Optimizing Refresh of a Set of Materialized Views

Nathan Folkert, Abhinav Gupta, Andrew Witkowski, Sankar Subramanian,
Srikanth Bellamkonda, Shrikanth Shankar, Tolga Bozkaya, Lei Sheng

Oracle Corporation
500 Oracle Parkway
Redwood Shores CA 94065
Firstname.Lastname@oracle.com

Abstract

In many data warehousing environments, it is common to have materialized views (MVs) at different levels of aggregation of one or more dimensions. The extreme case of this is relational OLAP environments, where, for performance reasons, nearly all levels of aggregation across all dimensions may be computed and stored in MVs. Furthermore, base tables and MVs are usually partitioned for ease and speed of maintenance. In these scenarios, updates to the base table are done using Bulk or Partition operations like add, exchange, truncate and drop partition. If changes to base tables can be tracked at the partition level, join dependencies, functional dependencies and query rewrite can be used to optimize refresh of an individual MV. The refresh optimizer, in the presence of partitioned tables and MVs, may recognize dependencies between base table and the MV partitions leading to the generation of very efficient refresh expressions. Additionally, in the presence of multiple MVs, the refresh subsytem can come up with an optimal refresh schedule such that MVs can be refreshed using query rewrite against previously refreshed MVs. This makes the database server more manageable and user friendly since a single function call can optimally refresh all the MVs in the system.

Terminology

PMOP: partition maintenance operation.

Conventional Refresh refers to MV refresh using changes to the base tables stored in materialized view logs. The logs record images of individual rows that have changed since the last refresh.

Partition Change Tracking (PCT) Refresh refers to MV refresh using only the changed partitions of base tables of an MV. This refresh method is possible only if the base tables are partitioned and changes to base tables are tracked on a partition basis - hence the name, Partition Change Tracking Refresh.

Enhanced Partition Change Tracking (EPCT) Refresh refers to PCT based refresh applied to MVs containing columns that are *partition-join dependent* on the partitioning column of the base table.

**Proceedings of the 31st VLDB Conference,
Trondheim, Norway, 2005**

1. Introduction

We present an approach to optimize refresh [8], [10], [13], [15] of a single materialized view by taking advantage of partitioning of base tables and materialized views. In addition, we present an algorithm to derive an optimal schedule of refreshing a set of materialized views. All methods presented in this paper are available in Oracle Database Release 10*g*.

2. Schema and Cubes

We use the following example star schema in the rest of the paper. There is a fact table *sales* and three dimension tables – *times*, *product* and *customers*. Sales of each product (*prod_id*) sold to a particular customer (*cust_id*) on a given day (*day_id*) is stored in the *sales* table. The schema looks as follows:

- *sales (day_id, cust_id, prod_id, amount)*: The sales table is partitioned by range on day_id (a date column) such that each partition contains a month of data. Partitions are named correspondingly as `SALES_JAN_2003_PART`, `SALES_FEB_2003_PART`, .., etc.

- *times (day_id, month, quarter, year)*: contains the hierarchy (*day_id -> month -> quarter -> year*). Assume that *day_id* records days.
- *Customer (cust_id, city, state):* contains the hierarchy (*cust_id -> city -> state*).
- *Product(prod_id, package, group):* contains the hierarchy (*prod_id -> package -> group*).

The hierarchy relationships present in the dimension tables (i.e., month rolls up to a quarter, city rolls up to a state, etc) can be defined using Oracle Dimension objects [3], [14]. For example, the rollup relationship present in the *times* table can be defined using the following DIMENSION statement:

```
CREATE DIMENSION time_dim
   LEVEL day   IS times.day_id
   LEVEL month IS times.month
   LEVEL quart IS times.quarter
   LEVEL year  IS times.year
   HIERARCHY calendar (
      day   CHILD OF month CHILD OF
      quart CHILD OF year)
```

In OLAP environments, analysts use cubes [2], [5], [9] over the star schema. We distinguish between two representations of the cube: Rollup Cube and Federated Cube.

The *Rollup Cube* stores rolled up aggregation values along all dimensional hierarchies in a single MV. For example:

```
CREATE MATERIALIZED VIEW rollup_cube_mv AS
SELECT year, quarter, month, day_id, state,
       city, cust_id, group, package,
       prod_id, sum (amount) amt
FROM sales s, times t,customer c, product p
WHERE s.day_id =t t.day_id AND s.cust_id =
      c.cust_id AND s.prod_id = p.prod_id
GROUP BY
   ROLLUP(year, quarter, month, day_id),
   ROLLUP(state, city, cust_id),
   ROLLUP(group, package, prod_id)
```

An alternative way of representing a Cube is storing each grouping in a separate MV. The *Federated Cube* is a collection of MVs each representing an aggregation over one level from each dimension. For example, the following are two such MVs in a Federated cube:

```
Q0:
CREATE MATERIALIZED VIEW quart_city_pack_mv
AS
SELECT quarter, city, package,
       sum(amount) amt
FROM sales s, times t,customer c, product p
WHERE s.day_id =t t.day_id AND s.cust_id =
      c.cust_id AND s.prod_id = p.prod_id
GROUP BY quarter, city, package
```

```
CREATE MATERIALIZED VIEW
       year_city_pack_mv AS
SELECT year, city, package,sum(amount) amt
FROM sales s, times t,customer c, product p
WHERE s.day_id =t t.day_id AND s.cust_id =
      c.cust_id AND s.prod_id = p.prod_id
GROUP BY year, city, package
```

A full Federated Cube in our example schema consists of 80[1] separate MVs. We concatenate the levels from each of the dimensions (<time_customer_product_mv>) to designate the individual MVs.

In many cases, users prefer a Federated Cube over a Rollup Cube: it usually takes less space since the null values of rolled up columns [9] are not stored. Bitmap indexes created over cubes for efficient querying are also more compact for a federated cube as they don't index null values and therefore are much smaller in size.

