Maintenance of Data Cubes and Summary Tables in a Warehouse

Inderpal Singh Mumick
AT&T Laboratories
mumick@research.att.com

Dallan Quass
Stanford University
quass@cs.stanford.edu

Barinderpal Singh Mumick*
Lucent Technologies
bmumick@lucent.com

Abstract

Data warehouses contain large amounts of information, often collected from a variety of independent sources. Decision-support functions in a warehouse, such as *on-line analytical processing* (OLAP), involve hundreds of complex aggregate queries over large volumes of data. It is not feasible to compute these queries by scanning the data sets each time. Warehouse applications therefore build a large number of *summary tables*, or materialized aggregate views, to help them increase the system performance.

As changes, most notably new transactional data, are collected at the data sources, all summary tables at the warehouse that depend upon this data need to be updated. Usually, source changes are loaded into the warehouse at regular intervals, usually once a day, in a batch window, and the warehouse is made unavailable for querying while it is updated. Since the number of summary tables that need to be maintained is often large, a critical issue for data warehousing is how to maintain the summary tables efficiently.

In this paper we propose a method of maintaining aggregate views (the *summary-delta table* method), and use it to solve two problems in maintaining summary tables in a warehouse: (1) how to efficiently maintain a summary table while minimizing the batch window needed for maintenance, and (2) how to maintain a large set of summary tables defined over the same base tables.

While several papers have addressed the issues relating to choosing and materializing a set of summary tables, this is the first paper to address maintaining summary tables efficiently.

1 Introduction

Data warehouses contain information that is collected from multiple, independent data sources and integrated into a common repository for querying and analysis. Often, data warehouses are designed for *on-line analytical processing* (OLAP), where the queries aggregate large volumes of data in order to detect trends and anomalies. In order to speed up query processing in such environments, warehouses usually contain a large number of *summary tables*, which represent materialized aggregate views of the base data collected from the sources. The summary tables group the base data along various *dimensions*, corresponding to different sets of group-by attributes, and compute various aggregate functions, often called *measures*. As an example, the cube operator [GBLP96] can be used to define several such summary tables with one statement.

As changes are made to the data sources, the warehouse views must be updated to reflect the changed state of the data sources. The views either can be recomputed from scratch, or incremental maintenance techniques [BC79, SI84, RK86, BLT86, Han87, SP89, QW91, Qua96, CW91, GMS93, GL95, LMSS95, ZGHW95] can be used to calculate the changes to the views due to the source changes. It is common in a data warehousing environment for source changes to be *deferred* and applied to the warehouse views in large batches for efficiency. Source changes received during the day are applied to the views in a nightly batch window, during which time the warehouse is unavailable to readers.

The nightly batch window involves updating the base tables (if any) stored at the warehouse, and maintaining all the materialized summary tables. The problem with this approach is that the warehouse is typically unavailable to readers while the views are being maintained, due to the large number of updates that need to be applied. Since the warehouse must be made available to readers again by the next morning, the time required for maintenance is often a limiting factor in the number of summary tables that can be made available in the warehouse. Because the number of summary tables available has such a significant impact on OLAP query performance, maintaining the summary tables efficiently is crucial.

This paper addresses the issue of efficiently maintaining a set of summary tables in a data warehouse. Using efficient incremental maintenance techniques, it is possible to increase the number of summary tables available in the warehouse, or alternatively, to decrease the time that the warehouse is unavailable to readers. The paper includes the following contributions:

- We propose a new method, called the summary-delta tables method, for maintenance of aggregate views. The summary-delta tables method represents a new paradigm for incremental view maintenance.
- A general strategy to minimize the batch time needed for maintenance is to split the maintenance work into

*Work performed while at New Jersey Institute of Technology.

propagate and *refresh* functions [CGL+96]. Propagate can occur outside the batch window, while refresh occurs inside the batch window. We show how the propagate and refresh functions can be derived for aggregate views.

- We show how multiple summary tables can be related so that their maintenance can take advantage of the computation done to maintain other summary tables.

Paper outline: Section 2 presents a motivating example illustrating the importance of efficient incremental maintenance of summary tables. Background and notation is given in Section 3. Section 4 presents propagate and refresh functions for maintaining individual summary tables. Section 5 explains how multiple summary tables can be maintained efficiently together. A performance study of the summary-delta table method, based upon an experimental implementation, is presented in Section 6. Related work and conclusions appear in Section 7.

2 Motivating Example

Consider a warehouse of retail information, with point-of-sale (pos) data from hundreds of stores. The point of sale data is stored in the warehouse in a large pos table, called a *fact table*, that contains a tuple for each item sold in a sales transaction. Each tuple has the format

pos(storeID, itemID, date, qty, price).

The attributes of the tuple are the id of the store selling the item, the id of the item sold, the date of the sale, the quantity of the item sold, and the selling price of the item. The pos table is allowed to contain duplicates, for example, when an item is sold in different transactions in the same store on the same date.

In addition, a warehouse will often store *dimension tables*, which contain information related to the fact table. Let the stores and items tables contain store information and item information, respectively. The key of stores is storeID, and the key of items is itemID.

stores(storeID, city, region).
items(itemID, name, category, cost).

Data in dimension tables often represents *dimension hierarchies*. A dimension hierarchy is essentially a set of functional dependencies among the attributes of the dimension table. For our example we will assume that in the stores dimension hierarchy, storeID functionally determines city, and city functionally determines region. In the items dimension hierarchy, itemID functionally determines name, category, and cost.

In order to answer aggregate queries quickly, a warehouse will often store a number of summary tables, which are materialized views that aggregate the data in the fact table, possibly after joining it with one or more dimension tables. Figure 1 shows four summary tables, each defined as a materialized SQL view. We assume that these views have been chosen to be materialized, either by the database administrator, or by using an algorithm such as [HRU96].

Note that the names of the views have been chosen to reflect the group-by attributes. The character S represents storeID, I represents itemID, and D represents date. The notation sC represents the city for a store, sR represents the region for a store, and iC represents the category for an item. For example, the name SiC_sales implies that storeID and category are the group-by attributes in the view definition.

```
CREATE VIEW SID_sales(storeID, itemID, date, TotalCount,
                      TotalQuantity) AS
SELECT storeID, itemID, date, COUNT(*) AS TotalCount,
       SUM(qty) AS TotalQuantity
FROM pos
GROUP BY storeID, itemID, date

CREATE VIEW sCD_sales(city, date, TotalCount,
                      TotalQuantity) AS
SELECT city, date, COUNT(*) AS TotalCount,
       SUM(qty) AS TotalQuantity
FROM pos, stores
WHERE pos.storeID = stores.storeID
GROUP BY city, date

CREATE VIEW SiC_sales(storeID, category, TotalCount,
                      EarliestSale, TotalQuantity) AS
SELECT storeID, category, COUNT(*) AS TotalCount,
       MIN(date) AS EarliestSale,
       SUM(qty) AS TotalQuantity
FROM pos, items
WHERE pos.itemID = items.itemID
GROUP BY storeID, category

CREATE VIEW sR_sales(region, TotalCount,
                     TotalQuantity) AS
SELECT region, COUNT(*) AS TotalCount,
       SUM(qty) AS TotalQuantity
FROM pos, stores
WHERE pos.storeID = stores.storeID
GROUP BY region
```

Figure 1: Example summary tables

The views of Figure 1 could represent four of the possible points on a "data cube" as described in [GBLP96], except for the use of date as both a dimension and a measure. Another difference between this paper and previous work on data cubes is that in previous work the data being aggregated comes solely from the fact table, with dimension hierarchy information obtained implicitly. As mentioned earlier, data warehouses typically store dimension hierarchy information explicitly in dimension tables; in this paper we extend the data-cube concept to include explicit joins with dimension tables (see Section 3.3).

