WPI-CS-TR-00-16 May 2000

Scalable Maintenance in Distributed Data Warehousing Environment

by

Lingli Ding and Xin Zhang and Elke A. Rundensteiner

Computer Science
Technical Report
Series

WORCESTER POLYTECHNIC INSTITUTE

Computer Science Department
100 Institute Road, Worcester, Massachusetts 01609-2280

Scalable Maintenance in Distributed Data Warehousing Environment

Lingli Ding, Xin Zhang and Elke A. Rundensteiner

Department of Computer Science
Worcester Polytechnic Institute
Worcester, MA 01609-2280
(lingli | xin | rundenst)@cs.wpi.edu

June 4, 2000

Abstract

The maintenance of data warehouses is becoming an increasingly important topic due to the growing use, derivation and integration of digital information. Most previous work has dealt with one centralized data warehouse (DW) only. In this paper, we now focus on environments with multiple data warehouses that are possibly derived from other data warehouses. In such a large-scale environment, data updates from base sources may arrive in individual data warehouses in different orders, thus resulting in inconsistent data warehouse extents. We propose a registry-based solution strategy that addresses this problem by employing a registry agent responsible for establishing one unique order for the propagation of updates from the base sources to the data warehouses. With this solution, individual data warehouse managers can still maintain their respective extents autonomously and independently from each other, thus allowing them to apply any of the existing incremental maintenance algorithms from the literature. We demonstrate that this will indeed achieve consistency across all data warehouses. In order to achieve scalability of this approach, we further optimize this registry solution by partitioning the set of data warehouse managers into smaller data warehouse groups each equipped with their own registry. We present a partitioning algorithm for generating such a scalable DW group architecture. Finally, we analyze the performance of the proposed solutions based on a cost model, and demonstrate that the partitioned registry approach achieves substantially better performance than the registry solution.

Keywords: Distributed Data Warehousing, View Maintenance, Registry, Partition, Data Update.

1 Introduction

Data warehousing [WB97, GMLWZ98, GM96, ZR99] (DW) is a popular technology to integrate data from heterogeneous information sources (ISs) in order to provide data to, for example, decision support or data mining applications [CD97]. Once a DW is established, the problem of maintaining it consistent with underlying ISs under updates remains a critical issue. It is popular to maintain the DW incrementally [AASY97, MKK97, KLMR97] instead of recomputing the whole extent of the DW after each IS update, due to the large size of DWs and the enormous overhead associated with the DW loading process. The majority of such view maintenance algorithms [AASY97, ZGMW96, ZGMHW95] as of now are based on a centralized DW system in which materialized views are stored in a single site even if bases may be distributed.

Given the growth of digital on-line information and the distributed nature of sources found on the Internet, we can expect large collections of interrelated derived repositories instead of just one individual DW. Such a distributed environment composed of several autonomous data warehouses is emerging as a design solution for a growing set of applications [SDA99]. In this paper, we focus on the view maintenance in such distributed environments composed of multiple DWs, in particular, data warehouses that contain views possibly derived from views residing in other data warehouses.

1.1 Motivation Example

In such a distributed environment, data updates of information sources (ISs) may arrive in different data warehouses in different orders. Thus data updates from ISs may result in an inconsistency of the data warehouse extents. This means that the higher level DW built on top of those inconsistent ones couldn't be updated correctly by any traditional incremental view maintenance [AASY97, MKK97, KLMR97]. Here is an example.

Example 1 *Suppose there are two base relations B_1, B_2 in two ISs and three views V_0, V_1 and V_2 at three DWs that reside on different sites, defined as $V_0 = V_1 \bowtie V_2$, $V_1 = B_1 \bowtie B_2$ and $V_2 = B_1 \bowtie B_2$.*

Assume there is a data update at B_1 denoted by ΔB_1 and a data update at B_2 denoted by ΔB_2. These two DUs are then sent to views V_1 and V_2. Assume V_1 receives DUs in the order of ΔB_1 followed by ΔB_2. V_2 receives DUs in the different order of ΔB_2 followed by ΔB_1 due to different network traffic.

Both V_1 and V_2 compute new view extents to reflect base data updates. To distinguish view updates caused by the different data updates, we use a data update identifier, in short DU-id, to identify DUs from different ISs. For simplicity, we use an integer to denote DU-id with the first digit of the DU-id the same as the IS index number [1] *and rest of DU-id is the index of DUs from that IS. For example, $\Delta V_1/11$ denotes the ΔV_1 calculated for the first update (ΔB_1) from B_1, and $\Delta V_1/21$ means the ΔV_1 calculated for the first update (ΔB_2) from B_2. We can also rewrite ΔB_1 and ΔB_2 into $\Delta B_1/11$ and $\Delta B_2/21$, respectively. In a distributed system, each IS has an unique index number. Hence, each DU-id is unique for the whole system.*

[1] If there were a lot of bases, we could use more digits to represent base index number.

In our case, view maintenance for V_1 computes and updates V_1 with $\Delta V_1/11$ for $\Delta B_1/11$ and then $\Delta V_1/21$ for $\Delta B_2/21$. However, the view manager of V_2 computes and updates V_2 in a reverse order, i.e., $\Delta V_2/21$ for $\Delta B_2/21$ first and then $\Delta V_2/11$ for $\Delta B_1/11$. V_1 sends its update messages $\Delta V_1/11$, $\Delta V_1/21$ and V_2 sends $\Delta V_2/21$ and $\Delta V_2/11$ to V_0.

For both V_1 and V_2, the above view maintenance steps are correct. Their extents are correctly updated to reflect the changes in the ISs B_1 and B_2. However, V_0 receives inconsistent update messages from V_1 and V_2 which are as follows:

$\Delta V_1/11 = \Delta B_1/11 \bowtie B_2$.

$\Delta V_1/21 = \Delta B_2/21 \bowtie (B_1 + \Delta B_1/11)$.

$\Delta V_2/21 = B_1 \bowtie \Delta B_2/21$.

$\Delta V_2/11 = \Delta B_1/11 \bowtie (B_2 + \Delta B_2/21)$.

We notice that $\Delta V_1/11$ reflects the new state of B_1 but not B_2. But $\Delta V_2/11$ reflects both the new state of B_1 and B_2. When V_0 computes its new extent to reflect the update from base B_1 based on the update messages $\Delta V_1/11$ and $\Delta V_2/11$, its new extent incorporates aspects of two different states of B_2, i.e., the B_2 before and after update $\Delta B_2/21$.

$$\begin{aligned} V_0 &= (V_1 + \Delta V_1/11) \bowtie (V_2 + \Delta V_2/11) \\ &= (V_1 + \Delta B_1/11 \bowtie B_2) \bowtie (V_2 + \Delta B_1/11 \bowtie (B_2 + \Delta B_2/21)) \end{aligned} \tag{1}$$

The correct view V_0 extent should only reflect one state of B_2. For example:

$$V_0 = (V_1 + \Delta B_1/11 \bowtie B_2) \bowtie (V_1 + \Delta B_1/11 \bowtie B_2) \tag{2}$$

From the example, we can see that in this distributed DW environment though V_1 and V_2 can still be made to be consistent with ISs, the DW (V_0) built on top of them will result in an inconsistent state with respect to the ISs if there is no coordination between the maintenance of V_1 and V_2.

1.2 Related Work

Zhuge et al. [ZWGM97] defined the multiple views consistency (MVC) problem and proposed an architecture for handling multiple views consistency for multiple views specified in a single data warehouse. Stanoi et al. [SDA98] proposed a weak consistency maintenance approach that processes cumulative effect of a batch of updates. In their latter work [SDA99], they defined a distributed multi-view data warehouse model and proposed an incremental algorithm for maintenance. Their approach requires storage for a stable state with table of changes. A stable state 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 state prior to this safe state. The table of changes includes all the updates following the safe state of the actual materialized view. There is a dependency list appended to all entries in the table of changes [SDA99]. A materialized view is refreshed 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, but all its direct descendent need to wait for their corresponding direct descendents. If a lot of data updates continue to occur at bases and they are in a different sequence when they arrive at different views due to having been propagated through different derivation chains, it could take a long time to have a safe time. Hence, this could cause a DW to be updated only after a long delay especially in a DDWE with long view derivation chains. Hence, in the worst case, some DW state may never get refreshed. In our approach, we don't need any DW to keep a safe state table. Also by using the registry, there is no potential chance of an infinite wait. Our approach can have good performance for both a flat or a tall topology and hence is scalable.

1.3 Our Approach

In this paper, we propose a registry-based solution strategy that is able to coordinate the maintenance of multiple data warehouses in a non-intrusive manner by establishing a unique order among base update notifications for the environment. DW managers in this distributed environment exploit this order synchronization when determine how and when to process incoming update notifications and to commit their respective extent update, instead of blindly making use of the order in which the update messages are received from their parent. We demonstrate that this method, which we call RyCo for agent-based coordination for destination data warehouse maintenance will indeed achieve consistency across all data warehouses with little overhead. In our RyCo architecture, all views can be updated independently from one another, i.e., unlike in [SDA99], there is no need for a safe state. In order to achieve scalability of this approach, we further optimize this registry solution by partitioning the set of interrelated data warehouse systems into smaller groups each equipped with its own registry. Thus, in effect, we achieve a partitioned registry approach now responsible for only a small subset of the notification messages in the system, have called the partition-registry approach called PyCo. PyCo achieves a scalable DW group architecture that achieves substantially better performance than the central registry solution. We study the performance of the proposed solutions based on an analytical model, confirming our expectations that the partitioning approach PyCo has an overall better performance than the registry-only approach RyCo.

