WPI-CS-TR-99-14 March 1999

PSWEEP: Parallel View Maintenance Under Concurrent Data Updates of Distributed Sources

by

Xin Zhang
Elke A. Rundensteiner

Computer Science Technical Report Series

WORCESTER POLYTECHNIC INSTITUTE

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

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

Xin Zhang and Elke A. Rundensteiner

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

Abstract

Data warehouses (DW) are built by gathering information from several information sources (ISs) and integrating it into one repository customized to users' needs. Recent work has begun to address the problem of view maintenance of DWs under concurrent data updates of different ISs. SWEEP proposed by Agrawal et al. [AAS97] is one of the more popular solutions; even though its performance is limited due to enforcing a sequential ordering on the handling of data updates from ISs by the view maintenance module. We have overcome this limitation by developing a parallel algorithm for view maintenance, called PSWEEP, that still incorporates all benefits of SWEEP while offering substantially improved performance. In order to perform parallel view maintenance, we solve two issues: detecting **maintenance-concurrent** data updates in a parallel mode, and correcting the problem that the DW commit order may not correspond to the DW update processing order due to parallel maintenance handling. By decomposing SWEEP into an architecture of modular components, we then can insert a local timestamp assignment module for detecting **maintenance-concurrent** data updates without requiring any global clock synchronization. We introduce the negative counter concept as a simple yet sufficient solution to solve the **Variant-DW-Commit** problem of variant orders of committing effects of data updates to the DW. We have proven the correctness of PSWEEP to guarantee that our strategy indeed generates the correct final DW state. An evaluation of both SWEEP and PSWEEP is given that shows that PSWEEP has the potential of multi-fold performance improvement over SWEEP depending on the number of threads supportable in the given DW system implementation.

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

*This work was supported in part by the NSF NYI grant #IRI 94-57609. Dr. Rundensteiner would also like to thank IBM for the IBM Partnership Award as well as for IBM Fellowships for her students.

1 Introduction

1.1 Background on View Maintenance

Data warehousing [WB97, Wid95, GMLWZ98, GM96] is a newly developed technology to integrate data from heterogeneous information sources (ISs) in order to provide such data to for example decision support or data mining applications [CD97]. As depicted in Figure 1, a data warehousing (DW) system [LZW+97, YKL97, HGMW+95] typically has wrappers to detect changes at an IS and to propagate the changes in a generic form to the integrator. The integrator integrates the changes into the DW by merging the updates with the data already present in the DW and resolving possible update anomalies.

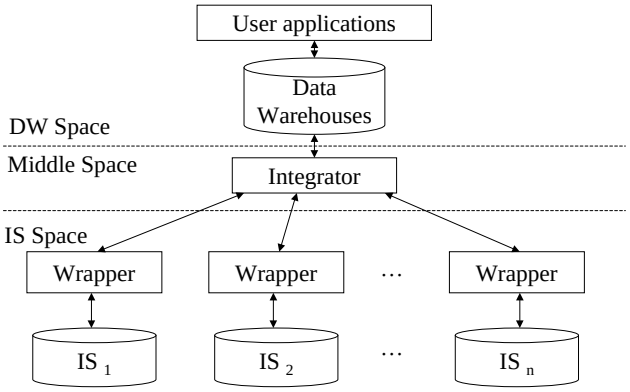

Figure 1: DW Framework

Once a DW is established, the problem of maintaining it consistent with the underlying ISs under updates remains a critical issue. It is popular to maintain the DW incrementally [AAS97, 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. 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, recent work has focused on performing the maintenance in a separate process, then called deferred view maintenance [CGL+96, KLM+97].

Since deferred view maintenance does not block the update transaction of the underlying ISs, it causes 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 defered view maintenance [WGL+96, MKK97, CM96, QGMW96, GJM97, KLMR97, ZGMW98]. ECA [ZGMHW95] handles view maintenance under concurrent data updates of one centralized IS, while Strobe [ZGMW96] and SWEEP [AAS97] handle distributed ISs. Any solution that manages to improve the performance of the view maintenance process

is critical, given that it would reduce the time during which the DW lags behind and thus is inconsistent with the ISs. Consequently, such optimization would lead to an improvement of the timeliness and hence quality of the DW content important for time-critical applications, like stock market and monitoring-control applications. Optimization of DW maintenance is hence the focus of our work.

1.2 Our Approach – PSWEEP

In this paper, we now propose a novel algorithm called Parallel SWEEP, in short PSWEEP, that has a higher performance of handling a set of concurrent data updates than the previous solutions. PSWEEP preserves all advantages of its predecessors, in particular SWEEP [AAS97], while overcoming their main limitation in terms of performance. 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. We investigate how best to introduce parallelism into SWEEP to overcome this limitation. We identify two research issues central to enabling a parallel solution of view maintenance. The first issue is concerned with how to detect conflicting concurrent data updates within a parallel execution mode. The second issue is concerned with how to handle the random commit order of effects of data updates at the DW, which is called the DW commit order, because of the parallel execution of the view maintenance without blocking. For the first issue, we introduce two data structures to store concurrent data updates to help PSWEEP detect the conflicting ones. We introduce a local timestamp mechanism to identify each data update and query result received by the middle layer in order to help detecting the concurrent data updates instead of global timestamp mechanism. For the second issue, we extend the meaning of counters [1] kept for tuples at the DW to also include negative counts. A negative counter now indicates the number of faulty deleted tuples due to the variant DW commit order. 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. Lastly, the evaluation of our cost model of the PSWEEP algorithm shows that PSWEEP dramatically increases the performance of SWEEP based on the number of threads that can be executed at the same time in the given system configuration.

1.3 Contributions

The main contributions of this paper are:

- Identify the performance limitation of the state-of-the-art view maintenance (VM) solutions in terms of sequential handling and characterize research issues to be addressed to achieve parallel execution.

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

- Introduce a local timestamp service and several auxiliary data structures that together solve the concurrent DU detection problem even in a parallel execution mode.
- Propose a solution to the parallel view maintenance process without blocking the commit phase to the DW by extending the range of DW duplicate counters to allow for negative values to keep track of and then compensate for faulty tuples.
- Prove the correctness of the PSWEEP solution to confirm that it achieves the same final DW extent as a complete re-computation algorithm.
- Present the detailed PSWEEP architecture that incorporates the proposed solution techniques based on identifying a minimal set of changes required to adapt the current solution (SWEEP) to achieve parallel view maintenance. This should allow for a quick adaption of PSWEEP by current systems.
- Perform an analytic evaluation of PSWEEP that clearly characterizes its performance advantages over previous solutions.

Outline: In the following section, we discussed related work in view maintenance. In Section 3 we introduce background material needed for this work. In particular, we review the key modules of the SWEEP system, pointing out how it detects concurrent data updates and characterizing its bottleneck. In Section 4 we introduce the PSWEEP solution, including the two open research issues and their solutions. In Section 5, we describe the cost-model and analytic evaluation of the performance of PSWEEP. At last, we discuss conclusions in Section 6. Proofs of the correctness of theorems and lemmas are given in the appendix.

2 Related Work

2.1 View Maintenance Issues

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 gives rise to information redundancy which might lead to inconsistencies. In addition, not all views are self-maintainable.

Consistency 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 problem and solves it partially, i.e., for one single IS. Strobe [ZGMW96] and SWEEP [AAS97] 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 queuing the view updates at the DW and commiting 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]. So far, consistency maintenance methods either have the infinite waiting deficiency or need a global time stamp service.

The SWEEP [AAS97] family of algorithms eliminates the above mentioned limitations by applying local compensation techniques. SWEEP uses special detection methods for concurrent data updates that don't need the global time stamp. It also requires no quiescence before being able to update the DW.

Most recent algorithms perform deferred view maintenance (VM), i.e., they maintain the DW extent in a transaction separate from the data update transaction of the information sources. This includes SWEEP [AAS97], Strobe [ZGMW96], ECA [ZGMHW95] and also our PSWEEP. Deferred VM has the following features. It is executed after a data update has been committed at the IS. It will either compensate for a query result affected by concurrent data updates, or reuse the query result to recursively handle next concurrent data updates. The DW will have a time period during which it is not consistent with the ISs.