As sales are made, changes representing the new point-of-sale data come into the warehouse. As mentioned in Section 1, most warehouses do not apply the changes immediately. Instead, changes are deferred and applied to the base tables and summary tables in the warehouse at night in a single batch. Deferring the changes allows analysts that query the warehouse to see a consistent snapshot of the data throughout the day, and can make the maintenance more efficient.

Although it is often the case that changes to a warehouse involve only insertions, for the sake of example in this paper we will assume that the changes involve both insertions and deletions. In order to correctly maintain an aggregate view in the presence of deletions it is necessary to include a COUNT(*) aggregate function in the view. Having COUNT(*) makes it possible to determine when all tuples in a group have been deleted (*i.e.*, when COUNT(*) for the group becomes 0), implying the deletion of the tuple for the group in the view. We have included COUNT(*) explicitly in the example views above, but it also could be added implicitly when the view is materialized in the warehouse.

For simplicity of presentation, we will usually assume in this paper that maintenance is performed in response to changes only to the fact table, and that the columns being aggregated do not include null values. However, the algorithms we present are easily extended to handle changes also to the dimension tables, as well as nulls in the aggregated columns. The effect of changes to dimension tables is considered in Section 4.1.4, and the effect of nulls in the aggregated columns is considered in Section 3.1.

2.1 Maintaining a single summary table

We will illustrate the summary-delta table method through an example, using it to maintain the SID_sales summary table of Figure 1. Later in Section 2.2, we show that much of the work in maintaining SID_sales can be re-used to maintain the other summary tables in the figure. The complete algorithms for maintaining a single summary table and a set of summary tables appear in Sections 4 and 5 respectively.

An important aspect of our maintenance algorithm is that the maintenance process is divided into two functions: propagate and refresh. The work of computing a summary-delta table happens within the propagate function, which can take place without locking the summary tables so that the warehouse can continue to be made available for querying by analysts. Summary tables are not locked until the refresh function, during which time the summary table is updated from the summary-delta table.

Propagate: The propagate function involves creating a *summary-delta table* from the deferred set of changes. The summary-delta table represents the net changes to the summary table due to the changes to the fact table. Let the deferred set of insertions be stored in table pos_ins and the deferred set of deletions be stored in table pos_del. Then the summary-delta table is derived using the following SQL statement, without accessing the base pos table.

```
CREATE VIEW sd_SID_sales (storeID, itemID, date, sd_Count,
                          sd_Quantity) AS
SELECT storeID, itemID, date, SUM(_count) AS sd_Count,
       SUM(_quantity) AS sd_Quantity
FROM ( (SELECT storeID, itemID, date, 1 as _count,
               qty as _quantity
        FROM pos_ins)
       UNION ALL
       (SELECT storeID, itemID, date, -1 as _count,
               -qty as _quantity
        FROM pos_del) )
GROUP BY storeID, itemID, date
```

To compute the summary-delta table, we first perform a projection on the inserted and deleted tuples so that we have 1 for count and qty for quantity from the inserted tuples, and the negative of those values from the deleted tuples. We then take the union of this result and aggregate it, grouping by the same group-by attributes as in the summary table. The resulting aggregate function values represent the net changes to the corresponding aggregate function values in the summary table. The propagate function is explained fully in Section 4.1.

Refresh: The refresh function applies the net changes represented in the summary-delta table to the summary table. The function to refresh SID_sales appears in Figure 2, and is described below. It takes as input the summary-delta table sd_SID_sales, and the summary table SID_sales, and updates the summary table to reflect the changes in the summary-delta table. For simplicity, we assume here that there are no null values in pos. Null values will be considered later when deriving the generic refresh algorithm.

The refresh function has been designed to run quickly. Except for certain cases involving MIN and MAX (see Section 4.2), the refresh function does not require access to the base pos table, and all aggregation is performed in the propagate function. Each tuple in the summary-delta table causes a single update to the summary table, and each tuple in the summary table is updated at most once.

```
For each tuple δt in sd_SID_sales:
  Let tuple t =
    (SELECT *
     FROM SID_sales d
     WHERE d.storeID = δt.storeID AND
           d.date = δt.date AND
           d.itemID = δt.itemID)
  If t is not found,
    Insert tuple δt into SID_sales
  Else /* if t is found */
    If δt.sd_Count + t.TotalCount = 0,
      Delete tuple t from SID_sales
    Else
      Update tuple t.TotalCount += δt.sd_Count,
             t.TotalQuantity += δt.sd_Quantity
```

Figure 2: Refresh function for SID_sales.

Intuitively, the refresh function of Figure 2 can be written as an embedded SQL program using cursors as follows. A cursor c_1 is opened to iterate over each tuple δt in the summary-delta table sd_SID_sales. For each δt, a query is issued and a second cursor c_2 is opened to find a matching tuple t in the summary table SID_sales (there is at most one matching t since the match is on the group-by attributes). If a matching tuple t is not found, then the δt tuple is inserted into the summary table. Otherwise, if t is found it is updated or deleted using cursor c_2, depending upon whether all tuples in t's group have been deleted. The refresh function is explained fully in Section 4.2, including an explanation of how it can be optimized when certain integrity constraints on the changes hold.

2.2 Maintaining multiple summary tables

We now give propagate functions that create summary deltas for the remaining summary tables of Figure 1. Efficiently maintaining multiple summary tables together allows more opportunity for optimization than maintaining each summary table individually, because the summary-delta table computed for the maintenance of one summary table often can be used to compute summary-delta tables for other summary tables. Since a summary-delta table already involves some aggregation over the changes to the base tables, it is likely to be smaller than the changes themselves, so using a summary-delta table to compute other summary-delta tables will likely require fewer tuple accesses than computing each summary-delta table from the changes directly.

The queries defining summary-delta tables for sCD_sales, SiC_sales, and sR_sales are shown in Figure 3. The summary-delta tables for sCD_sales and SiC_sales both reference the summary-delta table for SID_sales, and the summary-delta table for sR_sales references the summary-delta table for sCD_sales.

```
CREATE VIEW sd_sCD_sales(city, region, date, sd_Count,
                         sd_Quantity) AS
SELECT city, region, date, sum(sd_Count) AS sd_Count,
       sum(sd_Quantity) AS sd_Quantity
FROM sd_SID_sales, stores
WHERE sd_SID_sales.storeID = stores.storeID
GROUP BY city, region, date

CREATE VIEW sd_SiC_sales(storeID, category, sd_Count,
                         sd_EarliestSale, sd_Quantity) AS
SELECT storeID, category, sum(sd_Count) AS sd_Count,
       min(date) AS sd_EarliestSale,
       sum(sd_Quantity) AS sd_Quantity
FROM sd_SID_sales, items
WHERE sd_SID_sales.itemID = items.itemID
GROUP BY storeID, category

CREATE VIEW sd_sR_sales(region, sd_Count, sd_Quantity) AS
SELECT region, sum(sd_Count) AS sd_Count,
       sum(sd_Quantity) AS sd_Quantity
FROM sd_sCD_sales
GROUP BY region
```

Figure 3: Propagate Functions

Note that the summary-delta table sd_sCD_sales includes the region attribute, which is not necessary to maintain sCD_sales. Region is included so that later in the definition of sd_sR_sales we do not need to join sd_sCD_sales with stores. Including region in sd_sCD_sales does not affect the maintenance of sCD_sales because in the dimension hierarchy for cities we have specified that city functionally determines region, *i.e.*, every city belongs to a single region, so grouping by (city, region, date) results in the same groups as grouping by (city, date).

