Decentralized Incremental Maintenance of Multi-View Data Warehouses (Extended Abstract)

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

1 Introduction

Decision support systems make up a big percentage of data base servers. Presently, their size increases due to the necessity of more detailed information, and the extended time range of interest. As a result, data mining queries also span larger sets of data. To achieve fast response time, a subset of the relevant information is sometimes materialized in views, separate from the database sources [4]. A data warehouse is an example of storage that integrates information from multiple sources, which may be stand-alone databases as well as sites such as the Internet. Although data warehousing is a powerful concept for supporting analytical processing, building a data warehouse runs into several pragmatic problems. The cost of building a data warehouse that integrates disparate data sources in an organization can easily exceed millions of dollars. A more severe problem that arises in constructing a data warehouse is the time needed to make it operational; past experience indicates that building a data warehouse for a large enterprise can easily take 12-18 months. As a consequence of these practical difficulties, an alternative to a data warehouse is emerging which is 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.

The distributed multi-view data warehouse model is important since it can be used to integrate and summarize information from multiple and diverse data sources which may be geographically dispersed. This integration and summarization is necessary to execute decision making and analytical processing queries on aggregated and derived data. 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 is expected that the present centralized model would evolve into a distributed system. As a result of this distribution, there will be multiple materialized views maintained at different sites distributed over a large network. We expect that these views 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 provide the possibility of accessing data at different granularity levels and facilitate operations such as drill-down or roll-up.

Most algorithms previously developed for the maintenance of a distributed data warehouse are specifically designed for a single view or independent multiple views [3, 1, 11, 12, 8, 6]. In this paper we develop an efficient incremental algorithm for the maintenance of multi-views in a distributed data warehouse. In a multiple view environment, a single 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*. [10], have proposed using a synchronization processing unit for ensuring consistency, thus often requiring additional delay of view updates. This approach is appropriate for centralized multi-view data warehouses. A decentralized approach has also been proposed [9], based on calcualting the cummulative effect of a batch of updates. In contrast, our method pertains to decentralized systems, and allows each view to incorporate updates independently. By differentiating between the data stored in a materialized view in a safe state and subsequent updates, a view is able 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 views in a large distributed setting. An update to a view can be determined solely through communication with the views or data sources from which it directly derives data. Furthermore, no unnecessary delays are imposed on the incorporation of new updates and no centralized coordination is necessary. This is especially important when the data is derived from various autonomous sources such as in an internet setting.

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 a decentralized incremental algorithm for updating the information stored by views (Section 4). We conclude with a discussion of the proposed algorithm in Section 5.

2 The Multi-View Data Warehouse 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. 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 not only from the need to correctly propagate updates from base sources to views, but also from issues related to the maintenance of a consistent state between different views that comprise a distributed data warehouse. 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 [7]. 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 a distributed multi-view data warehouse.

Consider a data 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 $\sigma(B)$; and $\sigma_m(B)$ is the m'th state in this sequence $\sigma(B)$. 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 $\{S_1, S_2, \ldots S_k\}$, 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 S_i's state is derived from state $\sigma_m(B)$ and S_j's state is derived from from state $\sigma_n(B)$, $\sigma_m(B)$ must be the same as $\sigma_n(B)$.

The above definition simply states that a view V is *common base consistent* if its data at any time reflects at most one state of each base source in the data warehouse. We also use a View Dependency Graph [3] to describe the system. This graph has nodes for sites hosting views and base sources, and the directed edges between nodes indicate direction of the flow of update messages. An edge directed towards any view S connects S to one of its ancestors. Figure 1 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. For common base consistency, 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.

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 $S_1 \bowtie S_2 \bowtie \ldots \bowtie S_n$. Furthermore, each source S_i has experienced a set of possible empty updates (inserts and deletes) ΔS_i. View V now 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 $V + \Delta V$, calculated as follows:

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

Note that $S_1 \bowtie S_2 \ldots \bowtie S_i \ldots \bowtie S_n$ 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. All ΔS_i in the above equation are derived from the same change at a data source. By using the commutativity property of the deriving function, the above equation can be rewritten as follows:

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

Note that we made use of the properties of the deriving function not only to decompose an update query into subqueries, but also to reorder some of the terms. Similar to prior methods of decomposing update queries [1], for update ΔS_i we first need to traverse the ancestors to the left of S_i (in our case in decreasing order of their indices) and then to the right (in increasing order of indices). If subqueries are evaluated in this manner the associativity requirement of the merging function can be relaxed. All subqueries start with a term of the form $\mathbf{\Delta S_i}$, and therefore 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. Further optimizations can be incorporated, by choosing an optimal order of query over the remaining terms [2]. 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.