2.2 A Comparison of SWEEP Versus Strobe Solutions

Both Strobe [ZGMW96] and SWEEP [AAS97] are among the most advanced solutions for VM that handle concurrent DUs for distributed ISs. Thus we study them in more depth to crystalize which of their principles to extract for our parallel VM work. From Table 1, both SWEEP and Strobe are using deferred View Maintenance strategies. Strobe is the first algorithm that is handling concurrent DUs for distributed ISs [ZGMW96]. Strobe processes DUs as they arrive, however because of Strobe's data structure, Strobe needs a quiescence time period to update the DW. Otherwise all changes are simply buffered in a temporary place and will never propagate to the DW. To address this limitation, [ZGMW96] also introduces the C-Strobe (Complete-Strobe) algorithm that doesn't need the quiescence period. It however significantly increases the number of messages required to handle each data update, i.e., it is O(n!) complexity with n the number of ISs.

Based on the study of the Strobe family of algorithms, Agrawal et al. [AAS97] determined that the previous algorithms for view maintenance require at least one of the following assumptions: 1. a quiescent state at the DW, 2. an exponential number of messages in terms of the number of data sources, or 3. restricted types of view definitions that require to include keys of every data source. SWEEP successfully released these previous limitations by executing compensation queries locally exploiting the information that is already available at the DW. Previous approaches had to send down compensation queries to ISs, which may lead to a potential ripple-effect of a remote compensation query requiring more compensation itself. SWEEP is a significant improvement over Strobe in terms of both the number of messages and data sizes transferred between the DW and the ISs. In particular, the message cost per update for the SWEEP

Similarities	Differences	
	Strobe	SWEEP
• Handle Concurrent Data Updates of Multiple Information Sources • Deferred View Maintenance Algorithm	• Compensate by Remote Query • Process DUs as They Arrive • Require Quiescence	• Compensate Locally • Process DUs Sequentially • Can Commit to DW without Delay

Table 1: Compare Strobe and SWEEP Approaches.

algorithm is O(n) compared to O(n!) of C-Strobe [ZGMW96]. It is hence this SWEEP approach that we chose as starting point for our solution of parallel view maintenance, though we succeed in dramatically increasing its performance as described in this paper.

Note that SWEEP handles the DUs in a sequential manner, as only after the middle layer fully handles a DU_i and commits it to the DW will it start processing the next DU_{i+1}. This decreases the overall performance for handling a set of DUs. In this report, we propose a new algorithm called PSWEEP that parallelizes the execution of SWEEP and thus results in a significant performance increase over previous solutions, while preserving all of SWEEP's benefits.

3 Background on the SWEEP Approach

3.1 The Basic SWEEP Solution

Since the PSWEEP approach proposed in this paper utilizes some of the ideas of the SWEEP algorithm [AAS97], we first present a brief introduction of SWEEP. Figure 2 shows the original design of SWEEP. In general, the algorithm is divided into two layers, the *data warehouse layer* and the *information source layer*. Those two layers are connected by a FIFO network. The *information source layer* contains two types of processes. One process, called *ProcessQuery*, associated with each IS is used to process all the queries from the *data warehouse layer* and to return the query result. The other process, called *SendUpdate*, is used to send the data updates of each IS up to the DW. Every IS will have two such processes. The *data warehouse layer* also contains two types of processes as well as one function called *ViewChange*. The *LogUpdates* process is used to buffer all the data updates received from ISs into the *Update Message Queue* (UMQ). The *UpdateView* process retrieves the data updates from the UMQ sequentially and applies the effect of those data updates to the DW one by one. The *ViewChange* function is used by the *UpdateView* process to calculate the effect of a given data update (using local compensation queries).

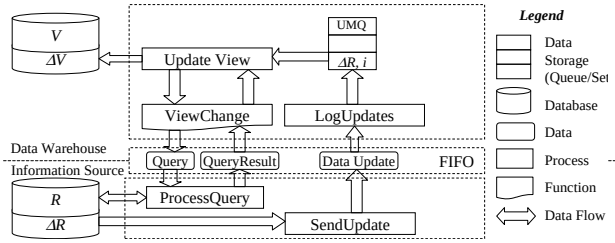

Figure 2: Modules of SWEEP

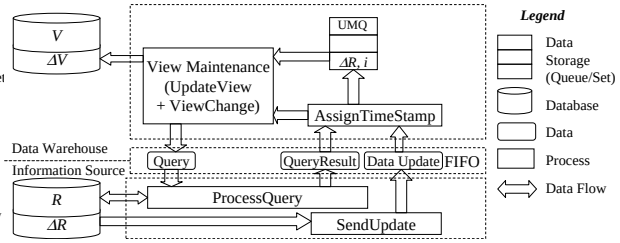

Figure 3: Overview of SWEEP's Modified Architecture

3.2 Proposed Enhancement to the SWEEP Architecture

Figure 3 shows the detailed design of the DW module described in Figure 2 now with several key extensions we propose. First, we change the *LogUpdates* process of SWEEP to the *AssignTimeStamp* process in Figure 3. In addition to simply logging data updates to the UMQ structure, this process assigns a local time stamp to each incoming message, including both data updates and query results, to help with the detection of concurrent data updates. The original DU detection of SWEEP [AAS97] assumes there is no system expenses associated with receiving query results and checking the UMQ, i.e., that this is instantaneous. They treat all the updates in the UMQ from the same IS as the query result as concurrent data updates. However, in a practical system, we cannot make sure there isn't a data update from the same IS which is put in the UMQ after receiving the query result but before SWEEP checks the UMQ. If that happen, SWEEP as defined in [AAS97] will mistakenly take that DU as a concurrent DU.

For example, we assume one IS named IS_i sends three messages to the middle layer. In the **IS-Commit-Order**, they are DU_2, QR_1, and DU_3, where QR_1 is the query result requested by the middle layer for handling the DU_1. From the above settings, we can see that DU_2 is a concurrent DU, while DU_3 is not. Based on the FIFO assumption, those three messages will arrive at the middle layer in the same order. The *LogUpdate* process will log DU_2, and DU_3 into the UMQ, and send QR_1 to the *ViewMaintenance* process. Then the *ViewMaintenance* process will check the UMQ for concurrent DUs. Please note here, the logging of DU_3 and checking of UMQ are done by two separate processes, namely the *LogUpdate* process and the *ViewMaintenance* process, respectively. So, there is no guarantee that the checking of UMQ will be executed earlier than the logging of DU_3. In other words, the DUs that are visible to the *ViewMaintenance* process when checking the UMQ may not all have happened before QR_1. Hence, if we simply treat all DUs in the UMQ as concurrent DUs, we will mistakenly treat DU_3 as a concurrent DU.

To avoid such an error, we modify the *LogUpdates* process to assign time stamps to all incoming messages to compare the data updates and the query results to figure out which one arrived first in the middle layer. 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. The UMQ only keeps the data updates, and the order of

the data updates in the UMQ is same as the **DW-Receive-Order**.

Second, we integrate both the *UpdateView* process and *ViewChange* function into one process, called the *View Maintenance* process, because the *UpdateView* process first calls the *ViewChange* function to calculate the effect of the data update and then commits the effect to the data warehouse. The coupling between the *UpdateView* process and *ViewChange* function is thus very high, i.e., they are not independent. So we put them together into one *View Maintenance* process to more precisely model the architectural structure of the SWEEP system.

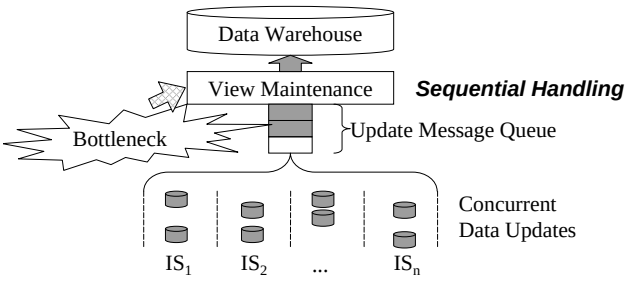

Figure 4: Performance Bottleneck of SWEEP

Figure 4 depicts the major drawback of SWEEP. That is, no matter how many data updates are generated by the ISs, they all must wait in the update message queue in a sequential order until the *View Maintenance* process has handled the previous update and is free to service the next. Unlike the Strobe algorithm [ZGMW96], SWEEP cannot handle the data updates as soon as they are received. It is this severe performance bottleneck of the SWEEP system which we are solving in this work.

4 PSWEEP: A Parallel View Maintenance Solution

4.1 Assumptions

Below we list the assumptions of SWEEP that we are also using in PSWEEP.

Assumption 1 *An IS will send a notification message to the DW middle layer only after the data update is committed or the query result is generated. The order in which the middle layer receives those messages is the same as the order in which each IS is sending out the messages. (FIFO network).*

Besides the assumptions that are explicitly defined in SWEEP, we note that the following assumptions also hold for SWEEP as well as for PSWEEP.