The refresh functions corresponding to the summary-delta tables of Figure 3 are not given in this section. In general they follow in a straightforward fashion from the example refresh function for SID_sales given in Section 2.1, with the exception of the MIN aggregate function in SiC_sales. In Section 4.2 we show how the refresh function handles MIN and MAX aggregate functions.

3 Background and Notation

In this section we review the concepts of self-maintainable aggregate functions (Section 3.1), data cube (Section 3.2), and the computation lattice corresponding to a data cube (Section 3.3).

3.1 Self-maintainable aggregate functions

In [GBLP96], aggregate functions are divided three classes: *distributive*, *algebraic*, and *holistic*. Distributive aggregate functions can be computed by partitioning their input into disjoint sets, aggregating each set individually, then further aggregating the (partial) results from each set into the final result. Amongst the aggregate functions found in standard SQL, COUNT, SUM, MIN, and MAX are distributive. For example, COUNT can be computed by summing partial counts. Note however, if the DISTINCT keyword is used, as in COUNT(DISTINCT E) (count the distinct values of E) then these functions are no longer distributive.

Algebraic aggregate functions can be expressed as a scalar function of distributive aggregate functions. Average is algebraic, since it can be expressed as SUM/COUNT. From now on we will assume that if a view is supposed to contain the AVG aggregate function, the materialized view will contain instead the SUM and COUNT functions.

Holistic aggregate functions cannot be computed by dividing into parts. Median is an example of a holistic aggregate function. We will not consider holistic functions in this paper.

Definition 3.1 (Self-maintainable aggregates): A set of aggregate functions is *self-maintainable* if the new value of the functions can be computed solely from the old values of the aggregation functions and from the changes to the base data. Aggregate functions can be self-maintainable with respect to insertions, with respect to deletions, or both. ■

In order for an aggregate function to be self-maintainable it must be distributive. In fact, all distributive aggregate functions are self-maintainable with respect to insertions. However, not all distributive aggregate functions are self-maintainable with respect to deletions. The COUNT(*) function can help to make certain aggregate functions self-maintainable with respect to deletions, by helping to determine when all tuples in the group (or in the full table if a group-by is not performed) have been deleted, so that the grouped tuple can be deleted from the view.

The function COUNT(*) is always self-maintainable with respect to deletions. Including COUNT(*) also makes the function COUNT(E), (count the number of non-null values of E), self-maintainable with respect to deletions. If nulls are not allowed in the input, then COUNT(*) also makes SUM(E) self-maintainable with respect to deletions. In the presence of nulls, both COUNT(*) and COUNT(E) are required to make SUM(E) self-maintainable.

MIN and MAX functions: MIN and MAX are not self-maintainable with respect to deletions, and cannot be made self-maintainable. For instance, when a tuple having the minimum (maximum) value is deleted, the new minimum (maximum) value for the group must be recomputed from the changes and the base data. Including COUNT(*) can help a little (if COUNT(*) reaches 0, there is no other tuple in the group, so the group can be deleted), but COUNT(*) cannot make MIN and MAX self-maintainable. (If COUNT(*) > 0 after a tuple having minimum (maximum) value is deleted, we still need to look up the base table.) COUNT(E) can also help in maintaining MIN(E) and MAX(E) (If Count(*) > 0 and COUNT(E) $= 0$, then MIN(E)= null), but COUNT(E) also cannot make MIN and MAX self-maintainable (if Count(*) > 0 and COUNT(E) > 0, and a tuple having minimum (maximum) value is deleted, then we need to look up the base table).

3.2 Data cube

The date cube [GBLP96] is a convenient way of thinking about multiple aggregate views, all derived from a fact table using different sets of group-by attributes. Data cubes are popular in OLAP because they provide an intuitive way for data analysts to navigate various levels of summary information in the database. In a data cube, attributes are categorized into *dimension attributes*, on which grouping may be performed, and *measures*, which are the results of aggregate functions.

Cube Views: A data cube with k dimension attributes is a shorthand for 2^k cube views, each one defined by a single SELECT-FROM-WHERE-GROUPBY block, having identical aggregation functions, identical FROM and WHERE clauses, no HAVING clause, and one of the 2^k subsets of the dimension attributes as the groupby columns.

EXAMPLE 3.1 An example data cube for the pos table of Section 2 is shown in Figure 4 as a lattice structure. Construction of the lattice corresponding to a data cube was

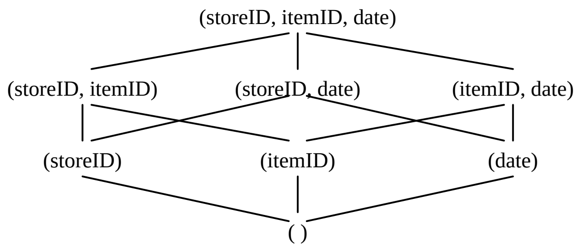


Figure 4: Data Cube Lattice.

first introduced in [HRU96]. The dimension attributes of the data cube are storeID, itemID, and date, and the measures are COUNT(*) and SUM(qty). Since the measures computed are assumed to be the same, each point in the figure is annotated simply by the group-by attributes. Thus, the point (storeID, itemID) represents the cube view corresponding to the query

(SI): SELECT storeID, itemID, COUNT(*), SUM(qty)
FROM pos
GROUP BY storeID, itemID .

■

Edges in a lattice run from the node above to the node below. Each edge $v1 \rightarrow v2$ implies that $v2$ can be answered using $v1$, instead of accessing the base data. The edge defines a query that derives view $v2$ below from the view $v1$ above by simply replacing the table in the FROM clause with the name of the view above, and by replacing any COUNT aggregate function with the SUM aggregate function. For example, the edge from $v1 =$ (storeID, itemID, date) to $v2 =$ (storeID, itemID) defines the following query equivalent to query SI above (assuming that the aggregate columns in the views are named count and qty).

(SI'): SELECT storeID, itemID, SUM(count), SUM(qty)
FROM v1
GROUP BY storeID, itemID .

Generalized Cube Views: However, in most warehouses and decision support systems, the set of summary tables do not fit into the structure of cube views—they differ in their aggregation functions and the joins they perform with the fact tables[1]. Further, some views may do aggregation on columns used as dimension attributes in other views. We will call these views *generalized cube views*, and define them as traditional cube-style views that are extended in the following ways:

- different views may compute different aggregate functions,
- some views may compute aggregate functions over attributes that are used as group-by attributes in other views,
- views may join with different combinations of dimension tables (note that dimension-table joins are always along foreign keys).

[1]They may also differ in the WHERE clause, but we do not consider differing WHERE clauses in this paper.

3.3 Dimension hierarchies and lattices

As mentioned in Section 2, the various dimensions represented by the group-by attributes of a fact table often are organized into dimension hierarchies. For example, in the stores dimension, stores can be grouped into cities, and cities can be grouped into regions. In the items dimension, items can be grouped into categories.

The dimension hierarchy information can be stored in separate dimension tables, as we did in the stores and items tables. In order to group by attributes further along the dimension hierarchy, the fact table must be joined with the dimension tables before doing the aggregation. The joins between the fact table and dimension tables are always along foreign keys, so each tuple in the fact table is guaranteed to join with one and only one tuple from each dimension table.

A dimension hierarchy can also be represented by a lattice, similar to a data-cube lattice. We can construct a lattice representing the set of views that can be obtained by grouping on each combination of elements from the set of dimension hierarchies. It turns out that a *direct product* of the lattice for the fact table along with the lattices for the dimension hierarchies yields the desired result [HRU96]. For example, given that stores are grouped into cities and then regions, and items are grouped into categories, Figure 5 shows the lattice combining the fact table lattice of Figure 4 with the dimension hierarchy lattices of store and item.