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

The efficiency of maintaining views incrementally rather than through recomputation has been recognized in previous work [5, 6]. 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, 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 1, 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. In reality their ordering may be inconsistent with each other.

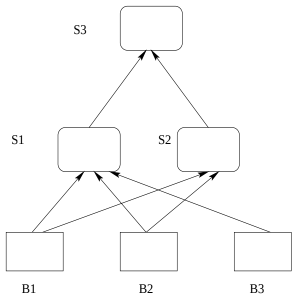

Figure 1: Issues in Maintaining a Hierarchy of Views in a Consistent State

We use the following notation to reflect 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.

Consider, for illustration purposes, a scenario in which update messages are delivered to different views in Figure 1 in the order below:

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	–	–	$\Delta S_1[\Delta B_1|\Delta B_2, \Delta B_3]$

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. According to the order defined in Section 2, these queries result in the following changes at the views:

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

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

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 equation 2, in order to calculate update $\Delta S_3[\Delta B_1]$, S_3 needs to evaluate: $\Delta S_3[\Delta B_1] = \Delta S_1[\Delta B_1] \bowtie (S_2 + \Delta S_2[\Delta B_1]) + \Delta S_2[\Delta B_1] \bowtie S_1$ 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:

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

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 for 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 illustrating 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.

materialized state of S_1: $(B_1 \bowtie B_2 \bowtie B_3)$

table of changes:

$\Delta S_1[\Delta B_3] = (\Delta B_3 \bowtie B_2 \bowtie B_1)dep_{\Delta B_3}$

$\Delta S_1[\Delta B_2|\Delta B_3] = (\Delta B_2 \bowtie B_1 \bowtie B_3)dep_{\Delta B_2} + (\Delta B_2 \bowtie B_1 \bowtie \Delta B_3)dep_{\Delta B_2,\Delta B_3}$

$\Delta S_1[\Delta B_1|\Delta B_2,\Delta B_3] = (\Delta B_1 \bowtie B_2 \bowtie B_3)dep_{\Delta B_1} + (\Delta B_1 \bowtie \Delta B_2 \bowtie B_3)dep_{\Delta B_1,\Delta B_2} + (\Delta B_1 \bowtie B_2 \bowtie \Delta B_3)dep_{\Delta B_1,\Delta B_3} + (\Delta B_1 \bowtie \Delta B_2 \bowtie \Delta B_3)dep_{\Delta B_1,\Delta B_2,\Delta B_3}$

materialized 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 illustrating example, according to the above update order at each site. The first table includes both the initial states and the insert updates at base sources B_1, B_2 and B_3, while the second table contains the $safe$ state as well as the table of changes of S_1 and S_2. 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	ΔB_2	ΔB_3
(2,2)	(2,3)	(3,1)

S_1		S_2	
(A,B,C,D)	**dep**	**(A,B,C)**	**dep**
materialized state		materialized state	
(1,2,4,1)	–	(1,2,4)	–
table of changes		table of changes	
(1,2,3,4)	$\{\Delta B_2\}$	(2,2,4)	$\{\Delta B_1\}$
(1,2,3,1)	$\{\Delta B_2, \Delta B_3\}$	(1,2,3)	$\{\Delta B_2\}$
(2,2,4,1)	$\{\Delta B_1\}$	(2,2,3)	$\{\Delta B_1, \Delta B_2\}$
(2,2,3,4)	$\{\Delta B_1, \Delta B_2\}$	–	–
(2,2,3,1)	$\{\Delta B_1, \Delta B_2, \Delta B_3\}$	–	–

The algorithm 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. Since commutativity is transitive, this property holds even if ΔB_i and ΔB_j are not consecutive updates. An important consequence of this observation is that all entries that include ΔB_i reflect all and exactly the effects of having incorporated the update ΔB_i, as if this was 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 calculate 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 Description

We now present the algorithm for the incremental maintenance of views. Figure 2 illustrates the code for answering update queries, while Figure 3 outlines the steps of a view's 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 a *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. For example, for S_3 to incorporate update ΔB_1 it selects the set of updates $\Delta S_1[\Delta B_1|\Delta B_2, \Delta B_3]$ from S_1 (which follows $\Delta S_1[\Delta B_2|\Delta B_3]$ and $\Delta S_1[\Delta B_3]$ in the message queue) and $\Delta S_2[\Delta B_1]$ from the list of messages received from S_2.

```
MODULE QueryServer;

    CONSTANT
        MyIndex = j;

    PROCESS AnswerQuery;
    LOOP
        RECEIVE ⟨ΔV, IgnoreSet[S_j], ΔB⟩ FROM descendant V;
        IF BaseRelation(j) THEN
            ΔV = ComputeJoin(ΔV, S_j);
        ELSE IF j < i 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 j > i 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 2: The Query Server at Ancestor j

Once the set of ΔS_i is selected, the view must determine the actual form of each ΔS_i, as required by the update phase. In our example, at the time when S_3 updates according to 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 $(\Delta B_1 \bowtie \Delta B_2 \bowtie \Delta B_3)dep_{\Delta B_1,\Delta B_2,\Delta B_3}$. In