In summary, this paper offers the following contributions:

- We propose a registry-based approach for coordinating view maintenance in distributed data warehousing environments called RyCo. RyCo has the following advantages:
  - First, each DW is maintained by a separate mediator from the literature and can be updated independently from other DWs.
  - Second, all the DWs are guaranteed to be maintained consistent with each other and the ISs.
  - Third, unlike previous work [SDA99], our approach avoids any requirement for a safe time to update a view.

- For better scalability of our solution, we propose PyCo, a partitioning algorithm that optimizes the registry approach for a large distributed data warehousing environment by partitioning the registry into multiple smaller registry agents. It has the following advantages:
  - First, the three advantages of RyCo listed above all still hold for PyCo.
  - Second, the system is scalable in terms of the total number of DWs and bases in the environments.
  - Third, we prove the uniqueness of the partition composed of atomic DW groups generated by our proposed algorithm.
- At last, we have done a preliminary evaluation of the two solutions , demonstrating that the partitioning approach has an overall better performance than the registry approach.

Outline. The rest of the paper is organized as follows. The system definitions, assumptions and view consistency levels are presented in Section 2. Section 3 presents the architecture and algorithm of the registry-based solution. The partitioning architecture and a partitioning algorithm are presented in Section 4. We study the performance and define the cost model in Section 5. Finally, we conclude in Section 6.

2 Background: A Distributed Data Warehousing Environment

2.1 View Dependency Graph

A distributed data warehousing environment (DDWE) is composed of multiple possibly interrelated DWs. Individual DWs are independent processes running possibly on different sites. Each DW has its own views. Views may be defined on views of its own DW, on other views, on base relations in ISs, or on a combination of them.

Definition 1 *A view V is defined as below.*

$$V = S_1 \bowtie S_2 \bowtie ... \bowtie S_n \tag{3}$$

where S_i $(1 \leq i \leq n)$ could be either a base relation or a view. We say S_i is a **parent** *of the view V and the view V is the* **direct descendent** *of S_i. If any S_i in Equation 3 is a view, then the parent P of S_i is called* **ancestor** *of the view V and V is* **indirect descendent** *of P. The bases B_1, ..., B_m from which V is ultimately derived, i.e., all ancestors of V that are bases, are called* **base-ancestor***s of V. If a base B is a base-ancestor of both a view V and a view V', then we say B is a* **common base-ancestor** *of V and V'.*

A view-dependency graph [CKL+96] represents the hierarchical relationships among the views in DWs and bases in ISs. The dependency graph shows how a view is derived from bases and/or other views.

Definition 2 *The* **view dependency graph** *of the view V in a DDWE is a graph $G_v = (N, E)$ with N the set of bases or views that are ancestors of V including V itself and E the set of of directed edges $E_{ij} = (N_i, N_j)$ for each N_j that is a direct descendent of N_i, where $N_i, N_j \in N$. All nodes N_i are labeled by the name of the bases or views that they represent.*

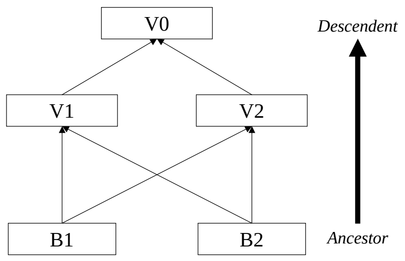

Figure 1: A Distributed DW System Example

Example 2 *Figure 1 depicts the view dependency graph for the view definition in Example 1. For example, views V_1 and V_2 are* **parents** *of the view V_0. B_1 and B_2 are* **ancestors** *of V_0. V_0 is the* **direct descendent** *of V_1 and V_2. The* **base-ancestors** *of V_0 (as well as V_1) are B_1 and B_2; while B_1 and B_2 are also* **common-bases ancestors** *of V_1 and V_2.*

2.2 Base Consistency and View Consistency

We assume that the execution of IS and DW updates is serialized, i.e., one update at IS or DW at a time and sequential network.

Definition 3 *A* **state change sequence** *of a base (view) is defined as the sequence of states of the base (view) with a snap shot of each state taken after each update commits.*

Definition 4 *Let a view V be defined by Definition 1 with S_i $(1 \leq i \leq n)$ a base or a view and B_1, ..., B_m be the base-ancestors of V. V is said to be* **base consistent** *if each DW state in the state change sequence of V corresponds to one single state in the state change sequence of each B_1, ..., B_m respectively.*

Based on Definition 4, a view is base consistent if at all times its extent reflects at most one real state of each of the ISs in DDWE. In other words, the view V cannot reflect two different states of a base B_j.

Definition 5 *Views V_1 and V_2 are said to be* **view consistent** *with each other if V_1 and V_2 are both base consistent and after committing each data update (DU_i) from one of the common bases, the states of V_1 and V_2 correspond to the same state in the state change sequence for each common base.*

Example 3 *In Example 1, assume n data updates happened at B_1, m data updates at B_2, the state change sequence of B_1 is sb_{11}, sb_{12}, $...sb_{1n}$, and the state change sequence of B_2 is sb_{21}, sb_{22}, $...sb_{2m}$. After* *committing all base data updates, the state change sequence of V_1 is sv_{11}, sv_{12}, ...sv_{1k} (k less than $n+m$) and the state change sequence of V_2 is sv_{21}, sv_{22}, ...$sv_{2k'}$ (k' less than $n+m$).*

In the state change sequence of V_1, if every sv_{1i} (i from 1 to k) corresponds to exactly one base state of B_1, say sb_{1i}, and of B_2, say sb_{2j}, then we say V_1 is **base consistent**.

If after committing a DU of B_1, the state of V_1 and V_2 always correspond to the same base state of B_1, then V_1 and V_2 are said to be **view consistent** *with each other.*

2.3 View Maintenance Issues in Distributed Data Warehouse Systems

Assume the view V is defined as in Equation 3. Further assume the views S_i (for $1 \leq i \leq n$) have multiple common bases B_k. If there is any base data update from B_k, the views S_1, ..., S_n and V need to be maintained **view consistent** with each other. In the following discussion, we use the notations defined in Table 1.

Notation	**Meaning**
B_i	Represent a base information source (IS) i.
V_i	Represent a view (DW) i.
S_i	Represent a base or a view i.
DU-id	Data Update Identifier.
$\Delta V/ij$	A view update based on data update with DU-id ij that is jth data update from base i.

Table 1: Notation for Views, Bases, and Updates

For example, assume one of the data updates that happened at base B_k, denoted as $\Delta B_k/kj$, is propagated up to S_1, ..., S_n. Then all $\Delta S_i/kj$ (i from 1 to n) are calculated where some of the $\Delta S_i/kj$ could be null if B_k is not a base of the view S_i, and sent to the view V. If all $\Delta S_i/kj$ only contain the effect of the data update of $\Delta B_k/kj$ for the base B_k, we can calculate the new view extent V by the following equation:

$$V + \Delta V = (S_1 + \Delta S_1/kj) \bowtie (S_2 + \Delta S_2/kj) \bowtie ... \bowtie (S_i + \Delta S_i/kj) \bowtie ... \bowtie (S_n + \Delta S_n/kj) \quad (4)$$

Hence, the ΔV expression is shown as follows:

$$\begin{aligned} \Delta V &= \Delta S_1/kj \bowtie S_2 \bowtie ... \bowtie S_i \bowtie ... \bowtie S_n \\ &+ (S_1 + \Delta S_1/kj) \bowtie \Delta S_2/kj \bowtie S_3 \bowtie ...... \bowtie S_i \bowtie ... \bowtie S_n + ... \\ &+ (S_1 + \Delta S_1/kj) \bowtie ... \bowtie (S_{i-1} + \Delta S_{i-1}/kj) \bowtie \Delta S_i/kj \bowtie S_{i+1} \bowtie ... \bowtie S_n + ... \quad (5) \\ &+ (S_1 + \Delta S_1/kj) \bowtie (S_2 + \Delta S_2/kj) \bowtie ... \bowtie (S_i + \Delta S_i/kj) \bowtie ... \bowtie (S_{n-1} + \Delta S_{n-1}/kj) \bowtie \Delta S_n/kj \end{aligned}$$

In practice, multiple data updates may have happened concurrently at different bases. These DUs may arrive at different views in a different order. Then it could happen that some of the $\Delta S_i/kj$ do not only contain the effect of $\Delta B_k/kj$ but in addition they may also incorporate the effect of some other DUs. In other words, S_i and S_j $(i \neq j)$ are not view consistent with each other. Hence, view V derived from S_i $(1 \leq i \leq n)$ is not base consistent.