3.4 Partially-materialized lattices

A partially-materialized lattice is obtained by removing some nodes of the lattice, to represent the fact that the corresponding views are not being materialized. When a node n is removed, all incoming and outgoing edges from node n are also removed, and new edges are added between nodes above and below node n. For every incoming edge $(n1, n)$, and every outgoing edge $(n, n2)$, we add an edge $(n1, n2)$. The query defining view $n2$ along the edge $(n1, n2)$ is obtained from the query along the edge $(n, n2)$ by replacing view n in the FROM clause with view $n1$. Note that if the top and/or bottom elements of the lattice are removed, the resulting partially-materialized lattice may not be a lattice - it represents a partial order between nodes without a top and/or a bottom element.

4 Basic Summary-Delta Maintenance Algorithm

In this section we show how to efficiently maintain a summary table given changes to the base data. Specifically, we give propagate and refresh functions for maintaining a generalized cube view of the type described in Section 3.2, including joins with dimension tables. We require that the aggregate functions calculated in the summary table either be self-maintainable, be made self-maintainable by adding the appropriate COUNT functions as described in Section 3.1, or be MIN or MAX aggregate functions (in which case the circumstances under which they are not self-maintainable are detected and handled in the refresh function). For simplicity, we start out by considering changes (insertions and deletions) only to the base fact table. We consider changes to the dimension tables in Section 4.1.4.

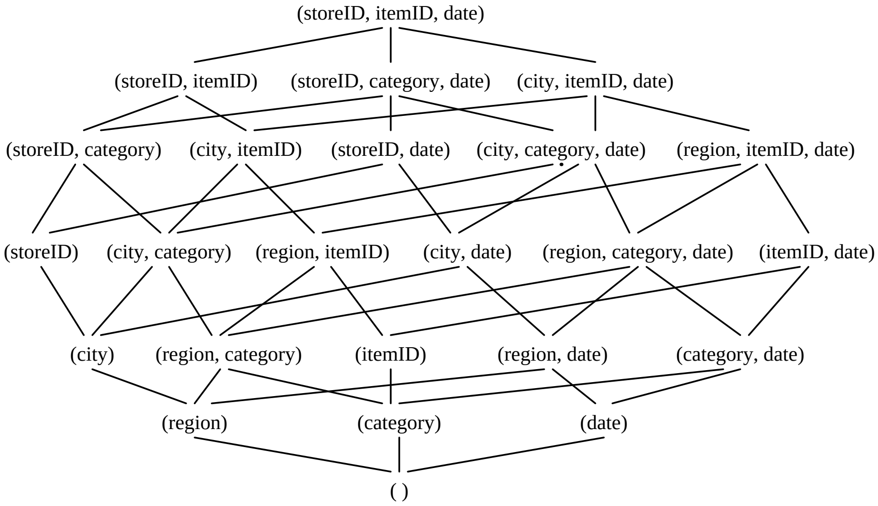


Figure 5: Combined lattice.

4.1 Propagate function

As described briefly in Section 2, the general intuition for the propagate function is to create a *summary-delta table* that contains the net effect of the changes on the summary table. Since the propagate function does not affect the summary table, the summary table can continue to be available to readers while the propagate function is computed. Therefore, the goal of the propagate function is to do as much work as possible so that the time required by the refresh function is minimized.

4.1.1 Preparing changes

In order to make the computation of the summary-delta table easier to understand, we split up some of the work by first defining three *virtual* views: *prepare-changes*, *prepare-insertions* and *prepare-deletions*. The prepare-changes virtual view is defined simply as the union of prepare-insertions and prepare-deletions, which are described below. In Section 4.1.2 we will see that the summary-delta table is computed from the prepare-changes virtual view.

The prepare-insertions and prepare-deletions views derive the changes to the aggregate functions caused by individual insertions and deletions, respectively, to the base data. They take a projection of the insertions/deletions to the base data, after applying any selections conditions and joins that appear in the definition of the summary table. The projected attributes include

- each of the group-by attributes of the summary table, and
- *aggregate-source* attributes corresponding to each of the aggregate functions computed in the summary table.

An aggregate-source attribute computes the result of the expression on which the aggregate function is applied. For example, if the summary table included the aggregate function `sum(A*B)`, the prepare-insertions and prepare-deletions virtual views would each include in their select clause an aggregate-source attribute computing either $A{*}B$ (for prepare-insertions), or $-(A * B)$ (for prepare-deletions). We will see later that the aggregate-source attributes are aggregated when defining the summary-delta table.

The aggregate-source attributes are derived according to Table 1. The column labeled prepare-insertions describes how they are derived for the prepare-insertions view; the column labeled prepare-deletions describes how they are derived for the prepare-deletions view. The `COUNT`($expr$) row uses the SQL-92 case statement [MS93].

	prepare-insertions	prepare-deletions
`COUNT(*)`	1	-1
`COUNT`($expr$)	case when $expr$ is null then 0 else 1	case when $expr$ is null then 0 else -1
`SUM`($expr$)	$expr$	$-expr$
`MIN`($expr$)	$expr$	$expr$
`MAX`($expr$)	$expr$	$expr$

Table 1: Deriving aggregate-source attributes

EXAMPLE 4.1 Consider the SiC_sales view of Figure 1. The prepare-insertions, prepare-deletions, and the prepare-changes virtual views for SiC_sales are shown in Figure 6. The prepare-insertions view name is prefixed by "`pi_`," the prepare-deletions view name is prefixed by "`pd_`," and the prepare-changes view name is prefixed by "`pc_`." The aggregate sources are named _count, _date, and _quantity, respectively. ■

4.1.2 Computing the summary-delta table

The summary-delta table is computed by aggregating the prepare-changes virtual view. The summary-delta table has the same schema as the summary table, except that the attributes resulting from aggregate functions in the summary delta represent changes to the corresponding aggregate functions in the summary table. For this reason we name attributes resulting from aggregate functions in the summary-delta table after the name of the corresponding attribute in the summary table, prefixed by "`sd_`."

Each tuple in the summary-delta table describes the effect of the base-data changes on the aggregate functions of a corresponding tuple in the summary table (*i.e.*, a tuple in the summary table having the same values for all group-by attributes as the tuple in the summary-delta table). Note

```
CREATE VIEW pi_SiC_sales(storeID, category, _count, _date,
                         _quantity) AS
SELECT storeID, category, 1 AS _count, date AS _date,
       qty AS _quantity
FROM pos_ins, items
WHERE pos_ins.itemID = items.itemID

CREATE VIEW pd_SiC_sales(storeID, category, _count, _date,
                         _quantity) AS
SELECT storeID, category, -1 AS _count, date AS _date,
       -qty AS _quantity
FROM pos_del, items
WHERE pos_del.itemID = items.itemID

CREATE VIEW pc_SiC_sales(storeID, category, _count, _date,
                         _quantity) AS
SELECT *
FROM (pi_SiC_sales UNION ALL pd_SiC_sales)
```

Figure 6: Prepare changes example

that a corresponding tuple in the summary table may not exist, and in fact it is sometimes necessary in the refresh function to insert a tuple into (or delete a tuple from) the summary table due to the changes represented in the summary-delta table.

The query to compute the summary-delta table follows from the query computing the summary table, with the following differences:

- The FROM clause is replaced by prepare-changes.
- The WHERE clause is removed. (It is already applied when defining prepare-insertions and prepare-deletions.)
- The expressions on which the aggregate functions are applied are replaced by references to the aggregate-source attributes of prepare-changes.
- COUNT aggregate functions are replaced by SUM.

Note that computing the summary-delta table involves essentially aggregating the tuples in the changes. Thus, techniques for parallelizing aggregation can be used to speed up computation of the summary-delta table.

