Modeling and Maintaining Multi-View Data Warehouses

I. Stanoi D. Agrawal A. El Abbadi
Department of Computer Science, University of California
Santa Barbara, CA 93106

Abstract

Data warehouses are designed mostly as centralized systems, and the majority of updating maintenance algorithms are tailored for this specific model. Maintenance methods have been proposed either under the assumption of a single view data warehouse, a multi-view centralized model, or a multi-view distributed system with strict synchronization restrictions. We argue that extending this model to a multi-view distributed one, is a practical generalization of the data warehouse system, but also the base of a growing number of applications based on the idea of cooperative views. In this context, existing maintenance protocols either solve the multi-view synchronization and consistency problem by grouping out of order updates in batches, or by creating multiple versions of the data. In this paper we develop a general framework for modeling the maintenance of multi-views in a distributed, decentralized data warehouse, together with an efficient incremental algorithm. To our knowledge, there is no other proposal for a method that incorporates individually and asynchronously updates to different views that are related through derivation dependencies.

1 Introduction

A data warehouse is an example of a storage component that derives, summarizes and integrates information from multiple sources [4], which may be stand-alone databases as well as sites on the Internet. Although data warehousing is a powerful concept for supporting analytical processing, building a data warehouse runs into several pragmatic problems, such as high cost and the large time frame to make it operational. As a consequence of these practical difficulties, an alternative to a data warehouse is emerging, referred to as a data mart. A data mart is a specialized data warehouse that focuses on a single analytical function such as sales, marketing, or finance. Since the size of the warehousing is scaled down significantly in a data mart, it can be created at a fraction of the cost and time. Multiple data marts may co-exist within an enterprise and they can be integrated to form a virtual data warehouse for the entire organization.

We are interested in analyzing the *distributed* multi-view data warehouse/data marts model, since it can be used to derive information from multiple and diverse data sources that may be geographically dispersed. As the use of views becomes a popular technique for integrating diverse data sources (e.g., data stored in different databases, file-systems, or even Web sites), it can be anticipated that the present centralized model would evolve into a distributed one. We expect that views materialized at different sites may depend not only on base sources, where data is generated, but may also depend on other derived views. This may be due to data security restrictions, requirements on how data flows through an organization, or the sharing of information between companies. The hierarchical arrangement of views also provides the possibility of accessing data at different granularity levels and facilitates operations such as drill-down or roll-up. It is essential to note that the idea of multiple views deriving data from possibly common sources pertains to a much more general model than the data marts example mentioned above. The distributed multi-view data warehouse is emerging as a design solution for a growing set of applications, such as storage of messages from newsgroups [5] or cooperative web caches for the Internet. We therefore believe that it is important for the database community to continue to realize the possibilities offered by exploring the distributed multi-view model.

Some effort in recent literature has been directed towards the efficient definition of views with respect to already materialized views and base sources. The total operational cost of a multi-view data warehouse however, involves both the evaluation and materialization of queries as well as the maintenance of views. Most methods previously proposed for the maintenance of a distributed data warehouse are specifically designed for a single view or independent multiple views [3, 1, 13, 14, 10, 7]. In this paper we develop a general framework for modeling the maintenance of multi-views in a distributed data warehouse together with an efficient incremental algorithm. The key issue is that in a multiple view environment, unlike in the case of the single view model, a view may derive information from the source data via multiple paths through intermediate views. For consistency, the views should reflect the same state of common data sources.

Zhuge et al. [12] have proposed using a synchronization processing unit for ensuring consistency, appropriate for a centralized multi-view data warehouses. A decentralized approach has been proposed in [11], based on calculating the cumulative effect of a batch of updates. The consistency criterion there was relaxed, allowing a view to be less restrictive, and derive data from multiple states of a base source. In contrast, our method pertains to decentralized systems, and allows each view to correctly incorporate updates independently. The model we propose is based on differentiating the data stored in a materialized view into a safe state and subsequent updates. This allows our maintenance algorithm to recreate any necessary state irrespective of the order in which updates are actually delivered or incorporated.

Our solution is novel in its ability to maintain a set of interdependent views, derived from various autonomous base sources. It differs from previous work in its capacity to integrate key advantages in the context of a decentralized, distributed setting: i)an update to a view can be determined solely through communication with the views or data sources from which it directly derives data; ii)no unnecessary delays are imposed due to the grouping of new updates in batches and no centralized coordination is necessary; iii)our method does not require two copies of the data to be stored in the view [10], avoids rematerialization, and queries return the most up to date information available at the view.

In Section 2 we define the multi-view data warehouse model and the appropriate consistency criterion associated with it. Section 3 specifies in more detail, with the use of an example, the complexity of incremental view maintenance in the multi-view data warehouse. We then develop in Section 4 a decentralized incremental algorithm for updating the information stored by views. This algorithm is further extended in to accommodate the grouping of multiple updates generated from the same source. We define in Section 5.1 a cost formulation for the number of messages exchanged during the updating of a hierarchy of views, and we compare different organizations of dependencies for the same set of views and base sources. A discussion of the proposed algorithms concludes this paper.

2 The Multi-View Data Warehouse Model

2.1 The Hierarchical Model

In the multi-view distributed data warehouse model, data sources and derived views may be stored at different sites. We assume that pairwise communication among sites is both reliable and FIFO. Views are allowed to derive information from base sources as well as from other views.

We first model a distributed multi-view data warehouse/data mart using the notion of a View Dependency Graph [3]. This type of graph has nodes for sites hosting views and base sources, and the directed edges between nodes indicate the direction of the flow of update messages. An edge directed towards any view S, connects S to one of its ancestors. Figure 2 is an example of a View Dependency Graph where B_1, B_2 and B_3 are base sources and S_1, S_2 and S_3 are derived views. S_1 and S_2 are *direct ancestors* of S_3, and one of their *common ancestor* is B_2. S_3 is a *direct descendent* of both S_1 and S_2. It is easy to see, with the use of the View Dependency Graph, the paths of propagation of updates from base sources to views and the therefore the implicit update dependencies between different sites. The maintenance of such interdependent views brings out issues that are not present in the single view model. The complexity of a maintenance algorithm arises from the need to correctly propagate updates from base sources to views and *simulate* synchronicity in an asynchronous system. A view should integrate the effects of base source updates as if they are integrated and propagated by its descendents in the same order. Note that this issue does not arise if views are restricted to directly derive data exclusively from base relations.

Colby et al. [3] have proposed a notion of consistency for multiple views and described constraints under which different integration policies can be combined to maintain consistency of multiple views. Although multiple views were considered, all views are assumed to be stored at a single site. Hence, their notion of consistency enforces a temporal order on the updates and ensures that views are updated in this order. We argue that in the more general case, as well as in a multiple view distributed data warehouse where multiple sites are involved, a time-based consistency constraint is too strong since a global notion of time is unrealistic and unrealizable [9]. Instead, an application dependent consistency criterion that exploits the properties of different views may be more appropriate. We therefore propose a consistency criterion, *common base consistency* specifically for distributed multi-view data warehouses.

Consider a base source B that executes a sequence of atomic updates (for simplicity, we assume that a single base relation exists per data source). Such an *update* to a base relation, may actually refer to either a single insert or delete operation, or to a set of changes that make up local atomic transactions. These changes result in a sequence of states where we use the notation $\theta_m(B)$ for the m'th state in this sequence. We now formally define the notion of *Common Base Consistency.*

Definition 1 *Common Base Consistency.* Assume a view V whose state is derived directly from a set of sites $\{S_1, S_2, \ldots S_k\}$ and their states $\{\theta(S_1), \theta(S_2), \ldots \theta(S_k)\}$, where S_i is either a base source or a view. We say that V is common base consistent if for every pair (S_i, S_j) and for every base source B such that $\theta(S_i)$ is derived from state $\theta_m(B)$ and $\theta(S_j)$ is derived from state $\theta_n(B)$, $\theta_m(B)$ must be the same as $\theta_n(B)$.

The above definition simply states that a view V is *common base consistent* if at any time its data reflects at most one state of each base source in the data warehouse. In the example in Figure 2, for common base consistency, any state $\theta(S_3)$ of S_3 can only incorporate states of S_1 and S_2 that reflect the same state of B_2.

Data at the view is integrated from the sources according to a function, which we assume to be commutative, associative, and distributive. In the case of a data warehouse where base sources are relational databases, the deriving functions generally are operations such as join, select, project, which filter a subset of tuples over which the view may further execute local aggregations. In order to simplify our presentation, we will refer to the deriving function as only a *join* operation. While the commutativity property is necessary for the correctness of our algorithm, the distributivity and associativity play an important role in reducing the communication cost of view maintenance.

Since information about the initial individual tuples is lost through select/project or join operations, multiple entries in the base relation can be mapped to the same tuple in the resulting view. As a result, the action of a delete operation on a view can lead to an incorrect state. To avoid such inconsistencies, we assume that a count is associated with each tuple that is part of the materialized view [13], and it refers to the number of ways in which a tuple may be derived. The count should be associated with the tuples as soon as they are derived, and included in any updating messages containing the tuples.

