DaWaK2001 PAPER SUBMISSION

PVM: Parallel View Maintenance Under Concurrent Data Updates of Distributed Sources*

Xin Zhang, Elke A. Rundensteiner and Lingli Ding

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

Abstract

Data warehouses (DW) are built by gathering information from distributed information sources (ISs) and integrating it into one customized repository. In recent years, work has begun to address the problem of view maintenance of DWs under concurrent data updates of different ISs. Popular solutions such as ECA and Strobe achieve such concurrent maintenance however with the requirement of quiescence of the ISs. More recently, the SWEEP solution releases this quiescence requirement using a local compensation strategy that now processes all update messages in a sequential manner. To optimize upon this sequential processing, we have developed a parallel view maintenance algorithm, called PVM, that incorporates all benefits of previous maintenance approaches while offering improved performance due to parallelism. In order to perform parallel view maintenance, we have identified two critical issues: (1) detecting maintenance-concurrent data updates in a parallel mode, and (2) correcting the problem that the DW commit order may not correspond to the DW update processing order due to parallel maintenance handling. In this work, we provide solutions to both issues. Given a modular component-based system architecture, we insert a middle-layer timestamp assignment module for detecting maintenance-concurrent data updates without requiring any global clock synchronization. In addition, we introduce the negative counter concept as a simple yet elegant solution to solve the problem of variant orders of committing effects of data updates to the DW. We have proven the correctness of PVM to guarantee that our strategy indeed generates the correct final DW state. We have implemented both SWEEP and PVM in our EVE data warehousing system. Our performance study demonstrates that a multi-fold performance improvement is achieved by PVM over SWEEP.

Keywords: Data Warehousing, Parallel View Maintenance, Concurrent Data Updates, Performance Evaluation.

*This work was supported in part by several grants from NSF, namely, the NSF NYI grant #IRI 97–96264, the NSF CISE Instrumentation grant #IRIS 97–29878, and the NSF grant #IIS 97–32897. Dr. Rundensteiner would like to thank our industrial sponsors, in particular, IBM for the IBM partnership award, and Verizon Laboratories Incorporated for partial support of Xin Zhang.

1 Introduction

Background. Data warehousing [WB97, Wid95, GMLWZ98, GM96] is an important technology to integrate data from distributed 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 the underlying ISs under updates becomes a critical issue. After IS updates, it is typically more practical to maintain the DW incrementally [AASY97, MKK97, KLMR97, GMS93] instead of recomputing the whole extent of the DW due to the large size of DWs and the enormous overhead associated with the DW loading process. Because the DW usually connects with ISs through the network and the network is generally not very stable, it's unacceptable to block the update transaction at an IS in order for the integrator to immediately accomplish the view maintenance process. Instead, the maintenance is typically conducted in a separate asynchronous process, then called deferred view maintenance [CGL$^+$96, KLM$^+$97].

Since deferred view maintenance does not block the update transactions of the ISs, there is a time period during which the extent of the DW will be temporarily inconsistent with that of the ISs. In recent years, there have been a number of algorithms proposed for such deferred view maintenance [WGL$^+$96, MKK97, CM96, QGMW96, GJM97, KLMR97, ZGMW98, NR99]. ECA [ZGMHW95] handles view maintenance under concurrent data updates of one centralized IS, while Strobe [ZGMW96] and SWEEP [AASY97] handle distributed ISs. Any solution that improves the performance of view maintenance is critical, given that it would reduce the time during which the DW lags behind and thus is inconsistent with the ISs. Such optimization would lead to an improvement of the timeliness and hence quality of the DW content, which is important for time-critical applications like stock market and monitoring-control systems. Such performance optimization of DW maintenance is the focus of our work.

Our Approach. In this paper, we propose an efficient algorithm called Parallel View Maintenance, or short PVM, that preserves all advantages of its predecessors, including SWEEP [AASY97], while overcoming their main limitation in terms of performance caused by sequential processing. In particular, SWEEP handles data updates *sequentially* enforcing all data updates to queue at the DW manager until the data updates (DUs) received before them have been fully processed. Such sequential processing time in a DW system build over distributed ISs requires the system responsible for DW processing to wait for the processing of the DBMS server in each IS as well as for transmissions of results or update messages over the network. Although a variation called Nested-SWEEP handles multiple DUs in a more efficient way by reusing part of the query results, it puts more requirements on the DUs, e.g., require the DUs to be non-interfering with each other in order to terminate. We will show that PVM more efficiently handles a set of concurrent data updates than previous solutions by parallelizing the DU handling process.

For this purpose, we have investigated how best to introduce parallelism into view maintenance. We have identified two research issues central to enabling a parallel solution of view maintenance. The first issue is how to detect conflicting concurrent data updates within a parallel execution paradigm. The second issue is how to handle the random commit order of effects of data updates at the DW, which ws call the **Out-of-Order-DW-Commit** order, caused by the parallel execution of the view maintenance process without blocking incoming DUs.

For the first issue, we introduce a middle-layer timestamp service that can be explicitly identify each incoming data update and query result once received by the middle layer and hence to help detect concurrent data updates. For the second issue, we extend the meaning of counters[1] kept for

[1]Here we assume the representation in the DW is unique tuples with associated duplicate counters.

tuples in the DW to also include negative counts. A negative counter now indicates the number of faulty deleted tuples due to the **Out-of-Order-DW-Commit** problem. We keep track of this in order to fix the faulty tuples once the delayed data inserts are detected and need to be compensated for. The correctness of our solution based on negative counters is proven. A full implementation of PVM and the most popular contender in the literature (SWEEP)within our EVE data warehousing management system has been accomplished. Our evaluation based both on a cost model as well as on experimental studies has shown that PVM achieves a many-fold performance improvement over SWEEP.

Contributions. The main contributions of this paper are:

- Identify the performance limitation of the state-of-art view maintenance (VM) [ZGMW96, AASY97] solutions in terms of performance and limitations and characterize research issues to be addressed to achieve parallel execution, in particular, the **maintenance-concurrent** DU detection problem and the **Out-of-Order-DW-Commit** problem.
- Introduce a middle-layer timestamp service and associated algorithms to solve the **maintenance-concurrent** DU detection problem within a parallel execution paradigm.
- Develop a solution to the **Out-of-Order-DW-Commit** problem using the negative counter concept.
- The correctness of the PVM solution can be proven, which then confirms that it always generates the same final DW content as a complete re-computation algorithm. This proof is presented in the technical report only [ZDR99] due to space limitations.
- Design a modular PVM architecture that incorporates the proposed solution techniques based on minimally adapting the state-of-the-art solution (e.g., SWEEP) to achieve parallel view maintenance. This should allow for a quick adaption of PVM by current systems.
- Implement both PVM and a primary contender from the literature namely, the SWEEP, algorithm in a uniform testbed, the EVE data warehousing system [RKZ+99], to allow for fair comparative evaluation. To the best of our knowledge, there is no implementation or experimented performance study of the SWEEP system has been reported in the literature thus far – rather only cost-model based discussion of the algorithm's complexity.
- Conduct a series of experimental studies that assess the performance benefits and limitations of PVM by itself as well as compared to SWEEP [AASY97]. We conclude that PVM has multi-fold performance improvement over SWEEP.

