materialized views or materialized views based on views, and it will use relationships that are not enforced as well as those that are enforced. It also 'trusts' declared but not ENABLED VALIDATED primary/unique key constraints and data relationships specified using dimensions.

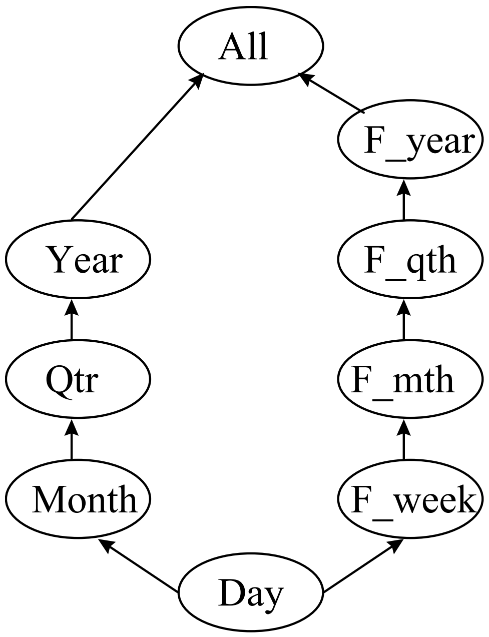
In the ENFORCED mode which is the default, the optimizer will only use materialized views that it knows contain fresh data and it will only use those relationships that are based on ENABLED VALIDATED primary/unique/ foreign key constraints. Since it only uses validated relationships, it will also not use dimension objects. Therefore you may find that query rewrite may not occur using this method if some constraints have not been validated, even though it occurs using the less restrictive TRUSTED or STALE_TOLERATED modes.

Why Dimensions help Query Rewrite

To get the most out of using materialized views, dimensions should be defined which describe the hierarchical (parent/child) relationships between columns, where all the columns do not have to come from the same table. While defining a dimension it is also possible to describe between the dimension columns and other columns in the table.

Figure 8 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.

Figure 8 Illustrates the Time Dimension



Each bubble represents a *level* in the dimension and is declared using the LEVEL clause. The dimension *hierarchy* is declared using the HIERARCHY clause 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*.

Once these dimensions have been defined, then query rewrite can use them to perform rollup and join back operations.

By having dimensions, it means that it avoids the need to define materialized views at each level in the hierarchy, thus reducing the number of materialized views that have to be defined and maintained which reduces maintenance time and disk space requirements.

The Different Types of Query Rewrite

One of the really nice features of query rewrite is that the definition of the materialized view does not have to be exactly the same as the SQL query. That is why there are a number of different types of query rewrite available, so that a wide range of queries can be satisfied by a limited number of materialized views.

These rewrite techniques are known as:

- Exact Match
- Materialized View Join Back
- Materialized View Rollup
- Materialized View Aggregation
- Filtering
- PCT Rewrite

Since all these rewrite methods are available, one financial institution in the UK, for one of their Management information systems has defined only five materialized views, but these satisfy up to 60% of the queries being processed.

To illustrate how a single materialized view can satisfy a range of queries, lets define the following materialized view.

```
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 to use the materialized view all_cust_sales_mv

Materialized View JoinBack

Some times a query may contain a reference to a column which is not stored in a 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 customer 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*. Since we know from the dimension that there is a 1:1 relationship between c.cust_last_name and c.cust_id. 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.

Materialized View 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 for every customer. A customer dimension has been created which describes the relationship between customer and region. 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;
```

Materialized View Aggregation to All

Rather than rolling up to a higher level of aggregation than is defined in the materialized view, we may want to aggregate at the level at which the materialized view is defined.

In our example, the materialized view is grouped by product_id and customer_id. So suppose we wanted to find out what each customer has purchased from us and didn’t care as to what products they had bought. Then we can use the materialized view *all_cust_sales_mv* to produce the report where it will aggregate together all the data for a customer.

```
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;
```

So hopefully now you are beginning to appreciate how our single materialized view has already satisfied 4 different queries.

Filtering

So far all of the materialized views which we have seen contain all of the data, but this could still result in a very large materialized view. Instead a materialized view can be defined so that it only contains part of the data, as shown below, where we only have data for the cities Dublin, Galway, Hamburg and Istanbul.

```
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. In addition, this query is also using a rollup to show sales at 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
AND      c.cust_state_province IN ('Dublin','Galway')
GROUP BY  c.cust_state_province;
```

PCT Rewrite

The final type of rewrite that is possible is known as PCT (Partition Change Tracking) rewrite and this allows query rewrite to occur when only some of the data in the materialized views is fresh. Oracle keeps track of which partitions in the original tables have been updated, so it can then determine whether the data in the materialized view is from a fresh or a stale partition. If it’s still fresh, then query rewrite can occur without the need to refresh the entire materialized view.

CONCLUSION

We have taken huge steps forward from the days when a report writer would have to query the summary tables directly. Today, they write their query as if accessing the original tables, and the optimizer transparently selects query rewrite. With all the different types of query rewrite methods that are available, they can be used to satisfy a wide variety of queries. Therefore, if you had given up hope of tuning your database, thinking that query performance could be improved no further. If you haven’t yet tried materialized views and query rewrite, then you could be missing a golden opportunity to significantly improve query response times in your database.

Improving Query Performance using Query Rewrite in Oracle Database 10g
December 2003
Author: Dr Lilian Hobbs
Contributing Authors:

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
www.oracle.com