There are obvious variations of the Rollup and Federated cubes which contain only a subset of the groupings due to space constraints. A useful example of a partial Rollup Cube is one which rolls up on all dimensions except time. The non-rolled up time level is then used as a partitioning column. For example consider the following materialized view using Oracle partitioning syntax:

```
CREATE MATERIALIZED VIEW
       rollup_cube_month_ mv AS
PARTITION BY RANGE (month)
(
  PARTITION VALUES LESS THEN 'M1_2003',
  PARTITION VALUES LESS THEN 'M2_2003',
       ....
)
SELECT month, state, city, cust_id, group,
       package, prod_id,sum (amount) amt
FROM sales s, times t,customer c, product p
WHERE s.day_id =t t.day_id AND s.cust_id =
      c.cust_id AND s.prod_id = p.prod_id
GROUP BY month,
   ROLLUP(state, city, cust_id)
   ROLLUP(group, package, prod_id)
```

In this paper we focus on refresh of (partial) cubes that are partitioned by the time dimension or any other dimension serving as a granule of data warehouse maintenance. We assume that each partition contains data from a single time granule. For example, in the *rollup_cube_month_mv* cube, each partition contains a month of data. We use symbolic partition names, like

[1] There are 5 levels in time dimension (day, month, quarter, year and all years). Similarly, there are 4 levels in customer dimension and 4 levels in product dimension. 5x4x4 = 80 groupings.

ROLLUP_CUBE_MONTH_M1_2003_MV for the first month (January) partition.

Of course, the proposed refresh method works if time level partitions are coarser. For example, each partition of *rollup_up_cube_mv* can contain multiple months instead of storing a single month as described above.

3. Motivation

Consider a manufacturer supplying products packaged in different ways to be sold at its various retail outlets. All the sales data is centrally collected in a database server in a star schema described earlier for decision support and analysis. Every month, new data is added to the sales table corresponding to sales data received from the retailers for the previous month. Users are interested in the total sales of the current quarter for earnings announcement and previous quarters for comparative analysis. The sales analysts are also interested in tracking monthly sales to measure growth with respect to the previous month and also the same month of the previous year. These analyses are required at different levels of the dimensional hierarchy. Hence, there is a need to materialize a "cube" to slice and dice data along various dimensions of interest for fast responses to such queries.

Let's assume there is a Federated Cube materialized in the database. The MVs *quart_city_pack_mv* and *year_city_pack_mv* are part of the cube. Each of the materialized views comprising the cube has to be incrementally refreshed to keep it in sync with the data in the sales table. It is common practice to store base data as a rolling window of time (for instance, last 24 months) in the warehouse. Hence the MVs are also stored as a rolling window of time. In the example discussed, every month a new partition of data is added to the *sales* table and the data for the oldest month is removed and possibly archived. We assume that the lowest level of time granule is day.

After these rolling window operations, the above MVs can be refreshed using the following observations:

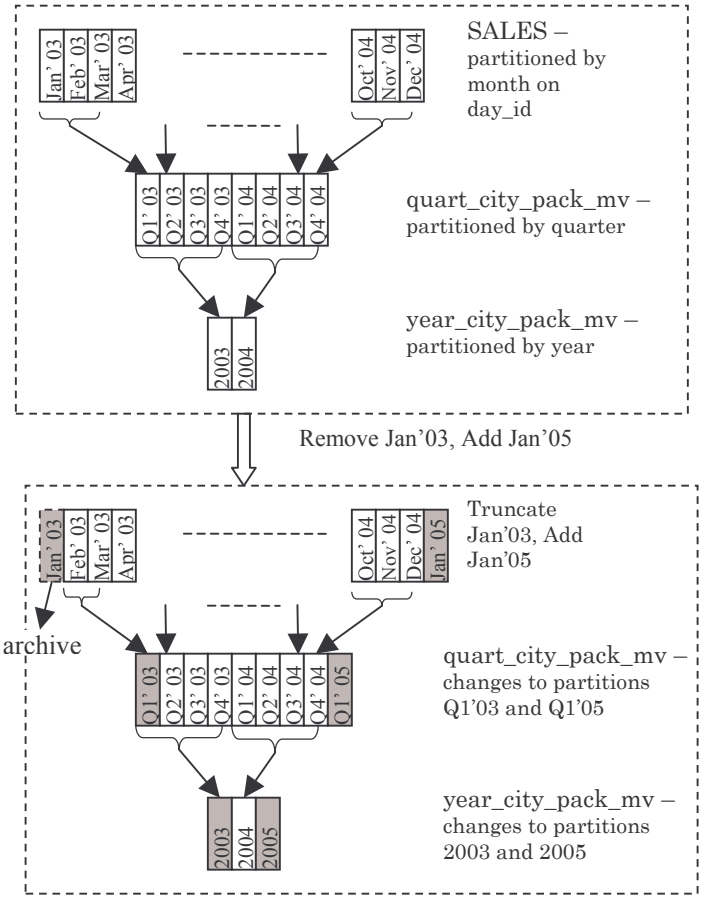
quart_city_pack_mv:

- If *quart_city_pack_mv* is partitioned on quarter, one option of refreshing is to truncate the affected partitions and recreate them. Only two quarters of data in the MV are affected. These quarters can be identified by the following simple query on *times* dimension table:

```
SELECT DISTINCT quarter
FROM times t
WHERE
<predicate on day_id for the
affected partitions of sales table>
```

Truncating a partition is an instantaneous operation since it requires simple changes to the *quart_city_pack_mv* storage meta-data. Inserting new data is also relatively fast since it can be appended in new storage segments. However, recreating data to be inserted is expensive as it requires aggregating data of all the quarters in the affected MVs.

The method of refresh described above uses *join dependency* of *quarter* column in the MV on the partitioning column *day_id* of *sales*.



- Another option is to apply the deltas from the *sales* table to the MV. This works well if number of rows to be updated is small. However, update can be expensive since it requires a join of aggregated delta from SALES table with the MV *quart_city_pack_mv*. In Oracle, this method of refresh is called *conventional incremental refresh*.