Assumption 2 *We assume each IS uses the 2 phase commit protocol to guarantee that the data update DU2 that is committed after DU1 at one IS will incorporate all effects of previously committed data updates, such*

as DU1.

Assumption 3 *All ISs are independent from each other. That means, data updates at one IS will not propagate into other ISs.*

Assumption 4 *The view definition will not be changed during the view maintenance process.*

Assumption 5 *Every IS will report all and only those changes caused by a data update at the tuple level after having committed the data update, including the direct effect of that data update as well as the indirect effects of that data update on that IS.*

```
CREATE TABLE R1( A CHAR(11),
                 D INTEGER,
                 E DATE,
                 PRIMARY KEY (A, D),
                 FOREIGN KEY (A) REFERENCES R2,
                 ON DELETE CASCADE)
```

```
CREATE TABLE R2 ( A CHAR(11),
                  B CHAR(11),
                  C INTEGER,
                  PRIMARY KEY (A))
```

Table 2: Schema Definition of an Example of the Indirect Effect of a Data Update.

As an illustration of assumption 5, assume we have two relations R1 and R2 at IS1 as defined in Table 2. As we can see, R1 has a foreign key referring to R2. When there is a data update DU1 that deletes tuple T1 from R2, the indirect effect of DU1 would be the deletion of these tuples that have been referred to by T1 from R1.

4.2 Problem Definition of Maintenance-Concurrent Data Update

From the DW framework described in Figure 1, we can see that the overall framework includes both the IS space and the DW space. Depending on where messages are received or generated, we distinguish between five different orders of messages inside the DW framework. They are **IS-Start-Order**, **IS-Commit-Order**, **DW-Receive-Order**, **DW-Handle-Order** and **DW-Commit-Order** as defined in Table 3.

	Where	Name	Definition
Orders of Message	IS	**IS-Start-Order**	Order in which Data Update Transactions start at ISs.
		IS-Commit-Order	Order in which Data Update Transactions commit at ISs.
	DW	**DW-Receive-Order**	Order in which Data Updates are received by middle layer.
		DW-Handle-Order	Order in which Data Updates are handled by middle layer.
		DW-Commit-Order	Order in which Data Update effects are committed to DW.

Table 3: Five Orders of Messages in the Data Warehousing Framework

Figure 5 described the timing relationship between the five message orders. First DUs occur at the IS in the **IS-Start-Order**. After the DUs are committed at an IS in the **IS-Commit-Order**, the IS is indeed

changed. Then the IS reports the changes to the DW. The DW receives the notification of data updates in the **DW-Receive-Order**. Then the DW starts handling the data updates in the **DW-Handle-Order**. Finally, the DW commits the effect of the data updates in the **DW-Commit-Order**.

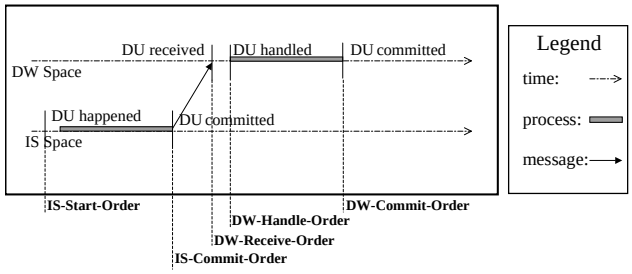

Figure 5: Timing Relationship Between the Five Message Orders

Among the five orders at different stages of processing data updates, **IS-Start-Order** is transparent to DW maintenance, and **IS-Commit-Order** is the same as **DW-Receive-Order** because of the FIFO assumption [2]. From the three orders local to the DW, namely, **DW-Receive-Order**, **DW-Handle-Order** and **DW-Commit-Order**, PSWEEP system can detect concurrent data updates and handle them locally without any global timestamp service. **DW-Receive-Order** is set by the IS space and the network and decides whether data updates are concurrent or not. The **DW-Handle-Order** will not affect the effects of data updates on the DW. However, the **DW-Commit-Order** may change the final state of DW if it is not properly handled. The three DW orders are identical to each other in the SWEEP system, i.e., **DW-Receive-Order** = **DW-Handle-Order** = **DW-Commit-Order**, since the SWEEP system is sequentialized. Now given that PSWEEP is a parallel system, **DW-Commit-Order** may not be the same as **DW-Receive-Order**. Hence, **DW-Commit-Order** will cause the **Variant-DW-Commit** problem in PSWEEP as further discussed in Section 4.6.

Based on the five message orders in the data warehousing system, we now define the problem of view maintenance of concurrent data updates. Because the data updates are happening concurrently at the ISs, one data update DU_j at IS_m could interfere with the query processing of a query sent by the middle layer to this IS_m to handle another data update DU_i. We call this data update DU_j a **maintenance-concurrent** data update as it takes place concurrently with the maintenance process servicing alternate updates.

Definition 1 *Let DU_i and DU_j denote two DUs on IS_m and IS_n respectively. We say that the data update DU_j is* **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., $i < j$)

[2] It is in fact possible to also drop this assumption of order, but due to space limitations this part of the solution is omitted here.

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.
$SQ_{i,m}[t]$	$SQ_{i,m}$ with Unique Timestamp t.
$SQR_{i,m}[t]$	$SQR_{i,m}$ with Unique Timestamp t.

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.*

We say that an update DU_p is **maintenance-concurrent**, *if there exists at least one DU_q for some $q < p$ such that $DU_p \vdash DU_q$.*

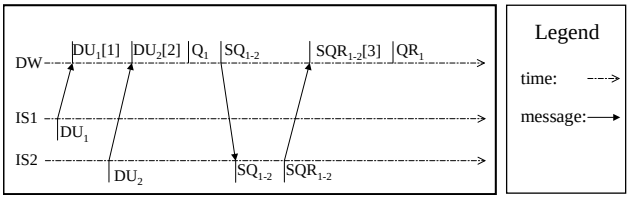

Figure 6: Definition of a **maintenance-concurrent** Data Update

For example, in Figure 6 DU_1 occurs at IS_1 and DU_2 occurs at IS_2 respectively [3]. We assume that DU_1 is received at time 1 which is earlier than DU_2 at time 2 by the DW. Then DU_1 is renamed as $DU_1[1]$ with timestamp 1, and DU_2 is renamed as $DU_2[2]$ with timestamp 2 by the *AssignTimeStamp* process. Then the VM process generates a query Q_1 in order to handle DU_1. The query Q_1 will then break down into sub-queries to be handled by each IS. The sub-query $SQ_{1,2}$ that is generated from Q_1 is sent to IS_2. Since DU_2 occured before IS_2 received $SQ_{1,2}$, and because of assumption 1, DU_2 is received earlier than the query result of $SQ_{1,2}$, denoted as $SQR_{1,2}$, at the DW. Then, *AssignTimeStamp* will assign timestamp 3 to $SQR_{1,2}$ and denotes it as $SQR_{1,2}[3]$ 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 $SQR_{1,2}[3]$. This means that DU_2 affected the sub-query result $SQR_{1,2}$. By Definition 1, DU_2 is said to be a **maintenance-concurrent** data update. As demonstrated by this example, we can discover the **maintenance-concurrent** problem solely based on the local timestamps associated with sub-query results and update messages at the DW middle layer.

[3] Here 1 and 2 are two index numbers of ISs. They are independent from the 1 and 2 of DUs' subscripts.

4.3 Multi-Threading of Processes in PSWEEP

From Section 3.1, we can see that the major drawback of the SWEEP algorithm is that it cannot handle data updates in a parallel fashion. In order to improve the performance of SWEEP, we analyze its characteristics to access which processes are potential candidates for being multi-threaded.