2.2 Query Processing during the Update Phase

One of the main challenges in maintaining a data warehouse is to consistently incorporate into views the effects of changes in the source data. Typically, this process is initiated when a view receives information regarding an update to an ancestor that has incorporated new data. Consider a view V that integrates data from sources $S_1, S_2 \ldots S_n$, which may be data sources or derived views. Without loss of generality, assume that V is defined as the join operations on $S_1 \bowtie S_2 \bowtie \ldots \bowtie S_n$. Each source S_i experiences a possibly empty update due to some change at the base sources (inserts and deletes) ΔS_i. Note that all

ΔS_i $(1 \leq i \leq n)$ included in the update phase of V must be derived from the same update at a data source. A descendent view V, needs to initiate an *update phase* where it communicates with all its direct ancestors through *update queries* to evaluate, for each ΔS_i, the effects on the remaining ancestors. As a result of V's update phase, its new state will become $V + \Delta V$, which is shown in Figure 1.

$$
\begin{array}{rcl}
V + \Delta V & = & (S_1 + \Delta S_1) \bowtie (S_2 + \Delta S_2) \ldots \bowtie (S_n + \Delta S_n) \\
 & = & (S_1 \bowtie S_2 \ldots \bowtie S_i \ldots \bowtie S_n) \\
\mathbf{(1)} & + & \mathbf{\Delta S_1} \bowtie (S_2 + \Delta S_2) \ldots \bowtie (S_i + \Delta S_i) \ldots \bowtie (S_n + \Delta S_n) \\
\mathbf{(2)} & + & S_1 \bowtie \mathbf{\Delta S_2} \bowtie (S_3 + \Delta S_3) \ldots \bowtie (S_i + \Delta S_i) \ldots \bowtie (S_n + \Delta S_n) \\
 & + & \cdots \\
\mathbf{(i)} & + & S_1 \ldots \bowtie S_{i-1} \bowtie \mathbf{\Delta S_i} \bowtie (S_{i+1} + \Delta S_{i+1}) \ldots \bowtie (S_n + \Delta S_n) \\
 & + & \cdots \\
\mathbf{(n)} & + & S_1 \bowtie S_2 \ldots \bowtie S_i \ldots \bowtie S_{n-1} \bowtie \mathbf{\Delta S_n}
\end{array}
$$

Figure 1: Expression for $V + \Delta V$.

In the above equation $S_1 \bowtie S_2 \ldots \bowtie S_i \ldots \bowtie S_n$ is the initial state of V. View V needs to formulate update subqueries which are then summed up to calculate the resulting effect on V, ΔV. For common base consistency, each update phase that V undergoes has to reflect the effects of only one update at the base sources.

If all subqueries start with the term of the form ΔS_i, the computation of the different steps in the subquery involves only join operations based on the *change* at one of the views. The messages are therefore small as compared to subqueries that start by forwarding whole relations. Although we assume that V queries its ancestors in the order described by the equation, the only condition we actually impose is that subqueries start with a term of the form ΔS_i.

If the update of V involves operations such as select and project, they are performed according to the order specified by the query. Notice that there are two cases: either this additional operation is defined over individual relations (i.e. $(\sigma S_i) \bowtie S_j$) or over the result of a join (i.e. $\sigma(S_i \bowtie S_j)$). In the case of a delete, for example, this order is important, as different positioning of the select/project may return inconsistent results. Since the correctness of the computation depends on this order, we enforce it as part of the querying procedure of a view's ancestors. In the first case, the select/project is performed by the ancestor view itself. If a tuple in the answer set is derived from more than one tuple in the view or base relation, the associated count should indicate this. In the second case, the updating view will perform the operation before forwarding the result of the join to other ancestors. In order to simplify the notation, throughout the description of the maintenance algorithm, we focus our attention on the join operation and assume that the select/project operations are performed according to the definition of the updating view.

3 Difficulties in Maintaining a Hierarchy of Views in a Consistent State

The advantages of maintaining views incrementally rather than through recomputation has been recognized in previous work [6, 7]. Several algorithms have been proposed to ensure consistent updates at a view. However, they are mostly tailored for the case of a single view incorporating changes from single or multiple data sources. In the case when views are allowed to derive updates from other views, there is a need for synchronization of updates between a view and its ancestors. Due to the physical distribution of warehouse views/data marts, it is necessary that this synchronization cost be as low as possible. If a view S_3 receives information from views S_1 and S_2 (Figure 2), then the messages received from these ancestors should reflect the same order of updates. Since sites are autonomous, this assumption is very restrictive. Rather than imposing synchronization on the arrival of messages, our approach allows two views S_1 and S_2 to answer the updating queries of descendent S_3's update phase as though they did undergo the same order of updates. Unfortunately, in reality the update orders at S_1 and S_2 may be inconsistent with each other. We next illustrate this problem through an example.

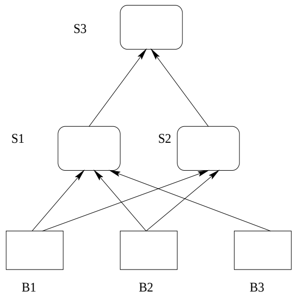

Figure 2: Issues in Maintaining a Hierarchy of Views in a Consistent State

We use the following notation for updates at a view S_i: $\Delta S_i[\Delta B_j]$ refers to changes at view S_i as a consequence of the update ΔB_j and $\Delta S_i[\Delta B_j|\Delta B_1, \ldots, \Delta B_k]$ refers to an update to a view S_i due to the change ΔB_j with the condition that the local updates resulting from $\Delta B_1, \ldots, \Delta B_k$ have already been incorporated. Since the final effect of updates $\Delta B_1, \ldots, \Delta B_k$ is cumulative, their ordering in the notation for $\Delta S_i[\Delta B_j|\Delta B_1, \ldots, \Delta B_k]$ is not important.

Consider, for illustration purposes, a scenario in which update messages are delivered to the different views in Figure 2 in the order specified in Figure 3.

TIME	S_1	S_2	S_3
t_1	ΔB_3	–	–
t_2	ΔB_2	–	–
t_3	ΔB_1	ΔB_1	$\Delta S_2[\Delta B_1]$
t_4	–	ΔB_2	$\Delta S_2[\Delta B_2\|\Delta B_1]$
t_5	–	–	$\Delta S_1[\Delta B_3]$
t_6	–	–	$\Delta S_1[\Delta B_2\|\Delta B_3]$
t_7	–	–	$\Delta S_1[\Delta B_1\|\Delta B_2, \Delta B_3]$

Figure 3: Order of Update Delivery to Views S_1, S_2 and S_3.

Views S_1,S_2 and S_3 will undergo update phases to calculate one by one the effects of the updates on the local data. View S_1 incorporates ΔB_3, followed by ΔB_2 and ΔB_1, and S_2 changes its data to reflect the updates ΔB_1 followed by ΔB_2. The different update phases that views S_1 and S_2 undergo involve queries to their direct ancestors, which result in the following changes at the views (Figure 1):

$$\begin{aligned} \text{initial } S_1 &= B_1 \bowtie B_2 \bowtie B_3 \\ \Delta S_1[\Delta B_3] &= B_1 \bowtie B_2 \bowtie \Delta B_3 \\ \Delta S_1[\Delta B_2|\Delta B_3] &= B_1 \bowtie \Delta B_2 \bowtie (B_3 + \Delta B_3) \\ \Delta S_1[\Delta B_1|\Delta B_2, \Delta B_3] &= \Delta B_1 \bowtie (B_2 + \Delta B_2) \bowtie (B_3 + \Delta B_3) \end{aligned}$$

$$\begin{aligned} \text{initial } S_2 &= B_1 \bowtie B_2 \\ \Delta S_2[\Delta B_1] &= \Delta B_1 \bowtie B_2 \\ \Delta S_2[\Delta B_2|\Delta B_1] &= (B_1 + \Delta B_1) \bowtie \Delta B_2 \end{aligned}$$

Assume that view S_3 decides to calculate first the effect of the change propagated from base source B_1, then from B_2 followed by B_3. First note that to ensure common base consistency, S_3 cannot compute the update $\Delta S_3[\Delta B_1]$ until it has received the corresponding changes reflecting the incorporation of ΔB_1 into S_1 and S_2, i.e., $\Delta S_1[\Delta B_1|\Delta B_2, \Delta B_3]$ and $\Delta S_2[\Delta B_1]$. According to Figure 1, in order to calculate update $\Delta S_3[\Delta B_1]$, S_3 needs to evaluate:

$$\Delta S_3[\Delta B_1] = \Delta S_1[\Delta B_1] \bowtie (S_2 + \Delta S_2[\Delta B_1]) + S_1 \bowtie \Delta S_2[\Delta B_1]$$

where $\Delta S_1[\Delta B_1]$ reflects a change due to ΔB_1 but not ΔB_2 or ΔB_3. *This form of* $\Delta S_1[\Delta B_1]$ *is not the actual update to* S_1, as the update message received from S_1 is $\Delta S_1[\Delta B_1|\Delta B_2, \Delta B_3]$ and hence the update maintenance protocol needs to develop a technique to extract that information with as little overhead as possible. This example indicates that when an update of the form $\Delta S_1[\Delta B_1|\Delta B_2, \Delta B_3]$ arrives at S_3, and S_3 is only interested in incorporating the effects of ΔB_1, enough information must be available for S_3 to extract $\Delta S_1[\Delta B_1]$ from $\Delta S_1[\Delta B_1|\Delta B_2, \Delta B_3]$.