Ideally, the database server should explore all possible options of refreshing a particular MV and pick the cheapest method.

year_city_pack_mv:

- The method for refreshing *year_city_pack_mv* is similar to those for *quart_city_pack_mv*. If refresh of *quart_city_pack_mv* is done prior to refresh of *year_city_pack_mv*, then, *year_city_pack_mv* can be refreshed by *rewriting* refresh expressions against *quart_city_pack_mv* by exploiting the *functional dependency* of year on quarter (i.e. quarter determines year). Since functional dependencies are expensive to validate in real time, Oracle provides syntax to declare the functional dependencies using DIMENSION [3], [14] statements. These dependencies are not validated, but can be used for optimizing queries.

If functional dependencies are used for refresh, the data contained in the materialized view may no longer be valid. Therefore, we introduce a new state for materialized views called *trusted state*. A materialized view in trusted state can be used to rewrite a user query only if the user session has been enabled to go against trusted MVs.

In this paper, we present the concepts of join dependency and trusted MVs and propose the use of join dependency, functional dependency, MV partitions and query rewrite to optimize refresh of MVs. If a complete Federated cube is created on the schema discussed above, 80 materialized views will be required. Increasing the number of levels in dimensions and increasing number of dimensions can require the creation of hundreds of materialized views. It is a nightmare to find the correct order of refreshing such materialized views. Therefore, we present a simple API that can be used to refresh all the materialized views that depend on a set of changed tables. Internally, an MV refresh scheduler (see Section 5) computes an optimal order of refreshing the MVs and minimizes the overall refresh time by exploiting dependencies and hence enabling optimal rewrites.

4. Optimizing Refresh of a single MV

In Oracle Database Release 9*i*, Oracle provided an MV refresh mechanism called the Partition Change Tracking (PCT) refresh. It tracks partition maintenance operations (PMOPs) like ADD and DROP PARTITION as well as DML changes to the base data on a partition basis. If an MV contains the partitioning key of changed base tables and is partitioned by these keys, then based on this tracking information, we generate very efficient refresh expressions by issuing truncate partition operations on the affected MV partitions followed by a bulk INSERT into these partitions. We have observed multi-fold performance improvements for large changes to the base tables when using this technique compared to the conventional MV log-based refresh.

Unfortunately, PCT refresh has limited applications because in many scenarios, MVs are partitioned at different time granules than the base data. This section describes extensions to the PCT refresh that allows PCT refresh to be applied to MVs containing columns that are *join dependent* on the partitioning column of the base table. This improved refresh also supports query rewrite of the refresh expressions using un-enforced functional dependencies. We refer to this PCT refresh as Enhanced PCT refresh (EPCT).

We note that EPCT can be used not only when a partition operation has been performed on a base table, but also when a significant portion of a partition has been updated with a conventional DML such as Insert or Delete.

4.1. Use of join dependencies

Consider a SQL query block Q of an MV defined on a partitioned table T. **Partition-Join Dependent Expression on T** is an expression consisting of columns from tables directly or indirectly joined through equi-joins to a partitioned table T on its partitioning key. The tables containing these expressions and the tables in the join path from these tables to T are called PJoin-Dependent tables. For example, in *quart_city_part_mv*, the quarter column is PJoin-Dependent on the partitioning key, *day_id*, of *sales* table. Observe that the value of partition key determines the value of PJoin-Dependent Expression. Hence, if an MV is partitioned by a PJoin-Dependent column and if we add or delete a partition in T, then we can easily determine affected partitions in the MV by executing the constrained form of Q. This is the intuition behind Enhanced PCT refresh.

EPCT refresh is enabled if the MV has, in its select list, expressions which are PJoin-Dependent on the partitioning column(s) of a base table that has changed. Below, we consider two methods of refreshing an MV exploiting PJoin-Dependency. Assume a partitioned base table T has changed.

EPCT refresh with DELETE and PJoin-Dependency

There are 2 phases in Enhanced PCT refresh:

Delete phase removes all the affected rows from the MV.
Insert phase inserts the recomputed rows into the MV.

Both phases make use of a sub-query predicate on the base tables to remove and compute the affected rows in the MV. The predicate takes the following form:

```
Q1:
  <pj_depend_exp_list> IN
       (SELECT <pj_depend_exp_list>
        FROM <tab_list>
        WHERE <join_pred> AND <part_pred>)
```

Here,

- <pj_depend_exp_list> is the list of expressions in the SELECT list of the MV which are PJoin-Dependent on the partitioning columns of the changed table T
- <tab_list> is the list of PJoin-Dependent tables on table T
- <join_pred> is the predicate which joins the tables in <tab_list>
- <part_pred> is the predicate identifying the changed partitions in T

For example, consider *quart_city_pack_mv* from our example in Section 2 (see Q0). Assume that partition SALES_JAN_2003_PART corresponding to January-2003 of the *sales* undergoes a series of DML operations. EPCT refresh of the MV has to re-compute all the data in the affected MV quarter. The quarter column of the MV is PJoin-Dependent on the partitioning key of the *sales*, hence the DELETE phase refresh statement is:

```
Q2:
DELETE quart_city_pack_mv
WHERE quarter IN
   (SELECT quarter
    FROM times t
    WHERE (t.day_id >= '01-01-2003' AND
           t.day_id < '02-01-2003')
```

The corresponding INSERT phase refresh statement is:

```
Q3:
INSERT INTO quart_city_pack_mv
SELECT quarter,city,package,sum(amount) amt
FROM sales s, times t,customer c, product p
WHERE s.day_id=t t.day_id AND s.cust_id =
    c.cust_id AND s.prod_id = p.prod_id AND
    t.quarter  IN
       (SELECT quarter
        FROM times t
        WHERE t.day_id >= '01-01-2003' AND
              t.day_id < '02-01-2003')
GROUP BY quarter, city, package
```

Observe that the IN sub-query references only *times* table and not the changed *sales* table. The *times* table is connected in the MV definition by equi-joins to the partitioning columns of *sales*, so it is sufficient to transfer the changed range to the times table only. This removes an expensive join from the sub-query. This join pruning technique is used when the sub-query predicate of Q1 is employed for refresh. Hence <tab_list> from Q1 will not contain the changed table T as we transfer constant partition predicates on T to its closest equi-joined table and prune T from Q1.

We note that the above refresh expressions are general and can be used independent of whether the MV is partitioned or not.