To maintain view V base consistent by Equation 4, we need to update the view V based on the update messages from all its parents S_1, S_2, ... S_n that reflect the same state of the same base. That is all the $\Delta S_i/kj$ (i from 1 to n) must reflect the same states of all the bases B_1, ..., B_m

In such a distributed data warehousing environment, we can't control that S_i and S_j (i $\neq$ j) always reflect the same base extents because S_i and S_j belong to different DWs. But to maintain a view V correctly, we first need to assure all its parents S_i to be view consistent with each other. We propose a solution to this problem in the next section.

2.4 Assumptions

Assumption 1 *For simplicity, in the remainder of this paper, we assume each DW only has one view, though the extension of our work to multiple views per DW is straightforward.*

Based on this assumption, we can use the terms DW and view interchangeably in the remainder of our discussion.

Assumption 2 *We assume each information source (IS) has one base relation.*

We can use the MRE wrapper proposed in [DZR99] to release this assumption.

Assumption 3 *In a distributed environment, the order in which data update (DU) messages from the* **same** *base (or DW) arrive at a direct descendent reflects the order in which they actually happened and have been sent out from that base (or DW). That is, we assume the point to point network connection is FIFO.*

3 The Registry-Based Coordination for Distributed DW Maintenance

3.1 View Maintenance Order and Consistency

If views in our environment are maintained independently from other DWs, then they may not be consistent with each other. If views are not view consistent with each others, it is difficult to maintain a view that is defined on top of already inconsistent views to be base consistent. If we could maintain the lower layer views consistent with each other, we could also correctly maintain descendent views defined on top of these consistent parents to be consistent. For example, in Figure 1, if we succeed to maintain V_1 and V_2 consistent with each other [2], then we can also easily maintain V_0.

[2]If V_1 and V_2 are updated according to the same order of base data updates, then V_1 and V_2 are view consistent with each other by Definition 5.

Definition 6 *Given a set of base data updates in a distributed DW system, we call the order in which a DW receives these data updates* **receive-message-order***. We call the order in which a DW updates its extent* **update-message-order***.*

Lemma 1 *Given a set of views V_i that are view consistent with each other. Then any descendent view V defined directly on top of these consistent views V_i is base consistent and view consistent with V_i, if maintained by a traditional single-DW algorithm.*

In a distributed data warehousing environment, different DWs could have a different receive-message-order and have no knowledge of the receive-message-order of other DWs. But if DWs were to be maintained not based on their respective receive-message-order but rather on the same enforced update-message-order, that means that the DWs are updated and reflect base changes in the same order. Hence DWs are consistent with each other and thus are base consistent.

Lemma 2 *If all DWs in the distributed data warehousing environment are updated by the same update-message-order, then they are all base consistent by Definition 4.*

To generate the unique update-message-order, the key idea we propose is to use the concept of registry service. All DWs and bases are registered with the registry. The registry generates a unique update-message-order and sends it to all DWs in the system. All the DW extents are updated by this update-message-order and hence all DWs are then consistent with each other.

The main purpose of the registry is to generate one unique update-message-order for all DWs. Whenever there is a data update at any base, it sends this data update message annotated with an unique identifier DU-id to its direct descendent DWs. It also sends the DU-id only, which is very short message, to the registry agent. The registry generates one update-message-order based on the order of receiving DU-id from all bases and sends an ordered sequence of DU-ids as the update-message-order to all DWs that have registered with it. DWs receive DUs with DU-id from bases and put them into the received-message-queue (RMQ), while the ordered sequence of DU-id from the registry are put into the update-message-queue (UMQ). All the DWs in the system are updated by the update-message-order instead of the receive-message-order. We assume that a DW knows its related base set and all its direct descendents.

3.2 System Architecture of RyCo Approach

A **r**egistry-based **co**ordinate data warehousing environment is composed of three types of components as depicted in Figure 2. First, the registry agent is used to generate a unique update order. Second, one wrapper at each base sends DU-ids and data updates to related views as well as processes queries. Third, the mediator of each data warehouse maintains the data warehouse based on the updates in the order decided by the registry. Figure 2 depicts the relationships between these three components. As we can see there is one single registry that connects to multiple base wrappers and mediators of different DWs.

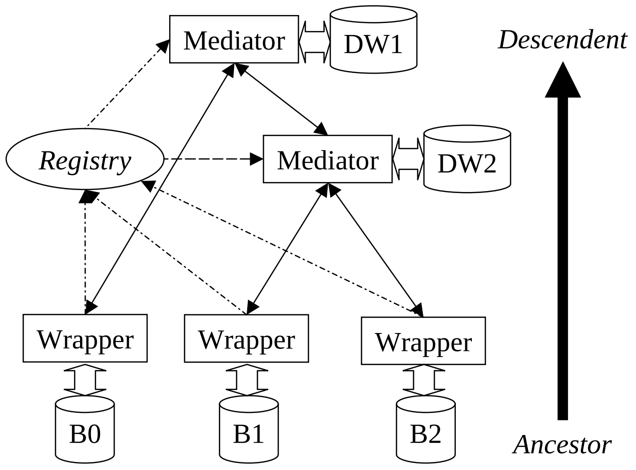

Figure 2: Overall Architecture of Registry System

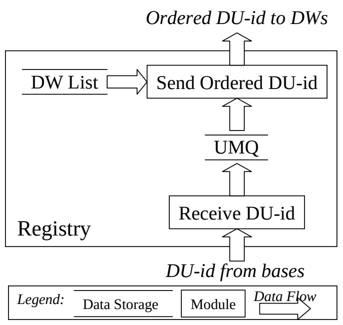

Figure 3: Structure of Registry

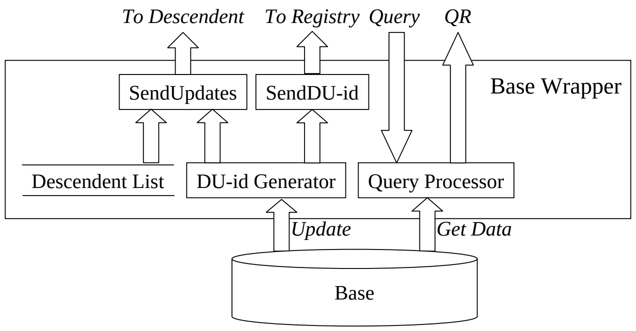

Figure 4: Structure of Base Wrapper

Table 2 gives out all the notations used in the description of detailed architecture of registry agent, base wrapper, and DW mediator.

Notation	**Meaning**
Query	Maintenance query.
QR	Query result.
RMQ	Receive-Message-Queue in DW for buffering DU message from its parent.
UMQ	Update-Message-Queue in DW for buffering ordered DU-ids from registry.

Table 2: Notation and Meaning

Figure 3 shows the simple structure of the agent. The registry keeps the information of registered DWs in the DW list. It can further add or remove a DW from its DW list when a DW is added or removed from DDWE. It receives DU-id from all bases and orders those IDs based on their receive order and then forwards those IDs in that unique order to all DWs in the system. The bases only send a DU-id to the registry, as there is no need to send real update messages. The base will still send data update messages with DU-ids to o all their direct descendent DWs as before.

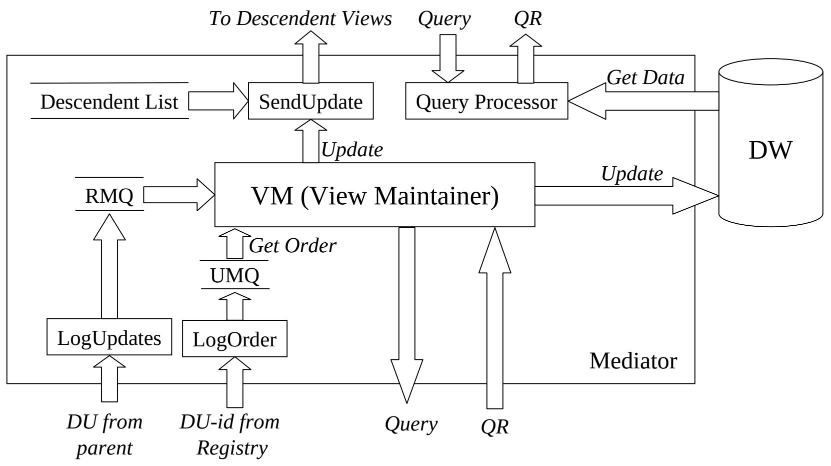

Figure 5: Structure of DW Mediator

Based on Figure 5, we can see that the structure of a DW mediator is more complicated than that of centralized DW systems. A DW mediator in the distributed data warehousing environment mainly consists of the VM (View Maintainer) and the Query Processor. The VM (as shown in Figure 5) has the same function as in centralized DW systems. The VM processes one update at a time and updates the DW view to be consistent with the base after this update. Any incremental view maintenance algorithm from the literature, e.g., SWEEP [AASY97] could be employed as VM here with minor modification. One such modification of VM is that the new extent is computed based on the order from the update-message-order.