As we will show next, sites also have to be able to compute *states*, not only *changes* to the states. Consider now the update phase that S_3 undergoes in order to integrate the effects of ΔB_2:

$$\begin{aligned}\Delta S_3[\Delta B_2|\Delta B_1] &= \Delta S_1[\Delta B_2|\Delta B_1] \bowtie (S_2 + \Delta S_2[\Delta B_1] + \Delta S_2[\Delta B_2|\Delta B_1]) \\ &+ (S_1 + \Delta S_1[\Delta B_1]) \bowtie \Delta S_2[\Delta B_2|\Delta B_1]\end{aligned}$$

This update phase needs to include an update query to S_1, based on the state, $S_1 + \Delta S_1[\Delta B_1]$, which reflects updates from B_1 but not B_2 and B_3. In this case the *necessary state is not consistent with the sequence of states that exist at* S_1, and therefore cannot be retrieved from the data stored at S_1 (recall that S_1 incorporated update ΔB_1 after updates ΔB_2 and ΔB_3). Any naive attempt at S_1 to compute this state would involve querying indirect ancestors to retrieve information about earlier states, which in the worst case may propagate to the level of base sources. This example illustrates the need for a site to answer a query based on a state which is not part of its sequence of states. In particular, since S_3 dispatches subquery $\Delta S_2[\Delta B_2|\Delta B_1] \bowtie (S_1 + \Delta S_1[\Delta B_1])$, to S_1, S_1 should have enough information available either locally or through its ancestors so that it can extract $S_1 + \Delta S_1[\Delta B_1]$ from $S_1 + \Delta S_1[\Delta B_3] + \Delta S_1[\Delta B2|\Delta B_3] + \Delta S_1[\Delta B_1|\Delta B_2, \Delta B_3]$.

4 Incremental Maintenance of Multiple Views

In this section we propose an incremental algorithm for maintaining views in a multi-view setting. We first define the storage requirements, then give a detailed explanation of the updating method together with an illustrative example, and finally we discuss some of the properties of the algorithm.

4.1 Storage and Control Structures

Updates are incorporated into a view in an incremental and permanent manner, that is, an update is never undone, and two consecutive states differ from one another by exactly one update to a base relation. Each view maintains a materialized state that is a snapshot of a *safe* state in the view's history such that the view will not need to answer update queries based on a prior state. This snapshot is stored as a relational table that is the actual materialized view. For any subsequent states we need to store additional information, so that we can simulate subtraction of update effects as requested by any of the view's descendents. Hence we require that a view implement a separate storage, the *table of changes*, that includes all the updates following the safe state of the actual materialized view. Although a view is able to calculate older states of its data for user queries, it will also have access to the most current state based on its actual materialized data as well as the table of changes. The table of changes stores updates in the order they are incorporated at a view, and delete operations are noted as negative updates. Since this order may not be the same as that requested by descendents trying to update their views, a view answering update queries needs more information to be able to locally exclude the effects of unwanted out-of-order updates. We therefore append a dependency list (referred to as *dep*) to all entries in the table of changes. Each entry is a term returned as answer for an update query, and it reflects one or more changes to different base sources. The dependency list is a set of *identifiers* ΔB_i which indicate that the entry is dependent on changes to the corresponding B_i.

Consider again the example presented in the previous section. The following are the materialized state and table of changes of S_1 and S_2 following their update phases. We also include the dependency lists that correspond to terms in the table of changes.

safe state of S_1: $(B_1 \bowtie B_2 \bowtie B_3)$
table of changes:
$\Delta S_1[\Delta B_3] = (\Delta B_3 \bowtie B_2 \bowtie B_1)dep_{\Delta B_3}$
$\Delta S_1[\Delta B_2|\Delta B_3] = (\Delta B_2 \bowtie B_1 \bowtie B_3)dep_{\Delta B_2} + (\Delta B_2 \bowtie B_1 \bowtie \Delta B_3)dep_{\Delta B_2,\Delta B_3}$
$\Delta S_1[\Delta B_1|\Delta B_2, \Delta B_3] = (\Delta B_1 \bowtie B_2 \bowtie B_3)dep_{\Delta B_1} + (\Delta B_1 \bowtie \Delta B_2 \bowtie B_3)dep_{\Delta B_1,\Delta B_2} + (\Delta B_1 \bowtie B_2 \bowtie \Delta B_3)dep_{\Delta B_1,\Delta B_3} + (\Delta B_1 \bowtie \Delta B_2 \bowtie \Delta B_3)dep_{\Delta B_1,\Delta B_2,\Delta B_3}$

safe state of S_2: $(B_1 \bowtie B_2)$
table of changes:
$\Delta S_2[\Delta B_1] = (\Delta B_1 \bowtie B_2)dep_{\Delta B_1}$
$\Delta S_2[\Delta B_2|\Delta B_1] = (\Delta B_2 \bowtie B_1)dep_{\Delta B_2} + (\Delta B_2 \bowtie \Delta B_1)dep_{\Delta B_1,\Delta B_2}$

To show the use of both the storage separation and the dependency lists, we describe below an instance of the illustrative example, according to the above update order at each site. The first table, Figure 4 includes both the initial states and the insert updates at base relations B_1 with attributes (A, B), B_2 with attributes (B, C) and B_3 with attributes (C, D). The second table, Figure 5 contains the $safe$ state as well as the table of changes of S_1 and S_2. Assume relation S_1 has attributes (A, B, C, D), and the attributes of S_2 are (A, B, C). Dependency lists are associated with the different entries in the table of changes for each view.

B_1	B_2	B_3
(A,B)	**(B,C)**	**(C,D)**
(1,2) –	(2,4) –	(4,1) (3,4)
ΔB_1 (2,2)	ΔB_2 (2,3)	ΔB_3 (3,1)

Figure 4: Updates to Base Sources B_1, B_2 and B_3.

S_1 (A,B,C,D)		
safe state (1,2,4,1)		
table of changes –	**dep** –	update $\Delta S_1[\Delta B_3]$
(1,2,3,4) (1,2,3,1)	$\{\Delta B_2\}$ $\{\Delta B_2, \Delta B_3\}$	$\Delta S_1[\Delta B_2\|\Delta B_3]$
(2,2,4,1) (2,2,3,4) (2,2,3,1)	$\{\Delta B_1\}$ $\{\Delta B_1, \Delta B_2\}$ $\{\Delta B_1, \Delta B_2, \Delta B_3\}$	$\Delta S_1[\Delta B_1\|\Delta B_2, \Delta B_3]$

S_2 (A,B,C)		
safe state (1,2,4)		
table of changes (2,2,4)	**dep** $\{\Delta B_1\}$	update $\Delta S_2[\Delta B_1]$
(1,2,3) (2,2,3)	$\{\Delta B_2\}$ $\{\Delta B_1, \Delta B_2\}$	$\Delta S_2[\Delta B_2\|\Delta B_1]$

Figure 5: Updates to Views S_1 and S_2.

Following the update $\Delta S_1[\Delta B_3]$, there are no new tuples to be stored at the view S_1. The first tuples inserted in the table of changes of S_1 are $(1, 2, 3, 4)$ and $(1, 2, 3, 1)$, due to $\Delta S_1[\Delta B_2|\Delta B_3]$. The remaining tuples in S_1's table of changes are a result of the last update, $\Delta S_1[\Delta B_1|\Delta B_2, \Delta B_3]$. In the table of changes of S_2, the tuple $(2, 2, 4)$ is generated by the computation of $\Delta S_2[\Delta B_1]$, while tuples $(1, 2, 3)$ and $(2, 2, 3)$ are due to $\Delta S_2[\Delta B_2|\Delta B_1]$.

The incremental view update model we present is based on the observation that *for a view, the effects of incorporating an update ΔB_i over the entries affected by ΔB_j are the same tuples as those reflecting the change ΔB_j over the entries affected by ΔB_i*:

Theorem 1 Let a view V undergo updates due to changes at base sources $\Delta B_1, \cdots, \Delta B_i, \cdots, \Delta B_n$. The exclusion of the updates whose dependency lists include ΔB_i leads to a correct state of V based on updates $\Delta B_1, \cdots, \Delta B_{i-1}, \Delta B_{i+1} \cdots, \Delta B_n$.

The proof of Theorem 1 is included as an Appendix.

All entries that include ΔB_i reflect all and exactly the effects of having incorporated the update ΔB_i, as the latest change. Hence by excluding the entries that include ΔB_i in their attached dependency lists, we **undo** the effects of the update ΔB_i. Referring back to our example, if we want to compute the state of S_1 as if its local data has been affected by only one update ΔB_1, we have to ignore the effects of ΔB_2 and ΔB_3 from the table of changes. To undo the change due to ΔB_2, we decide on the set of entries that represent all the effects of ΔB_2 and therefore include ΔB_2 in their dependency lists, i.e., {(1,2,3,4), (1,2,3,1),(2,2,3,4) ,(2,2,3,1)}. Since this set already includes all the effects of ΔB_3, the only entry that is valid for $\Delta S_1[\Delta B_1]$ is {(2,2,4,1)}.

4.2 Algorithm: Overview

We now present the algorithm for the incremental maintenance of views. Figure 15 in the Appendix B illustrates the code for answering update queries, while Figure 16 outlines the steps of a view's update phase.