Outline: The following section contains background material needed for this work. In Section 3, the PVM solution, including the two open research issues and their solutions, are introduced. Section 4 presents results of our experimental performance study. In Section 5, related work is described, and conclusions are discussed in Section 6.

2 Background on View Maintenance

2.1 The Data Warehouse Model

A data warehouse (DW) stores integrated information from multiple information sources (ISs) in materialized views. When the data in any IS changes, the materialized views at the DW need to be updated accordingly, which is called view maintenance (VM) process. The VM process is always happened at the program, called middle-layer, between the DW and the ISs.

Consistency. Zhuge et al. [ZGMHW95, ZGMW96] define different notions of consistency of a view depending on how and when IS updates are incorporated into the view at the data warehouse:

- *Convergence:* when all the updates are incorporated into the materialized views and all activities have ceased then the view is consistent with the relations at the information sources.
- *Strong consistency:* when the order of the DW states corresponds to the order of the states of the IS space. Some IS space states may not result in a separate DW state.
- *Complete consistency:* There is a complete order-preserving mapping between all the states of the DW and the states of the information sources.

2.2 Incremental View Maintenance

View maintenance is concerned with maintaining the materialized views up-to-date when the underlying information sources are modified over time. The straightforward approach of recomputing the whole view extent for each data update is not realistic. Instead, an incremental solution that only calculates the effect of each data update from the ISs and updates the data warehouse incrementally is typically more efficient. If two data updates are separated far enough in time so that they do not interfere with each other during maintenance, then this incremental approach is straightforward. In this case, when a data update ΔR_i happened at base relation R_i and is received at the middle layer, the incremental changes are computed by sending down subqueries of the following query [GM95, GMS93] to the respective ISs:

$$\prod_{attributes} \sigma_{predicates}(R_1 \bowtie \ldots \bowtie \Delta R_i \bowtie \ldots \bowtie R_n) \tag{1}$$

To explain the generation and execution order of the subqueries of the Equation 1, we introduced the notations described in Table 1.

Notation	**Meaning**
DU_i	A Data Update with unique subscript i.
Q_i	A Query for handling DU_i.
QR_i	The Query Result of Q_i.
IS_m	An Information Source with unique index m.
$SQ_{i,m}$	A Sub-Query of Q_i being sent to IS_m.
$SQR_{i,m}$	A Sub-Query Result of $SQ_{i,m}$.
$DU_i[t]$	DU_i with Unique Timestamp t.
ΔV_i	ΔV is Effect of DU_i.
$SQ_{i,m}[t]$	$SQ_{i,m}$ with Unique Timestamp t.
$SQR_{i,m}[t]$	$SQR_{i,m}$ with Unique Timestamp t.

Table 1: Notations

The subqueries are generated in the sequence that starts from joining the relations on the left side of ΔR_i, and then continues to join the relations on the right side. Let $DU_j = \Delta R_i$. In the order of the generation sequence, we have [2]:

Scan left:	Scan right:
$SQ_{j,i-1} = \prod \sigma(R_{i-1} \bowtie \Delta R_i)$	$SQ_{j,i+1} = \prod \sigma(SQR_{j,1} \bowtie R_{i+1})$
$SQ_{j,i-2} = \prod \sigma(R_{i-2} \bowtie SQR_{j,i-1})$	$SQ_{j,i+2} = \prod \sigma(SQR_{j,i+1} \bowtie R_{i+2})$
...	...
$SQ_{j,1} = \prod \sigma(R_1 \bowtie SQR_{j,2})$	$SQ_{j,n} = \prod \sigma(SQR_{j,n-1} \bowtie R_n)$

Please notice we don't need to send any subquery to the IS_i that initially had reported the update ΔR_i.

This simple approach will succeed in the case that the updates are spaced far enough in time so to allow for completing the incremental computation of the view before any new update occurs. However, due to the autonomous nature of the information sources participating in the data warehouse, this kind of separation in time cannot always be guaranteed. Hence, we have to maintain the data warehouse under concurrent data updates that are not separated far enough to accomplish the naive incremental maintenance illustrated above.

2.3 View Maintenance under Concurrent Data Updates

Because the data updates are happening concurrently at the ISs, one data update DU_2 at IS_j could interfere with the query processing of a query sent by the middle layer to this IS_j to handle another data update DU_1. We call this data update DU_2 a **maintenance-concurrent** data update as it takes place concurrently with the maintenance process servicing other updates.

Definition 1 *Let DU_i and DU_j denote two DUs on IS_m and IS_n respectively. The data update DU_j is called* **maintenance-concurrent** *with the update DU_i, denoted by $DU_j \vdash DU_i$, iff:*
i) DU_i is received earlier than DU_j at the middle layer, (i.e., $i < j$)
ii) DU_j is received at the DW **before** *the answer of the sub-query $SQ_{j,n}$ on the same IS_n generated by the DW for handling DU_i. An update DU_p is* **maintenance-concurrent***, if there exists at least one DU_q for some $q < p$ such that $DU_p \vdash DU_q$.*

Like previous work [ZGMW96, AASY97], we also make the following assumption to facilitate the definition of **maintenance-concurrent**.

Assumption 1 *The order in which the middle layer receives the messages from an IS is the same as the order in which the IS sends out the messages. It is known as FIFO Assumption.*

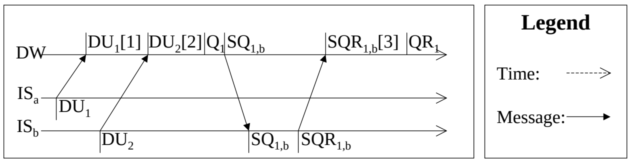

Figure 1: Definition of a **maintenance-concurrent** Data Update

[2]For simplicity, we omit the selection conditions and project attributes applied to the related relations for each subquery.

For example, in Figure 1, DU_1 occurs at IS_a and DU_2 occurs at IS_b respectively. We assume that DU_1 is received at time $t = 1$, denoted as $DU_1[1]$, which is earlier than DU_2 at time $t = 2$, denoted as $DU_2[2]$, at the DW middle-layer. Then the DW middle-layer generates a query Q_1 in order to handle $DU_1[1]$. The query Q_1 will then be broken into sub-queries to be handled by each IS. The sub-query $SQ_{1,b}$ that is generated from Q_1 is sent to IS_b. Since DU_2 occured before IS_b received $SQ_{1,b}$, and because of Assumption 1, DU_2 is received earlier than the query result of $SQ_{1,b}$, denoted as $SQR_{1,b}$, at the DW. Then, the DW will assign $t = 3$ to $SQR_{1,b}$ and denotes it as $SQR_{1,b}[3]$ in the middle layer. At the DW middle layer, based on the timestamp, we can see that $DU_1[1]$ was received earlier than $DU_2[2]$ and $DU_2[2]$ was received earlier than $SQR_{1,b}[3]$. This means that DU_2 affected the sub-query result $SQR_{1,b}$. By Definition 1, DU_2 is said to be a **maintenance-concurrent** data update.