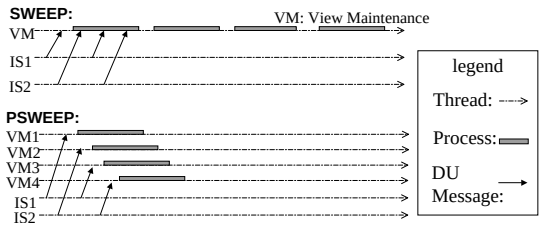

Figure 7: Ideal Execution Plan of PSWEEP

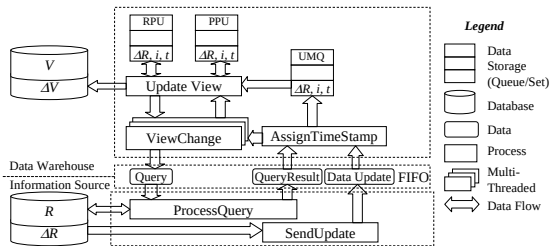

Figure 8: PSWEEP Architecture

From Figure 2, we can see that the *UpdateView* process is responsible for simply committing the effect of a DU to the materialized view of the DW, while 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) handling multiple queries per each DU, (2) processing remote queries to ISs, and (3) handling these remote queries one by one in some sequential order. Hence, most of the time the ViewChange function is idle waiting for either messages to be shipped across the FIFO network or ISs to compute query requests (see top of Figure 7). Hence, it is the one functionality in the system that should be multiplied to handle more than one data update at the same time. Our goal here is to interleave the handling of many DUs by running parallel VM threads, and leading to potential performance improvements as depicted in Figure 7.

We thus propose the new architectural design for PSWEEP given in Figure 8. As we can see the *ViewChange* function is now promoted to be a multi-threaded process. There are two more modules in addition to those of SWEEP designed to help solve the problem of the detection of concurrent DUs, called the *Related Processed Update* (RPU) set and the *Parallel Processing Update* (PPU) set. Details of their functionality are given in Section 4.4. The other change of the PSWEEP architecture is that every item in the *Update Message Queue* (UMQ) now has an annotated timestamp, denoted by t in Figure 8.

4.4 Detection of Maintenance-Concurrent DUs under Parallel Execution

In order to handle concurrent data updates in a parallel fashion, 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, PSWEEP 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 first glance, it may appear that PSWEEP could

possibly extend SWEEP's concurrent DU detection scheme by checking both the UMQ and PPU. We now show that this checking would not be sufficient by presenting one illustrating example in which PSWEEP would fail to detect a concurrent data update.

4.4.1 Example of the Maintenance Concurrent Detection Problem

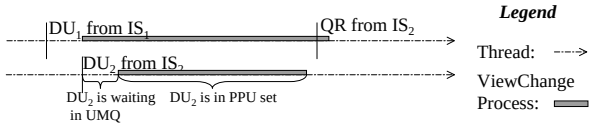

Figure 9: **Maintenance-Concurrent** DU Detection Problem.

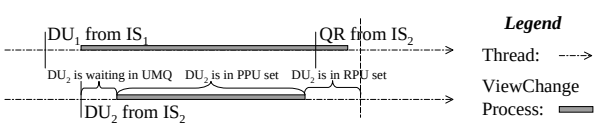

Figure 10: Solution of **Maintenance-Concurrent** DU Detection Problem.

In Figure 9 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_1 processed at IS_2, which is used to calculate the effect of DU_1. So DU_2 is a **maintenance-concurrent** data update by Definition 1. In Figure 9 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 PSWEEP system after having been handled. Note that we assumed that DU_2 is handled faster than the processing of DU_1. So, before DU_1 receives the query result QR_1 from the IS_2, DU_2 has already been removed from the PSWEEP system. Hence, PSWEEP can no longer detect the **maintenance-concurrent** data update DU_2. 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.

4.4.2 Solution of Maintenance Concurrent Detection Problem

The reason for this problem is we haven't keep track of data updates that while having been completely handled already may still be needed for the purpose of detecting future concurrent data updates. We propose to address this problem by keeping the completely handled and actually committed data updates in a special holding place where we can check for potentially concurrent data updates. It is important to note that we do not require the delay of committing the DU to the DW as in Strobe, i.e., quiescence is still not required by PSWEEP. We call these data updates *related processed updates.*

Definition 2 [Related Processed Updates]*: A data update DU_i already completely handled (and committed) 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.*

This definition identifies the situation in which a committed DU_j could have caused a **maintenance-concurrent** detection problem, and hence the knowledge about the occurrence of this DU_j must be kept track of by DW. In order to solve this problem, we put a second data storage into the PSWEEP system

(Figure 8), called the *related processed updates set* or *RPU* in short. It will maintain all related processed updates until their retention becomes no longer necessary. After the *ViewChange* process handles one data update DU_i and commits the effect of that DU_i to the data warehouse, if the DU_i is found to be a *related processed update*, the *ViewChange* process will put it into the RPU set.

Figure 10 illustrates the relative time intervals for the DU along the time-line of the *ViewChange* process. In Figure 10 we can see that the concurrent data update DU_2 is not immediately removed from the system after committing the effect of it into the data warehouse. Instead we put the DU_2 into RPU, because it was handled later than DU_1 but finished before the handling of DU_1. Hence, DU_2 is a *related processed update*. When the middle layer receives the query result QR_1 from IS_2, it checks UMQ, PPU and RPU for concurrent data updates. This time, it finds the DU_2 from RPU, and compensates the effect of DU_2 from the query result QR_1. Hence, QR_1 is corrected and the effect of DU_1 is computed correctly. This ensures the final DW state will be consistent with the IS space.

The *ViewChange* process also cleans up the RPU to remove all *unrelated* DUs. This can be done by simply checking the timestamps and IS indices of DUs in the PPU and the RPU.

Definition 3 *A data update DU_i in RPU is said to be unrelated if the following condition holds: there does not exist any DU_j in PPU such that the timestamp t_j for DU_j is smaller than the timestamp t_i for DU_i.*

Lemma 1 *A data update DU_i in PPU that is* **unrelated** *by Definition 3 can be safely removed from the RPU set while guaranteeing that no* **maintenance-concurrent** *DU related to this DU_i will be missed, i.e., no potentially conflicting DU_j could ever be added to the PPU or could exist in the UMQ after this removal time.*

Proof can be given intuitively based on Definition 2 and Definition 3.

4.5 Correction of Maintenance-Concurrent Updates

The SWEEP's local compensation query is an individual query. It works because the local compensation query is simple and done locally only. Once we have detected all concurrent updates [AAS97], no additional solution needed to make it also work in PSWEEP.

4.6 The Variant DW Commit Order

We now show that even if individual DU_i can be handled correctly by using some local compensation technique, the final DW state after committing these correctly computed effects can still lead to an inconsistent DW state due to variant commit orders.

4.6.1 The Problem Caused by Variant DW Commit Orders

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 and move the IS state from ISS_1 to ISS_2. No matter in which order they are executed the IS space will be unique, denoted by the state ISS_2.

Assume DW defined upon the IS space is affected by these two data updates. If we are doing parallel incremental view maintenance of the DW, different orders of committing the effects of the two data updates may occur and hence different final extents of DW may result. However, only one of them can be the correct one. The extent of the DW is correct if it is the same as the view extent recomputed directly from the IS space with state ISS_2.

Definition 4 *We define the DW extent to be* **correct** *under multiple IS updates if it is equal to the extent we get when first updating the IS space with DU_1 and DU_2 and then recomputing the DW extent from scratch (Figure 11).*

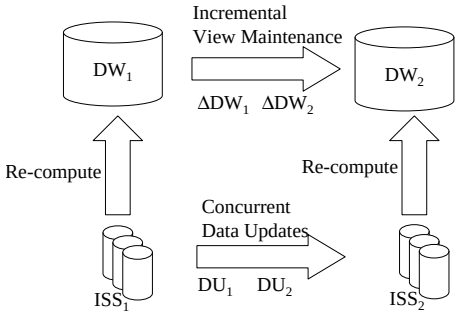

Figure 11: Correctness Criteria of Extent of DW

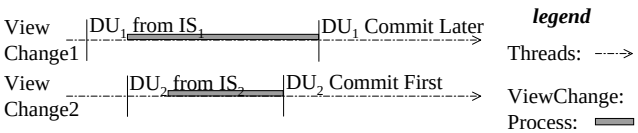

Figure 12: Variant DW Commit Order Problem

Definition 5 *We call the problem of inconsistency between the DW and the IS space caused by the variant* **DW-Commit-Order** *the* **Variant-DW-Commit** *problem.*

Theorem 1 *The* **Variant-DW-Commit** *problem defined by Definition 5 will only occur when first an add-tuple and then a delete-tuple, which will modify same tuples in DW, are received by the middle layer and both are handled in parallel by PSWEEP.*

Proof: Please see Appendix B for the proof.

Figure 12 illustrates the **Variant-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, since PSWEEP does not require any quiescence but rather commits updates as soon as their handling is completed.

4.6.2 An Example of the DW-Commit Problem

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 13. 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 [4] defined by $A \times DU_1$, is shown in Figure 13. 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 13 [5]. 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 14, we get the correct DW state.