MultiView Data Warehouse Maintenance:

1) Begin Update Phase. The update phase is triggered by the receipt of a set of changes to the ancestors and their attached dependency lists. Each view maintains these updates in an *update message queue*. Once a view V has received all messages from its ancestors S_i regarding a particular update ΔB, i.e., $\Delta S_i[\Delta B| \ldots]$, it initiates the update phase. Since update messages are not synchronized, the update ΔS_i chosen may not be the first in the message queue from S_i. Views that derive data directly from a base source B, can correctly create an identifier for the update ΔB from source B. Since communication is reliable and FIFO between a pair of sites, the views can create this identifier based on the sender and the order in which updates are received from the sender. Views that derive information from other views, receive the identifier in the dependency list attached to the update message.

2) Determine the set of ΔS_i. Once the set of ΔS_i is selected, the updating view must determine the actual form of each ΔS_i, as required by the update phase. A deriving view stores enough information in its message queue to correctly eliminate the effects of unwanted updates ΔB_k from a message $\Delta S_i[\Delta B| \ldots, \Delta B_k, \ldots]$. This is done by eliminating from $\Delta S_i[\Delta B| \ldots, \Delta B_k, \ldots]$ all terms whose dependency lists include the identifier of the updates ΔB_k.

In our example, at the time when S_3 enters the update phase to integrate the effects of the change ΔB_1 at the base source, the message $\Delta S_1[\Delta B_1|\Delta B_2, \Delta B_3]$ includes also changes to B_2 and B_3 which should not affect the current update phase of S_3. The term $\Delta S_1[\Delta B_1] = \Delta B_1 \bowtie B_2 \bowtie B_3$ is obtained by eliminating from $\Delta S_1[\Delta B_1|\Delta B_2, \Delta B_3]$ all entries whose dependency lists include ΔB_2 and ΔB_3, i.e., the terms $(\Delta B_1 \bowtie \Delta B_2 \bowtie B_3)dep_{\Delta B_1, \Delta B_2}$, $(\Delta B_1 \bowtie B_2 \bowtie \Delta B_3)dep_{\Delta B_1, \Delta B_3}$, and $(\Delta B_1 \bowtie \Delta B_2 \bowtie \Delta B_3)dep_{\Delta B_1, \Delta B_2, \Delta B_3}$.

3) Dispatch Queries As a view V enters the updating phase in order to incorporate the effects of an update ΔB, it dispatches update queries to its direct ancestors. A query to an ancestor S_j includes the partial result of previously queried ancestors (within the same update query ΔV_i of computing the subquery starting with $\Delta S_i[\Delta B]$), and a list of delayed updates that V received before the effects of ΔB from S_j, but have not been integrated yet at V. If any of the queried ancestors S_j is a base relation, then the updating view receives an answer based on the current version of S_j, and locally subtracts effects of unnecessary updates. The technique used is similar to the one described in [1]. Otherwise, if the queried ancestor is a view, it is expected to locally calculate the correct answer to the update query (see **QueryServer**), and no additional compensation is needed by the querying view.

4) Refresh the Materialized View A materialized view is refreshed to incorporate an update from the table of changes only when the new state is considered safe, i.e., when the respective updates are in sequence, and they have been integrated by all the corresponding direct descendents. When updates are included in the actual materialized view, delete operations can be safely incorporated by deleting the appropriate tuples in the relation. As dependency lists are no longer necessary to describe tuples in a $safe$ state, they will not be included in the actual materialized views.

Hence for example $\Delta S_1[\Delta B_1|\Delta B_2, \Delta B_3]$ can be incorporated into the materialized view only after both $\Delta S_3[\Delta B_3]$ and $\Delta S_3[\Delta B_2|\Delta B_3)$ are incorporated into S_3. In contrast, view S_2 can refresh from its table of changes sooner, by incorporating $\Delta S_2[\Delta B_1]$ immediately after S_3 integrates the update $\Delta S_3[\Delta B_1]$

End MultiView Data Warehouse Maintenance

Now we consider that a view S_j that is queried for a partial result in the update phase of its descendent V. Assume the current update phase computes the effects due to the changes to a base source B. A view S_j may or may not be asked to include in this set the effects of ΔB, depending on whether S_j is to the right or to the left of S_i. This information is passed to S_j together with the query request message and the set of updates to be excluded from the computation, $IgnoreSet$.

QueryServer:

1) Compute the Partial Query Answer View S_j performs the join over its materialized view and the table of changes from which it excludes the terms containing in their dependency lists updates that have not been incorporated yet by the descendent. The entries to be ignored are selected based on the information in the IgnoreSet.

In our example, this case is illustrated by the computation of $\Delta S_3[\Delta B_2]$. View S_1 can compute the state $S_1 + \Delta S_1[\Delta B_1]$ required during S_3's updating phase by eliminating from the table of changes all tuples that include identifiers of ΔB_2 and ΔB_3. The state will consist of the data stored by the actual materialized view, and the term $(\Delta B_1 \bowtie B_2 \bowtie B_3) dep_{\Delta B_1}$ from the table of changes.

2) Send the Answer to the Updating View Once a site has computed the correct response to an update query, it appends the appropriate dependency lists and sends the answer to the querying view. Following the termination of the update phase, the updating view V will include the corresponding identifier ΔB in the dependency lists of all new entries calculated during the update phase. After a site incorporates an update, it asynchronously sends this change and the corresponding dependency lists to its descendents.

END QueryServer

In a distributed system, messages from different ancestors can arrive at a view out of order. As the number of messages involved in the updating phase of V does not change, the only difference is in the local computation of the ancestors in answering the update query. In our algorithm, the local elimination of unwanted updates is easily executed during the computation of the deriving function, by ignoring the respective entries in the table of changes. The only additional effort is in scanning the dependency lists associated with the different entries, which is independent of the number of tuples to be ignored. Therefore the local computational effort in answering queries does not depend on the order of updates to be integrated by the querying view. Hence, the reordering of updates is achieved at no additional computational and communication cost and some storage overhead (i.e, storage necessary to maintain dependency information).

4.3 Discussion

The number of messages dispatched during the update phase by a view V with n direct ancestors is $O(n^2)$. That is, in the worst case an update to a base source is propagated to all of V's direct ancestors, and the effects on each of these ancestors must be calculated with respect to all other direct ancestors. The number of messages can be limited to $O(n)$, with the tradeoff of their increasing size. Depending on the definition of the view, the reduction in the number of messages might be advantageous in some systems. Consider again the equation used for calculating the update ΔV of a view with ancestors $S_1 \dots S_n$. The subqueries $\{(1),(2)\dots,(n)\}$ (Figure 1) can be processed concurrently, through the dispatch of only two queries to each of the ancestors. The query processing is divided into two steps. First, each site S_i will receive a set of subquery results from S_{i-1}, calculate the effects on the local data, $S_i + \Delta S_i$, and forward the results of the computations to the next site, S_{i+1}. Site S_i also forwards through V its ΔS_i to S_{i+1}. (S_1 will only send ΔS_1 to S_2.) At the end of this first step, only one of the subqueries if Figure 1, (1), is complete, and sent to the updating view V. In the second phase, a view S_i forwards all results of the subqueries received from S_{i+1} to S_{i-1}, with the exception of the subqueries whose computation had started at S_i or later during the first phase. These subqueries are joint with the data of S_i, which excludes the new update ΔS_i. As soon as the result of a subquery is fully computed, it is sent to the updating view V. In the following, the emphasized part of the query represents the computation during the second phase.

Since the updating algorithm is not dependent on the order of query, it can be easily adapted to the two-phase querying method. A recent paper ([8]) proposed an ordering of the subqueries $\{(1), \cdots, (n)\}$, that would optimize the cost of their computation. However, the results described do not hold in the case of a view deriving data from two or more sites that have a common ancestor. Since the purpose of this paper is to introduce a maintenance algorithm for a general hierarchy of views, we do not enforce this reordering of the subqueries.

Further optimizations can improve the update queries, by choosing an optimal order over the different terms of a subquery [2]. Since we are analyzing a general hierarchical model, we cannot make further assumptions on the query optimizations. The challenge we are addressing in this paper is to perform these join operations with the appropriate state, i.e. one that reflects the correct set of updates according to the definition of common base consistency. Although we consider that the correct (to a determined state) updating of distributed views in a decentralized setting is an important and challenging problem, this issue has not been addressed until now.

The number of messages exchanged depends on the structure of the updating query, as well as on the clustering of updates. If updates to base sources are frequent, then the single-update incremental maintenance algorithm may be inefficient as compared to an algorithm that groups updates in batches. The maintenance algorithm we proposed in Section 4.2 is flexible enough to easily integrate the grouping of updates so that, for an updating view V, each ΔV computes the effects of a batch of updates to a base source B. Previous to dispatching the update query, V groups the incoming update messages from its ancestors S_i in separate lists of $\sum \Delta S_i(\Delta B)$, for changes propagated from base source B. Then for any S_i and S_j ancestors of V, if both S_i, S_j derive from a source B, then a change ΔB must be either in both ($\sum \Delta S_i(\Delta B)$ and $\sum \Delta S_j(\Delta B)$) or in neither of them. The general order of query is the same as discussed in the single-update case,the only difference being in the replacement of ΔS_i with $\sum \Delta S_i$.

5 Message Cost Analysis

In this section we present a message cost formulation specific to the updating of a hierarchy of distributed views. Based on this cost formulation, we compare different hierarchical structures of the same set of views and base sources.