A DW mediators (in Figure 5) in the system has two queues, namely the receive-message-queue (RMQ) and update-message-queue (UMQ). The RMQ is for buffering update messages received from its parents in the receive-message-order. The UMQ is responsible for buffering ordered DU-id messages from the registry. Views are maintained and updated in the order of the UMQ as follows. DU-ids coming from the base are appended to the tail and the registry keeps on taking out DU-id from head of the queue to submit to the DWs in the system. First, VM removes the next DU-id from the head of the UMQ and checks whether the DU-id is related to this view. Second, if the DU-id is not related, VM will send a empty update [3] with same DU-id to all direct descendent DWs; otherwise, VM will wait for all updates with this DU-id from all its parents, then incrementally calculate ΔV for those data updates. Third, the VM updates the view with ΔV and sends it with the same DU-id to all direct descendent DWs.

Because the registry will send out all DU-ids to all the views, some of them may not be related to the views. When a view manager gets a DU-id that is not from its related base, it simply sends a NULL extent with the DU-id to its direct descendent. When a DU-id arrives from a base that is related to the view, then the view manager first calculates ΔV and then updates the view by $V + \Delta V$. It also sends ΔV with the same DU-id to its direct descendent. The Query Processor component is used to process any query sent

[3]That means no update for this specific base DU.

from its children. Its functionality is similar to an IS query processor in centralized DW systems.

The structure of the IS wrapper is shown in Figure 4. It consists of the Query Processor and the DU-id generator. The query processor has the same function as the processer of an IS wrapper in centralized DW systems. The DU-id generator will generate a unique identifier for each data update.

We can observe that, using the registry, all DWs in our distributed data warehousing environment incorporate the DUs in the same order. Hence they are consistent with each other and all views are base consistent.

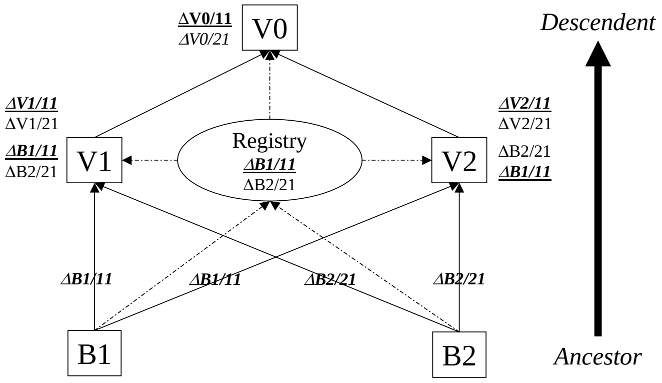

Figure 6: Registry is Used for View Maintenance

The example shows how the registry is used to enable view maintenance in distributed data warehousing environments.

Example 4 *Based on Example 1, data updates* $\Delta B_1/11$ *and* $\Delta B_2/21$ *arrive at the views* V_1 *and* V_2 *in different orders. If* V_1 *and* V_2 *were updated by this receive-message-order, then the two views would not be consistent with each other. Now, let's add the registry into the system as depicted in Figure 6. Whenever a data update happens, the DU-id of the data update is sent to the registry. Assume that the registry receives DU-ids in the order of* 11 *and* 21*. Then the registry forwards this unique update-message-order to all views, namely* V_1, V_2 *and* V_0*. The new view extents of* V_1 *and* V_2 *are calculated and updated in the update-message-order. Then* $\Delta V_1/11$, $\Delta V_1/21$, $\Delta V_2/11$, *and* $\Delta V_2/21$ *are sent to the view* V_0 *in the order of* 11, 21. $\Delta V_1/11$ *and* $\Delta V_2/11$ *only have the effect of the data update* $\Delta B_1/11$. $\Delta V_1/21$ *and* $\Delta V_2/21$ *have the effect of both* $\Delta B_1/11$ *and* $\Delta B_2/21$*. The view* V_0 *is then updated in the same update-message-order. That is, it first incorporates the effect of* $\Delta B_1/11$ *based on* $\Delta V_1/11$ *and* $\Delta V_2/11$ *, then incorporates the effect of* $\Delta B_2/21$ *based on* $\Delta V_2/21$, $\Delta V_2/21$*. Because views* V_1, V_2 *and* V_0 *are updated by the same update-message-order generated by the registry, the views are consistent with each other and thus also are base consistent.*

3.3 Algorithm of Registry Approach

Based on the previous discussion, we now present an algorithm for incremental view maintenance based on the registry. In general, there are three modules called BaseWrapper, Registry and Mediator. The pseudo-code of base wrapper and registry are described in Figures 7 and 8, respectively. The base wrapper has two functions, i.e., sending data updates and processing queries. The registry agent will order the DU ids received and broadcast to all the DWs registered in it.

```
MODULE BaseWrapper
  CONSTANT
    BaseIndex = i;

  PROCESS SendUpdates;
  BEGIN
    LOOP
      RECEIVE $\Delta$ R from R;
      GenID for $\Delta$ R;
      SEND ($\Delta$ R, DUid) TO AllDirectDesDWS;
      SEND (DUid) TO Registry;
    FOREVER;
  END SendUpdates;

  PROCESS ProcessQuery;
  BEGIN
    LOOP
      RECEIVE ($\Delta V$, $DWIndex$) FROM DirectDesDWS ;
      $\Delta V$ = ComputeJoin($\Delta V$, R);
      SEND ($\Delta V$, $BaseIndex$, $DWIndex$) TO DirectDes-
DWS;
    FOREVER;
  END ProcessQuery;

BEGIN /* Initialization */
  StartProcess(SendUpdates);
  StartProcess(ProcessQuery);
END
```

Figure 7: Pseudo Code of Module Base Update and Query

```
MODULE Registry
  PROCESS GetUpdateIds
  BEGIN
    LOOP
      RECEIVE DU-id from Base;
      APPEND DU-id to UMQ;
    FOREVER
  END GetUpdateIds.

  PROCESS SendIds
  BEGIN
    LOOP
      REMOVE DU-id from UMQ;
      SEND to all DWs;
    FOREVER
  END SENDIds.
  END

BEGIN
  StartProcess(GetUpdateIds);
  StartProcess(SendIds);
END
```

Figure 8: Pseudo Code of Registry Module

Figure 9 depicts the software module that is employed at the DW mediator. At initialization, the Data Warehouse module will start four processes: *LogUpdates*, *LogUpdateIds*, *UpdateView* and *Query Processor*. The *LogUpdates* and *LogUpdateIds* processes assign a unique local timestamp to the messages coming into the DW including data updates and query results. The *ViewChange* process will be called by the *UpdateView* to calculate the effect of one data update on the view extent. It uses a local compensation technique to remove concurrent data update effects [AASY97]. Besides view maintenance, a DW needs a *Query Processor* to answer queries from its direct descendent DWs in the distributed data warehousing. The query processor accepts any query coming from its direct descendents and sends back the query result.

The *UpdateView* process monitors the UMQ to check if there is any DU-id in the UMQ. If there is any, the *UpdateView* process will remove the first DU-id from the UMQ, say kj, and then wait for all the update messages from its parents with DU-id kj, i.e., the $\Delta S_i/jk$ $(1 \leq i \leq n)$ [4]. The VM will calculate ΔV by

[4]If a B_j is not a base-ancestor of S_i, then $\Delta S_i/jk$ will be an empty message