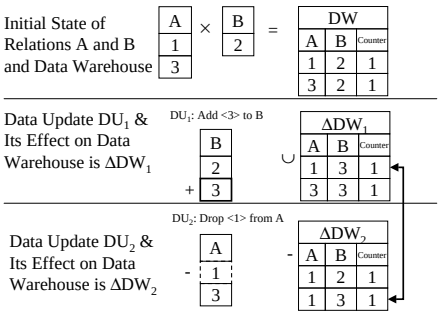

Figure 13: Environment of Example

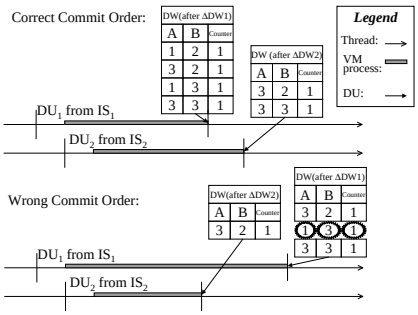

Figure 14: Example of Variant DW Commit Order Problem

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 14. 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, we get the wrong DW extent depicted in Figure 14, 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 14.

4.6.3 Solution of the DW-Commit Problem

We now provide a solution, called negative-counter solution, that guarantees that the extent of the DW after the incremental view maintenance of DU_1 and DU_2 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

[4] 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.

[5] 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.

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.

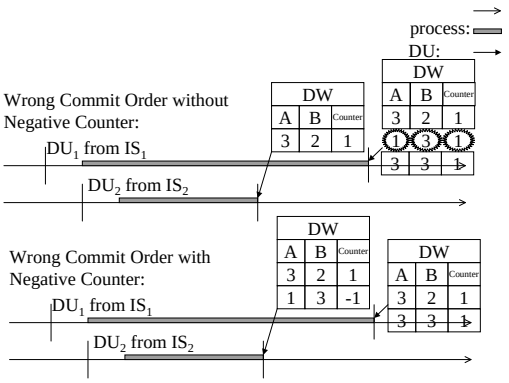

Figure 15: Example of Solution of Variant DW Commit Order Problem

Figure 15 compares the extent of the DW from our previous example from Section 4.6.1 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. Hence, this deletion has no effect. 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 15, 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 $< 1,3 >$ 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. 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 2 *Given the negative counter mechanism described in Section 4.6, the counters of all tuples will always be positive when the DW reaches a quiescence state.*

The proof for Lemma 2 is shown in Appendix C.

Lemma 3 *The negative counter mechanism described in the algorithm in Figures 16 and 17 correctly solves the* **Variant-DW-Commit** *problem defined in Definition 5 if any two data updates are handled in parallel by the DW.*

The proof of Lemma 3 can be found in Appendix D.

Theorem 2 *The negative-counter based algorithm (Figure 16 and 17) correctly solves the* **Variant-DW-Commit** *problem for any number of* **maintenance-concurrent** *DUs.*

As shown in Appendix E, the proof of this theorem can be given based on induction on the number of concurrent tuples by using Lemma 3. This theorem confirms the correctness of the final state of the DW after applying the effects of the data updates in any order.

4.7 Algorithms for the PSWEEP Processes

Based on the previous description of the key features of PSWEEP, we now can give details of the PSWEEP processes. The update and query server component employed at each IS is given in Figure 16. They are identical to those of the SWEEP [AAS97] algorithm.

```
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 16: Pseudo Code of Module Update&QueryServer of PSWEEP

Figure 17 depicts the software module that is employed at the DW for view maintenance. *ViewChange* is the main process that is invoked for every update (ΔR, 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 however also run a dedicated process to clean up the RPU set at some fixed time interval.

```
MODULE DataWarehouse;
  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(Δ R: Relation; UpdateSource:
    INTEGER; TimeStamp: INTEGER): RELATION
  VAR
    Δ V, TempView: RELATION;
    j: INTEGER;
  BEGIN
    Δ V = Δ R;
    /* Compute the left part of the incremental
    view resulting for Δ R */
    FOR (j = UpdateSource -1; j ≥ 1; j–) DO
      Δ V = ProcessSubQuery(Δ V, j, TimeStamp)
    ENDFOR;
    /* Compute the right part to the incremental
    view resulting from Δ R */
    FOR (j=UpdateSource+1; j ≤ n; j++) DO
      Δ V = ProcessSubQuery(Δ V, j, TimeStamp)
    ENDFOR;
    V = V + (Δ V);
    /* erase the unrelated Updates */
    CleanUp RelatedProcessedUpdate;
    CRITICAL AREA
      REMOVE (Δ R, i, t)
        FROM ParallelProcessingUpdate;
      IF (Δ R, i, t) is related
      THEN PUT (Δ R, 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( Δ V: RELATION; j:
    INTEGER; t: INTEGER): RELATION
    TempView = Δ V;
    SEND Δ V TO Data Source j;
    /* The Δ V in the next line has
    already time stamp assigned by AssignTimeStamp
    process */
    RECEIVE Δ V 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 Δ V = Δ V - 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 (Δ R, i, t) TO UpdateMessageQueue;
        ELSE
          Assign getCurrentTime() to Δ V
        ENDIF
    FOREVER;
  END AssignTimeStamp;

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

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

Figure 17: Pseudo Code of Module DataWarehouse

5 PSWEEP Evaluation

5.1 Level of DW Consistency

[ZGMHW95, ZGMW96] define different notions of consistency of a view depending on how the updates are incorporated into the view at the data warehouse:

- *Weak consistency* where the updates are eventually incorporated into the materialized view and every state of view corresponds to a valid state of the IS space.
- *Complete consistency* where every state of the data sources is reflected as a distinct state at the data warehouse. The ordering among the state transformations at the data sources are preserved at the data warehouse by having an equivalent ordering of distinct DW states.

PSWEEP reaches the *Weak Consistency* level of consistency. If we enforce that effects must be committed in the order that as they arrive, then it could achieve the *Complete Consistency* level of consistency.

5.2 Experimental Set: Parameters and Notations

Factors of the Experiments	
n	Number of Concurrent Data Updates that happened at the same time or before the middle layer starts to handle them.
m	Number of ISs.
p	Size of PPU (maximal number of parallel threads).
Measurement of the Experiments	
M	Number of Messages.
T	Elapsed Time of Handling a Set of Data Updates.
t	Elapsed Time of Handling one Data Update.

Table 4: Parameters of the Preliminary Evaluation

In the reminder of this section, we will discuss the performance of PSWEEP compared to the performance of SWEEP in terms of number of messages and total time. The factors we used in this analysis are described in Table 4. For simplicity, we assume we have m ISs and n data updates, and n mod m is 0. The view is defined over all of m ISs and is assumed to be affected by all n DUs. We also assume the data updates are evenly distributed, and all data updates happened before any processing of the data updates. So they are all concurrent with one another. For simplicity, we label the ISs in an increasing order from 1 to m and number the data updates from 1 to n as follows:

$$
\begin{array}{cccccc}
 & & IS\ index\ (j) \longrightarrow & & & \\
1 & 2 & 3 & 4 & \cdots & m \\
\hline
DU_1 & DU_2 & DU_3 & DU_4 & \cdots & DU_m \\
DU_{m+1} & DU_{m+2} & DU_{m+3} & DU_{m+4} & \cdots & DU_{2m} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
DU_{n-m+1} & DU_{n-m+2} & DU_{n-m+3} & DU_{n-m+4} & \cdots & DU_n
\end{array}
$$

5.3 Comparison in Number of Messages

The cost models for both SWEEP and PSWEEP are the same in terms of the total number of messages M. The reason for this is that PSWEEP simply parallelizes the execution of SWEEP instead of changing the way the view maintenance for each data update is done. There are however two kinds of messages send by the (P)SWEEP algorithms: remote queries and local queries. The number of the remote queries is denoted by M_{remote}, and the number of local queries by M_{local}, i.e., we have: $M = M_{local} + M_{remote}$.