5.1 Cost Formulation

In general, we can express all dependencies within a hierarchy of views and base sources with the help of a simple binary matrix. There are two types of dependencies that play an important role in the number of messages exchanged during a view's update phase: dependencies on direct ancestors, and dependencies (direct or indirect) on base sources. Although the second type of information can be inferred from the first, we explicitly separate the two. A row i in the dependency matrix summarizes the dependencies of a view indexed S_i. The columns in the matrix describing dependencies on base sources, matrix A, correspond to each of the sources. The second matrix, B, has [number of base sources + number of views]x columns, for all the possible direct dependencies. For our example, (Figure 2), the matrix is the following:

matrix A			
	B_1	B_2	B_3
S_1	**1**	**1**	**1**
S_2	**1**	**1**	**0**
S_3	**1**	**1**	**1**

matrix B						
B_1	B_2	B_3		S_1	S_2	S_3
S_1	**1**	**1**	**1**	**0**	**0**	**0**
S_2	**1**	**1**	**0**	**0**	**0**	**0**
S_3	**0**	**0**	**0**	**1**	**1**	**0**

Let $NrUpdates_{B_i}$ be the number of batches or single updates at B_i, $NrViews$ and $NrSources$ be the number of views in the hierarchy and, respectively, the number of base sources.

If **U(i)** = number of queries dispatched by view S_i during its update phase, corresponding to the subqueries { (1),(2), $\cdots$, (n)} in Figure 1, Section 2.2. then U(i) is the sum of all possible dependencies on a base source, times the number of updates to the corresponding base source.

$\bullet U(i) = \sum_{j=1}^{NrSources}(NrUpdates_{B_j} * (\sum_{k=1}^{NrViews}(B[i,(k + NrSources)] \bigcap A[k,j]) + B[i,j]))$

For a view S_k, $(B[i,(k + NrSources)] \bigcap A[k,j]) = 1$ if view S_i derives data directly from S_k and the update B_j is one of S_k's ancestors. Considering again our example, the number of queries computed during the update phases of views S_1, S_2 and S_3 are:

U(1) = $NrUpdates_{B_1} + NrUpdates_{B_2} + NrUpdates_{B_3}$
U(2) = $NrUpdates_{B_1} + NrUpdates_{B_2}$
U(3) = $NrUpdates_{B_1} * 2 + NrUpdates_{B_2} * 2 + NrUpdates_{B_3} * 1$

The number of queries differs from the number of messages exchanged, as a query that calculates the effects of a change at one of the ancestors involving messages to all the other direct ancestors. That is, a query involves (n-1) messages, where n is the number of direct ancestors. Let M(i) be the number of queries dispatched to compute all the update queries in a view S_i's update phase. Then M(i) is tightly related to U(i) and the number of direct ancestors of S_i, $NrAn_{S_i}$. The value of $NrAn_{S_i}$ is easily calculated as the sum of the entries (which store either a "1" or a "0") in matrix B, row i.

$\bullet M(i) = (NrAn_{S_i} - 1) * U(i)$

$\bullet M(i) = (NrAn_{S_i} - 1) * \sum_{j=1}^{NrSources} (NrUpdates_{B_j} * (\sum_{k=1}^{NrViews} (B[i,(k+NrSources] \bigcap A[k,j]) + B[i,j]))$

M(1) = $NrUpdates_{B_1} * 2 + NrUpdates_{B_2} * 2 + NrUpdates_{B_3} * 2$
M(2) = $NrUpdates_{B_1} * 1 + NrUpdates_{B_2} * 1$
M(3) = $NrUpdates_{B_1} * 2 * 1 + NrUpdates_{B_2} * 2 * 1 + NrUpdates_{B_3} * 1 * 1$

Another measure for the number of messages exchanged during the updating of the views in the hierarchy, is the *responsibility*, R, of each view or base source. For a site S_i, R(i) is the number of subqueries dispatched by a descendent of S_i, in which S_i is asked to perform some computation.

$$\bullet R(i) = \begin{cases} \sum_{j=1}^{NrViews}(B[j, i + NrSources] * U(j) - \sum_{k=1}^{NrSources}(A[i,k] * NrUpdates_{B_k}))\text{if } S_i \text{ is view} \\ \\ \sum_{j=1}^{NrViews}(B[j,i] * U(j) - NrUpdates_{B_i})\text{if i is a base source} \end{cases}$$

R(B1) = U(1) - $NrUpdates_{B_1}$ + U(2) - $NrUpdates_{B_1}$
R(B2) = U(1) - $NrUpdates_{B_1}$ + U(2) - $NrUpdates_{B_1}$
R(B3) = U(1) - $NrUpdates_{B_1}$

R(1) = U(3) - ($NrUpdates_{B_1} + NrUpdates_{B_2} + NrUpdates_{B_3}$)
R(2) = U(3) - ($NrUpdates_{B_1} + NrUpdates_{B_2}$)
R(3) = 0

Note that $\sum_{i=1}^{NrViews+NrSources}$ U(i), $\sum_{i=1}^{NrViews+NrSources}$ M(i) and $\sum_{i=1}^{NrViews+NrSources}$ R(i) represent the different cost evaluations for the whole hierarchy of views and base sources, and $\sum_{i=1}^{NrViews+NrSources}$ M(i) = $\sum_{i=1}^{NrViews+NrSources}$ R(i).

5.2 Tall or Flattened Hierarchy?

Based on the cost model proposed, we analyze the behavior of a given set of views and base sources in a spectrum of hierarchical structures. Figure 6 illustrates one end of the spectrum, in which a view is placed in the highest level of the hierarchy while still able to derive the necessary information. At the other end of the spectrum is the flat arrangement of the views, where all views derive their data directly from base sources (Figure 8). An intermediary structure is shown in Figure 7.

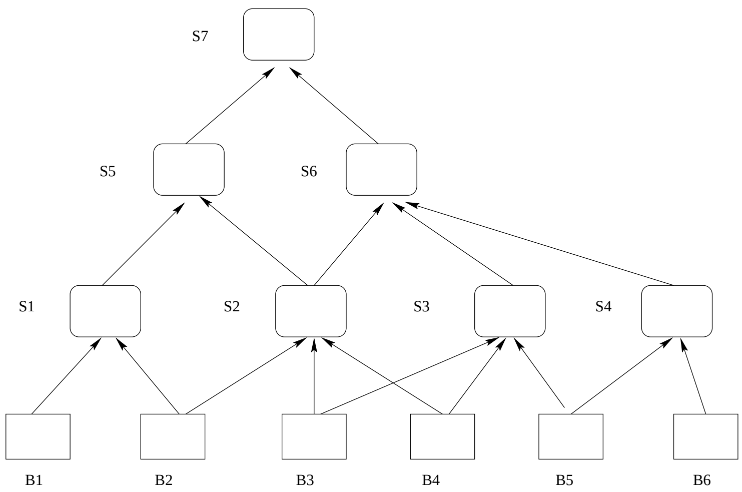

Figure 6: Three Level Hierarchy (tall)

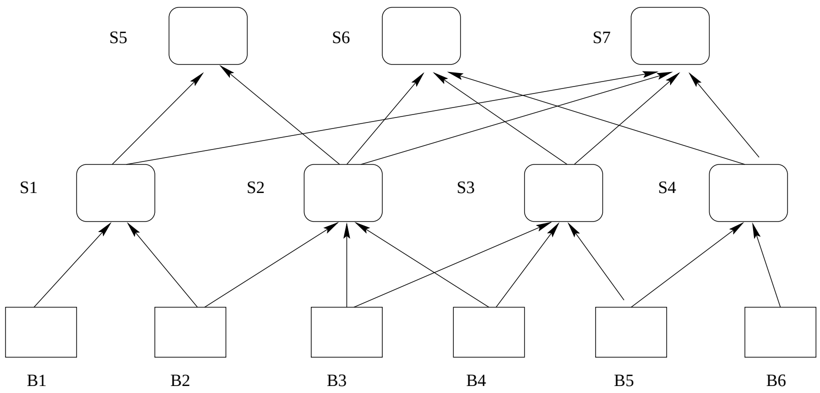

Figure 7: Two Level Hierarchy (medium)

The views in the three different hierarchies are similarly defined over the base sources. However, the total responsibility of the system, $\sum_{i=1}^{NrViews+NrSources}$R(i), and therefore the total number of messages exchanged, $\sum_{i=1}^{NrViews+NrSources}$M(i), differs from one case to another. Figure 9(a) shows that, for equal frequencies of updates at the sources, the total number of messages exchanged during the updating of the views increases when the height of the hierarchy is reduced. If only one update is performed at each base source, then the total responsibility on the system for the three cases is: 46, 67 and 78 messages for the tall, middle and flat hierarchy. These numbers increase linearly with the number of individual updates (or batches) to the sources if the distribution of updates over the base sources remains the same.