EXAMPLE 4.2 Consider again the SiC_sales view of Figure 1. The query computing the summary-delta table for SiC_sales is shown below. It aggregates the changes represented in the prepare-changes virtual view, grouping by the same group-by attributes as the summary table.

```
CREATE VIEW sd_SiC_sales(storeID, category, sd_Count,
                         sd_EarliestSale, sd_Quantity) AS
SELECT storeID, category, sum(_count) AS sd_Count,
       min(_date) AS sd_EarliestSale,
       sum(_quantity) AS sd_Quantity
FROM pc_SiC_sales
GROUP BY storeID, category
```

The astute reader will recall that in Section 2.2 the summary-delta table for SiC_sales was defined using the summary-delta table for SID_sales. In this example we defined the summary-delta table using instead the changes to the base data. ■

4.1.3 Pre-aggregation

As a potential optimization, it is possible to pre-aggregate the insertions and deletions before joining with some of the dimension tables. In particular, joins with dimension tables whose attributes are not referenced in the aggregate functions, can be delayed until after pre-aggregation. Delaying joins until after pre-aggregation reduces the number of tuples involved in the join, potentially speeding up the computation of the summary-delta table. The decision of whether or not to pre-aggregate could be made in a cost-based manner by a query optimizer. The notion of pre-aggregation follows essentially from the idea of pushing down aggregation presented in [CS94, GHQ95, YL95].

4.1.4 Changes to dimension tables

Up to now we have considered changes only to the fact table. Changes to the dimension tables can also be incorporated into our method. Due to space constraints we will only give the intuition underlying the technique.

Applying the incremental view-maintenance techniques of [GMS93, GL95], we can start with the changes to a dimension table, and derive dimension-table-specific prepare-insertions and prepare-deletions views that represent the changes to the aggregate functions due to changes to the dimension table. For example, the following view definition calculates prepare-insertions for SiC_sales due to insertions to items (made available in items_ins).

```
CREATE VIEW pi_items_SiC_sales(storeID, category, _count,
                               _date, _quantity) AS
SELECT storeID, category, 1 AS _count, date AS _date,
       qty AS _quantity
FROM pos, items_ins
WHERE pos.itemID = items_ins.itemID
```

Prepare-changes then takes the union of all such prepare-insertions and prepare-deletions views, representing changes to the fact table and all dimension tables, and the summary-delta computation proceeds as before.

4.2 Refresh function

The refresh function applies the changes represented in the summary-delta table to the summary table. Each tuple in the summary-delta table causes a change to a single corresponding tuple in the summary table (by corresponding we mean a tuple in the summary table having the same values for all group-by attributes as the tuple in the summary delta). The corresponding tuple in the summary table is either updated, deleted, or if the corresponding tuple is not found, the summary-delta tuple is inserted into the summary table.

The refresh algorithm is shown in Figure 7. It generalizes and extends the example refresh function given in Section 2.1, by handling the case of nulls in the input and MIN and MAX aggregate functions. In the algorithm, for each tuple δt in the summary-delta table the corresponding tuple t in the summary table is looked up. If t is not found, the summary-delta tuple is inserted into the summary table. If t is found, then if COUNT(*) from t plus COUNT(*) from δt is zero, then t is deleted.[2] Otherwise, a check is performed for each of the MIN and MAX aggregate functions, to see if a value less than or equal to the minimum (greater than or equal to the maximum) value was deleted, in which case the new MIN or MAX value of t will probably need to be recomputed. The only exception is if COUNT(e) from t plus COUNT(e) from δt is zero, in which case the new min/max/sum/count(e) values are null.

[2] Note that COUNT(*) from t plus COUNT(*) from δt can never be less than zero, and that if COUNT(*) from δt is less than zero, then the corresponding tuples t must be found in the summary table.

For each tuple δt in the summary-delta table,
  % get the corresponding tuple in the summary table
  Let tuple t = tuple in the summary table having the same values for its group-by attributes as δt
  If t is not found,
    % insert tuple
    Insert tuple δt into the summary table
  Else
    % check if the tuple needs to be deleted
    If $\delta t.\texttt{COUNT(*)} + t.\texttt{COUNT(*)} = 0$,
      Delete tuple t
    Else
      % check if min/max values must be recomputed
      recompute = false
      For each `MIN` and `MAX` aggregate function $m(e)$ in the summary table,
        If ((m is a `MIN` function AND
          $\delta t.\texttt{MIN}(e) \leq t.\texttt{MIN}(e)$ AND
          $t.\texttt{COUNT}(e) + \delta t.\texttt{COUNT}(e) > 0$) OR
         (m is a `MAX` function AND
          $\delta t.\texttt{MAX}(e) \geq t.\texttt{MAX}(e)$ AND
          $t.\texttt{COUNT}(e) + \delta t.\texttt{COUNT}(e) > 0$))
          recompute = true
      If (recompute)
        Update tuple t by recomputing its aggregate functions from the base data for t's group.
      Else
        % update the tuple
        For each aggregate function $a(e)$ in the summary table,
          If $t.\texttt{COUNT}(e) + \delta t.\texttt{COUNT}(e) = 0$,
            $t.a$ = null
          Else If a is `COUNT` or `SUM`,
            $t.a = t.a + \delta t.a$
          Else If a is `MIN`,
            $t.a = \texttt{MIN}(t.a, \delta t.a)$
          Else If a is `MAX`,
            $t.a = \texttt{MAX}(t.a, \delta t.a)$

Figure 7: The Refresh function

As the last step in the algorithm, the aggregation functions of tuple t are updated from the values in δt, or (if needed) by recomputing a min/max value from the base data for t's group. For simplicity in the recomputation, we assume that when a summary table is being refreshed, the changes have already been applied to the base data. However, an alternative would be to do the recomputation before the changes have been applied to the base table by issuing a query that subtracts the deletions from the base data and unions the insertions. As written, the refresh function only considers the `COUNT`, `SUM`, `MIN`, and `MAX` aggregate functions, but it should be easy to see how any self-maintainable aggregation function would be incorporated.

The above refresh function may appear complex, but conceptually it is very simple. One can think of it as a left outer-join between the summary-delta table and the summary table. Each summary table tuple that joins with a summary-delta tuple is updated or deleted as it joins, while a summary-delta tuple that does not join is inserted into the summary table. The only complication in the process is an occasional recomputation of a min/max value. The refresh function could be parallelized by partitioning the summary-delta table and summary table on the group-by attributes. Such a *"summary-delta join"* needs to be implemented in the database server, and should be implemented by database vendors that are targeting the warehouse market.

5 Efficiently maintaining multiple summary tables

In the previous section we have shown how to compute the summary-delta table for a generalized cube view, directly from the insertions and deletions into the base fact table.

We have also seen that multiple cube views can be arranged into a (partially-materialized) lattice (Section 3.3). We will now show that multiple summary tables, which are generalized cube views, can also be placed in a (partially-materialized) lattice, which we call a $\mathcal{V}$-lattice. Further, all the summary-delta tables can also be written as generalized cube views, and can be placed in a (partially-materialized) lattice, which we call a $\mathcal{D}$-lattice. It turns out that the $\mathcal{D}$-lattice is identical to the $\mathcal{V}$-lattice, modulo renaming of tables.

5.1 Placing generalized cube views into a lattice

The principle behind the placement of cube views in a lattice is that a cube view $v2$ should be derivable from the cube view $v1$ placed above $v2$ in the cube lattice. The same principle can be adapted to place a given set of generalized cube views into a (partially-materialized) lattice. We will show how to define a *derives relation* $v2 \prec v1$ between the given set of generalized cube views. The derives relation, $\prec$, can be used to impose a partial ordering on the set of generalized views, and to place the views into a (partially-materialized) lattice, with v_1 being an ancestor of v_2 in the lattice if and only if $v_2 \prec v_1$.