adding up every ΔV_i $(1 \leq i \leq n)$ that is calculating from every $\Delta S_i/jk$ removed from the UMQ. The order in which we compute the $\Delta S_i/jk$ (i from 1 to n) will not affect the correctness of the result, hence we can compute them in a different order, or even in parallel. After calculated ΔV, VM will update the DW with ΔV and also send it to all its direct descendent DWs if any.

```
MODULE Mediator
  CONSTANT
    ViewIndex = i;      n: INTEGER /* Total Number of Parents */
  GLOBAL DATA
    V: RELATION; /* Initialized to the correct view */
    UpdateMessageQueue: QUEUE initially 0;
    ReceiveMessageSet: Set initially 0;
    TempArray: RELATION /* Array size is n,
      used for $\Delta S$ from Parent */;
    DeltaAray: RELATION /* Array size is n, used for $\Delta V$ */;

  PROCESS ViewChange($\Delta S$: Relation; DU-id:INTEGER;
    UpdateSource:INTEGER; TimeStamp: INTEGER): RELATION
  VAR
    $\Delta V$, TempView: RELATION;
    j: INTEGER; /* loop variable */
  BEGIN
    $\Delta V = \Delta S$;
    /* Compute the left part of the incremental
    view resulting for $\Delta R$ */
    FOR ($j = UpdateSource - 1$; j $\geq$ 1; $j - -$) DO
      TempView = $\Delta V$;
      SEND ($\Delta V$, ViewIndex) TO Parent j;
      /* The $\Delta V$ in the next line has
      already time stamp assigned by AssignTimeStamp
      process */
      RECEIVE $\Delta V$ FROM Parent j;
      /* Remove the error due to concurrent update
      if any ( maybe more than one ) by local compensation*/
      FOR ALL $\Delta S$ from UpdateSource DO
        $\Delta V = \Delta V - \Delta S \bowtie$ TempView;
      ENDFOR;
    ENDFOR;
    /* Compute the right part to the incremental
    view resulting from $\Delta S$ */
    FOR ($j = UpdateSource + 1$; j $\leq$ n; $j + +$) DO
      TempView = $\Delta V$;
      SEND ($\Delta V$, ViewIndex) TO Parent j;
      /* The $\Delta V$ in the next line has
      already time stamp assigned by AssignTimeStamp
      process */
      RECEIVE $\Delta V$ FROM Parent j;
      /* Remove the error due to concurrent update
      if any ( maybe more than one ) by local compensation*/
      FOR ALL $\Delta S$ from UpdateSource DO
        $\Delta V = \Delta V - \Delta S \bowtie$ TempView;
      ENDFOR;
    ENDFOR;
    RETURN $\Delta V$;
  END ViewChange;

  PROCESS ProcessQuery;
  BEGIN
    LOOP
      RECEIVE ($\Delta V$, index) FROM DirectDesDWS ;
      $\Delta V$ = ComputeJoin($\Delta V$, S);
      SEND $\Delta V$ TO DirectDesDWS $index$;
    FOREVER;
  END ProcessQuery;

  PROCESS LogUpdates;
  VAR
    t: TIME; /* current system time at the DW */
  BEGIN
    LOOP
      RECEIVE Message FROM ParentSource i
        as received order;
        IF Message is ($\Delta S$, ID) THEN
          $t = getCurrentTime()$;
          APPEND ($\Delta S$, ID, t) TO ReceiveMessageSet;
        ELSE /* it is query result $\Delta V$
          Assign $getCurrentTime()$ to $\Delta V$
        ENDIF
    FOREVER;
  END LogUpdates;

  PROCESS LogUpdateIds;
  BEGIN
    LOOP
      RECEIVE UpdateId FROM Registry
        as received order;
        APPEND $UpdateId$ TO UpdateMessageQueue;
    FOREVER;
  END LogUpdateIds;

  PROCESS UpdateView;
    VARIABLE Id:INTEGER;
    UpdateSource:INTEGER;
    BEGIN
      LOOP
        GetOrdered Id from UMQ;
        FOR ($i = 1$; i $\leq$ n; $i + +$) DO
          Remove ($\Delta S_i$,Id,t) from RMQ;
          IF ($\Delta S$,Id) is $NULL$ THEN
            $\Delta V$ is $NULL$;
          ELSE
            BEGIN
              $TempArray[i]$ = ($\Delta S$,Id);
              $\Delta V_i$ = ViewChange($\Delta S_i$,Id, i,t);
              $DeltaArray[i] = \Delta V_i$;
            END
        ENDFOR
        $\Delta V = 0$;
        FOR (i = 1; i $\leq$ n; i++) DO
          $\Delta V = DeltaArray[i] + \Delta V$;
        ENDFOR;
        V = V + ($\Delta V$);
        SendUpdate( $\Delta V$, Id) to AllDirectDesDWS;
        Clean TempArray;
        Clean DeltaArray;
      FOREVER
    END;
  END UpdateView;

BEGIN /* Start DataWarehouse Processes */
  StartProcess(LogUpdates);
  StartProcess(LogUpdateIds);
  StartProcess(UpdateView);
  StartProcess(ProcessQuery);
END DataWarehouse
```

Figure 9: Pseudo Code of Data Warehouse Mediator Module

4 Partitioning for Scalable Distributed Data Warehousing Maintenance

We may have many bases in a distributed data warehousing environment. In our registry approach, views are maintained in the update-message-order generated by the registry. The update-message-order orders all the data updates of all bases even if the bases are not referenced by all the views. If a view only relates to several but not necessarily all bases, it has to unnecessarily maintain extra information, e.g., update message from un-related bases, that later turn out to be irrelevant to the view. If most of the views in such a large-scale system depend on only a subset of all possible bases, then a lot of irrelevant DU-id messages are received from the registry, stored and then checked in the different view mediators. This is likely to lead to low system performance.

To make the system more scalable in terms of the number of bases and views in the system, we now propose an optimization technique called PyCo divides the system into different clustered groups, called DW groups (DWG). Each group then is quipped with its own dedicated registry. Figure 10 shows an example of the system with a partition into three DW groups. We can see the views in each group are closely related to each other, while views in different groups are not.

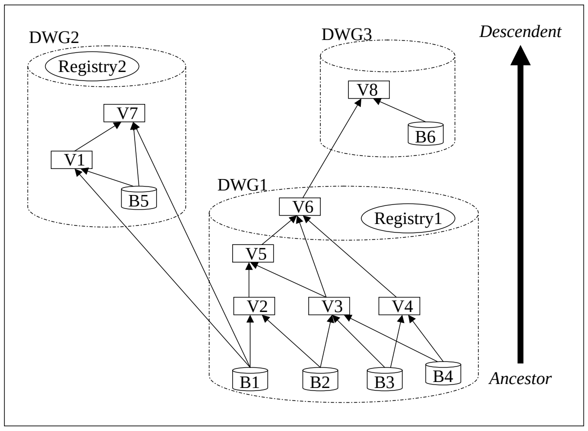

Figure 10: A Partition Hierarchical DW System

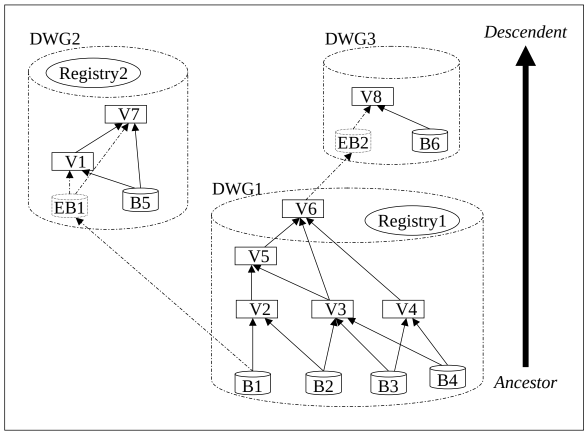

Figure 11: A Partition with New Base Notation

4.1 DW Group

A DW group is composed of a subset of the views and bases available in the system with an optional registry. We call a base which is inside a DW group an **internal-base**. When one DWG_1 group has a parent in another DWG_2 group, we treat this parent as a base of the DWG_1 and call it an **external-base** of the DWG_1. We call the bases of a DW group, which include both internal-bases and external-bases, **group-base**s. If a group base is not a view, we call it a **real-base**. If a group base is a view, we call it **virtual-base**.

All group-bases need to be registered with its group registry. If a DW group DWG_1 has n different parents in other groups then these n parents are treated as n separate external-bases. If a DW group has two views that have the same parents in another group, then they are treated as one external-base of this group.

Example 5 *Based on our definition, we now present in Figure 11 a variation of Figure 10 using the new notation. The partition has three DW groups $DWG1$, $DWG2$ and $DWG3$. $DWG1$ has 4 internal-bases. $DWG2$ has one internal-base B_5 and one external-base $EB1$. The views V_1 and V_7 in $DWG2$ both have parents in DWG1 but they are from the same base B_1. We treat their parent as one external-base shown* *as $EB1$. $DWG3$ also has one external-base and one internal-base. We notice that there is no registry in DWG3 because there is only one view in this group. Any group with one view will always be consistent with its bases.*

DW groups are used to help scale the registry solution. Hence all the views in one DW group have to be consistent with each other. They also have to be consistent with their bases. We call such a group a valid DW group as defined in Definition 8.

Definition 7 *If two views or bases have no common ancestors, then these two views or bases are said to be* **independent**. *Otherwise, they are* **dependent**.

Lemma 3 *If the direct parent(s) of two views or bases are independent, then these two views or bases are independent.*

Definition 8 *We say a DW group is* **valid** *if all group-bases are independent.*

Lemma 4 *All views in a valid group are base consistent.*

If group-bases of a DW group G are independent, the group-bases have no common base-ancestors. Their updated order from group-bases is the same with that of their base-ancestors. Views in group G are updated in the same order of the base-ancestors order. Hence, Views in G are base consistent. Theorem 1 can be easily derived from Lemma 4.

Theorem 1 *Assume a view V has parents $S_1, ..., S_m$ which has common bases $B_1, ..., B_k$. Then view V, parents $S_1, ..., S_m$ and bases $B_1, ..., B_k$ have to be put into one DW group G in order to make V base consistent (necessary condition) but the group G may not be a valid group yet (not necessarily sufficient condition).*

4.2 Partition Validation

To make our system more scalable, we propose to decompose our environment into multiple DW groups, called a partition. With such a partition, we still keep the DWs consistent with each other while we reduce the unrelated and empty messages sent by the registry and handled in each DW group. To ensure view consistency, we define the notion of a valid partition.