```
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]);
            FOR EACH ancestor j ≠ i DO /* j = i − 1 DOWNTO 1 followed by j = i + 1 TO n */
                TempView = ΔV_i;
                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 3: The Incremental View Construction Algorithm for Distributed Multi-view Data Warehouses

general, 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.

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. Assuming that the base source returns an answer $\Delta V_i \bowtie (CurrentState)$ where ΔV_i was included in the querying message, then V can easily compute $\Delta V_i \bowtie (CurrentState)$ - $\Delta V_i \bowtie (UnnecessaryUpdates)$. This is possible because a view deriving directly from a base source has the update messages in its message queue in the order of their delivery. Otherwise, if the queried ancestor is a view, it is expected to locally calculate the correct answer to the update query, and no additional compensation is needed by the querying view.

Now we consider this later case of a view S_j that is queried for a partial result in the update phase of its descendent V. This update phase computes the effects due to the changes to a base source B. 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, $IgnoreSet$. View S_j performs the join over its materialized view and the table of changes from which it excludes (by executing function $EliminateEntries$) 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.

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. 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. 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.

As a result of incorporating all the updates, S_3 will eventually store the following information:

$\underline{\text{materialized state of } S_3} = (S_1 \bowtie S_2)$

$\underline{\text{table of changes:}}$

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

$\Delta S_3[\Delta B_2|\Delta B_1] = (\Delta S_1[\Delta B_2] \bowtie S_2 + \Delta S_1[\Delta B_2] \bowtie \Delta S_2[\Delta B_2])dep_{\Delta B_2} + (\Delta S_1[\Delta B_2] \bowtie \Delta S_2[\Delta B_1]dep_{\Delta B_1,\Delta B_2} +$
$(\Delta S_2[\Delta B_2] \bowtie S_1)dep_{\Delta B_2} + (\Delta S_2[\Delta B_2] \bowtie \Delta S_1[\Delta B_1])dep_{\Delta B_1,\Delta B_2}$

$\Delta S_3[\Delta B_3|\Delta B_2,\Delta B_1] = (\Delta S_1[\Delta B_3] \bowtie S_2)dep_{\Delta B_3} + (\Delta S_1[\Delta B_3] \bowtie \Delta S_2[\Delta B_1])dep_{\Delta B_1,\Delta B_3} + (\Delta S_1[\Delta B_3] \bowtie \Delta S_2[\Delta B_2])dep_{\Delta B_2,\Delta B_3}$

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. Hence for example $\Delta S_1[\Delta B_1|\Delta B_2,\Delta B_3]$ can be incorporated into the materialized view only after $\Delta S_3[\Delta B_3]$ and $\Delta S_3[\Delta B_2|\Delta B_3)$ are incorporated into S_3. In contrast, view S_2 can refresh its table of changes sooner, by incorporating $\Delta S_2[\Delta B_1]$ immediately after S_3 integrates the update $\Delta S_3[\Delta B_1]$. As dependency lists are no longer necessary to describe tuples in a $safe$ state, they will not be included in the actual materialized views.

4.3 Discussion

The total number of messages exchanged during the update phase of a view using our protocol is independent of the order of update propagation. For a view V with n direct ancestors, the upper bound on the number of messages involved in the incorporation of an update is $O(n^2)$. That is, the update of the base source is at most propagated to all of V's direct ancestors, and the query for computing the effects of each of these subsequent changes involves at most a pass through all the other direct ancestors. Note that, in general, the number of direct ancestors is expected to be much smaller than the number of base sources in the system. Furthermore, it is possible to pipeline these subqueries together at the cost of larger individual message size but linear message cost.

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. Although we gave an intuitive argument for the correctness of the eliminating method, due to space limitations we are not including a formal proof. 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).

An alternative to our approach, that still maintains consistency, is the preordering of update messages by a synchronizatio module. Such a module must take a conservative decision, since it may forward messages to more than one view and restrictions on the order of updates depend on all these views. In contrast, a local decision is limited by less restrictions, and thus it eliminates some of the delays imposed by the ordering of update messages.

5 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 [10] where the order of updates to all views must be the same, thus imposing unnecessary delays when incorporating new updates.

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

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

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

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.