Assumptions. Below are the assumptions held by any previous VM from the literature.

Assumption 2 *All ISs are independent from each other, in the sense that a data update at one IS will not propagate into other ISs.*

Assumption 3 *The updates to the base relations are assumed to be inserts and deletes of tuples. A modify is modeled as a delete followed by an insert.*

2.4 Comparison of State-of-Art View Maintenance Algorithms

The ECA [ZGMHW95], Strobe [ZGMW96] and SWEEP [AASY97] algorithms all solve the **maintenance-concurrent** problem. A comparison of their features is shown in Table 2 inspired by [AASY97]. Due to its features of no quiescence requirement, complete consistency, and high performance for handling one data update, we choose SWEEP as baseline algorithm and we now aim to further improve its performance for *a set of data updates.*

Algorithm	**Architecture**	**Consistency**	**Message Cost**	**Quiescence**	**Key Points**	**Execution**
ECA	Centralized	Strong	O(1)	Required	Remote compensation, Quadratic message size	Single-thread
Strobe	Distributed	Strong	O(n)	Required	Unique key assumption	Multi-thread
C-Strobe	Distributed	Complete	O(n!)	Not Required	Unique key assumption	Multi-thread
SWEEP	Distributed	Complete	O(n)	Not Required	Local compensation	Single-thread
Nested SWEEP	Distributed	Strong	O(n)	Required non-interference of DUs	Local compensation	Single-thread

Table 2: Comparison of ECA, Strobe and SWEEP

One of the major limitations of SWEEP is that it handles the DUs in a sequential manner, as only after the middle layer has fully handled a DU_i and has committed it to the DW will it start processing DU_{i+1}. This decreases the overall performance for a set of DUs forcing the unnecessary delay due to having to wait for a response to maintenance queries from each of the ISs. In this paper, we propose a new algorithm called PVM that enables parallel view maintenance while still keeping all other advantages of SWEEP as listed in Table 2.

On-line Error Correction of Incremental View Computation in SWEEP. The SWEEP [AASY97] algorithm is based on the technique called *on-line error correction*. Instead of doing the compensation after completely evaluating a query as done in ECA [ZGMHW95] and in Strobe [ZGMW96] algorithms, SWEEP eliminates the effects of concurrent data updates during the evaluation of the incremental view maintenance query (Equation 1 on page 3) as soon as they are detected at the middle layer.

For example, assume during the process of an incremental view maintenance of data update ΔR_i, the query has been partially evaluated from R_1 to R_j ($j > i$). The current query result is $Q_j = R_1 \bowtie ... \bowtie \Delta R_i \bowtie ... \bowtie R_j$. The next step will calculate $Q_{j+1} = Q_j \bowtie R_{j+1}$. An update ΔR_{j+1} happens before Q_{j+1} is evaluated at R_{j+1}. As a result of the FIFO property between the middle layer and each IS (Assumption 1), the middle layer will receive ΔR_{j+1} before it receives the query Q_{j+1}. From this, the middle layer know there is an error term $Q_j \bowtie \Delta R_{j+1}$, which can be evaluated locally and eliminated to restore the expected answer $Q_{j+1} \bowtie R_{j+1}$.

In the following sections, we are going to demonstrate the PVM solution by using SWEEP as an example.

3 PVM: A Parallel View Maintenance Solution

3.1 Architecture of PVM

Based on all the pros and cons of the previous VM algorithms, we now introduce the architecture of the PVM system in Figure 2 as an extension of the initial architecture adopted by other VM systems. We can see that the *UpdateView* process is responsible for updating the materialized view of the DW and the *ViewChange* function is responsible for calculating the effect of every data update on the DW. So *ViewChange* is the function with the largest overhead as it is (1) waiting for processing remote queries to be processed by ISs, (2) handling multiple queries, and (3) possibly restricted by processing these remote queries one by one in some sequential order. This results in waits due to IS processes and transmissions over the network. Hence, it is the one functionality in the system that needs to be multi-threaded to handle more than one data update at the same time.

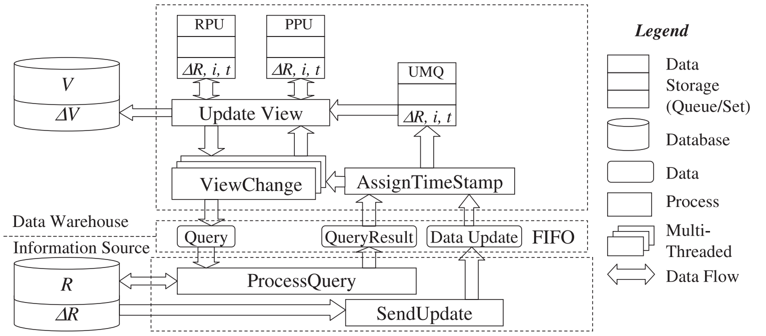

Figure 2: Modules of PVM

The *AssignTimeStamp* process assigns a middle-layer timestamp to each incoming message, including both data updates and query results, to help with the detection of concurrent data updates. So that every item in the UMQ now has an annotated timestamp, denoted by t in Figure 2. If one update and one query result are received at the same time, we know that they come from different ISs due to Assumption 1. That means, the query result will not be affected by the data update. Hence, the order assigned to those two messages will not cause any additional concurrencies. Only the data updates in the UMQ received earlier than the query result and from the same IS as the query result will be treated as concurrent data updates.

In order to parallelize the view maintenance process, we introduce two associated modules, i.e., *Related Processed Update* (RPU) set and the *Parallel Processing Update* (PPU) set. Details of their functionality are given in Section 3.2.

3.2 Issue 1: Detection of Maintenance-Concurrent DUs under Parallel Execution

In order to handle concurrent data updates in the parallel system, the first issue we need to address is how to correctly detect all concurrent DUs. The original data update detection by SWEEP is based on storing data updates in the UMQ. Because SWEEP is a sequential system, all the data updates will wait in the UMQ in the order in which they arrived except the one that is currently handled by SWEEP. However, PVM parallelizes the execution of the *ViewChange* process, i.e., it removes multiple DUs from the UMQ and puts them into the PPU during the parallel handling process. At a first glance, it may appear that PVM could possibly extend SWEEP's concurrent DU detection scheme by checking both the UMQ and PPU. We now show that this checking would still not be sufficient by presenting one example illustrating that PVM would fail to detect a concurrent data update.