Then M_{local} for every DU_i can be represented by a matrix of the number of messages in the number of DUs and number of ISs as shown below:

		$IS\ index\ (j) \longrightarrow$				
		1	2	3	$\cdots$	m
No.	1	0	$\frac{n}{m}$	$\frac{n}{m}$	$\cdots$	$\frac{n}{m}$
of	2	$\frac{n}{m}-1$	0	$\frac{n}{m}$	$\cdots$	$\frac{n}{m}$
DU	3	$\frac{n}{m}-1$	$\frac{n}{m}-1$	0	$\cdots$	$\frac{n}{m}$
(i)	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\downarrow$	$m+1$	0	$\frac{n}{m}-1$	$\frac{n}{m}-1$	$\cdots$	$\frac{n}{m}-1$
	$m+2$	$\frac{n}{m}-2$	0	$\frac{n}{m}-1$	$\cdots$	$\frac{n}{m}-1$
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
	$n-1$	0	0	0	$\cdots$	1
	n	0	0	0	$\cdots$	0

So the message cost M_{local} of DU_i at IS_j is defined by:

$$M_{local} = \begin{cases} \frac{n}{m} - k & j \leq (i-2) \bmod m + 1 \\ \frac{n}{m} - (k-1) & j > (i-2) \bmod m + 1 \\ 0 & i \bmod m = j \ or \ \frac{n}{m} < k \ or \ \frac{n}{m} < k-1 \end{cases}$$

with j is the index of ISs, and i is index of DUs.

$$k = \begin{cases} \frac{(i-2)}{m} + 1 & i > 1 \\ 0 & i = 1 \end{cases}$$

$M_{local} = \sum_{i=1}^{n-1} i - m \sum_{i=0}^{\frac{n}{m}-1} n = (\frac{m-1}{2m})n^2$

And, from [AAS97] we know that:

$M_{remote} = n \times (m-1) = (m-1)n.$

5.4 Comparison in Total Execution Time

Although the number of messages are the same for PSWEEP and SWEEP, the execution time of them are different. If we assume the overall time spend on a local-join is C_{local} and on a remote-join is C_{remote} per query, then we get the total time T for SWEEP as:

$$\begin{aligned} T_{SWEEP} &= M_{remote} \times C_{remote} + M_{local} \times C_{local} \\ &= (m-1)n \times C_{remote} + (\tfrac{m-1}{2m})n^2 \times C_{local} \end{aligned}$$

Because C_{remote} is much larger than C_{local}, for small numbers of n we can simplify the equation to:

$T_{SWEEP} = (nm-n) \times C_{remote} \approx O(n \times m \times C_{remote}).$

This shows that the time performance of SWEEP is linear in the product of the number of ISs and the number of data updates. Hence, for one DU, the estimate time is:

$t_{SWEEP} = (m-1) \times C_{remote}.$

For simplicity, we are going to use t instead of t_{SWEEP} for the following discussion. For PSWEEP, the size of the PPU set, denoted by p, as well as the query capability q of the IS space will affect the performance of PSWEEP. q shows how many queries can be processed at the same time at the IS space. We assume the overhead of generating a new thread and waiting for locks to access the critical area is a. Depending on the

value of q and p, there are two cases.

Case 1: $p < q$ means that compared to how many threads can be executed in parallel the database has much larger (infinite) query processing capabilities. Then for every data update in the UMQ, we can assess how long it will wait for being handled. The first data update DU_1 does not need to wait, DU_2 needs to wait time a, DU_3 need to wait time $2a$, and so on $DU_p = (p-1)a$. DU_{p+1} needs to wait time t because DU_{p+1} can start to be handled right after DU_1 is handled which is t. So, we get the following waiting sequence:

DU_i	1	2	3	$\cdots$	$p-1$	p	$p+1$	$\cdots$	$2p$	$2p+1$	$\cdots$
$Wait$	0	a	$2a$	$\cdots$	$(p-2)a$	$(p-1)a$	t	$\cdots$	$t+(p-1)a$	$2t$	$\cdots$

Hence for DU_i, the waiting time $Wait(i)$ is: $Wait(i) = \lfloor \frac{i-1}{p} \rfloor t + [(i-1) \bmod p]a$, with t, p, i, and a as defined above.

Case 2: $p \geq q$ means that the query capabilities of ISs limit the number of threads that should practically be run at the same time in order for the benefits of each thread in terms of performance improvement to out-weight its overhead in terms of system resources. This case is similar to the case 1. The waiting time $Wait(i)$ is: $Wait(i) = \lfloor \frac{i-1}{q} \rfloor t + [(i-1) \bmod q]a$

So in general, the waiting time for DU_i is:

$Wait(i) = \lfloor \frac{i-1}{min(p,q)} \rfloor t + [(i-1) \bmod min(p,q)]a$.

Then the total execution time of PSWEEP is: $T_{PSWEEP} = \max_{i=1\ldots n}(Wait(i)) + t$.

Let $k = min(p,q)$. If we assume $(n-1) \bmod k = 0$ and $t > (n-2)a$, then we can simplify T_{PSWEEP}:

$T_{PSWEEP} = \left(\frac{n-1}{k} + 1\right) t$

Comparing the performance of T_{SWEEP} and T_{PSWEEP} we observe that PSWEEP has approximately a k-fold better performance than SWEEP with $k = min(p,q)$[6]. This implies for example that if your DW implementation platform comfortably supports 10 concurrent threads, then a 10 fold performance improvement may possibly be achieved.

All the previous analysis is based on the assumption of DUs even distributed over the ISs. If the DUs are not evenly distributed, for example all the d DUs of IS_1 will be handled after the DUs of the IS_2, and so on, the overall performance of PSWEEP will be d times lower until we improve the query capability of IS_1.

There is a trade-off between C_{local} and C_{remote}. From previous discussion we know that for each DU, the handling time for it is $t = (m-1) \times C_{remote} + (\frac{m-1}{2m})n \times C_{local}$. Let's denote the first part as t_{remote}, and the second part as t_{local}. The ratio between t_{local} and t_{remote} is:

$\frac{t_{local}}{t_{remote}} = \frac{n}{2m} \times \frac{C_{local}}{C_{remote}}$

[6] Because we assume every IS is sequentially processing the incoming queries, the query capability of every IS is 1. Hence, the whole IS space with m ISs has query capability m, which means $k = min(p,m)$.

If we know $\frac{C_{local}}{C_{remote}}$ is 0.01, we have 10 ISs, and we want $\frac{t_{local}}{t_{remote}}$ less than 0.1 in order to ignore the cost of local join queries, then the maximum **maintenance-concurrent** DUs we can have is: 200. If we have more than 200 **maintenance-concurrent** DUs, the t_{local} will be no longer neglectable.

6 Conclusions

In this paper, we investigate the problem of parallel view maintenance. For this, we utilize the state-of-the-art VM solution called SWEEP as foundation for our work, because it has the least limitations as compared to alternate contenders in the literature. However, we identified the bottleneck of the SWEEP [AAS97] in terms of sequential handling of DUs. In this work, we identify several open issues that must be addressed to achieve parallel view maintenance, notable, concurrent data update detection in a parallel execution mode, and the variant DW commit order problem. We then present an integrated solution called PSWEEP that is indeed capable of handling these problems. We present the PSWEEP algorithm as an extension of the SWEEP algorithm and prove its correctness in terms of generating a correct final DW state. By analyzing the cost model of the SWEEP and the PSWEEP systems, we demonstrate that PSWEEP dramatically improves SWEEP's performance. We are now in the process of implementing both SWEEP and PSWEEP in our EVE data warehousing system in order to verify our analytic findings in a practical environment [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 his help with the EVE design and implementation. We also thank Prof. Nabil Hachem as qualification advisor as well as for his help and guidance with some of the ideas incorporated into this paper.

References

[AAS97] D. Agrawal, A. E. Abbadi, and A. Singh. 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] S. Chaudhuri and U. 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] L. S. Colby and I. S. Mumick. Staggered Maintenance of Multiple Views. In *Workshop on Materialized Views: Techniques and Applications*, pages 119–128, 1996.

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

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

[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.

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

[HGMW+95] J. Hammer, H. Garcia-Molina, J. Widom, W. Labio, and Y. Zhuge. The Stanford Data Warehousing Project. *Data Engineering Bulletin*, 18(2):41–48, 1995.

[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] A. Kawaguchi, D. F. Lieuwen, I. S. Mumick, D. Quass, and K. A. Ross. Concurrency Control Theory for Deferred Materialized Views. In *ICDT*, pages 306–320, 1997.

[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.

[LZW+97] W. Labio, Y. Zhuge, J. L. Wiener, H. Gupta, H. Garcia-Molina, and J. Widom. The WHIPS Prototype for Data Warehouse Creation and Maintenance. In *Proceedings of SIGMOD*, pages 557–559, 1997.

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

[QGMW96] D. Quass, A. Gupta, I. S. Mumick, and J. 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, and A. Nica. Evolvable View Environment. In *Demo Session Proceedings of SIGMOD'99*, 1999. to appear.