The base sources in the cases shown in Figure 6 and 7 have the same set of descendents. Therefore, the responsibility load on the sources remains the same. However, as seen in Figure 9(b) by comparing the flat and the tall hierarchies of the same structure, if the number of views directly dependent on base sources

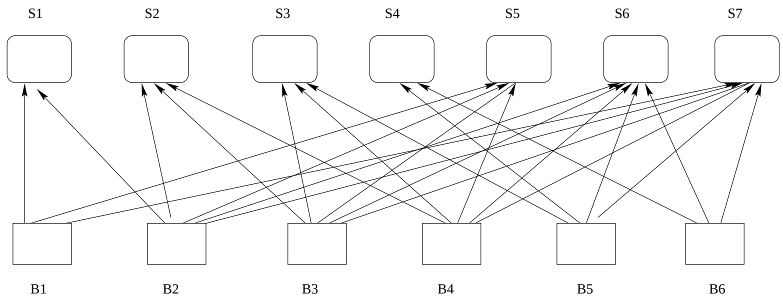

Figure 8: One Level Hierarchy(short)

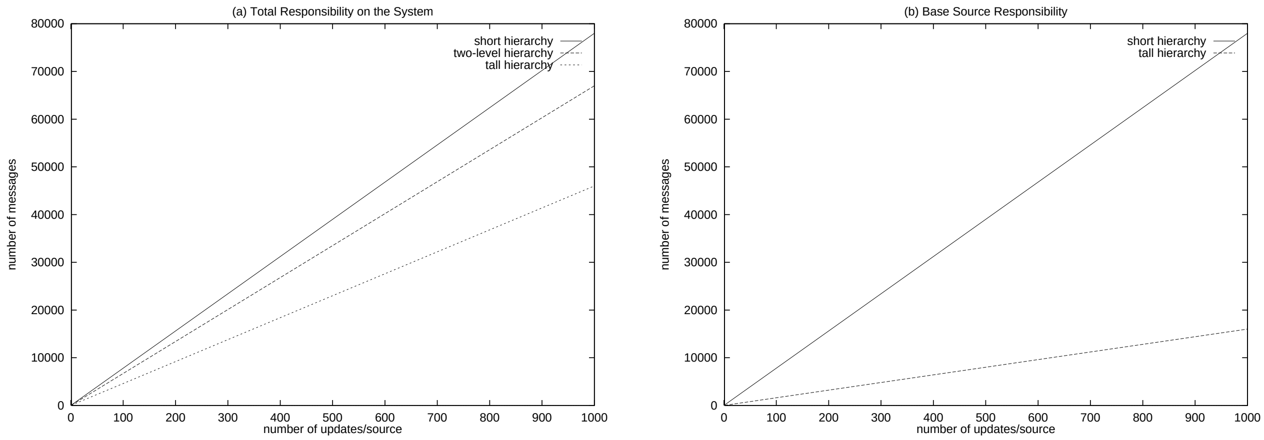

Figure 9: Comparison of message cost between the tall, two-level and short hierarchy

increases, then the total responsibility $\sum_{i=1}^{NrSources}$R(i) of the base sources increases significantly. The fact that tall hierarchies impose a reduced communication and execution load on the sources and in general on the system, supports the idea that, when possible, it is advantageous to allow derivation dependencies between different views.

6 Conclusion

Several maintenance algorithms have been proposed for the data warehouse model. They are specific to one view, or a set of views that have no interdependencies. For multi-view warehouses, previously proposed methods either include a synchronization module, or required a notion of synchronized global time. Both of these requirements are too restrictive in a distributed setting, which is the primary application area of multi-view warehouses, i.e., data marts. In this paper, we first developed a notion of consistency for such a setting of distributed multi-view data warehouse. We then developed an incremental algorithm for asynchronously incorporating updates into views. Our approach requires the maintenance of a stable state separate from a table of changes. This table of changes gives sites the flexibility of incorporating updates in different orders, and at the same time allow them to respond to queries requiring alternate orders of execution, without significant computational and storage overhead (simply checking the dependency lists). Moreover, local computation of ancestors in answering a view's updating query does not increase if reordering of messages at the view becomes necessary. This decentralized approach then allows sites to make independent decisions on when to incorporate an update, unlike a synchronization module based approach [12] where the order of updates to all views must be the same, thus imposing unnecessary delays when incorporating new updates. We further extend the single-update maintenance algorithm to an algorithm based on computing changes in batches, more appropriate for systems that support a high frequency of updates to the base sources. After developing a cost model for the update messages through the hierarchy, we argue that, if possible, views should be defined to derive data from other views rather than base sources, and reduce the communication and execution load on the sources.

References

[1] D. Agrawal, A. El Abbadi, A. Singh, and T. Yurek. Efficient View Maintenance in Data Warehouses. In *Proceedings of the 1997 ACM International Conference on Management of Data*, pages 417–427, May 1997.

[2] Surajit Chaudhuri. An Overview of Query Optimization in Relational Systems. In *Proceedings of the ACM PODS*, pages 34–43, June 1998.

[3] Latha S. Colby, Akira Kawaguchi, Daniel F. Lieuwen, and Inderpal Singh Mumick. Supporting Multiple View Maintenance Policies. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 405–416, May 1997.

[4] A. Gupta and I. S. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Bulletin of the Technical Committee on Data Engineering*, 18(2):3–18, June 1995.

[5] H. Gupta and D. Srivastava. The Data Warehouse of Newsgroups. In *Proceedings of the Seventh International Conference on Database Theory (ICDT)*, pages 471–488, 1999.

[6] E. N. Hanson. A Performance Analysis of View Materialization Strategies. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 440–453, May 1987.

[7] N. Huyn. Multiple-View Self-Maintenance in Data Warehousing Environments. In *Proceedings of the International Conference on Very Large Data Bases*, 1997.

[8] W. Labio, R. Yerneni, and H. Garcia-Molina. Shrinking the Warehouse Update Window. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, May 1999.

[9] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[10] D. Quass and J. Widom. On-line Warehouse View Maintenance. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, May 1997.

[11] I. Stanoi, D. Agrawal, and A. El Abbadi. Weak Consistency in Distributed Data Warehouses. In *Proceedings of the International Conference of Foundations of Data Organization*, November 1998.

[12] Y. Zhuge, J. L. Wiener, and H. Garcia-Molina. Multiple View Consistency for Data Warehousing. In *Proceedings of the 1997 IEEE International Conference on Data Engineering*, 1997.

[13] Yue Zhuge, Hector Garcia-Molina, Joachim Hammer, and Jennifer Widom. View Maintenance in a Warehousing Environment. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 316–327, May 1995.

[14] Yue Zhuge, Hector Garcia-Molina, and Janet L. Wiener. The Strobe Algorithms for Multi-Source Warehouse Consistency. In *Proceedings of the International Conference on Parallel and Distributed Information Systems*, December 1996.

A Proof

Proof:

Without loss of generalization, we assume that the index of an update ΔB_i represents both the delivery order of updates to V and the ordering of base sources.

We can formally express Theorem 1 as :
$\Delta V[\Delta B_1] + \Delta V[\Delta B_2|\Delta B_1] + \Delta V[\Delta B_3|\Delta B_1, \Delta B_2] + \cdots +$
$\Delta V[\Delta B_i|\Delta B_1, \cdots, \Delta B_{i-1}] + \cdots + \Delta V[\Delta B_n|\Delta B_1, \cdots, \Delta B_i, \cdots, \Delta B_{n-1}] - \sum \Delta V dep_{\ldots,\Delta B_i,\cdots} =$
$\Delta V[\Delta B_1] + \Delta V[\Delta B_2|\Delta B_1] + \Delta V[\Delta B_3|\Delta B_1, \Delta B_2] + \cdots + \Delta V[\Delta B_{i-1}|\Delta B_1, \cdots, \Delta B_{i-2}] +$
$\Delta V[\Delta B_{i+1}|\Delta B_1, \cdots, \Delta B_{i-1}] + \cdots + \Delta V[\Delta B_n|\Delta B_1, \cdots, \Delta B_{i-1}, \Delta B_{i+1}, \cdots, \Delta B_{n-1}]$, where $\sum \Delta V dep_{\ldots,\Delta B_i,\cdots}$ are all tuples tagged with a dependency list that includes ΔB_i.

The proof consists of two steps. First we show the format of all the entries resulting from the sets of updates, and select the entries (marked in boldface) containing ΔB_i in their dependency lists. The selected set can be written as $\sum \Delta V dep_{\ldots,\Delta B_i,\ldots}$. In the second part of the proof we show that $\sum \Delta V dep_{\ldots,\Delta B_i,\ldots}$ is a set equal to the set $\Delta V[\Delta B_i|\Delta B_1, \cdots, \Delta B_{i-1}, \Delta B_{i+1}, \cdots, \Delta B_n]$, i.e., the tuple resulting from incorporating the change due to ΔB_i $after$ all changes resulting from $\Delta B_1, \cdots, \Delta B_{i-1}, \Delta B_{i+1}, \cdots, \Delta B_n$ have been included into V

Since V undergoes updates due to the propagation of $\Delta B_1, \cdots, \Delta B_i, \cdots, \Delta B_n$, its table of changes will store the entries enumerated in the three tables shown in (Figure 10, 11, 12). For updates computed after the integration of the effects of ΔB_i, we separated the entries that include ΔB_i in their dependency lists. For the purpose of the proof, we extend the notation to capture the presence or absence of a ΔB in the dependency list. For example, an entry $\Delta V dep_{\ldots,+/-\Delta B_j,\ldots}$ denotes two different possible entries in the table of changes, entries that include or do not include ΔB_j in the dependency lists.