EPCT refresh with TRUNCATE and Join Dependency

If the MV itself is partitioned[2] and its partitioning columns are PJoin-Dependent on the partitioning columns of a changed base table and no other table has changed, the refresh system can issue TRUNCATE instead of DELETE. Note that we assume that only one table T has changed which is usually the case for a data warehouse with a rolling window on a fact table. PJoin-Dependent refresh expressions when multiple base tables have changed exist, but are much more complex and hence not shown.

EPCT refresh using TRUNCATE also consists of two phases, but the first phase uses TRUNCATE (instead of DELETE) to remove the changed rows from the materialized view. To get a list of MV partitions to be truncated, we use an Oracle internal function, PARTNAME, which for a given partitioned table maps its partitioning key to the partition name. The following query gets the names of the affected MV partitions:

```
Q4:
SELECT DISTINCT
 PARTNAME(<mv_name>, <partition_exp_list>)
FROM (SELECT DISTINCT <partition_exp_list>
       FROM <tab_list>
       WHERE <join_pred> AND <part_pred>)
```

Here, <mv_name> is the MV being refreshed, <partition_exp_list> is the list of <pj_depend_exp_list> expressions in Q1 that are also the partitioning columns of the MV. <tab_list>, <join_pred> and <part_pred> are the same as in query Q1.

Once the partitions to be truncated are identified, we generate the corresponding TRUNCATE statements and then construct the INSERT-SELECT statement which re-computes the affected MV partitions. The INSERT query goes against the data in base table partitions that affect these MV partitions. Note that the INSERT-SELECT statement will populate *freshly* truncated partitions. In this case, Oracle uses an optimized bulk insertion that loads data into new extents of a partition that is more efficient than conventional row-by-row insert since no undo logs are created.

[2] Range or List partitioning is supported.

If there is more than one MV partition to populate, then instead of issuing an INSERT query for each partition that may also lead to multiple scans of the same base table, we use a (ANSI SQL) MULTI-TABLE INSERT-SELECT [14] statement whose targets are affected MV partitions. Each branch of this statement inserts into an individual MV partition obtained from Q2.

For example, assume that *quart_city_pack_mv* is partitioned by quarter, and that partition SALES_JAN_2003_PART of the *sales* table has undergone a series of DML operations. We determine the affected MV partition using:

```
Q5:
SELECT DISTINCT
 PARTNAME(quart_city_pack_mv, quarter)
FROM (SELECT DISTINCT quarter
      FROM times
      WHERE (t.day_id >= '01-01-2003' AND
             t.day_id < '02-01-2003')
```

Assume that the PARTNAME function returns partition name QUART_CITY_PACK_Q1_PART. Then, the TRUNCATE statement is:

```
Q6:
ALTER TABLE quart_city_pack_mv
TRUNCATE PARTITION quart_city_pack_q1_part
```

To construct the INSERT statement, we first retrieve partition boundaries ('Q1_2003' and 'Q2_2003') of this partition from dictionary based on the partition name. The INSERT statement is then:

```
Q7:
INSERT INTO quart_city_pack_mv
    PARTITION quart_city_pack_q1_part
SELECT
 quarter, city, package, sum(amount) amt
FROM sales s, times t,customer c, product p
WHERE s.day_id =t t.day_id AND s.cust_id =
      c.cust_id AND s.prod_id = p.prod_id
      AND t.quarter >= 'Q1_2003' AND
          t.quarter < 'Q2_2003'
GROUP BY quarter, city, package
```

Note that in this case only a single partition of MV is affected, so we used INSERT-SELECT rather than MULTI-TABLE insert. Also, note that we used the boundary of the partition in the refresh query.

MV partitions may be larger than the affected data in the MV. For example, quarter partitions in *quart_city_pack_mv* may contain two or more calendar quarters. In such a scenario, there are 2 options:

- Use TRUNCATE that will end up removing a lot more data but is very fast. The corresponding INSERT will also end up re-computing more data.
- Use DELETE to remove only the data corresponding to the affected partitions. This DELETE will be slower than TRUNCATE but the corresponding INSERT will compute only the affected MV rows.

We decide between the two options based on the optimizer provided cost estimates of the refresh expressions – see Section 4.4.

Restrictions on EPCT

There are some restrictions on applicability of EPCT based on the structure of query block Q defining the MV.

- The <pj_depend_exp_list> expressions from Q1 should not be rolled up in the GROUP BY of Q as the dependency is lost due to NULLs generated by rollup.
- If the Q has window functions [20] in the select list or a SQL MODEL clause [18], their PARTITION BY columns should include a common subset of the PJoin-Dependent expressions. This guarantees that partitions of window functions or SQL MODEL clause are PJoin-Dependent on the partitioning columns of the base table. This gives us an opportunity to refresh MVs containing window functions or SQL Model clause incrementally – an option not available before.

4.2. Use of functional dependencies and query rewrite

Oracle provides query rewrite [3] with MVs which in most cases is a superset of rewrites presented in [6], [7], [12], [16], [15], [19]. The rewrite is controlled by two session parameters: *query_rewrite_integrity*, controls integrity of rewrite and supports ENFORCED (rewrite with enforced constraints only) and TRUSTED (rewrite with declared but not enforced constraints) rewrites. *Query_rewrite_enabled* controls if rewrite occurs and is FALSE (no rewrite), TRUE (cost based query rewrite), or FORCE (forced rewrite). We have made query rewrite available for refresh expressions.

Consider two MVs, *month_city_pack_mv* and *quart_city_pack_mv* that belongs to the federated cube discussed in Section 2. If *month_city_pack_mv* has already been refreshed, the INSERT statement Q7 for refreshing of *quart_city_pack_mv* can be rewritten as:

```
Q8:
INSERT INTO quart_city_pack_mv
    PARTITION quart_city_pack_q1_part
SELECT
 quarter, city, package, sum(amount) amt
FROM month_city_pack_mv mv,
     (SELECT DISTINCT month, quarter
      FROM times) t
```

```
WHERE mv.month = t.month AND
      t.quarter >= 'Q1_2003' AND
      t.quarter < 'Q2_2003'
GROUP BY quarter, city, package
```

Observe that *month_city_pack_mv* MV doesn't contain required *quarter* column, hence, the rewrite engine uses a join back of the MV the dimension table *times* to get the quarter values. The join-back uses the *functional dependency* that months rollup to quarters. The resulting data is then aggregated to get sales data at quarter level.