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

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

[WGL+96] J. L. Wiener, H. Gupta, W. Labio, Y. Zhuge, H. Garcia-Molina, and J. 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.

[YKL97] J. Yang, K. Karlapalem, and Q. Li. A Framework for Designing Materialized Views in Data Warehousing Environment. In *International Conference Distributed Computing Systems*, 1997.

[ZGMHW95] Y. Zhuge, H. Garcia-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. Garcia-Molina, and J. L. Wiener. The Strobe Algorithms for Multi-Source Warehouse Consistency. In *International Conference on Parallel and Distributed Information Systems*, December 1996.

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

A Correctness Proofs of the PSWEEP algorithm

A.1 Notations and Algebra for Proofs

Notations used in the proofs are listed in Table 5.

$DU_j[i]$	a data update at IS and assigned time-stamp i by middle layer.
m	A **DW-Commit-Order** order of a-s and d-s denoted by a regular expression [7].
$E(DU)$	effect of the DU on the data warehouse.
$E < m >$	Extent after commiting a set of DUs following the **DW-Commit-Order** m.
$=$	means "has same final DW state as".
$V(R_1, R_2, ..., R_n)$	extent of V generated based on the extents of $R_1, R_2, ...R_n$.
a_i	add-tuple operation at IS.
d_i	is delete-tuple operation at IS.

Table 5: Notations

We assume all the views are select-project-join (SPJ) views.

B Proof of Theorem 1

Theorem 1: The **Variant-DW-Commit** problem defined by Definition 5 will only occur when first an add-tuple and then a delete-tuple, which will modify same tuples in DW, are received by the middle layer and both are handled in parallel by PSWEEP.

Proof: In order to prove that, we are going to examine all possible **DW-Receive-Order**s of two data updates, i.e., $E < a_1\ a_2 >$, $E < a_1\ d_2 >$, $E < d_1\ a_2 >$, $E < d_1\ d_2 >$, where a_i denotes an add-tuple operation at position i in **DW-Receive-Order**, and d_j denotes an delete-tuple operation at position j in **DW-Receive-Order**, where $1 \leq i, j \leq 2$.

Using this notation, we represent the Theorem 1 as: Only $E < a_1\ d_2 > \neq E < d_2\ a_1 >$, while $E < a_1\ a_2 > = E < a_2\ a_1 >$, $E < d_1\ a_2 > = E < a_2\ d_1 >$, and $E < d_1\ d_2 > = E < d_2\ d_1 >$, with the order in which the a_i and d_j are listed in the sequence denoting their commit order.

CASE 1: Add-Tuple Update Followed by Delete-Tuple Update: $E < a_1\ d_2 > \neq E < d_2\ a_1 >$.

We are going to show via an example that the following may hold: $E < a_1\ d_2 > \neq E < d_2\ a_1 >$.

Let's define the DW by $DW = V(R_1, R_2)$. R_1 is modified by data update 'a_1' that is adding one tuple. The updated R_1 is denoted by R_1'. R_2 is modified by data update 'd_2' that is dropping one tuple. The updated R_2 is denoted by R_2'. We assume the middle layer received 'a_1' before 'd_2'. So, $E(a_1) = (\cup)V(a_1, R_2)$, and $E(d_2) = (-)V(R_1', d_2)$. The updated DW, denoted by DW', is $V(R_1', R_2')$. The expected commit order is $E(a_1)$ then $E(d_2)$. If we follow the expected order we will get $DW' = DW + E < a_1\ d_2 > = DW \cup V(a_1, R_2) - V(R_1', d_2)$, which is $V(R_1', R_2')$. So, $DW + E < a_1\ d_2 > = DW'$.

The reason for this is the following. First we can divide $V(R_1', d_2)$ into two parts: $V(R_1, d_2)$ and $V(a_1, d_2)$. Since $d_2 \in R_2$, we know that $V(a_1, d_2)$ is in $V(a_1, R_2)$. $DW = V(R_1, R_2)$ and $d_2 \in R_2$, and hence $V(R_1, d_2)$

[7] For example, $(a|d)^2$ means that a sequence of a-s and d-s of length 2, e.g. $< a\ a >$, $< a\ d >$, $< d\ a >$, $< d\ d >$; $< a\ d >$ means a followed by d.

in DW. So, $V(a_1, d_2) + V(R_1, d_2)$, which is $V(R_1', d_2)$, is in $DW \cup V(a_1, R_2)$. So, no error occurs at the deletion of $V(R_1', d_2)$. $DW + E < a_1\ d_2 >$ actually represents the new extent DW'.

If **DW-Commit-Order** is the reverse order of **DW-Receive-Order**, which means that we commit $E(d_2)$ first and then $E(a_1)$, we get $DW + E < d_2\ a_1 >= DW - V(R_1', d_2) \cup V(a_1, R_2)$. This is not equal to DW' as explained below:

Note that $V(a_1, d_2)$ is missing from the equation which implies we have the **Variant-DW-Commit** problem. The reason for it is that $a_1 \notin R_1$, so $V(a_1, d_2) \notin DW$, since $DW = V(R_1, R_2)$. As a result, subtracting $V(a_1, d_2)$ will do nothing to the DW, so we can treat it as an empty set. In this case, we will return the faulty tuples $V(a_1, d_2)$ in the final result.

As the result, it shows $E < a_1\ d_2 > \neq E < d_2\ a_1 >$.

Case 2: Two Add-Tuple Updates: $E < a_1\ a_2 > = E < a_2\ a_1 >$

There is no **Variant-DW-Commit** problem for the DW under two add-tuple updates, simply because $DW \cup E(a_1) \cup E(a_2) = DW \cup E(a_2) \cup E(a_1)$.

Case 3: Delete-Tuple Updates: $E < d_1\ d_2 >= E < d_2\ d_1 >$

Assume we have relations R_1 and R_2, and the DW is defined by $V(R_1, R_2)$. Two delete data updates are d_1 of R_1 and d_2 of R_2. We denote R_1' (R_2') as updated relation R_1 (R_2), defined by $R_1 - d_1$ ($R_2 - d_2$). We denote the updated DW as DW', defined by $V(R_1', R_2')$. $E(d_1)$ and $E(d_2)$ are calculated below based on the **DW-Receive-Order** that d_1 is received before d_2. $E(d_1) = V(d_1, R_2)$ is the effect of d_1, and $E(d_2) = V(R_1', d_2)$ is the effect of d_2.

Case 3.A: If **DW-Commit-Order** is that $E(d_1)$ is committed before $E(d_2)$, then:

- Because $d_1 \in R_1$, we know that $E(d_1) \in DW$. Hence, $DW - E(d_1)$ doesn't have any tuple with negative count. So $DW - E(d_1)$ equals $V(R_1', R_2)$.
- Because $d_2 \in R_2$, $E(d_2) = V(R_1', d_2) \in V(R_1', R_2)$. Hence, $DW - E(d_1) - E(d_2)$ doesn't have any tuple with negative count. So $DW - E(d_1) - E(d_2) = DW - E < d_1\ d_2 >$ equals $V(R_1', R_2')$.

Case 3.B: if we reverse the **DW-Commit-Order** that $E(d_2)$ is committed before $E(d_1)$, then:

- Because $R_1' \subset R_1$ and $d_2 \in R_2$, $DW - E(d_2)$ doesn't have any tuple with a negative count. $DW - E(d_2) = V(R_1', R_2') \cup V(d_1, R_2)$.
- Then $E(d_1)$ belongs to $DW - E(d_2)$ because $E(d_1) = V(d_1, R_2)$ and $DW - E(d_2) = V(R_1', R_2') \cup V(d_1, R_2)$. Then $DW - E(d_2) - E(d_1) = DW - E < d_2\ d_1 >$ has no tuple with a negative count. Hence, $E < d_2\ d_1 >= V(R_1', R_2')$.

As conclusion, because d_1 and d_2 are both in the original DW, the **Variant-DW-Commit** problem does not occur. So $E < d_1\ d_2 >= E < d_2\ d_1 >$.

Case 4: Delete-Tuple Update followed by Add-Tuple Update: $E < d_1\ a_2 >= E < a_2\ d_1 >$

Assume we have relations R_1 and R_2. The DW is defined by $V(R_1, R_2)$. The two data updates are: delete-update d_1 of R_1 and add-update a_2 of R_2. The **DW-Receive-Order** is $< d_1 a_2 >$ that means d_1 received earlier than a_2. As before, we denote R_1' (or R_2') as updated relation R_1 (or R_2), defined by $R_1 - d_1$ (or $R_2 \cup a_2$). We denote DW' as the updated DW, defined by $V(R_1', R_2')$.