Definition 9 *Given a distributed data warehousing environment composed of bases and DWs, then a* **partition** *is a collection of DW groups such that views in different DW groups are disjoint and the union of all DW groups is covering all bases and DWs. We say a partition of P is* **valid** *if all DW groups in P are valid by Definition 8.*

Lemma 5 *If a partition is valid, then all views in the environment can be maintained base consistent as defined by Definition 4.*

All DW groups are valid in a valid partition. Hence, views in each DW group are base consistent by Definition 8. Hence, all the views in a valid partition are base consistent.

There is no any central registry in the partition, except for those created for a specific DW group. In a valid partition, DW groups have the following relationships:

1. If a group has multiple external-bases which reside in different groups, these external-bases are independent.
2. One group can have multiple direct descendents in other groups.
3. A group DWG_i can have only either parent(s) or descendent(s) in another group DWG_j, but not both, because the parent(s) and descendent(s) are dependent by Definition 7.
4. Each group has at least one group-base, which could be either an internal-base or an external-base.

Lemma 6 *Assume a group G has parents $GB_1, ..., GB_m$ and a subset of parents $GB_{s_1}, ..., GB_{s_k}$ (here $k <= m$) are dependent. If $GB_{s_1}, ..., GB_{s_k}$ are in valid DW groups $G_1, ..., G_j$ (Here some or all of $G_1, ..., G_j$ could be the same), the group G and groups $G_1, ..., G_j$ must be merged into one big DW group G' in order to generate a valid DW group (necessary condition) but the group G' may not be a valid group yet (not necessarily sufficient condition).*

The Lemma 6 can be easily proved by Theorem 1.

4.3 Atomic Group

The notion of a valid partition ensures view consistency in the distributed DW environment. There is not necessarily one unique valid partition. In this section, we hence introduce the concept of an atomic group that leads to a unique partition.

Definition 10 *An* **atomic group** *is a valid DW group that cannot be split into more than one valid group.*

Definition 10 states that an atomic group G is the smallest valid DW group. Thus it would no longer be a valid DW group if we remove any base or DW from G. It can easily be shown that a group which contains only one real-base is an atomic DW group. And also a group which contains one view defined only on bases is an atomic DW group.

Lemma 7 *If an atomic group G consists of views or bases $S_1, S_2, ..., S_n$, we cannot construct another atomic group by replacing one of the S_i (i from 1 to n) with another S' with S' being another view or base from the environment.*

Lemma 7 states that no portion of an atomic group is replaceable. For example, if an atomic group G consists of views V_1, V_2 and V_3, then the bases of V_1, V_2 and V_3 must be dependent. Assume that another atomic group G' consists of V_1, V_2 and V_5, then the bases of V_1, V_2 and V_5 must also be dependent. Hence, the bases of V_1, V_2, V_3 and V_5 are dependent. Then views V_1, V_2, V_3 and V_5 must be in the same group. Hence, the group containing V_1, V_2 and V_3 was not an atomic group to begin with.

Theorem 2 *For a given system, the partition is valid, unique and maximal in its number of DW groups if a partition consists of only atomic groups.*

Proof

1. The partition is a *valid* partition if it consists of atomic groups.

   By Definition 10, all atomic groups are valid DW groups. Then by Definition 9, this partition is valid because it consists of valid DW groups.

2. This partition is *unique* by contradiction.

   Assume that this partition is not unique. Then there are at least two different partitions P_1 and P_2 that are composed of atomic groups. Because P_1 and P_2 are not identical, then there is at least one atomic group G in P_1 which is different from any atomic group G' in P_2. Hence without loss of generality there is at least one view in G which is not in G'. But the atomic group is minimal and not-replaceable by Definition 10 and Lemma 7. This implies that there is no overlap between two atomic groups. Hence, either G or G' is not an atomic group. This is contradiction. So, the partition is unique.

3. All the DW groups are atomic groups that are smallest valid groups. Hence, this partition has the *maximal* number of DW groups.

4.4 PyCo Algorithm of Atomic Groups in DDWE

Definition 11 *Based on the steps involved in a view definition, we assign all bases and views in the system a level. A level of a base or views is defined as:*

$$level(S) = \begin{cases} 0 & : \ if\, S\, is\, a\, base \\ 1 & : \ if\, S\, is\, a\, view\, which\, is\, defined\, only\, on\, bases \\ max((level(S1)), ..., (level(Sn))) + 1 & : \ if\, S\, is\, defined\, by\, Equation\, 3 \end{cases}$$

Before presenting the partitioning algorithm, we define the data structures used in the algorithm:

1. **Base Dependency List** (BDL) of a base B_i includes all views (include the base itself) that are derived from this real-base. We maintain a base dependency list for each real-base. Hence, if the system has n real-bases, then we generate n base dependency lists.

2. **View Parent List** (VPL) of a view V_i includes all direct parents of V_i. We have one list for each view. Hence, if the system has m views, then we generate m view parent lists.

3. **Group-Base Set** (GB) of a group G includes all the group bases of a group.

4. **Dependent Group-Base Set** (DGB) of a group G includes all group bases of G that are dependent.

5. **Group Set** (GS) of a given distributed DW environment includes all valid groups generated by the partitioning algorithm.

6. **Un-group Set** (UGS) of a given distributed DW environment includes all bases and views that have not yet been partitioned.

We now present the partitioning algorithm (PyCo)in Figure 12. First, the algorithm calls *InitProc* to initialize the system. The *InitProc* generates a base dependency list for each base and a view parent list for each view in the system based on the view definition. It assigns a level to all the bases and views according to Definition 11. If the un-group set (UGS) is empty, we are done with the partition. Otherwise, we pick a view V with the lowest view level from UGS, and then use the *CheckGroup* and *MergeGroup* processes to generate a valid group which has V in it.

CheckGroup process checks whether a given group G is valid group or not by Definition 8. *MergeGroup* process will merge a invalid group G' with other groups by Lemma 6 until the group becomes valid. Because we generate a valid group starting from the lower level views and moving up to higher level views, all merging will happen in the group set GS.

Theorem 3 *The partitioning algorithm in Figure 12 generates only atomic DW groups.*

Proof Assume N is the total number of bases or views in the group set GS. We use a proof by induction over N.

Base: With $N = 1$. The only case is that there is one base. Obviously this is an atomic group.

With $N = 2$. There are two possibilities: both are bases or one is a base and the other a view in the GS. They both are atomic groups.

Hypothesis: Assume with $N = n$ bases and/or views in GS, all groups in GS are atomic groups.

Induction Step: With $N = n + 1$, we would put one more base and/or view S_i from UGS to GS using the partitioning algorithm depicted in Figure 12, and all the groups in GS are only atomic groups.

Case1: If S_i is a base or a view with independent parents then this S_i is an atomic group itself. Done.

Case2: If S_i is a view, say V, that has dependent parents, we show that GS still continues to contain only atomic groups after V has been placed into GS by our algorithm from Figure 12.

Assume the dependent parents of V are V_1, ..., V_m. The algorithm forms groups from the lower level to the higher level. Hence, V_1, ..., V_m have already been assigned into groups and they all belong to atomic

```
MODULE PartitionGen;
  CONSTANT
    TotalView = m;
    TotalBase = n;
  GLOBAL DATA STRUCTURE
    GB: Group Base Set.
    DGB: Dependent Group List.
    BDL: Base Dependent List.
    VPL: View Parent List.
    GS: Group Set.
    UGS: Ungroup Set.
  VAR
    g = 0; /*total group number*/
    r = n; /*total views left in UGP*/
    LV: View Level.

  FUNCTION CheckGroup(Group G): Boolean
  VAR
    i,j,k,: INTEGER;
  BEGIN
    $GB = \{GB_1, ..., GB_{m'}\}$;
    FOR (i = 1; j $\leq$ m; $j++$) DO
      j = 0;
      FOR (k = 1; k $\leq$ m'; $k++$) DO
        IF $GB_1$ belong $BDL_i$
          THEN $j++$
        IF j $\geq$ 1
          THEN dep = 1;
      ENDFOR
    ENDFOR
    IF dep = 0, THEN return TRUE.
    ELSE return FALSE.
  END CheckGroup.

  PROCESS MergeGroup(Group G);
  BEGIN
    $GB = \{GB_1, ..., GB_{m'}\}$;
    Find dependent group base $DGB = \{GB_1, ...GB_k\}$.
    FOR (i = 1; i $\leq$ k; $i++$)
      IF $GB_i$ belong to $G_j$
        $G = G$ UNION $G_j$ /* merge group */
        $GB = GB$ UNION $GB_j$
      ENDIF
    ENDFOR
    IF CheckGroup(Group G) is TRUE
      THEN RETURN /* Get a valid group*/
      ELSE MergeGroup(Group G)
    ENDIF
  END MergeGroup;

  PROCESS AssViewLevel()
  BEGIN
    FOR (i = 1; i $\leq$ m;i++)
      LV($B_i$) = 0;
    ENDFOR.
    FOR (i = 1; i $\leq$ n; i++)
      IF VPL($V_i$) include only bases
        LV($V_i$) = 1;
      ELSE
        LV($V_i$) = -1
      ENDIF
    ENDFOR
    FOR (i = 1; i $\leq$ n; i++)
      IF (LV($V_i$) $\leq$ 0)
        LV($V_i$) = ViewLevel($V_i$)
      ENDIF
    ENDFOR
  END AssViewLevel

  FUNCTION ViewLevel($V$)
  BEGIN
    temp = VPL($V_i$)/*get parent list of $V_i$*/
    WHILE (temp not empty)
      Remove an element $S$ from temp
      IF (LV($S$) = -1)
        LV($S$) = ViewLevel($S$)
      ENDIF
      IF (LV($V$) $\leq$ LV($S$))
        LV($V$) = LV($S$)
      ENDIF
    ENDWHILE
    LV($V$) = LV($V$)+1
    RETURN LV($V$)
  END ViewLevel

  PROCESS InitProc()
  BEGIN
    /*Initialize BDL and VPL base on the view definition */
    Compute BDL for each base;
    Compute VPL for each view;
    AssViewLevel();
    Put all bases and views into $UGP$.
    $GP$ = empty.
    FOR (i = 1; i $\leq$ m; i++) /*Move all bases to GP*/
      Move $B_i$ from $UGP$ to $GP$;
      g++;
    ENDFOR;
      FOR (i = 1; i $\leq$ n; i++)          /* Move all view with level 1 to
GP*/
        IF (LV($V$)==1)
          Move view $V$ from $VGP$ to $GP$.
          r–; /* total view left in UGP */
          g++;
        ENDIF
      ENDFOR
  END InitProc

BEGIN /* Start PartitionGen Processes */
  InitProc().
  $i$ = 2;/*lowest view level in UGP */
  WHILE (UGP not empty)
    FOR (i = 1; i $\leq$ r; i++)
      IF (LV($V$) == $i$).
        THEN Move $V$ from $UGP$
        IF (Check($V$)==TRUE)
          Put $V$ into $GP$.
        ELSE
          MergeGroup($V$).
        ENDIF
      ENDIF
    ENDFOR
    i++;
  ENDWHILE
END PartitionGen
```