If the query_rewrite is enabled for the refresh session, query rewrite engine will automatically rewrite the MV query on base tables to go against the already refreshed materialized views. We note that while rewriting, Oracle may have many candidate MVs to rewrite against. For example, *quart_city_pack_mv* can be rewritten against *day_city_pack_mv* as well as the much smaller *month_city_pack_mv*. Our rewrite engine uses simple heuristics to choose the best candidate MV with least cost (sum of sizes of the MV and all the tables on the join back list) for rewrite.

Trusted MVs

When an MV is refreshed, we can use other already fresh MVs for rewriting the refresh queries, as mentioned above. Query Rewrite [3] can use functional dependencies defined in Oracle Dimensions and others like foreign key relationships, join equivalence, etc. Some of the dependency contraints may not be enforced. For example, Oracle Dimensions only declare, rather than enforce, the rollup dependencies between various data elements. Similarly, the primary key foreign key (PK-FK) relationships can also be declared as RELY only constraints, which are not enforced by the database. Refresh using declarative constraints may lead to incorrect results if the declaration made is actually violated in the data. However, users employ such constraints frequently when constraints are guaranteed by the applications and these declarative constraints do not impose expensive RDBMS validation.

Oracle MV refresh system provides a new option to the user to use un-enforced constraints during refresh. The refresh property of an MV can be either set at the creation time of the MV or altered later. The allowed values of the property are TRUSTED and ENFORCED. A *trusted* MV can use un-enforced relationships like functional dependencies defined in Oracle Dimensions or PK-FK constraints in the RELY only mode for refresh. An *enforced* MV can only be refreshed using validated relationships known to return correct data. The declaration of this property by the user controls the use of TRUSTED constraints when query rewrite is enabled.

Once a trusted MV has gone through refresh using un-enforced functional dependencies, it may, in abnormal situations, contain incorrect data. Therefore, we make it available for *query_rewrite_integrity* in TRUSTED mode only.

4.3. Dynamic Partition Pruning

Oracle supports a performance enhancement technique called partition pruning. In a query, if there is a predicate on the partitioning column of a table, Oracle will try to limit access to the table only to the selected partitions. For example, in

```
SELECT quarter FROM times
WHERE (day_id >= '01-01-2003' AND
       day_id < '02-01-2003')
```

Oracle optimizer will create a plan which accesses only single partition, SALES_JAN_2003_PART, of the *sales* table. In the above case the optimizer can derive the relevant partition at compile time by analyzing the query predicate and partition boundaries of SALES table. We refer to this optimization as *static partition pruning*.

The EPCT refresh queries described in prior sections use sub-query predicates (see Q1) for partition pruning. For example, consider the SELECT query from Q3:

```
Q9:
SELECT
 quarter, city, package, sum (amount) amt
FROM sales s, times t,customer c, product p
WHERE s.day_id=t t.day_id AND s.cust_id =
      c.cust_id AND s.prod_id=p.prod_id AND
      t.quarter  IN
       (SELECT quarter
        FROM times t
        WHERE  t.day_id >= '01-01-2003' AND
               t.day_id < '02-01-2003')
```

The *sales* table is partitioned by *day_id* and is joined to the *times* table on its partitioning key via *t.day_id*. There is a sub-query predicate on *times*. This sub-query predicate can be evaluated and we could get the corresponding values of *t.day_d* column, and hence the *s.day_id,* via join transitivity. These values can then be used to prune partitions of the sales table. We refer to this technique as *Dynamic Partition Pruning* as the pruning values have to be retrieved by executing a query at run-time. In this particular case, the following query will be evaluated to get the values of *day_id* to do partition pruning on *sales* table:

```
Q10:
SELECT day_id FROM times
WHERE quarter IN
       (SELECT quarter
        FROM times t
        WHERE t.day_id >= '01-01-2003'  AND
```

```
t.day_id < '02-01-2003')
```

Since this pruning methodology requires execution of an extra query, we compare the cost of evaluating this query with the cost of scanning the complete *sales* table before deciding to do dynamic partition pruning.

In the refresh expressions that use Q1, we perform dynamic partition pruning on any partitioned table T joined directly or through other tables to the sub-query predicate <part_pred>.

Dynamic and static partition pruning are performed after query rewrite, hence rewritten queries also benefit from these techniques. For example, consider rewritten query Q8. If the MV, *month_city_pack_mv*, is partitioned by *month*, we use the predicate on the in-line view *t* to trigger dynamic partition pruning on the MV. The query is similar to Q10 except that we select months corresponding to quarters `Q1_2003 and Q2_2003` from the *times* table.

4.4. Chosing the optimal refresh method

When a materialized view is incrementally refreshed, the refresh system decides between conventional refresh and PCT refresh.

If there are PMOPs on the base tables, conventional incremental refresh can not be used as the PMOPs are not recorded in MV logs and the only available refresh method is (E)PCT. If there are no PMOPs on the base table, then the refresh system has to decide between conventional and PCT refresh and the decision is based on the cost estimates of the respective refresh methods.

If the MV is itself partitioned and its partitions are not fully contained in base table partitions, the refresh system can also choose between issuing TRUNCATE followed by an (bulk) INSERT and issuing DELETE followed by an INSERT. As described in 4.1, TRUNCATE is faster than DELETE, but it might end up removing more data than DELETE and hence, the refresh process will have to compute and load more data as a part of INSERT. The Oracle refresh system interacts with the optimizer to choose the best option. Oracle refresh system generates refresh expressions for conventional refresh, PCT refresh using TRUNCATE and PCT refresh using DELETE. It then uses the costs of these refresh expressions estimated by the optimizer to choose the best refresh option.