$E(d_1) = (-)V(d_1, R_2)$ is the effect of d_1, and $E(a_2) = (\cup)V(R1', a_2)$ is the effect of a_2.

Case 4.a: If **DW-Commit-Order** is $E(d_1)$ followed by $E(a_2)$:

- Because $d_1 \in R1$, $E(d_1) \in DW$. Hence, $DW - E(d_1)$ doesn't have any tuple with a negative count. It equals $V(R_1', R_2)$. And the union will also not generate any tuple with a negative count.

- $DW' = DW - E(d_1) \cup E(a_2) = V(R_1, R_2) - V(d_1, R_2) \cup V(R_1', a_2) = V(R_1', R_2) \cup V(R_1', a_2) = V(R_1', R_2')$. By the same argument as case 3, no tuple with a negative count appeared.

Case 4.b: If the **DW-Commit-Order** is that $E(a_2)$ is committed before $E(d_1)$, then it is trivial that $DW \cup E(a_2)$ has not generated any tuple with a negative count.

- $DW \cup V(R_1', a_2) = V(R_1, R_2) \cup V(R_1', a_2)$, and due to union only, no negative counters are generated.
- Because $d_1 \in R_1$ and $E(d_1) \in DW$, we know $E(d_1) \in (DW \cup E(a_2))$. Hence $(DW \cup E(a_2)) - E(d_1)$ doesn't have any tuple with a negative count.
- $DW' = DW \cup E(a_2) - E(d_1)$
  $= V(R_1, R_2) \cup V(R_1', a_2) - V(d_1, R_2)$
  $= V(R_1, R_2) - V(d_1, R_2) \cup V(R_1', a_2)$
  $= V(R_1', R_2) \cup V(R_1', a_2) = V(R_1', R_2')$

No **Variant-DW-Commit** problem occurs because the tuples to be deleted are in the original DW. So $E < d_1\ a_2 >= E < a_2\ d_1 >$. Q.E.D.

C Proof of Lemma 2

Lemma 2: Given the negative counter mechanism described in Section 4.6, the counters of all tuples will always be positive when the DW reaches a quiescence state.

Proof: First intuitively, if we recompute the view extent after data updates happened at the ISs, the DW will result in a state which only has tuples with positive counts. Hence, we know that tuples with negative counts will not appear in the stable (correct) state of DW.

Second, assume we have defined a view V upon two relations R_1 and R_2. The updated R_1' is defined by $R_1 + a_1$. The updated R_2' is defined by $R_2 - d_2$. In the proof of Theorem 1, we know that only a_1 being received earlier than d_2 by the DW can cause the **Variant-DW-Commit** problem. The tuple $V(a_1, d_2)$ will be negative after we apply $E(d_2)$ first. As we know, $E(a_1) = (\cup)V(a_1, R_2)$. Because $d_2 \in R_2$, so $V(a_1, d_2) \subseteq V(a_1, R_2)$, hence the negative tuple $V(a_1, d_2)$ will be erased by $E(a_1)$. This means the final result will be positive or an empty set. Q.E.D.

D Proof of Lemma 3

Lemma 3: The negative counter based mechanism described in algorithm in Figures 16 and 17 correctly solves the **Variant-DW-Commit** problem if any two data updates are handled in parallel by the DW.

Proof: We will prove Lemma 3 based on the Theorem 1 and Lemma 2. We can represent this lemma by the following equation:

$E < (a|d)^2 >= E < (a|d)^2 >$ with i a's, 0<=i<=2, and $(2 - i)$ d's.

This means that for two DUs no matter how they commit to the DW, the final state of the DW will be consistent with the IS space if we use the negative counter technique. We break this discussion into four sub cases:

- case1: $E < a_1\ a_2 >= E < a_2\ a_1 >$
- case2: $E < d_1\ d_2 >= E < d_2\ d_1 >$

- case3: $E < d_1\ a_2 >= E < a_2\ d_1 >$
- case4: $E < a_1\ d_2 >= E < d_2\ a_1 >$

The cases 1,2, and 4 have already been shown to not produce any **Variant-DW-Commit** problem by Theorem 1. Hence we do not need to discuss them further. So we will only need to explain how the negative count concept solves the **Variant-DW-Commit** problem of $E < a_1\ d_2 >= E < d_2\ a_1 >$.

Assume we have a data warehouse with two ISs that have R_1 and R_2 respectively, defined by $DW = V(R_1, R_2)$. Assume the two DUs are a_1 and d_2 from R_1 and R_2 respectively. We denote updated R_1 after applying a_1 as R_1', and updated R_2 after applying d_2 as R_2'. $E(a_1)$ represents the effect of a_1 on the data warehouse DW, which should be $E(a_1) = V(a_1, R_2)$ in the sequential case when a_1 is received before d_2 (**DW-Receive-Order** is $< a_1, d_2 >$). $E(d_2)$ represents the effect of d_2 on the data warehouse DW, which should be $E(d_2) = V(R_1', R_2)$ in the sequential case when d_2 is received after a_1.

If we keep tuples with a negative count, then the $V(a_1, d_2)$ will be not really deleted. Rather the tuple that has been prematurely deleted will be kept in the database with a negative count. When $E(a_1)$ is added, the tuples with a negative count will finally cause the $V(a_1, d_2)$ to be deleted from DW. So the final result will be correct.

So, $E < a_1\ d_2 >= E < d_2\ a_1 >$.

That means if we keep a negative count for the tuples at the DW, no matter how we commit the effects, we always get the correct final state of DW.

E Theorem 2 Proof

Theorem 2: The negative-counter based algorithm (Figures 16 and 17) correctly solve the **Variant-DW-Commit** problem for any number of **maintenance-concurrent** DUs.

Proof by induction:

Induction hypothesis: Theorem 2 is true for any set of updates of length n.

Induction basis:

For n = 1: It's trivial

For n = 2: It's proved by Lemma 3.

Induction assumption: Assume hypothesis holds for $n = k$.

Induction step: Need to prove hypothesis for n = k + 1.

Assume data update commit sequence $S_1 = O_1, O_2, O_3, ..., O_{k+1}$ is the sequence we want to prove correct. Assume the desired ordering when receiving order = commit order is instead $S_2 = O_{i_1}, O_{i_2}, ..., O_{i_{k+1}}$.

Because of Lemma 3, we can see that, for a pair of data updates a_i, d_{i+1} we have:

$E < p\ a_i\ d_{i+1}\ q >= E < p > E < a_i\ d_{i+1} > E < q >$
$= E < p > E < d_{i+1}\ a_i > E < q >$
$= E < p\ d_{i+1}\ a_i\ q >$

For same reason, we have following equations:

$E < p\ a_i\ a_{i+1}\ q >= E < p\ a_{i+1}\ a_i\ q >$
$E < p\ d_i\ a_{i+1}\ q >= E < p\ a_{i+1}\ d_i\ q >$
$E < p\ d_i\ d_{i+1}\ q >= E < p\ d_{i+1}\ d_i\ q >$

These formulae indicate that the sequences at both sides with only one pair of data updates having a switch order will result in the same DW state using the negative count technique.

By using this pair switch operation we can convert the sequence S_1 to the corresponding position in sequence S_2.

First, it always exists O_l in S_1 that is corresponding to O_{i_1} in S_2 which can move to the first place of S_1 by using pair switching.

$E < S_1 >= E < O_1, ..., O_{l-1}, O_l, ..., O_{k+1} >$
$= E < O_1, ..., O_l, O_{l-1}, ..., O_{k+1} >$
...
$= E < O_l, O_1, ..., O_{l-1}, O_{l+1}, ..., O_{k+1} >$
$= E < O_l > E < O_1, ..., O_{l-1}, O_{l+1}, ..., O_{k+1} >$
$= E < O_{i_1} > E < O_1, ..., O_{l-1}, O_{l+1}, ..., O_{k+1} >$.

While $E < S_2 >= E < O_{i_1}, O_{i_2}, ..., O_{i_{k+1}} >= E < O_{i_1} > E < O_{i_2}, ..., O_{i_{k+1}} >$

Then, the remaining sequence of S_1 has length k. We know by our induction assumption we can convert it to the remaining sequence of S_2. So, the whole sequence S_1 has been shown to be equal to sequence S_2. Hence, $E < S_1 >= E < S_2 >$ for $n = k + 1$.

Q.E.D.