$\Delta V[\Delta B_1]$	$\Delta V[\Delta B_2|\Delta B_1]$	$\cdots$	$\Delta V[\Delta B_i|\Delta B_1, \Delta B_2, \cdots, \Delta B_{i-1}]$
$\Delta V dep_{\Delta B_1}$	$\Delta V dep_{\Delta B_2}$ $\Delta V dep_{\Delta B_2,\Delta B_1}$	$\cdots$	$\mathbf{\Delta V dep_{\Delta B_i}}$ $\mathbf{\Delta V dep_{\Delta B_{i-1},\Delta B_i}}$ $\mathbf{\Delta V dep_{\Delta B_{i-2},+/-\Delta B_{i-1},\Delta B_i}}$ $\mathbf{\Delta V dep_{\Delta B_{i-3},+/-\Delta B_{i-2},+/-\Delta B_{i-1},\Delta B_i}}$ $\vdots$ $\mathbf{\Delta V dep_{\Delta B_1,+/-\Delta B_1\cdots,+/-\Delta B_{i-1},\Delta B_i}}$

Figure 10: Entries and Dependency Lists Due to Updates $\Delta B_1, \cdots \Delta B_i$.

$\Delta V[\Delta B_{i+1}|\Delta B_1, \Delta B_2, \cdots, \Delta B_i]$	
$\Delta V dep_{\Delta B_{i+1}}$	
	$\mathbf{\Delta V dep_{\Delta B_i,\Delta B_{i+1}}}$
$\Delta V dep_{\Delta B_{i-1},\Delta B_{i+1}}$	$\mathbf{\Delta V dep_{\Delta B_{i-1},\Delta B_i,\Delta B_{i+1}}}$
$\Delta V dep_{\Delta B_{i-2},+/-\Delta B_{i-1},\Delta B_{i+1}}$	$\mathbf{\Delta V dep_{\Delta B_{i-2},+/-\Delta B_{i-1},\Delta B_i,\Delta B_{i+1}}}$
$\vdots$	$\vdots$
$\Delta V dep_{\Delta B_2,+/-\Delta B_3,\cdots,+/-\Delta B_{i-1},\Delta B_{i+1}}$	$\mathbf{\Delta V dep_{\Delta B_2,+/-\Delta B_3,\cdots,+/-\Delta B_{i-1},\Delta B_i,\Delta B_{i+1}}}$
$\Delta V dep_{\Delta B_1,+/-\Delta B_2,\cdots,+/-\Delta B_{i-1},\Delta B_{i+1}}$	$\mathbf{\Delta V dep_{\Delta B_1,+/-\Delta B_2,\cdots,+/-\Delta B_{i-1},\Delta B_i,\Delta B_{i+1}}}$

Figure 11: Entries and Dependency Lists Associated with Update ΔV Due to ΔB_{i+1}.

We further make use of the fact that a term can be represented as $\Delta V dep_{\ldots,\Delta B_i,\cdots,\Delta B_j,\cdots}$, or $\Delta V dep_{\ldots,\Delta B_j,\cdots,\Delta B_i,\cdots}$, as they are just different notations for the same terms. Since the terms in the Figures 10, 11, 12 are included in a set, we can change the position within the set of the terms that are written in boldface. The

$\cdots$	$\Delta V[\Delta B_n \vert \Delta B_1, \Delta B_2, \cdots, \Delta B_{n-1}]$	
$\cdots$	$\Delta Vdep_{\Delta B_n}$ $\Delta Vdep_{\Delta B_{n-1},\Delta B_n}$ $\Delta Vdep_{\Delta B_{n-2},+/-\Delta B_{n-1},\Delta B_n}$ $\Delta Vdep_{\Delta B_{n-3},+/-\Delta B_{n-2},+/-\Delta B_{n-1},\Delta B_n}$: $\Delta Vdep_{\Delta B_{n-i},+/-\Delta B_{n-i+1},\cdots+/-\Delta B_{n-1},\Delta B_n}$ $\Delta Vdep_{\Delta B_{n-i},+/-\Delta B_{n-i+1},\cdots+/-\Delta B_{n-1},\Delta B_n}$: $\Delta Vdep_{\Delta B_{i+1},+/-\Delta B_{i+2},\cdots+/-\Delta B_{n-1},\Delta B_n}$ $\Delta Vdep_{\Delta B_{i-1},+/-\Delta B_{i+1},\cdots+/-\Delta B_{n-1},\Delta B_n}$ $\Delta Vdep_{\Delta B_{i-2},+/-\Delta B_{i-1},+/-\Delta B_{i+1},\cdots+/-\Delta B_{n-1},\Delta B_n}$: $\Delta Vdep_{\Delta B_1,+/-\Delta B_2,+/-\Delta B_{i-1},+/-\Delta B_{i+1},\cdots,\Delta B_n}$	$\mathbf{\Delta Vdep_{\Delta B_i,+/-\Delta B_{i+1},\cdots+/-\Delta B_{n-1},\Delta B_n}}$ $\mathbf{\Delta Vdep_{\Delta B_{i-1},\Delta B_i,+/-\Delta B_{i+1},\cdots+/-\Delta B_{n-1},\Delta B_n}}$ $\mathbf{\Delta Vdep_{\Delta B_{i-2},+/-\Delta B_{i-1},\Delta B_i,+/-\Delta B_{i+1},\cdots+/-\Delta B_{n-1},\Delta B_n}}$: $\mathbf{\Delta Vdep_{\Delta B_1,\cdots,+/-\Delta B_{i-1},\Delta B_i,+/-\Delta B_{i+1},\cdots,\Delta B_n}}$

Figure 12: Entries and Associated Dependency Lists for Updates Following ΔB_{i+1}

table in Figure 14 consists of the entries whose dependency lists reflect the change due to ΔB_i, in two different notations. The entries that are not dependent on ΔB_i are listed in Figure 13.

$\Delta V[\Delta B_1]$	$\cdots$	$\Delta V[\Delta B_{i+1} \vert \Delta B_1, \Delta B_2, \cdots, \Delta B_{i-1}]$	$\cdots$	$\Delta V[\Delta B_n \vert \Delta B_1, \cdots, \Delta B_{n-1}] - \Delta Vdep_{\cdots,\Delta B_i,\cdots}$
$\Delta Vdep_{\Delta B_1}$	$\cdots$	$\Delta Vdep_{\Delta B_{i+1}}$ $\Delta Vdep_{\Delta B_{i-1},\Delta B_{i+1}}$ $\Delta Vdep_{\Delta B_{i-2},+/-\Delta B_{i-1},\Delta B_{i+1}}$: $\Delta Vdep_{\Delta B_2,+/-\Delta B_3,\cdots,+/-\Delta B_{i-1},\Delta B_{i+1}}$ $\Delta Vdep_{\Delta B_1,+/-\Delta B_2,\cdots,+/-\Delta B_{i-1},\Delta B_{i+1}}$	$\cdots$	$\Delta Vdep_{\Delta B_n}$ $\Delta Vdep_{\Delta B_{n-1},\Delta B_n}$ $\Delta Vdep_{\Delta B_{n-2},+/-\Delta B_{n-1},\Delta B_n}$ $\Delta Vdep_{\Delta B_{n-3},+/-\Delta B_{n-2},+/-\Delta B_{n-1},\Delta B_n}$: $\Delta Vdep_{\Delta B_{n-i},+/-\Delta B_{n-i+1},\cdots+/-\Delta B_{n-1},\Delta B_n}$ $\Delta Vdep_{\Delta B_{n-i},+/-\Delta B_{n-i+1},\cdots+/-\Delta B_{n-1},\Delta B_n}$ $\Delta Vdep_{\Delta B_{i+1},+/-\Delta B_{i+2},\cdots+/-\Delta B_{n-1},\Delta B_n}$ $\Delta Vdep_{\Delta B_{i-1},+/-\Delta B_{i+1},\cdots+/-\Delta B_{n-1},\Delta B_n}$ $\Delta Vdep_{\Delta B_{i-2},+/-\Delta B_{i-1},+/-\Delta B_{i+1},\cdots+/-\Delta B_{n-1},\Delta B_n}$: $\Delta Vdep_{\Delta B_1,+/-\Delta B_2,+/-\Delta B_{i-1},+/-\Delta B_{i+1},\cdots,\Delta B_n}$

Figure 13: Entries Without ΔB_i in the Dependency Lists.

The last table, shown in Figure 14, enumerates the entries whose dependency lists include ΔB_i. These entries were computed at one of the steps $\{\ \Delta V[\Delta B_i|\Delta B_1, \cdots, \Delta B_{i-1}], \cdots, \Delta V[\Delta B_n|\Delta B_1, \cdots, \Delta B_{n-1}]\}$. We argue that the selected set of entries, $\Delta Vdep_{\cdots,\Delta B_i,\cdots}$, are exactly the entries due to the propagation of ΔB_i **after** $\{\ \Delta B_1, \cdots, \Delta B_{i-1}, \Delta B_{i+1}, \cdots, \Delta B_n\ \}$, i.e., $\Delta V[\Delta B_i|\Delta B_1, \cdots, \Delta B_n]$. We can prove the equality of the two sets by showing that any element in the second set, $\Delta V[\Delta B_i|\Delta B_1, \cdots, \Delta B_n]$, must be in the first, $\Delta Vdep_{\cdots,\Delta B_i,\cdots}$, and that the first set does not include any additional entries with respect to the sencond set. To see that the first statement is true, consider the following cases for possible entries due to the change $\Delta V[\Delta B_i|\Delta B_1, \cdots, \Delta B_n]$

1. if the entry is of the form $\Delta Vdep_{\Delta B_i}$, then this entry is the first entry from the column of $\Delta V[\Delta B_i|\Delta B_1, \cdots, \Delta B_{i_1}]$ (Figure 10 and 14).