5. Scheduling Refresh of MVs

While refreshing a set of materialized views, there are opportunities for improving the performance of refresh of certain materialized views by rewriting their refresh expressions against other materialized views in the refresh set. Using the query optimizer and considering all applicable methods of refresh for each materialized view, we can construct an optimal schedule for refreshing the entire set.

We start the construction of the schedule by creating the Best Refresh graph, a dependency graph where each node represents a materialized view in the refresh set, and each edge represents containment of another materialized view in the refresh set in the best rewrite of that materialized view.

5.1. Creating best rewrite graph:

The first step in the construction of the graph is finding the best refresh expression for each materialized view.

For each materialized view, we determine all the refresh methods applicable for that materialized view. The different methods include incremental conventional refresh, PCT refresh using DELETE, PCT refresh using TRUNCATE and complete refresh. Complete Refresh is always possible for an MV, although other refresh methods are not always applicable.

Each refresh method will have a set of associated SQL DML and DDL expressions implementing the refresh. For example, a complete refresh might use a DDL TRUNCATE statement followed by the INSERT statement that defines the materialized view. A PCT refresh might use a series of TRUNCATE statements or a DELETE statement to empty partitions or subsets of a materialized view, followed by an INSERT statement repopulating those partitions with fresh data from the defining query.

For each of these expressions, we will use the query optimizer to estimate the total cost for executing that part of the refresh. The query rewrite phase of the optimizer will attempt to find other materialized views that could be used to improve performance. The cost of executing all expressions will be summed together to find the total cost of that refresh method, which can then be compared against other refresh methods available for that materialized view.

For the purposes of building the Best Refresh graph, the optimizer performs a few specialized operations during the estimation of this cost.

For query rewrite, we assume that all the materialized views contained in the refresh set (excluding the current materialized view) are in a "fresh" state and hence, available for rewrite for the current refresh expression, provided that other query containment requirements are satisfied. For each expression that we cost, we collect the materialized views in the refresh set that are rewritten against the optimized query.

For the case of deferred build materialized views (MVs that are not yet populated and thus lack statistics), we use the cardinality of the defining query against the base data to estimate the cost of rewriting against that materialized view. In some cases, if there has been a significant change in the base data since the last refresh of a materialized view, we might consider defining query cardinality as well, since the existing statistics for the materialized view before refresh may be significantly different than the actual statistics for the materialized view after refresh.

Once we have determined the best refresh method available for the materialized view by comparing the total costs of the expressions implementing each of its available refresh methods, we use the list of materialized views rewritten against the expressions of the best refresh method to create rewrite dependency edges in the Best Refresh graph.

5.2. Finding an acyclic graph:

Data Warehouses that can expect to benefit most from such a refresh scheduling service are those that store rollups over different dimensions and at different levels of the dimensional hierarchy in several distinct materialized view containers, with the larger MVs closer to the base being partitioned on the same dimension (though possibly at a different level in the dimensional hierarchy) as the base tables, thus allowing efficient PCT refresh with rewrite against other materialized views. In such cases, we would expect the Best Refresh graphs to be acyclic and tree-shaped, with each node having at most one incoming edge.

But while we expect that in most practical cases the Best Refresh graph will be acyclic, there may be cases where cyclical dependencies occur. One case where this might arise would be where two materialized views are defined on the same base query. The best refresh method for each will rewrite against the other, creating a cycle in the graph. In order to schedule refresh, however, we want to have an acyclic graph, so we will need a means to eliminate cyclical paths from the Best Refresh graph.

We start by analyzing the Best Refresh graph using Tarjan's algorithm [17] for finding strongly connected components. A strongly connected component (SCC) is a set of nodes where each node is connected to all the nodes in the set (including itself) by some path over the edges.

We then traverse the graph, and, when encountering SCCs, break the cycles by removing, arbitrarily or heuristically, edges from within the SCC until the nodes contained in the SCC are no longer strongly connected.

In practice, it rarely makes a difference in overall cost of the refresh set which edge is removed, provided that after a node is liberated from the SCC, its rewrite dependencies are recomputed, but this time only considering MVs that do not have incoming paths from that node.

If no SCCs are present (i.e., the graph is already acyclic), the schedule will be the same as the original graph, and will represent the optimal topological ordering for refresh, with respect to how the optimizer itself estimates the cost of queries and chooses MVs to rewrite against.

If SCCs are present, the schedule produced may not be optimal, because we do not exhaustively determine the best node to use to break the cycle. As noted above, however, this has not had a significant effect in practice.

5.3. Executing the refresh:

The easiest implementation of refresh schedule execution is to simply topologically sort the acyclic graph in some deterministic way and then successively refresh each MV in the ordered list. This will ensure that while refreshing an MV, all the MVs that are used for rewriting its best refresh expressions will be available for rewrite.

Since nodes on separate branches in the schedule are independent, we have an opportunity to use concurrency during execution.

The simplest concurrency scheme would proceed as an ordinary traversal of the original graph. We would find source nodes that have no incoming edges from MVs that have not been refreshed. The MVs in the source set can be refreshed concurrently by spawning a separate process or thread. When an MV refresh is finished, we determine if any of its dependent nodes are now source nodes and then submit them for refresh. This would proceed until the entire schedule graph had been traversed, at which point all MVs would have been refreshed.

The problem with this concurrency scheme is that it does not consider the resources required to execute each of these refreshes, including both memory resources and processing resources. The number of processes available for concurrent refresh may be limited in the system. Each refresh might be executed in parallel, and there may be a limited number of processes available for parallel operations. Large refreshes may consume significant memory resources for sorts or other intermediate computations, and thus may be most efficient to execute by themselves but impossible to execute simultaneously given the available resources in the system.

There are many possible load balancing algorithms to balance resources between concurrently running jobs. The

estimated cost of the refresh, found during the construction of the Best Refresh graph, can be leveraged to aid in producing an intelligent plan for resource allocation.

The approach we use traverses the graph as above, examining each node. If a node's cost exceeds a threshold representing the cost at which a single refresh typically requires the majority of system resources to run efficiently, determined by experiment, we run that MV alone but with full parallelism. Since the federated cube schemas most benefited by this approach have very large root MVs with the branch and leaf MVs successively smaller, any nodes that fall above the threshold are the first nodes encountered, and thus all such MVs are refreshed one at a time at the beginning of the refresh set.