Figure 12: Pseudo Code of Partition Generation Algorithm

groups in GS. When we merge V with some of these groups to generate a valid group, all merging happens inside GS. This means the total number of bases and views N in GS is not changed.

Because the parents of V are dependent, the partitioning algorithm merges V with V_1, ..., V_m resulting in a new group G'. Then *CheckGroup* checks whether G' is valid or not. If G' is not a valid group, then there is another round of merging until we get a valid group G that contains the view V. If G consists of atomic groups G_1, ..., G_k (Here $k <= n$), we can show that the group G is an atomic group:

Assume group G is not an atomic group. Then we can split the group G into a set of smaller valid groups. There are three possible ways to split this group G. We discuss each case below:

- If we take V out of G, then V itself is not a valid group by assumption. Done.
- If we take any view V' from its atomic group G_i then called G_i', then G_i' is not valid because an atomic group is minimal by Definition 10. Done
- If we take any atomic group G_i from G, then G becomes G' without containing G_i. Obviously, G' is

not a valid group because the bases of G' and G_i must be dependent otherwise they would not have been placed into one group G by the algorithm (see Figure 12). Done.

Based on the above discussion, we cannot divide G into smaller valid groups. Thus G is an atomic group by Definition 10.

We have proven that the group set contains only atomic groups after the $(n+1)th$ base or view S_{n+1} is put into GS by our partitioning algorithm. Thus the partition generated by PyCo consists of only atomic groups based on the above proof.

The example below illustrates the main ideas of the PyCo algorithm.

Example 6 *We use the view definition from Figure 10.*

1. **Initialization:**

   - *Base dependency lists:*

     $BDL(B_1) = \{B_1, V_1, V_2, V_5, V_6, V_7, V_8\}$,$BDL(B_2) = \{B_2, V_2, V_3, V_5, V_6, V_8\}$,

     $BDL(B_3) = \{B_3, V_3, V_4, V_5, V_6, V_8\}$, $BDL(B_4) = \{B_4, V_3, V_4, V_5, V_6, V_8\}$,

     $BDL(B_5) = \{B_5, V_1, V_7\}$,$BDL(B_6) = \{B_6, V_8\}$.

     View parent lists:

     $VPL(V_1) = \{B_1, B_5\}$,$VPL(V_2) = \{B_1, B_2\}$,$VPL(V_3) = \{B_2, B_3, B_4\}$, $VPL(V_4) = \{B_3, B_4\}$, $VPL(V_5) = \{V_2, V_3\}$,$VPL(V_6) = \{V_3, V_4, V_5\}$,$VPL(V_7) = \{B_1, B_5, V_1\}$, $VPL(V_8) = \{B_6, V_6\}$.

   - *Assign a level to each view and base. B_1, ..., B_6 have level 0, V_1, V_2, V_3, V_4 have level 1, V_5, V_7 have level 2, V_6 has level 3, and V_8 has level 4.*

   After initialization, GS has all bases B_1, ..., B_6 and all views with level 1, namely V_1, V_2, V_3, V_4, with each base or view in a separate group.

2. *Next, PyCo takes out a view, say V_5, with the lowest level 2 from UGS. The parent list $VPL(V_5)$ of V_5 includes V_2 and V_3, which are in $BDL(B_2)$. Hence, V_2 and V_3 are dependent. PyCo merges V_5 with the valid groups $G_2 = \{V_2\}$ and $G_3 = \{V_3\}$. This results in a new group $G' = \{V_5, V_2, V_3\}$. The new group-bases $GB(G') = \{B_1, B_2, B_3, B_4\}$ are independent. So we get a valid group G' that contains V_5 in GS.*

3. *Next, PyCo takes out another view with the lowest still available level from UGS (in our case, this is level 2) and repeats the same steps as in step 2 for V_5 until UGS becomes empty.*

4. *When UGS is empty, then the PyCo algorithm terminates with the result shown in Figure 13*

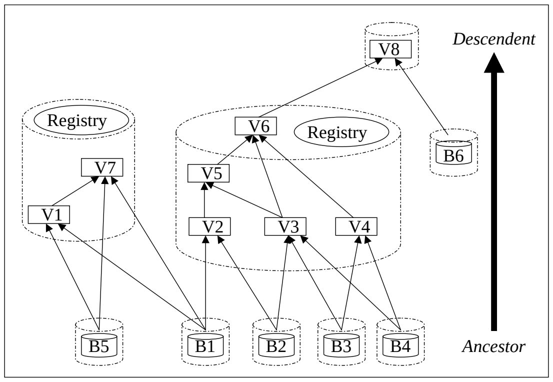

Figure 13: A Partition by Using Partition Algorithm

5 Preliminary Evaluation Using a Cost Model

The cost of updating a view is dependent on many factors, such as the VM algorithm employed at each DW, the communication cost of the network, the size of the data files shipped between DWs, the topology of the DDWE in terms of the DWs and their inter dependencies. In this paper, we have proposed two methods: registry and partitioning. The VM algorithm and the size of the view extents are the same for both the registry and the partitioned-registry approach. We hence focus on the remaining costs which are the communication cost and the update time cost. To simplify the problem, we assume that the communication cost between any pair of sites is fixed for a unit of transmission and that there are no capacity constraints for either the sites or the communication links.

5.1 Message Communication Cost Evaluation

There are four kinds of communication messages in the registry-based distributed data warehousing environment, namely:

1. Data update identifiers (DU-ids) send from bases to the registry. The total number is denoted by M_r.
2. Ordered DU-ids from the registry to all views. The total number is denoted by M_v.
3. DUs from bases or views to their direct descendents. The total number is denoted by M_d.
4. Queries and query results between views and their parents to compute the new view extent after receiving the DU-id from the registry and the corresponding DUs from the real-bases through their parents. The total number is denoted by M_q.

So, the total message communication cost M is: $M = M_r + M_v + M_d + M_q$.

M_d and M_q both are the same for both the registry and partitioning methods. M_r and M_v may be different for our two methods. Hence, we focus on those next.

Assume a DDWE consists of m bases and n views. We discuss two cases. First, we assume data updates are distributed evenly and each base has k DUs. Second, we generalize that each base B_i has k_i data updates. The total number of data updates is then denoted by $K = \sum_{i=1}^{m} k_i$.

Message Cost of RyCo

Case 1: The total message costs are: $M_r = k * m$ and $M_v = k * m * n$.

Case 2: The total message costs are: $M_r = K = \sum_{i=1}^{m} k_i$ and $M_v = K * n$.

Message Cost of PyCo

Assume the system has a valid partition with g DW groups and each DW group has m_i real-bases and n_i views. The total real-bases and views are the same as in the registry approach, i.e., m and n. To discuss the message cost, we further assume that each group has f_i DUs that come from external bases.

Case 1: The total message costs are:

$M_r = k * m_1 + ... + k * m_g + f_1 + ... + f_g >= k * m + \sum_{i=1}^{g} f_i$

$M_v = \sum_{i=1}^{g} (k * m_i + f_i) * n_i$

Case 2: Assume k_{ij} denotes the data update from real-bases j in the group i. The total message cost is:

$M_r = \sum_{i=1}^{g} (f_i + \sum_{i=1}^{m_i} k_{ij}) >= K + \sum_{i=1}^{g} f_i$

$M_v = \sum_{i=1}^{g} (f_i + \sum_{i=1}^{m_i} k_{ij}) * n_i$

Comparing Message Costs With and Without Partitioning

Comparing the message exchanges for the registry and the partition.