For two generalized cube views v_1 and v_2, let $v_2 \prec v_1$ if and only if view v_2 can be defined using a single block `SELECT-FROM-GROUPBY` query over view v_1 possibly joined with one or more dimension tables on the foreign key. The $v_2 \prec v_1$ condition holds if

1. each group-by attribute of v_2 is either a groupby attribute of v_1, or is an attribute of a dimension table whose foreign key is a groupby attribute of v_1, and
2. each aggregate function $a(E)$ of v_2 either appears in v_1, or E is an expression over the groupby attributes of v_1, or E is an expression over attributes of dimension tables whose foreign keys are groupby attributes of v_1.

If the above conditions are satisfied using dimension tables $d_1, \ldots, d_m$, we will superscript the $\prec$ relation as $\prec^{d_1,\ldots,d_m}$.

EXAMPLE 5.1 For our running retailing warehouse example, the following *derives* relationships exist: sCD_sales $\prec^{\text{stores}}$ SID_sales, SiC_sales $\prec^{\text{items}}$ SID_sales, sR_sales $\prec^{\text{stores}}$ SID_sales, sR_sales $\prec^{\text{stores}}$ sCD_sales, and sR_sales $\prec^{\text{stores}}$ SiC_sales. SID_sales is the top and sR_sales is the bottom of the lattice. ■

The query associated with an edge from v_1 to v_2 is obtained from the original query for v_2 by making the following changes:

- The original `WHERE` clause is removed (it is not needed since the conditions already appear in v_1).

- The FROM clause is replaced by a reference to v_1. Further, if the $\prec$ relation between $v2$ and $v1$ is superscripted with dimension tables, these are joined into $v1$. (The dimension tables will go into the FROM clause, and the join conditions will go into the WHERE clause.)
- The aggregate functions of v_2 need to be rewritten to reference the aggregate function results computed in v_1. In particular,
  - A COUNT aggregate function needs to be changed to a SUM of the counts computed in v_1.
  - If v_1 groups by an attribute A and v_2 computes SUM(A), then SUM(A) will be replaced by SUM($A * Y$), where Y is the result of COUNT(*) in v_1. Similarly COUNT(A) will be replaced by SUM(Y).

5.2 Making summary tables lattice-friendly

It is also possible to change the definitions of summary tables slightly so that the *derives* relation between them grows larger, and we do not repeat joins along the lattice paths. The summary tables are changed by adding joins with dimension tables, adding dimension attributes, and adding aggregation functions used by other summary tables.

Let us consider the case of dimension tables and dimension attributes. Are joins with dimension tables all performed implicitly at the top-most view, or could they be performed lower down just before grouping by dimension attributes? Because the joins between the fact table and the dimension tables are along foreign keys—so that each tuple in the fact table joins with one and only one tuple from each dimension table—either approach, joining implicitly at the top-most view or just before grouping on dimension attributes, is possible.

Now, consider a dimension hierarchy. An attribute in the hierarchy functionally determines all of its descendents in the hierarchy. Therefore, grouping by an attribute in the hierarchy yields the same groups as grouping by that attribute plus all of its descendent attributes. For example, grouping by (storeID) is the same as grouping by (storeID, city, region).

The above two properties provide the rationale for the following approach to fitting summary tables into a lattice: join the fact table with all dimension tables at the top-most point in the lattice. At each point in the lattice, instead of grouping only by the group-by attributes mentioned at that point, we include as well each dimension attribute functionally determined by the group-by attributes. For example, the top-most point in the lattice of Figure 5 groups by (storeID, city, region, itemID, category, date).

The end result of the process can be to fit the generalized views into a regular cube (partially-materialized) lattice where all the joins are taken once at the top-most point, and all the views have the same aggregation functions.

EXAMPLE 5.2 For our running warehousing example, we can define all four summary tables as a groupby over the join of pos, items, and stores, computing COUNT(*), SUM(qty), and MIN(date) in each view, and retaining some or all of the dimension attributes City, Region, and Category. The resulting lattice represents a portion of the complete lattice shown in Figure 5. ∎

5.3 Optimizing the lattice

Although the approach of Section 5.2 is always correct, it does not yield the most efficient result. An important question is where best to do the joins with the dimension tables. Further, assuming that some of the dimension columns and aggregation functions have been added to the views just so that the view fits into the lattice, where should the aggregation functions and the extra columns be computed? Optimizing a lattice means pushing joins, aggregation functions, and dimension columns as low down into the lattice as possible.

There are two reasons for pushing down joins: First, as one travels down the data cube, the number of tuples at each point is likely to decrease, so fewer tuples need to be involved in the join. Second, joining with all dimension tables at the top-most view results in very wide tuples, which require more room in memory and on disk. For example, when computing the data cube in Figure 5, instead of joining the pos table with stores and items to compute the (storeID, itemID, date) view, it may be better to push down the join with stores until the (city, itemID, date) view is computed from the (storeID, itemID, date) view, and to push down the join with items until the (storeID, category, date) view is computed from the (storeID, itemID, date) view.

EXAMPLE 5.3 For the running retail warehousing example, optimization derives the lattice shown in Figure 8. The lattice edges are labeled with the dimension join required when deriving the lower view. For example, the edge from SID_sales to SiC_sales is labeled items to indicate that SID_sales needs to be joined with items to derive the SiC_sales view. The view sCD_sales is extended by adding

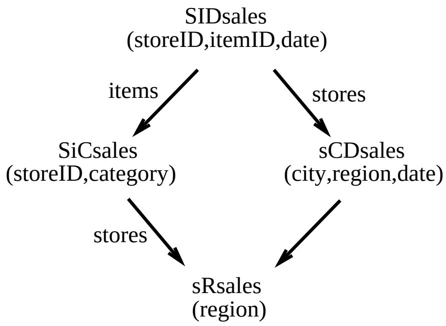


Figure 8: The $\mathcal{V}$-lattice for the retail warehousing example

the region attribute so that the view sR_sales may be derived from it without (re-)joining with the stores table. ∎

5.4 Summary-delta lattice

Following the self-maintenance conditions discussed in Section 3.1, we assume that any view computing an aggregation function is augmented with COUNT($\star$). A view computing SUM(E), MIN(E), and/or MAX(E) is further augmented with COUNT(E).

Given the set of generalized cube views in the partially-materialized $\mathcal{V}$ lattice, we would like to arrange the summary-delta tables for these views into a partially-materialized lattice (the $\mathcal{D}$, or delta, lattice). The hope is that we can then compute the summary-delta tables more efficiently by exploiting the $\mathcal{D}$ lattice structure, just as the views can be computed more efficiently by exploiting the $\mathcal{V}$ lattice structure.

The following theorem follows from the observation that the queries defining the summary-delta tables sd_v (Section 4.1) are similar to the queries defining the views v, except that some of the tables in the `FROM` clause are uniformly replaced by the *prepare-changes* table. The theorem gives us the desired $\mathcal{D}$-lattice. (A proof of the theorem appears in [Qua97].)

Theorem 5.1 *The $\mathcal{D}$-lattice is identical to the $\mathcal{V}$-lattice, including the queries along each edge, modulo a change in the names of tables at each node.* ■

Thus, each summary delta table can be derived from the summary-delta table above it in the partially-materialized lattice, possibly by a join with dimension tables, followed by a simple groupby operation. The queries defining the topmost summary-delta tables in the $\mathcal{D}$-lattice are obtained by defining a *prepare-changes* virtual view (Section 4.1). For example, the summary-delta $\mathcal{D}$-lattice for our warehouse example is the same as the partially-materialized $\mathcal{V}$-lattice of Figure 8.