2. if the entry is $\Delta Vdep_{+/-\Delta B_j,\cdots,+/-\Delta B_k,\Delta B-i}$ where $j < k$ <=i-1, then it is again one of the entries selected from the column of $\Delta V[\Delta B_i|\Delta B_1, \cdots, \Delta B_{i-1}]$ (Figure 10 and 14).

$\Delta Vdep_{\cdots,\Delta B_i,\cdots}$		
$\mathbf{\Delta Vdep_{\Delta B_i}}$		
$\mathbf{\Delta Vdep_{\Delta B_{i-1},\Delta B_i}}$		
$\mathbf{\Delta Vdep_{\Delta B_{i-2},+/-\Delta B_{i-1},\Delta B_i}}$		
$\mathbf{\Delta Vdep_{\Delta B_{i-3},+/-\Delta B_{i-2},+/-\Delta B_{i-1},\Delta B_i}}$		
:		
$\mathbf{\Delta Vdep_{\Delta B_1,+/-\Delta B_1\cdots,+/-\Delta B_{i-1},\Delta B_i}}$		
$\mathbf{\Delta Vdep_{\Delta B_i,\Delta B_{i+1}}}$	=	$\mathbf{\Delta Vdep_{\Delta B_{i+1},\Delta B_i}}$
$\mathbf{\Delta Vdep_{\Delta B_{i-1},\Delta B_i,\Delta B_{i+1}}}$	=	$\mathbf{\Delta Vdep_{\Delta B_{i-1},\Delta B_{i+1},\Delta B_i}}$
$\mathbf{\Delta Vdep_{\Delta B_{i-2},+/-\Delta B_{i-1},\Delta B_i,\Delta B_{i+1}}}$	=	$\mathbf{\Delta Vdep_{\Delta B_{i-2},+/-\Delta B_{i-1},\Delta B_{i+1},\Delta B_i,}}$
:	:	
$\mathbf{\Delta Vdep_{\Delta B_2,+/-\Delta B_3,\cdots,+/-\Delta B_{i-1},\Delta B_i,\Delta B_{i+1}}}$	=	$\mathbf{\Delta Vdep_{\Delta B_2,+/-\Delta B_3,\cdots,+/-\Delta B_{i-1},\Delta B_{i+1},\Delta B_i}}$
$\mathbf{\Delta Vdep_{\Delta B_1,+/-\Delta B_2,\cdots,+/-\Delta B_{i-1},\Delta B_i,\Delta B_{i+1}}}$	=	$\mathbf{\Delta Vdep_{\Delta B_1,+/-\Delta B_2,\cdots,+/-\Delta B_{i-1},\Delta B_{i+1},\Delta B_i}}$
:	:	
$\mathbf{\Delta Vdep_{\Delta B_i,+/-\Delta B_{i+1},\cdots+/-\Delta B_{n-1},\Delta B_n}}$	=	$\mathbf{\Delta Vdep_{+/-\Delta B_{i+1},\cdots+/-\Delta B_{n-1},\Delta B_n,\Delta B_i}}$
$\mathbf{\Delta Vdep_{\Delta B_{i-1},\Delta B_i,+/-\Delta B_{i+1},\cdots+/-\Delta B_{n-1},\Delta B_n}}$	=	$\mathbf{\Delta Vdep_{\Delta B_{i-1},+/-\Delta B_{i+1},\cdots,\Delta B_n,\Delta B_i}}$
$\mathbf{\Delta Vdep_{\Delta B_{i-2},+/-\Delta B_{i-1},\Delta B_i,+/-\Delta B_{i+1},\cdots+/-\Delta B_{n-1},\Delta B_n}}$	=	$\mathbf{\Delta Vdep_{\Delta B_{i-2},+/-\Delta B_{i-1},+/-\Delta B_{i+1},\cdots,\Delta B_n,\Delta B_i}}$
:	:	
$\mathbf{\Delta Vdep_{\Delta B_1,\cdots,+/-\Delta B_{i-1},\Delta B_i,+/-\Delta B_{i+1},\cdots,\Delta B_n}}$	=	$\mathbf{\Delta Vdep_{\Delta B_1,\cdots,+/-\Delta B_{i-1},+/-\Delta B_{i+1},\cdots,\Delta B_n,\Delta B_i}}$

Figure 14: All Entries that Contain ΔB_i in the Attached Dependency Lists

3. if the entry is of the form $\Delta V dep_{+/-\Delta B_j,\cdots,+/-\Delta B_k,\Delta B_i}$ where $k >= (i+1)$, then this entry is selected from the column of $\Delta V[\Delta B_k|\Delta B_1,\cdots,\Delta B_{k-1}]$ (corresponding to Figure 12, Figure 11 and 14).

All necessary entries for the computation of $\Delta V[\Delta B_i|\Delta B_1,\cdots,\Delta B_n]$ can therefore be found in the set of entries selected. We have to show that the two sets of entries are exactly the same, i.e, also that there are no unnecessary entries in the selected set. An entry is not necessary only if it includes a ΔB_j in its attached dependency list, where $j > n$. This is impossible, since the last computation integrated the effects of ΔB_n, and the indices represent the order of computation at V, that is $j <= n$.

[q.e.d.]

B Maintenance Algorithm

```
MODULE QueryServer;

    CONSTANT
        MyIndex = j;

    PROCESS AnswerQuery;
    LOOP
        RECEIVE ⟨ΔV, IgnoreSet[S_j], ΔB, direction⟩ FROM descendant V;
        IF BaseRelation(j) THEN
            ΔV = ComputeJoin(ΔV, S_j);
        ELSE IF direction = 0 THEN /* return S_j ⋈ ΔV */
            TempState= State of S_j that excludes the effects
                of updates in IgnoreSet[S_j], ΔB, and updates after ΔB;
            ΔV = ComputeJoin(ΔV, TempState);
        ELSE IF direction = 1 THEN /* return ΔV ⋈ (S_j + ΔS_j[ΔB]) */
            TempState= State of S_j that excludes the effects
                of updates in IgnoreSet[S_j], and updates after ΔB but includes ΔB;
            ΔV = ComputeJoin(ΔV, TempState);
        ENDIF;
        SEND ΔV TO DataWarehouse;
    FOREVER;

BEGIN /* Initialization */
    StartProcess(AnswerQuery);
END QueryServer.
```

Figure 15: The Query Server at Ancestor j

```
MODULE MultiView DataWarehouse;
    PERSISTENT DATA
    GLOBAL DATA
        V: RELATION; /* Initialized to the correct view */
        UpdateMessageQueue: QUEUE initially ∅;

    FUNCTION ViewChange( ΔB: RELATION): RELATION
    VAR
            ΔV_1, ..., ΔV_n, TempView: RELATION;
            i, j: INTEGER;
    BEGIN
        /* evaluate S_1 ⋈ ... ⋈ S_{i-1} ⋈ ΔS_i[ΔB] ⋈ (S_{i+1} + ΔS_{i+1}[ΔB]) ⋈ ... ⋈ (S_n + ΔS_n[ΔB]) */
        FOR i = 1 TO n DO
            if S_i is a base relation, then create identifier ΔB for S_i[ΔB]. Otherwise identifier will be included with the udpate.
            IgnoreSet[S_i] = {..., ΔB', ...} such thatΔS_i[ΔB'|...] delivered before ΔS_i[ΔB|...] but not incorporated at V;
            ΔV_i = EliminateEntries(ΔS_i[ΔB|...], IgnoreSet[S_i]);
            direction = 0 ;
            FOR EACH ancestor j ≠ i DO /* j = i - 1 DOWNTO 1 followed by j = i + 1 TO n */
                TempView = ΔV_i;
                IF i ≤ j THEN direction = 1 ;
                SEND ⟨ΔV_i, IgnoreSet[S_j], ΔB⟩ TO ancestor j;
                RECEIVE ΔV_i FROM ancestor j;
                IF BaseRelation(j) ∧ ∃ΔB_j ∈ UpdateMessageQueue THEN
                    ΔV_i = ΔV_i - ΔB_j ⋈ TempView;
                ENDIF;
            ENDFOR;
        ENDFOR;
        RETURN(ΔV_1 + ΔV_2 + ... + ΔV_n);
    END ViewChange;

    PROCESS LogUpdates;
    LOOP
        RECEIVE ΔS_i[ΔB|...] FROM Data Source i;
        APPEND (ΔS_i[ΔB|...], i) TO UpdateMessageQueue;
    FOREVER;

    PROCESS UpdateView;
    LOOP
        WAIT UNTIL ΔS_i[ΔB|...] ARRIVES FROM ALL ancestors i;
        REMOVE ALL (ΔS_i[ΔB|...], i) FROM UpdateMessageQueue;
        ΔV = ViewChange(ΔB)
        Add identifier of ΔB to dep of all entries in ΔV.
        V = ΔV
    FOREVER;

BEGIN /* Start DataWarehouse Processes */
    StartProcess(LogUpdates);
    StartProcess(UpdateView);
END MultiView DataWarehouse.
```

Figure 16: The Incremental View Construction Algorithm for Distributed Multi-view Data Warehouses