Nodes whose cost falls below the threshold are executed concurrently. Because at low levels in the federated cube tree we may have relatively few branches relative to available processors, when small numbers of cubes are available, we would like to allocate each a number of processors to run in parallel and concurrently. Towards the leaves, however, there are typically more source MVs available for refresh than processors. These MVs are also typically very small, especially since they are rewriting against rolled up data rather than base data. So we want to run as many of these concurrently as possible, typically in serial since parallelism does not offer performance gains against such small refreshes.

Our algorithm to perform this balancing of resources between refresh nodes is as follows:

```
get source nodes
if any source nodecost exceeds threshold
  refresh node MV with full parallelism
  return
for each node 1..n, ordered by cost
  if nodecost/runningsum*processes < 1
    break
  runningsum += nodecost
  k++
for each node k..1
  p = floor(nodecost/runningsum*processes)
  processes -= p
  runningsum -= nodecost
  refresh node MV with parallelism = p
```

Available nodes whose cost falls below the threshold are ordered by cost, and then, from largest to smallest, are added to an execution set until the expected number of parallel processes available for its execution (i.e., the cost of the refresh divided by the total sum of costs multiplied by the number of processes available) falls below one.

At that point, processes are allocated to the nodes in the execution set by allocating the share of processes from the smallest to the largest. The processes are allocated to the cheapest nodes first because fractional shares of processes will be aggregated and allocated to the more expensive refreshes.

Because costing estimates are imperfect, we do not want to wait for an entire set of refreshes to finish before starting a new one, as otherwise an unexpectedly expensive refresh could tie up idle processes by preventing their reallocation. Instead, we run this function whenever a refresh job finishes and resources become available again. The drawback is that subsequent calls to this function will have few available processes, since only the processes from a single job are freed.

Typically this does not matter. Close to the root, there are usually few available source nodes with fairly expensive MVs rewriting against the large root MV. So when any of these jobs finishes, a large number of processes becomes available, and the new source MVs, which are smaller since they are more rolled up, may still have resources available to run in parallel. Further out in the federated cube tree, however, where we have large numbers of leaf nodes with very small MVs, we typically only have one or two processes to balance between when running this function, so we end up running them concurrently but without parallelism.

6. Unified Refresh API

DBA intervention required with our refresh API is minimal. Because the schedule is generated automatically before each refresh, the DBA does not need to construct an ad hoc schedule for refresh using rewrite and maintain it as the schema evolves. Because the construction of the graph uses the query rewrite capability of the RDBMS itself, the DBA does not need to determine what rewrite the refresh will use in order to estimate the order in which to perform the refresh. Because all refresh methods are estimated and compared using the query optimizer, the DBA does not need to do any calculations or guesswork in deciding which refresh method to use for each MV in the refresh set. All of this is done automatically.

The DBA needs to submit only the desired list of MVs to be refreshed, and the system automatically finds the best schedule using the best refresh methods based on the DBMS's own query rewrite and query optimizer engines.

7. Performance Comparison

We conducted experiments on the APB benchmark [1] populated with 5.0 density data. The APB schema has a fact table with 4 hierarchical dimensions: channel with 2 levels, time with 3 levels, customer with 3 levels and product with 7 levels. We defined 168 MVs as part of a

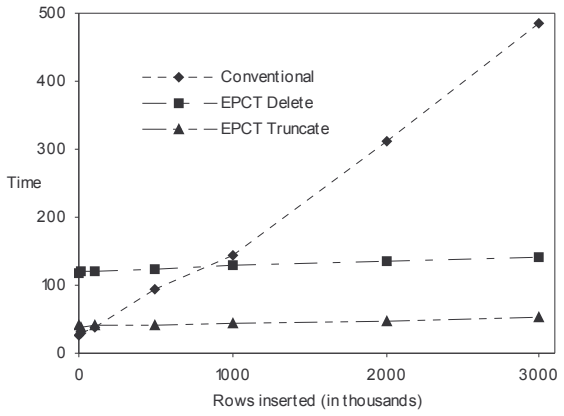
"federated cube" corresponding to a level combination for each of the 4 dimensions[3]. The fact table and MVs were partitioned on the time dimension. The fact table has 62 million rows which together with MVs expand to a total of 350 million rows. The experiments were conducted on a 24 CPU, 366 Mhz shared memory machine with a 24GB of memory.

Experiment 1: Building MVs using complete refresh.
The performance of building the 168 MVs using complete refresh improved more than 6 times in Oracle Database 10*g* compared to the naive method of building the MVs from the detail data in Oracle Database 9*i*. The naïve method spawned maximum number of refresh jobs allowed by the system in a random order.

Experiment 2: Incremental refresh of all MVs.
In this experiment, we measured the time taken to incrementally refresh all the MVs after inserting 1 new month of data into the fact table. Fact table has sales data for 17 months and we have inserted data for the 18th month, or 3.5 million rows. Our results show that refresh in Oracle 10g is 5 times better than in Oracle 9i.

Experiment 3: EPCT vs. Conventional Refresh.
This experiment compares the performance of EPCT refresh methods with the conventional refresh method. Number of rows inserted into the fact table is varied from 1000 to 3,000,000 and the time taken to refresh the MV on quarter, channel, customer, and product level is measured. This MV is partitioned on quarter and has 22 million rows. The results of this experiment are shown in Figure 1.