5.5 Computing the summary-delta lattice

The beauty of our approach is that the summary table maintenance problem has been partitioned into two subproblems — computation of summary-delta tables (propagation), and the application of refresh functions — in such a way that the subproblem of propagation for multiple summary tables can be mapped to the problem of efficiently computing multiple aggregate views in a lattice.

Propagation of changes to multiple summary tables involves computing all the summary-delta tables in the $\mathcal{D}$-lattice derived in Section 5.4. The problem now is how to compute the summary-delta lattice efficiently, since there are possibly several choices for ancestor summary-delta tables from which to compute a summary-delta. It turns out that that this problem maps directly to the problem of computing multiple summary tables from scratch, as addressed in [AAD+96, SAG96]. We can use their solutions to derive an efficient propagate strategy on how to sort/hash inputs, what order to evaluate summary-delta tables, and which of the incoming lattice edges (if there is more than one) to use to evaluate a summary-delta table. The algorithms of [AAD+96, SAG96] would be directly applicable but for the fact that they do not consider join annotations in the lattice. However, it is a simple matter to extend their algorithms by including the join cost estimate in the cost of the derivation of the aggregate view along the edge annotated with the join. We omit the details here as the algorithms for materializing a lattice are not the focus of this paper.

6 Performance

We have implemented the summary-delta algorithm on top of a common PC-based relational database system. We have used the implementation to test the performance improvements obtained by the summary-delta table method over recomputation, and to determine the benefits of using the lattice structure when maintaining multiple summary tables.

The implementation was done in Centura SQL Application Language (SAL) on a Pentium PC. The test database schema is the same as the one used in our running example described in Section 2. We varied the size of the `pos` table from 100,000 tuples to 500,000 tuples, and the size of the changes from 1,000 tuples to 10,000 tuples. The `pos` table had a composite index on (storeID, itemID, date), and each of the summary tables had composite indices on their groupby columns. We found that the performance of the refresh operation depended heavily on the number of updates/deletes vs. inserts to the summary tables. Consequently, we considered two types of changes to the `pos` table:

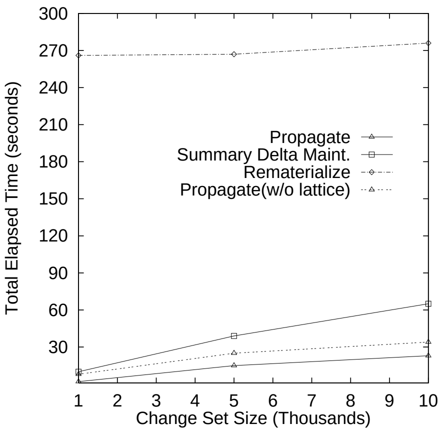
- **Update-Generating Changes:** Insertions and deletions of an equal number of tuples over existing date, store, and item values. These changes mostly cause updates amongst the existing tuples in summary tables.
- **Insertion-Generating Changes:** Insertions over new dates, but existing store and item values. These changes cause only inserts into two of the four summary tables (for whom date is a groupby column), and mostly cause updates into the other two summary-delta tables.

The insertion-generating changes are very meaningful since in many data warehousing applications the only changes to the fact tables are insertions of tuples for new dates, which leads to insertions, but no updates, into summary tables with date as a groupby column.

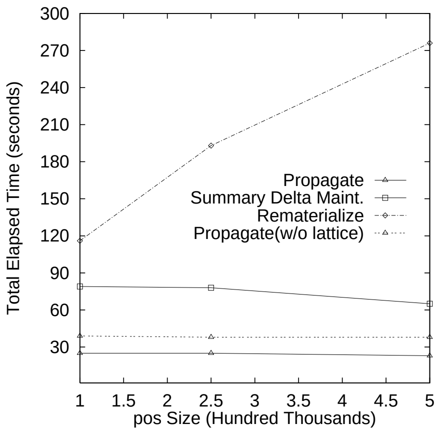
Figure 9 shows four graphs illustrating the performance advantage of using the summary-delta table method. The graphs show the time to rematerialize (using the lattice structure), and maintain all four summary tables using the summary-delta table method (using the lattice structure). The maintenance time is split into propagate and refresh, with the lower solid line representing the portion of the maintenance time taken by propagate when using the lattice structure. The upper solid line represents the total maintenance time (propagate + refresh). The time taken by propagate without using the lattice structure is shown with a dotted line for comparison.

Graphs 9(a) and 9(c) plot the variation in elapsed time as the size of the change set changes, for a fixed size (500,000) of the `pos` table. While 9(a) considers update-generating changes, graph 9(c) considers insertion-generating changes. We note that the incremental maintenance wins for both types of changes, but it wins with a greater margin for the insertion-generating changes. The difference between the two scenarios is mainly in the refresh times for the views SID_sales and sCD_sales; The refresh time going down by 50% in 9(c). The graphs also show that the summary-delta maintenance beats rematerialization, and that propagate benefits by exploiting the lattice structure. Further, the benefit to propagate increases as the size of the change set increases.

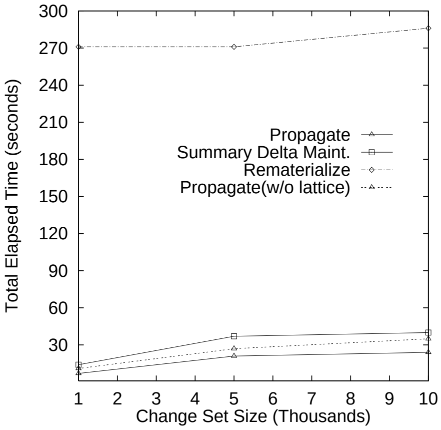
Graphs 9(b) and 9(d) plot the variation in elapsed time as the size of the `pos` table changes, for a fixed size (10,000) of the change set. Graph 9(b) considers update generating changes, and graph 9(d) considers insertion generating changes. We see that the propagate time stays virtually constant with increase in the size of `pos` table (as one would expect, since propagate does not depend on the `pos` table); However interestingly the refresh time goes down for the update generating changes. A close look reveals that when the `pos` table is small, refresh causes a significant number of deletions in addition to updates to the materialized views. When the `pos` table is large, refresh causes only updates to the materialized views, and this leads to a 20% savings in refresh time.



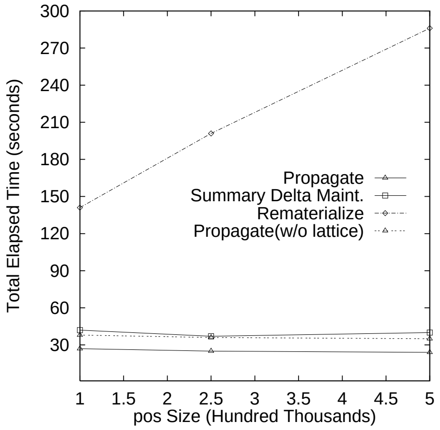
(a) Varying change size for update-generating changes



(b) Varying pos size for update-generating changes



(c) Varying change size for insertion-generating changes



(d) Varying pos size for insertion-generating changes

Figure 9: Performance of Summary-Delta Maintenance algorithm

7 Related Work and Conclusions

Both view maintenance and data warehousing are active areas of research, and this paper is in the intersection of the two areas, proposing new view maintenance techniques for maintaining multiple summary tables (aggregate views) over a star schema using a new *summary-delta* paradigm.