3.2.1 A Motivating Example Illustrating the Problem

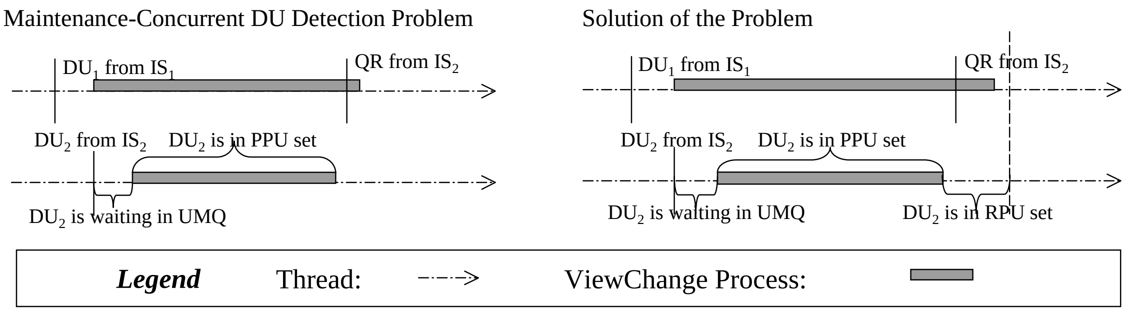

Figure 3: Maintenance-concurrent DU Detection Problem

In Figure 3 there are two data updates DU_1 and DU_2 from ISs IS_1 and IS_2, respectively. DU_1 arrived at the DW before DU_2. So DU_1 is handled first. However, DU_2 affects the query result QR processed at IS_2, which is used to calculate the effect of DU_1. So DU_2 is a **maintenance-concurrent** data update by the Definition 1. In Figure 3 we can see that DU_2 waited in the UMQ to be handled and then is stored in the PPU set while being handled and finally erased from the PVM system after having been handled. However, we assumed that DU_2 is handled faster than the processing of DU_1. So, before DU_1 receives the query result QR from the IS_2, DU_2 has already been removed from the PVM system. Hence, PVM can no longer detect the **maintenance-concurrent** data update DU_2 by only checking UMQ and PPU. Thus the final state of the DW after handling the DU_1 and DU_2 updates would be inconsistent with the state of the ISs.

3.2.2 Solution of Detecting Maintenance-Concurrent DU

The reason for this problem is that there is no tracking of data updates that having been completely handled already but may still be needed for the purpose of future concurrent data update detection. We address this problem by keeping the completely handled data updates in a special holding place where we can check for potential concurrent data updates. We call these data updates *related processed updates*, or RPUs.

Definition 2 [Related Processed Updates]: *A data update (DU) already completely handled by the middle layer is a* **related processed update** *if there exists at least one data update DU_j received before DU_i by DW that hasn't been completely handled by the ViewChange process yet.*

In order to solve this problem, we put a second data storage into the PVM system (Figure 2), called the RPU set. It will maintain all related processed updates until their retention becomes no longer necessary. The *ViewChange* process handles one data update DU_i and commits the effect of that DU_i to the data warehouse, it will first clean up the RPU to remove all the unrelated processed DUs that do not satisfy Definition 2. This can be done by simply checking the timestamps and IS indices of DUs in PPU and RPU. Then it will check whether the DU_i is a RPU. If it is a RPU, the *ViewChange* process will put it into the RPU set. Figure 3 illustrates the relative time intervals for the DU along the time-line of *ViewChange* process.

3.3 Issue 2: Out-of-Order DW Commit

3.3.1 A Motivating Example

In a parallel environment, view maintenance algorithms may work on multiple data updates. Some of the DUs may need a shorter processing time than others. For example, different DUs terminate at different iterations due to an empty query result or due to different network delays on connections to different IS. If every view maintenance thread could commit their change to the data warehouse independently, this could allow us to avoid such unnecessary waits and thus to increase system performance. However, in that case, the commit order of the updates will not match the order of the data updates received at the middle-layer. Hence, it will result in an inconsistent state of the DW extent, meaning that there is no IS space state that matches it.

Assume we have two relations A and B with DW defined by $DW = A \times B$. The extents of A, B and DW are shown in Figure 4. Two data updates DU_1 and DU_2 happened on B and A respectively, namely DU_1 adds $< 3 >$ to B, while DU_2 deletes $< 1 >$ from A. The middle layer received first DU_1 and then DU_2. The effect of DU_1, denoted by ΔDW_1 [3] defined by $A \times DU_1$, is shown in Figure 4. The effect of DU_2, denoted by ΔDW_2, is calculated after receiving DU_2. It is defined by $DU_2 \times B'$ as shown in Figure 4 [4]. After we apply the two Δ DW-s in the order of ΔDW_1 and ΔDW_2 as described in *Correct Commit Order* part of Figure 5, we get the correct DW state.

[3] The SWEEP assume that compensation queries are used so that each individual Q_i returns correct QR_i. So either way if there were no time conflicts or if we actually encounter time conflicts our results are the same.

[4] The strategy of how to calculate the ΔDWs is based on the time stamp of the DUs and query results and we assume that the local compensation strategy as adapted from SWEEP. If two DUs are received at the same time, we can choose any order to log them into the data warehouse. Then, the query for calculating the ΔDWs will depend on the selected order, and the DW will still be updated correctly.

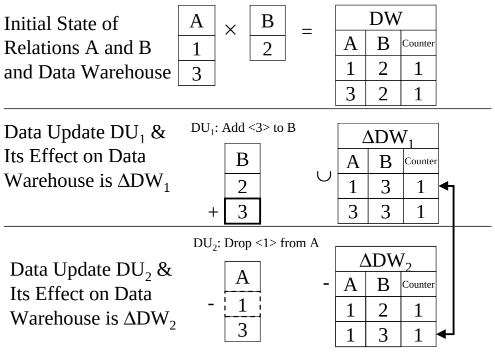

Figure 4: Environment of Example

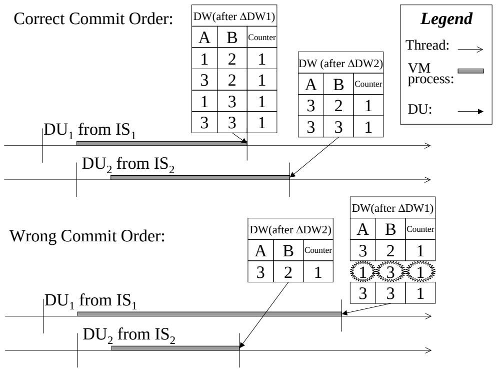

Figure 5: Example of **Out-of-Order-DW-Commit** Problem

If we reverse the commit order of the two Δ DW-s (and the original DW can only count positive tuples as is the state-of-the-art for current DBMS systems), we update the DW in the wrong commit order as shown at the bottom of Figure 5. First, we would subtract ΔDW_2, which given that $< 1, 3 >$ does not yet exist in DW, would be equal to a no-op. Then after union ΔDW_1 with the DW, we get the wrong DW extent depicted in Figure 5, which has a faulty tuple $< 1, 3 >$ with counter 1. The correct answer instead should have been the DW extent given in the *Correct Commit Order* part of Figure 5.

3.3.2 Problem Definition

Assume two data updates DU_1 and DU_2 happened on two different ISs IS_1 and IS_2 of the IS space with state ISS_1, respectively. Once the effect of these DUs has been propagated through the system, then no matter in which order they are executed the IS space will be stable, denoted by the unique state ISS_2. We say a correct extent of the DW should be the same one as if the view extent were recomputed directly from the IS space as state ISS_2.

If we are doing parallel incremental view maintenance of the DW, the order of committing the effects of the two data updates may result in different extents of DW even though only one of them can be the correct one. **DW-Commit-Order** is the order in which data update effects are committed to the DW.