1. Comparing the messages sent from the real-bases to the registry.

   With partitioning, there are more messages sent to the registry than for the RyCo approach. If the groups have no common real-bases, then there is the same amount of messages sent from real-bases to the registry.

2. Comparing messages sent from the registry to views. We find that there are more messages sent from the registry to the views for RyCo approach.

   If DUs from real-bases are distributed evenly, then the difference of messages, denoted by ΔM, sent from the registry to views with our two methods corresponds to:

$$\begin{aligned}\Delta M &= \sum_{i=1}^{g} (k * m_i + f_i) * n_i - k * m * n \\ &= \sum_{i=1}^{g} (k * m_i + f_i) * n_i - k * m * \sum_{i=1}^{g} n_i \\ &= \sum_{i=1}^{g} n_i * (f_i + k * m_i - k * m) \qquad (6)\end{aligned}$$

The value of $f_i + k * m_i - k * m$ in Formula 6 thus determines ΔM. f_i denotes the number of DUs coming from virtual bases. Because the number of group bases that is not real-bases is at most $m - m_i$, $f_i <= k * (m - m_i)$ based on the partition properties. Hence,

$$\Delta M = \sum_{i=1}^{g} n_i * (f_i + k * m_i - k * m) <= \sum_{i=1}^{g} n_i * (k * (m - m_i) + k * m_i - k * m) = 0$$

The difference in message costs being negative means that the total message cost from the registry to the views is less for algorithm PyCo than algorithm RyCo.

If DUs are not distributed evenly, we still have a result similar to above. Hence this is omitted here.

To summarize, the analysis of the message cost indicates that:

1. For the message cost from bases to the registry, the cost of the PyCo algorithm is larger than the cost of RyCo. But if every DW group has no virtual bases and no common bases, then the two approaches have the same costs.

2. For the message cost from the registry to the views, the cost of PyCo is less than the cost of RyCo. In the worst case when all groups include all the bases of DDWE, both approaches have the same cost.

5.2 Update Time Cost Evaluation

The update time refers to the elapsed time from when a DU was generated by a base and sent out to the DDWE to the completion of the last DW in DDWE updated to reflect the DU. We assume the transmission time between any pair of sites is fixed, denoted as t_t. The time to compute a ΔV is denoted as t_v. The time from when a DU arrives at a view until it is processed by VM is denoted by t_w. t_v and t_w are dependent on the system speed and how the views are defined, etc. Both of which are not constant.

1. With RyCo method, the update time for views after a data update is:

   views at level 1: $T_1 = 2 \times t_t + t_{v1} + t_{w1}$

   views at level 2: $T_2 = T_1 + (t_t + t_{v2}) + t_{w2} = (1 + 1) \times t_t + (t_{v1} + t_{v2}) + (t_{w1} + t_{w2})$

   ....

   views at level i: $T_i = (i + 1) \times t_t + \sum_{k=1}^{i} t_{vk} + \sum_{k=1}^{i} t_{wk}$

   ....

2. In the PyCo approach, the update time is also affected by the groups. Similar to the idea of level of views, there is a level of groups as defined as follows:

   Definition 12 *A level of a group G is:*

$$level(G) = \begin{cases} 0 & : \ if\ G\ has\ no\ external\text{-}bases \\ max((level(G_1)), ..., (level(G_n))) + 1 & : \ if\ G\ has\ external\text{-}bases\ in\ G_1..G_n \end{cases}$$

Hence the update time for a view at level i and in group at level j after a data update is:

For a view at level i and in group at level 0: $T_i = (i+1) * t_t + \sum_{k=1}^{i} t_{vk} + \sum_{k=1}^{i} t_{wk}$

For a view at level i and in group at level 1: $T_i = (i+1) \times t_t + \sum_{k=1}^{i} t_{vk} + \sum_{k=1}^{i} t_{wk} + t_t$

...

For a view at level i and in group at level j: $T_i = (i+1) \times t_t + \sum_{k=1}^{i} t_{vk} + \sum_{k=1}^{i} t_{wk} + j \times t_t$

...

The update time is influenced by t_t, t_v and t_w and is also dependent on the view level in both RyCo and PyCo and group level in PyCo. The lower the view level, the less time is needed to update. With or without partitioning, a view level is the same for a view V, and also t_v directly depends on the VM algorithm, so the difference between the update time of RyCo and PyCo is only the the group level, which decides the delay of the communication between registry and DWs in group at level j, i.e., $j \times t_t$. However, the t_w may be less in PyCo than RyCo, for there are no empty update message that must be passed around in the PyCo approach. When the average group level is low, the PyCo approach is expected to have overall better performance than the RyCo approach.

Summary Giving the above analysis, the PyCo approach has been shown to have an overall better performance than the RyCo approach if there are less or no common group bases between different DW groups. Key advantages of the partitioning approach beyond its performance are its increased scalability and its distributed and robust nature.

6 Conclusion

In this paper, we have primarily concerned with the consistency view maintenance in distributed data warehouse environment. We proposed two algorithms, namely the RyCo and PyCo approach that both handle view maintenance efficiently. Using the cost model, we have shown that the partitioning approach has better performance than registry in terms of the update time and overall message cost from registry to views. All the views in our system are maintained and updated independently according to the notification order from the registry. Our registry-based solution does not need any safe time to refresh a materialized view, as required by the only alternate solution in the literature thus far [SDA99]. With the partitioning algorithm, our system is more scalable and fault-tolerant in the sense of not relying on one single registry agent representing a potential bottleneck in the distributed DW environment.

References

[AASY97] D. Agrawal, A. E. Abbadi, A. Singh, and T. Yurek. Efficient View Maintenance at Data Warehouses. In *Proceedings of SIGMOD*, pages 417–427, 1997.

[CD97] S. Chaudhuri and U. Dayal. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Record*, 26(1):65–74, 1997.

[CKL$^+$96] L. Colby, A. Kawaguchi, D. Lieuwen, I. Mumick, and K. Ross. Supporting Multiple View Maintenance Policies. *AT&T Technical Memo*, 1996.

[DZR99] L. Ding, X. Zhang, and E. A. Rundensteiner. The MRE Wrapper Approach : Enabling Incremental View Maintenance of Data Warehouses Defined On Multi-Relation Information Sources. In *Proceedings of the ACM First International Workshop on Data Warehousing and OLAP (DOLAP'99)*, pages 30–35, November 1999.

[GM96] A. Gupta and I. S. Mumick. What is the data warehousing problem? (Are materialized views the answer?). In *International Conference on Very Large Data Bases*, page 602, 1996. Panel.

[GMLWZ98] H. García-Molina, W. Labio, J. L. Wiener, and Y. Zhuge. Distributed and Parallel Computing Issues in Data Warehousing . In *Symposium on Principles of Distributed Computing*, page 7, 1998. Abstract.

[KLMR97] A. Kawaguchi, D. F. Lieuwen, I. S. Mumick, and K. A. Ross. Implementing Incremental View Maintenance in Nested Data Models. In *Workshop on Database Programming Languages*, pages 202–221, 1997.

[MKK97] M. K. Mohania, S. Konomi, and Y. Kambayashi. Incremental Maintenance of Materialized Views. In *Database and Expert Systems Applications (DEXA)*, pages 551–560, 1997.

[SDA98] I. Stanoi, D.Agrawal, and A. E. Abbadi. Weak Consistency in Distributed Data Warehouses. In *Proceedings of the International Conference of Foundations o f Data Organization*, November 1998.

[SDA99] I. Stanoi, D.Agrawal, and A. E. Abbadi. Modeling and Maintaining Multi-View Data Warehouses. In *Proceedings of the 18th International Conference on Conc eptual Modeling (ER'99)*, pages 161–175, November 1999.

[WB97] M. Wu and A. P. Buchman. Research Issues in Data Warehousing. In *Datenbanksysteme in Büro, Technik und Wissenschaft*, pages 61–82, 1997.

[ZGMHW95] Y. Zhuge, H. García-Molina, J. Hammer, and J. Widom. View Maintenance in a Warehousing Environment. In *Proceedings of SIGMOD*, pages 316–327, May 1995.

[ZGMW96] Y. Zhuge, H. García-Molina, and J. L. Wiener. The Strobe Algorithms for Multi-Source Warehouse Consistency. In *International Conference on Parallel and Distributed Information Systems*, pages 146–157, December 1996.

[ZR99] X. Zhang and E. A. Rundensteiner. The SDCC Framework for Integrating Existing Algorithms for Diverse Data Warehouse Maintenance Tasks. In *International Database Engineering and Application Symposium*, pages 206–214, Montreal, Canada, August, 1999.

[ZWGM97] Y. Zhuge, J. L. Wiener, and H. García-Molina. Multiple View Consistency for Data Warehousing. In *Proceedings of IEEE International Conference on Data Engineering*, pages 289–300, 1997.