Figure 1: EPCT vs. Conventional Refresh

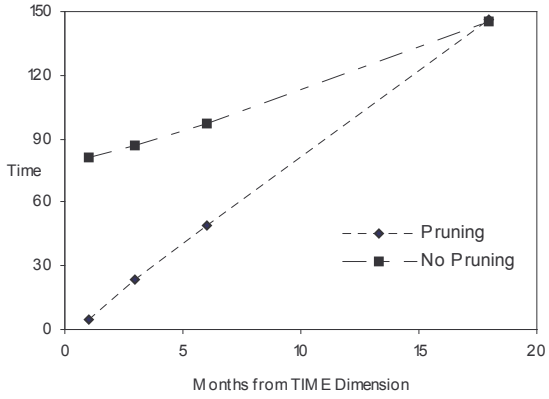


Our results show that EPCT using truncate and conventional refresh perform closely when few thousands of rows are inserted into the fact table. As expected, both EPCT refresh method's performance doesn't vary much on the number of rows inserted. Conventional refresh performance deteriorates as more and more rows are inserted into the fact table. EPCT with delete is more expensive than EPCT with truncate as we had to delete rows from the entire partition (one quarter) of the MV. Truncate is much faster in this case and hence EPCT with truncate is better. If the MV partition has data for more than one quarter, then EPCT with delete can be faster for cases involving fewer rows.

Experiment 4: Impact of dynamic partition pruning.
In this experiment, we illustrate the impact of dynamic partition pruning. The graph in Figure 2 shows the performance of a query joining the fact table with the TIME dimension. The query has a filter predicate on the month level of the TIME dimension and aggregates along the CHANNEL dimension. We varied the predicate selectivity from 1 to 18 months and measured the performance with and without dynamic partition pruning. In all the cases the query ran in serial and used a hash join.

The performance of the query with pruning is proportional to the number of partitions selected. Without pruning the performance is limited by the full table scan cost of the fact table. As expected, the performance with and without pruning was identical when all the months (18 months in Figure 2: Dynamic Partition Pruning) were selected. The numbers with a parallel plan show similar behavior.

Figure 2: Dynamic Partition Pruning



8. Conclusion

We presented an approach to optimizing refresh of a single materialized view by using partitioning of base

[3] TIME dimension doesn't have "ALL" level in its definition unlike other dimensions of APB schema. So, we have also created MVs with "ALL" level in TIME dimension and hence have 168 MVs.

tables and materialized views. In addition we showed an algorithm to derive an optimal schedule of refreshing a set of materialized views. We demonstrated the performance benefits of building and maintaining such materialized views.

9. Acknowledgements

We would like to thank Charles Sperry for his insightful comments about building and maintaining OLAP cubes. We are grateful to Jack Raitto and Shilpa Lawande for their insightful comments about Oracle Materialized View subsystem. We also like to thank Susy Fan and Saroj Sancheti for their assistance with the performance experiments. Finally, we thank John Haydu for his technical reviews.

References

[1] APB Benchmark Specifications. http://www.olapcouncil.org/research/APB1R2_spec.pdf

[2] S. Agarwal, R. Agrawal, P.M. Deshpande, A. Gupta, J.F. Naughton, R. Ramakrishnan, and S. Sarawagi, "On the computation of multidimensional aggregates", *Proceedings of 22nd Conference on VLDB*, Mumbai (Bombay), India.

[3] R. G. Bello, et al., "Materialized Views In Oracle", *Proceedings of 24th Conference on VLDB*, New York, USA.

[4] J. A. Blakeley, P. Larson, and F. W. Tompa, "Efficiently Updating Materialized Views", *Proceedings of ACM SIGMOD 1986*, Washington D.C., USA, 1986.

[5] D.Chatziantoniou and K.Ross, "Querying Multiple features of Groups in Relational Databases", *Proceedings of 22nd VLDB Conference*, Mumbai, India.

[6].S. Chaudhuri, R. Krishnamurthy, S. Potomianos, and K. Shim, "Optimizing queries with materialized views", *Proceedings of 11th Conference on Data Engineeing*, Taipei, Taiwan, 1995.

[7] S. Cohen, W. Nutt, and A. Serebrenik, "Rewriting aggregate queries using views*", Proceedings of 18th ACM PODS Symposium*, Philadelphia, USA, 1999.

[8] A. Gupta, I.S. Mumick, and V. S. Subrahmanian, "Maintaining views incrementally", *Proceedings of the ACM SIGMOD Conference 1993*, Washington D.C., USA, 1993.

[9] Jim Gray, Adam Bosworth, Andrew Layman, Hamid Pirahesh, "Data Cube: A Relational aggregation operator generalizing group-by, cross-tab, and sub-total", *Proceedings of 12th Conference on Data Engineering*, New Orleans, Louisiana, USA.

[10] T. Griffin and L. Libkin, "Incremental maintenance of views with duplicates", *Proceedings of ACM SIGMOD 1995*, San Jose, USA, 1995.

[11] W. J. Labio, J. Yang, Y. Cui, H. Garcia-Molina, and J. Widom, "Performance issues in incremental warehouse Maintenance", *Proceedings of 26th Conference on VLDB*, Cairo, Egypt, 2000.

[12] A. Levy, A. Rajaraman, and J. J. Ordille, "Answering queries using views", *Proceedings of 14th ACM PODS Symposium*, San Jose, USA, 1995.

[13] I. S. Mumick, D. Quass, and B. S. Mumick, "Maintenance of data cubes and summary tables in a warehouse", *Proceedings of ACM SIGMOD 1997*, Tucson, USA, 1997.

[14] Oracle 9*i* Datawarehouse manual. Multi-Table Insert

[15] R. Pottinger, A. Levy, "A scalable algorithm for answering queries using views", *Proceedings of 26th Conference on VLDB*, Cairo, Egypt, 2000.

[16] D. Srivastava, S. Rar, H. V. Jagadish and A. Levy, "Answering SQL queries using materialized views", *Proceedings of the 26th Conference on VLDB*, Mumbai, India, 1996.

[17] R. Tarjan, "Dept-first search and linear graph algorithms," *SIAM J. Computing*, 1997.

[18] A.Witkowski, et al., "Spreadsheets in RDBMS for OLAP", *Proceedings of ACM SIGMOD 2003,* San Diego, USA, 2003.

[19] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, and M. Urata, "Answering complex SQL queries using automatic summary tables", *Proceedings of ACM SIGMOD 2000*, Dallas, USA, 2000.

[20] F. Zemke "Rank, Moving and reporting functions for OLAP" 99/01/22 proposal for ANSI-NCTS.

[21] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom, "View maintenance in a warehouse environment", *Proceedings of ACM SIGMOD 1995*, San Jose, USA, 1995.

.