Definition 3 *The problem of inconsistency between the final state of DW and the IS space caused by the variant* **DW-Commit-Order** *is called the* **Out-of-Order-DW-Commit** *problem.*

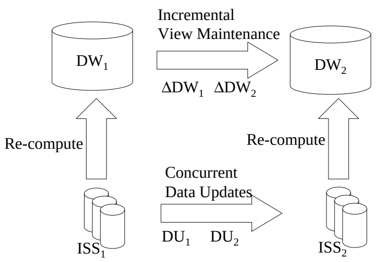

Figure 6: Correctness Criteria of Extent of DW

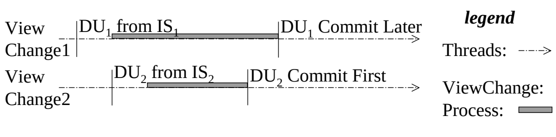

Figure 7: **Out-of-Order-DW-Commit** Order Problem

Theorem 1 *The* **Out-of-Order-DW-Commit** *problem defined by Definition 3 will only occur when first an add-tuple and then a delete-tuple, which both modify the same tuples in the DW, are received by the middle layer and both are handled in parallel by PVM.*

For space reasons, the proof is omitted here, but can be found in our technical report [ZDR99].

Figure 7 illustrates the **Out-of-Order-DW-Commit** problem with a time-line view. This problem may happen when the later handled data update DU_2 is handled much faster than the previously handled data update DU_1. In this case the DU_2 will complete first and commit its effect to the DW prior to the effect of DU_1 being committed to the DW.

3.3.3 Negative-Counter Solution

We now propose a technique called the negative-counter solution that guarantees that the extent of the DW after the parallel incremental view maintenance of multiple DUs will be correct. The basic principle underlying the solution is that we store the DW extent as unique tuples with a counter that shows how many duplicates exist of each tuple. For example, <1,3>[4] means four tuples with values <1,3> exist. The unique twist here is that we permit the counter to be **negative**. Then, for adding (deleting) one tuple, if the tuple already exists in the DW, we increase (decrease) the counter by one, else we create a new tuple with counter '1' ('-1'). We remove a tuple whenever its counter reaches '0'. When the user accesses the DW, any tuple with a counter ≤ 0 will be not visible.

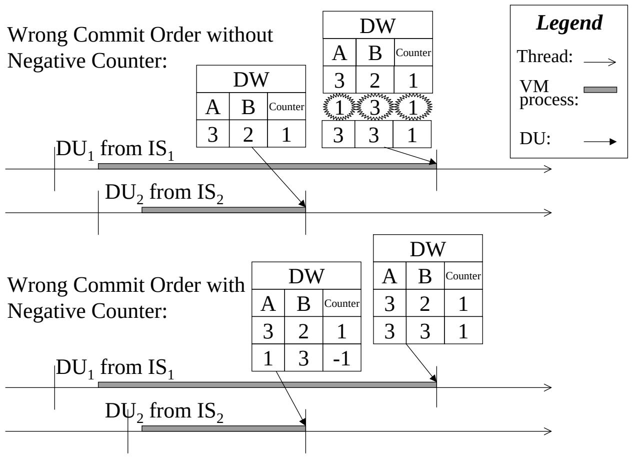

Figure 8: Example of Solution of Out-of-Order DW Commit Order Problem

Figure 8 compares the extent of the DW from our previous example from Section 3.3.2 with and without the negative counter. In the upper half of the figure, DW keeps only tuples with positive counters at any time. After the effect of DU_2, ΔDW_2, is committed, we removed $< 1, 2 >$ but the delete-tuple update $< 1, 3 >$ is invalid since $< 1, 3 >$ is not in the DW yet. Then, after we commit the ΔDW_1 later, we now have added the faulty tuple $< 1, 3 >$ into the DW, which should be deleted (however, the deletion was attempted too early).

In the lower half of Figure 8, the DW can keep tuples with a negative counter. Then the addition of that faulty tuple in the future will remove the tuple with the negative counter from the DW. So, the tuple with a negative counter effectively remembers what tuple should be deleted but cannot be deleted at the current time due to it not yet being in the DW. For example, in the lower part, we can see that DW keeps the tuple $< 1,3 >$ with negative counter **-1** and thus effectively remembers that the $< 1,3 >$ should be deleted eventually. After committing the ΔDW_1, the tuple $< 1,3 > [-1]$ is compensated with the tuple $< 1,3 > [1]$ in ΔDW_1 that given $< 1,3 > [-1] + < 1,3 > [1] = < 1,3 > [0]$, the tuple is finally completely removed from DW. Hence the final state of DW is consistent with the state of the ISs.

Lemma 1 *Given the negative counter mechanism described in Section 3.3, the counters of all tuples will always be positive when the DW reaches a quiescent state.*

The intuition for Lemma 1 is the following. In a quiescent state, there is no more **maintenance-concurrent** DU. The DW will be stable and consistent with the current IS space state. That is, there will be no more outstanding maintenance processes that must be completed and compensated for, as effectively indicated by a negative counter. Hence, there is no tuple with a negative counter in the DW.

PVM doesn't require a quiescent state to commit the updates to the DW. However, a user will only see the tuples with positive counters, while tuples with negative counters are invisible to end users. Such negative counters however indicate that full reflection of base IS updates may not yet have occured at the DW. Hence, only at the time of quiescent the DW maintenance process is fully completed and convergence is then reached.

Theorem 2 *The negative-counter based algorithm (Figures 9 and 10) correctly solves the* **Out-of-Order-DW-Commit** *problem for any number of* **maintenance-concurrent** *DUs.*

As shown in [ZDR99], the proof of this theorem can be given based on an induction on the number of concurrent tuples using Lemma 1. This theorem confirms the correctness of the final state of the DW after applying the effects of the data update in any order, and proves that PVM ensures consistency of type convergence.

3.4 Consistency Level

PVM in its purest form only ensures the *Convergence* level of consistency in order to allow for maximal parallelism of the maintenance process of each individual update. However, if it became important to not only achieve convergence but also *Complete Consistency*, we can enforce that the effects of data updates are committed in the order as they arrive at the DW. This could be very easily achieved in PVM by adding an extra buffer in the *Update View* module of Figure 2. That buffer will keep the ready to commit updates to the data warehouse in the order that they were received. The *Update View* module will only update the view when the head of the buffer is filled. In terms of the performance, the later DU may at times have to wait for the previous DU to commit. This is the price in terms of performance that would then be achieved due to limiting maximal parallism. In other words, the VM processes would still have full advantage of parallelizing, but the final commits of the refreshment to the DW may be delayed to preserve the one desired order as required for complete consistency.