Earlier view maintenance papers [BLT86, CW91, QW91, GMS93, GL95, JMS95, ZGHW95, CGL+96, HZ96, Qua96] have all used the *delta* paradigm - compute a set of inserted and deleted tuples that are then used to refresh the materialized view using simple union and difference operations. The new summary-delta paradigm is to compute a summary-delta table that represents a summary of the changes to be applied to the materialized view. The actual refresh of the materialized view is more complex than a union/difference in the delta paradigm, and can cause updates, insertions, and/or deletions to the materialized view. Amongst the above work on view maintenance algorithms, [GMS93, GL95, JMS95, Qua96] are the only papers that discuss maintenance algorithms for aggregate views. [GMS93, GL95, Qua96] develop algorithms to compute sets of inserted and deleted tuples into an aggregate view, while [JMS95] discusses the computational complexity of immediately maintaining a single aggregate view in response to a single insertion into a *chronicle* (sequence of tuples). It is worth noting that the previous papers do not consider the problem of maintaining multiple aggregate views, and are not as efficient as the summary-delta table method.

A formal split of the maintenance process into propagate and refresh functions was proposed in [CGL+96]. We build on the propagate/refresh idea here, extending it to aggregate views and to more complex refresh functions. Our notion of self-maintainable aggregation functions is an extension of self-maintainability for select-project-join views defined in [GJM96, QGMW96].

[GBLP96] proposed the cube operator linking together related aggregate tables into one SQL query, and started a mini-industry in warehousing research. The notion of cube lattices and dimension lattices was proposed in [HRU96], along with an algorithm to determine a subset of cube views to be materialized so as to maximize the querying benefit under a given space constraint. Algorithms to efficiently materialize all or a subset of the cube lattice have been proposed by [AAD+96, SAG96]. Next, we need a technique to maintain these cube views efficiently, and our paper provides the summary-delta table method to do so. In fact, we even map a part of the maintenance problem into the problem addressed by [AAD+96, SAG96].

Our algorithms are geared towards cube views, as well as towards generalizations of cube views that are likely to occur in typical decision-support systems. We have developed techniques to place aggregate views into a lattice, even suggesting small modifications to the views that can help generate a fuller lattice.

Finally, we have tested the feasibility and the performance gains of the summary-delta table method by implementing it on top of a relational database, and doing a performance study comparing the propagate and refresh times of our algorithm to the alternatives of doing rematerializations or using an alternative maintenance algorithm. We found that our algorithm provides an order of magnitude improvement over the alternatives. Another observation we made from the performance study is that our refresh function, when implemented outside the database system, runs much slower than what we had expected (while still being fast). The right way to implement the refresh function is by doing something similar to a *left outer-join* of the summary-delta table with the materialized view, identifying the view tuples to be updated, and updating them as a part of the outer-join. Such a *"summary-delta join"* operation should be built into the database servers that are targeting the warehousing market.

References

[AAD+96] S. Agarwal, R. Agrawal, P. Deshpande, A. Gupta, J. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In Vijayaraman et al. [TMB96], pages 506–521.

[AL80] M. Adiba and B. Lindsay. Database snapshots. In *Proceedings of the sixth International Conference on Very Large Databases*, pages 86–91, Montreal, Canada, October 1980.

[BC79] P. Buneman and E. Clemons. Efficiently monitoring relational databases. *ACM Transactions on Database Systems*, 4(3):368–382, September 1979.

[BLT86] J. Blakeley, P. Larson, and F. Tompa. Efficiently Updating Materialized Views. In *Proceedings of ACM SIGMOD 1986 International Conference on Management of Data*, pages 61–71, May 1986.

[CGL+96] L. Colby, T. Griffin, L. Libkin, I. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In Jagadish and Mumick [JM96].

[CS94] S. Chaudhuri and K. Shim. Including groupby in query optimization. In *Proceedings of the 20^{th} International Conference on Very Large Databases*, pages 354–366, Chile, September 1994.

[CS95] M. Carey and D. Schneider, editors. *Proceedings of ACM SIGMOD 1995 International Conference on Management of Data*, San Jose, CA, May 23-25 1995.

[CW91] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proceedings of the Seventeenth International Conference on Very Large Databases*, pages 108–119, Spain, September 1991.

[DGN95] U. Dayal, P. Gray, and S. Nishio, editors. *Proceedings of the 21^{st} International Conference on Very Large Databases*, Zurich, Switzerland, September 11-15 1995.

[GBLP96] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *Proceedings of the Twelfth IEEE International Conference on Data Engineering*, pages 152–159, New Orleans, LA, February 26 - March 1 1996.

[GHQ95] A. Gupta, V. Harinarayan, and D. Quass. Generalized projections: A powerful approach to aggregation. In Dayal et al. [DGN95].

[GJM96] A. Gupta, H. Jagadish, and I. Mumick. Data integration using self-maintainable views. In *Proceedings of the Fifth International Conference on Extending Database Technology*, Avignon, France, March 1996.

[GL95] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In Carey and Schneider [CS95].

[GMS93] A. Gupta, I. Mumick, and V. Subrahmanian. Maintaining views incrementally. In *Proceedings of ACM SIGMOD 1993 International Conference on Management of Data*, Washington, DC, May 26-28 1993.

[Han87] E. Hanson. A performance analysis of view materialization strategies. In *Proceedings of ACM SIGMOD 1987 International Conference on Management of Data*, pages 440–453, San Francisco, CA, May 1987.

[HRU96] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing data cubes efficiently. In Jagadish and Mumick [JM96], pages 205–216.

[HZ96] R. Hull and G. Zhou. A framework for supporting data integration using the materialized and virtual approaches. In Jagadish and Mumick [JM96].

[JM96] H. Jagadish and I. Mumick, editors. *Proceedings of ACM SIGMOD 1996 International Conference on Management of Data*, Montreal, Canada, June 1996.

[JMS95] H. Jagadish, I. Mumick, and A. Silberschatz. View maintenance issues in the chronicle data model. In *Proceedings of the Fourteenth Symposium on Principles of Database Systems (PODS)*, San Jose, CA, 1995.

[LMSS95] J. Lu, G. Moerkotte, J. Schu, and V. Subrahmanian. Efficient maintenance of materialized mediated views. In Carey and Schneider [CS95].

[MS93] J. Melton and A. Simon. *Understanding the New SQL: A Complete Guide*. Morgan Kaufmann, 1993.

[QGMW96] D. Quass, A. Gupta, I. Mumick, and J. Widom. Making views self-maintainable for data warehousing. In *Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems (PDIS)*, Miami Beach, FL, December 1996.

[Qua96] D. Quass. Maintenance expressions for views with aggregation. Presented at the Workshop on Materialized Views, June 1996.

[Qua97] D. Quass. *Materialized Views in Data Warehouses*. PhD thesis, Stanford University, Department of Computer Science, 1997.

[QW91] X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. *IEEE Transactions on Knowledge and Data Engineering*, 3(3):337–341, 1991.

[RK86] N. Roussopoulos and H. Kang. Principles and techniques in the design of ADMS$\pm$. *IEEE Computer*, pages 19–25, December 1986.

[SAG96] S. Sarawagi, R. Agrawal, and A. Gupta. On computing the data cube. Research report rj 10026, IBM Almaden Research Center, San Jose, California, 1996.

[SI84] O. Shmueli and A. Itai. Maintenance of Views. In *Proceedings of ACM SIGMOD 1984 International Conference on Management of Data*, pages 240–255, 1984.

[SP89] A. Segev and J. Park. Updating distributed materialized views. *IEEE Transactions on Knowledge and Data Engineering*, 1(2):173–184, June 1989.

[TMB96] T. Vijayaraman, C. Mohan, and A. Buchman, editors. *Proceedings of the 22^{nd} International Conference on Very Large Databases*, Mumbai, India, September 3-6 1996.

[YL95] W. Yan and P. Larson. Eager aggregation and lazy aggregation. In Dayal et al. [DGN95], pages 345–357.

[ZGHW95] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In Carey and Schneider [CS95], pages 316–327.