3.5 Algorithms for the PVM Processes

Based on the key features of PVM, we now give details of the PVM processes. The *sendUpdate* and *processQuery* components employed at each IS , given in Figure 9, are traditional as found in other VM systems, such as PSWEEP [AASY97].

```
MODULE Update&QueryServer
  CONSTANT
    MyIndex = i;

  PROCESS sendUpdates;
  BEGIN
    LOOP
      RECEIVE Δ R from R;
      SEND (Δ R, MyIndex) TO DataWarehouse;
    FOREVER;
  END SendUpdates;

  PROCESS processQuery;
  BEGIN
    LOOP
      RECEIVE Δ V FROM DataWarehouse;
      Δ V = ComputeJoin(Δ V, R);
      SEND Δ V TO DataWarehouse;
    FOREVER;
  END ProcessQuery;

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

Figure 9: Pseudo Code of Module Update&QueryServer

Figure 10 depicts the software module that is employed at the DW for view maintenance. *ViewChange* is the main process that is invoked for every update $(\Delta R, i)$ received at the DW. At initialization, the DataWarehouse module will start two processes: *AssignTimeStamp* and *UpdateView*. The *AssignTimeStamp* process assigns a unique local timestamp to all the messages coming into the DW including data updates and query results. The *UpdateView* process monitors the UMQ to check if there is any data update logged in the UMQ and the PPU set still has more room (i.e., we can create more threads to handle DUs). In this case, it would move the data update from the UMQ to the PPU set in one atomic operation, and then create a new instance of the *ViewChange* process to handle this data update now in the PPU set. The *ViewChange* process will first handle the data update, and then commit the effect of the data update to the DW. It will then move the data update from the PPU set to the RPU set if the data update is a *related processed update*, otherwise the data update is completely removed from the system. The RPU set is cleaned up at the end of the *ViewChange* process. Alternatively, one could also run a dedicated process to clean up the RPU set at some fixed time interval.

4 Performance Studies

4.1 Experimental Setup

We have implemented both the SWEEP and the PVM algorithms in JAVA within the EVE data warehouse system and a demonstration of them has been given at ACM SIGMOD'99 [RKZ$^+$99]. While the EVE system interacts with different data servers, our studies were conducted using Oracle 7.0 servers running on a network of NT machines. In our technical report [ZDR99], we have evaluated the performance of PVM and contrasted it with that of SWEEP both based on experimental studies as well as using an analytic cost model. Our experimental studies not only demonstrate the advantage of PVM over SWEEP but also confirm the validity of our cost model. Due to limited space, we now only review a small subset of the experimental findings here.

The parameters we used in this analysis are described in Table 3. We assume a set of ISs (on Oracle) each with one relation, which has two integer attributes named A and B. The DW is defined over all m ISs as given in Figure 11. The extents of all R_i $(1 \leq i \leq m)$are randomly generated and all are approximately the same number of tuples. Each relation has 10000 tuples. This database size while only reasonablly sized already illustrates the performance gains of PVM (parallel VM) over SWEEP (sequential VM), though larger databases at ISs are likely to lead to even further performance improvements due to increased IO processing costs.

```
MODULE DataWarehouseManager;
  CONSTANT
    n: INTEGER /* Size of CPQ */
  GLOBAL DATA
    V: RELATION; /* Initialized to the correct view */
    UpdateMessageQueue: QUEUE initially 0;
    ParallelProcessingUpdate: SET with length n
    RelatedProcessedUpdate: SET initially 0;

  PROCESS ViewChange(ΔR: Relation; UpdateSource:
    INTEGER; TimeStamp: INTEGER): RELATION
  VAR
    ΔV, TempView: RELATION;
    j: INTEGER;
  BEGIN
    ΔV = ΔR;
    /* Compute the left part of the incremental
    view resulting for ΔR */
    FOR (j = UpdateSource -1; j ≥ 1; j–) DO
      ΔV = ProcessSubQuery(ΔV, j, TimeStamp)
    ENDFOR;
    /* Compute the right part to the incremental
    view resulting from ΔR */
    FOR (j=UpdateSource+1; j ≤ n; j++) DO
      ΔV = ProcessSubQuery(ΔV, j, TimeStamp)
    ENDFOR;
    V = V + (ΔV);
    /* erase the unrelated Updates */
    CleanUp RelatedProcessedUpdate;
    CRITICAL AREA
      REMOVE (ΔR, i, t)
        FROM ParallelProcessingUpdate;
      IF (ΔR, i, t) is related
      THEN PUT (ΔR, i, t)
        INTO RelatedProcessedUpdate;
    ENDAREA
  END ViewChange;

  FUNCTION isConcurrent(ΔR: RELATION; j: INTEGER;
    t: INTEGER): BOOLEAN
    IF ∃(ΔR, j, t) ∈ UpdateMessageQueue
      or ∃(ΔR, j, t) ∈ ParallelProcessingUpdate
      or ∃(ΔR, j, t) ∈ RelatedProcessedUpdate
      THEN return TRUE; ELSE return FALSE
    ENDIF
  END isConcurrent

  FUNCTION ProcessSubQuery( ΔV: RELATION; j:
    INTEGER; t: INTEGER): RELATION
    TempView = ΔV;
    SEND ΔV TO Data Source j;
    /* The ΔV in the next line has
    already time stamp assigned by AssignTimeStamp
    process */
    RECEIVE ΔV FROM Data Source j;
    /* Remove the error due to concurrent update
    if any ( maybe more than one ) */
    FOR ALL ΔR from IS_j DO
      IF isConcurrent(ΔR, j, t)
      THEN ΔV = ΔV - ΔR ⋈ TempView; ENDIF;
    ENDFOR
  END ProcessSubQuery

  PROCESS AssignTimeStamp;
  VAR
    t: TIME; /* current system time at the DW */
  BEGIN
    LOOP
      RECEIVE Message FROM Data Source i
        as received order;
        IF Message is Δ THEN
          t= getCurrentTime();
          APPEND (ΔR, i, t) TO UpdateMessageQueue;
        ELSE
          Assign getCurrentTime() to ΔV
        ENDIF
    FOREVER;
  END AssignTimeStamp;

  PROCESS UpdateView;
  BEGIN
    LOOP
      IF ParallelProcessingUpdate not full THEN
      BEGIN
        CRITICAL AREA
          REMOVE (ΔR, i, t) FROM UpdateMessageQueue;
          APPEND (ΔR, i, t) TO ParallelProcessingUpdate;
        ENDAREA
        StartProcess(ViewChange(ΔR, i, t));
      END
      ENDIF
    FOREVER
  END UpdateView;

BEGIN /* Start DataWarehouse Manager */
  StartProcess(AssignTimeStamp);
  StartProcess(UpdateView);
END DataWarehouseManager
```

Figure 10: Algorithms for PVM

4.2 Performance of PVM

Figure 12 (on page 15) shows the behavior of PVM in response to different numbers of information sources with varied PPU sizes p (numbers of threads). In Figure 12, the different PPU sizes p are plotted on the x-axis and the time (in seconds) taken by PVM for different PPU sizes is presented on the y-axis. Each line corresponds to a different number of ISs. In all plots, the number of information sources vary from 2 to 6 with the number of data updates for each information source fixed to 10.

Based on Figure 12, we can make the following observations: (1) When the query capability of ISs ($m \times q$) is larger than the maximum number of threads p, then the total execution time T of PVM is reduced as the PPU size p increases. (2) After a certain point, for a fixed $m \times q$, time T can no longer be decreased by increasing p only. We call that point the "turning point". The two observations above show that the performance gain of PVM is based on both the query capability of the IS space and the system capability of the DW manager in terms of the maximum number of threads. We have found as also confirmed by our experiments that the performance of PVM denoted by T_{PVM} with PPU size p is proportional to $\approx T_{SWEEP}/min(p, m \times q)$.

The Parameters of the Experiments	
n	Number of Concurrent DUs.
m	Number of ISs.
p	Size of PPU (Maximum number of parallel threads).
q	Number of Queries could be Processed by one IS at one time (Query capability).
T	Elapsed Time of Handling a Set of DUs.
t	Elapsed Time of Handling one Data Update.

Table 3: Parameters of the Experimental Evaluation

```
CREATE VIEW  V AS
SELECT       R1.A, R2.A, R3.A, ..., Rm.A
FROM         R1, R2, R3, ..., Rm
WHERE        R1.A = R2.B, R2.A = R3.B, ...  Rm-1.A = Rm.B
```

Figure 11: Data Warehouse View Definition

4.3 Scalability of PVM

Figure 13 shows that our approach scales well with the number of DUs. In all plots, the number of information sources is fixed to 5, the numbers of parallel threads p in PVM is $\{3, 5, 7, 9\}$, and the number of concurrent data updates for information sources varied from 20 to 150. As we can see that, PVM's performance is almost linear in the number of concurrent data updates. The lines moved little bit up and down around the linear trend, because multiple threads may wait for each other to accomplish the query over the ISs. When the number of threads is high, and the ISs are fully loaded, the performance is again linear in number of concurrent DUs.

4.4 Comparison of PVM with SWEEP

Figure 14 (on Page 16) compares the performance of PVM with SWEEP in terms of seconds (depicted on the y-axis) for different numbers of data updates n (depicted on the x-axis). In all plots, the number of information sources is fixed to 2, p is set to 5 for PVM, the number of data updates for each information source varied from 4 to 40.

Comparing the response times, we notice that PVM is much faster than SWEEP. As explained in Section 4.2, the ratio of T_{SWEEP}/T_{PVM} is about $min(5, 2) = 2$. In this experiment this thus means that PVM achieves a 2-fold (100%) performance improvement.

Space constraints prevent us from presenting the results of our other experiments, however a detailed discussion of them can be found in [ZDR99].

5 Related Work

Self-maintenance [QGMW96, GJM97, SK98, GJM96, Huy96] is one way to maintain the materialized views at the DW without access to the base relations by replicating all or parts of the base data at the DW. As more and more data is added to the DW, it increases the space complexity and

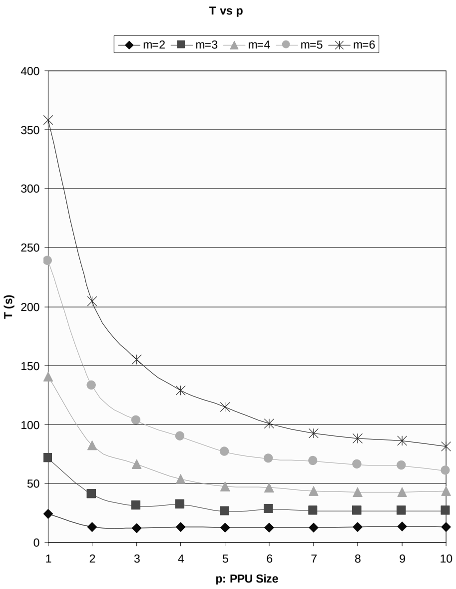

Figure 12: PVM Performance v.s. Number of Threads

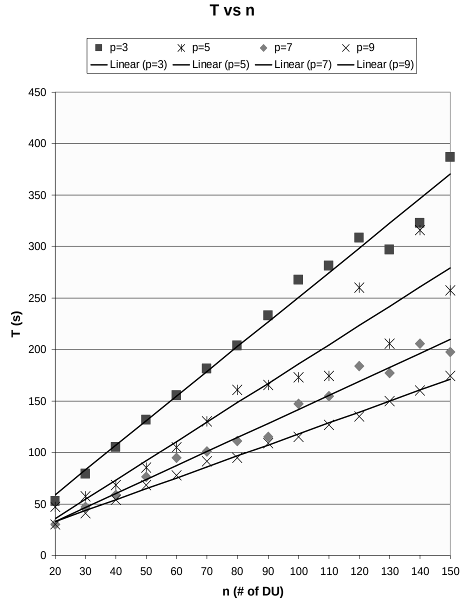

Figure 13: PVM Scale for Large Number of Concurrent DUs.

gives rise to information redundancy which might lead to an inconsistent DW extent. In addition, not all views are self-maintainable.

View maintenance methods concentrate instead on ensuring consistency of the DW when the materialized views are not self-maintainable. The ECA [ZGMHW95] family of algorithms introduces the maintenance problem and solves it partially, i.e., for one single IS. Strobe [ZGMW96] and SWEEP [AASY97] are two view maintenance algorithms for multiple ISs both of which focus on the concurrency of data updates.

The Strobe [ZGMW96] algorithm introduces the concept of queueing the view updates at the DW and committing the updates to the DW only when the unanswered query set (UQS) is empty. The algorithm solves the consistency problem but is subject to the potential threat of infinite waiting, i.e., the DW extent may never get updated. Other mechanisms are based on requiring the ISs to timestamp the view updates [BCS96], i.e., some global timing synchronization of the environment. So far, most consistency maintenance methods either have the infinite waiting deficiency or need a global timestamp service (see Table 2). Like SWEEP, PVM doesn't have any of the above limitations to handle multiple data updates.

The SWEEP [AASY97] family of algorithms however succeeded in eliminating the above mentioned limitations by applying local compensation techniques. SWEEP uses special detection methods for concurrent data updates that don't need global time stamp mechanism. It also requires no quiescent state before being able to update the DW. Beyond this, PVM now addresses the bottleneck of its sequential processing by exploiting the inherently parallelism of this view maintenance process due to the distributed ISs each with their independent servers.

Nested-SWEEP [AASY97] is used to handle a set of DUs by reusing the query results. Due to its recursive solution, it requires non-interference of the DUs, otherwise, an infinite recursive call may result in a maintenance failure. PVM doesn't apply any recursive process optimization to share the query result as in Nested-SWEEP, rather instead it parallelizes the SWEEP maintenance process. A more detailed comparative analysis between PVM and Nested-SWEEP remains to be

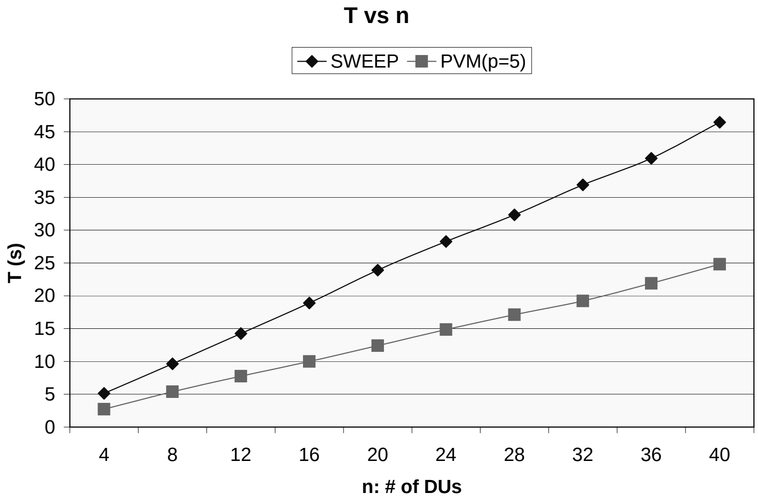

Figure 14: Comparison of PVM with SWEEP (m=2, m is number of ISs. So min(2,5) corresponds to a 2-folded speedup.)

undertaken.

The Posse [OAA99] framework is introduced to achieve the highest performance between the selection of different VM algorithms based on two factors: higher concurrency or smaller message size. They are using a DAG-like query plan to model their choice between different maintenance plans. They trade off between sending separate subqueries of one query Q to several ISs versus sending one subquery at a time with piggybacked intermediate results thus far to be joined remotely at that particular IS. The first option requires large result messages to be returned and joined in DW, and the second option enforces sequence between subquery processing. Instead, PVM is tackling a different problem that is, PVM is considering how we can parallelize the handling of a group of DUs to achieve higher performance.

6 Conclusions

In this paper, we have investigated the problem of parallel view maintenance. First, we identified the bottleneck in terms of sequential handling of DUs by the current state-of-the-art VM solution called SWEEP [AASY97]. The SWEEP algorithm has the least limitations compared to alternate contenders in the literature, but it adopts sequential processing when handling a set of data updates. In this work, we have identified several open issues to achieve parallel view maintenance, notably, concurrent data update detection in a parallel execution mode, and the out-of-order-DW-commit problem. We then present an integrated solution called PVM that is indeed capable of handling these problems. We have fully implemented both SWEEP and PVM in our EVE data warehousing system and conducted a set of experiments to study their performance. The experimental results demonstrate that the PVM achieves a multi-fold performance improvement compared to SWEEP [RKZ$^+$99].

Acknowledgments. The authors would like to thank students at the Database Systems Research Group at WPI for their interactions and feedback on this research. In particular, we are grateful to Andreas Koeller for helping to design and implement the EVE system in which PVM has been embedded into, and we also thank Prof. Nabil Hachem for his input on the ideas presented here.

References

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

[BCS96] E. Baralis, S. Ceri, and S.Paraboschi. Conservative TimeStamp Revised for Materialized View Maintenance in a Data Warehouse. In *Workshop on Materialized Views*, pages 1–9, 1996.

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

[CGL+96] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for Deferred View Maintenance. In *Proceedings of SIGMOD*, pages 469–480, 1996.

[CM96] Latha S. Colby and Inderpal Singh Mumick. Staggered Maintenance of Multiple Views. In *Workshop on Materialized Views: Techniques and Applications*, pages 119–128, 1996.

[GJM96] A. Gupta, H.V. Jagadish, and I.S. Mumick. Data Integration using Self-Maintainable Views. In *Proceedings of International Conference on Extending Database Technology (EDBT)*, pages 140–144, 1996.

[GJM97] Ashish Gupta, H. V. Jagadish, and Inderpal Singh Mumick. Maintenance and Self Maintenance of Outer-Join Views. In *Next Generation Information Technologies and Systems*, 1997.

[GM95] A. Gupta and I.S. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Engineering Bulletin, Special Issue on Materialized Views and Warehousing*, 18(2):3–19, 1995.

[GM96] Ashish Gupta and Inderpal Singh 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] Héctor García-Molina, Wilburt Labio, Janet L. Wiener, and Yue Zhuge. Distributed and Parallel Computing Issues in Data Warehousing . In *Symposium on Principles of Distributed Computing*, page 7, 1998. Abstract.

[GMS93] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. In *Proceedings of SIGMOD*, pages 157–166, 1993.

[Huy96] N. Huyn. Efficient View Self-Maintenance. In *Proceedings of the Workshop on Materialized Views: Techniques and Applications*, pages 17–25, June 1996.

[KLM+97] Akira Kawaguchi, Daniel F. Lieuwen, Inderpal Singh Mumick, Dallan Quass, and Kenneth A. Ross. Concurrency Control Theory for Deferred Materialized Views. In *ICDT*, pages 306–320, 1997.

[KLMR97] Akira Kawaguchi, Daniel F. Lieuwen, Inderpal Singh Mumick, and Kenneth A. Ross. Implementing Incremental View Maintenance in Nested Data Models. In *Workshop on Database Programming Languages*, pages 202–221, 1997.

[MKK97] Mukesh K. Mohania, Shin'ichi Konomi, and Yahiko Kambayashi. Incremental Maintenance of Materialized Views. In *Database and Expert Systems Applications (DEXA)*, pages 551–560, 1997.

[NR99] A. Nica and E. A. Rundensteiner. View Maintenance after View Synchronization. In *International Database Engineering and Applications Symposium (IDEAS'99)*, August, Montreal, Canada 1999.

[OAA99] Kevin O'Gorman, Divyakant Agrawel, and Amr El Abbadi. Posse: A Framework for Optimizing Incremental View Maintenance at Data Warehouses. In *Data Warehousing and Knowledge Discovery First International Conference*, 1999.

[QGMW96] Dallan Quass, Ashish Gupta, Inderpal Singh Mumick, and Jennifer Widom. Making Views Self-Maintainable for Data Warehousing. In *Conference on Parallel and Distributed Information Systems*, pages 158–169, 1996.

[RKZ+99] E. A. Rundensteiner, A. Koeller, X. Zhang, A. Lee, A. Nica, A. VanWyk, and Y. Li. Evolvable View Environment. In *Proceedings of SIGMOD'99 Demo Session*, pages 553–555, May/June 1999.

[SK98] S. Samtani and Vijay Kumar. Maintaining Consistency in Partially Self-Maintainable Views at the Data Warehouse. In *Database and Expert Systems Applications (DEXA)*, pages 206–211, 1998.

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

[WGL+96] Janet L. Wiener, Himanshu Gupta, Wilburt Labio, Yue Zhuge, Hector Garcia-Molina, and Jennifer Widom. A System Prototype for Warehouse View Maintenance. In *Workshop on Materialized Views: Techniques and Applications*, pages 26–33, 1996.

[Wid95] J. Widom. Research Problems in Data Warehousing. In *Proceedings of International Conference on Information and Knowledge Management*, pages 25–30, 1995.

[ZDR99] X. Zhang, L. Ding, and E. A. Rundensteiner. Parallel View Maintenance Under Concurrent Data Updates of Distributed Sources. Technical Report WPI-CS-TR-99-14, Worcester Polytechnic Institute, Dept. of Computer Science, May 1999.

[ZGMHW95] Y. Zhuge, Héctor 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, Héctor 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.

[ZGMW98] Yue Zhuge, Héctor García-Molina, and Janet L. Wiener. Consistency Algorithms for Multi-Source Warehouse View Maintenance. *Distributed and Parallel Databases*, 6(1):7–40, 1998.