The VLDB Journal (2004) 13: 22–48 / Digital Object Identifier (DOI) 10.1007/s00778-003-0086-0

Parallel multisource view maintenance

Xin Zhang, Lingli Ding, Elke A. Rundensteiner

Department of Computer Science, Worcester Polytechnic Institute, Worcester, MA 01609-2280, USA;
e-mail: {xinz, lingli, rundenst}@cs.wpi.edu

Received: November 12, 2001 / Accepted: December 18, 2002
Published online: July 31, 2003 – © Springer-Verlag 2003

Abstract. In a distributed environment, materialized views are used to integrate data from different information sources and then store them in some centralized location. In order to maintain such materialized views, maintenance queries need to be sent to information sources by the data warehouse management system. Due to the independence of the information sources and the data warehouse, concurrency issues are raised between the maintenance queries and the local update transactions at each information source. Recent solutions such as ECA and Strobe tackle such concurrent maintenance, however with the requirement of quiescence of the information sources. SWEEP and POSSE overcome this limitation by decomposing the global maintenance query into smaller subqueries to be sent to every information source and then performing conflict correction locally at the data warehouse. Note that all these previous approaches handle the data updates *one at a time.* Hence either some of the information sources or the data warehouse is likely to be idle during most of the maintenance process. In this paper, we propose that a set of updates should be maintained in parallel by several concurrent maintenance processes so that both the information sources as well as the warehouse would be utilized more fully throughout the maintenance process. This parallelism should then improve the overall maintenance performance. For this we have developed a parallel view maintenance algorithm, called PVM, that substantially improves upon the performance of previous maintenance approaches by handling a set of data updates at the same time. The parallel handling of a set of updates is orthogonal to the particular maintenance algorithm applied to the handling of each individual update. In order to perform parallel view maintenance, we have identified two critical issues that must be overcome: (1) detecting maintenance-concurrent data updates in a parallel mode and (2) correcting the problem that the data warehouse commit order may not correspond to the data warehouse update processing order due to parallel maintenance handling. In this work, we provide solutions to both issues. For the former, we insert a middle-layer timestamp assignment module for detecting maintenance-concurrent data updates without requiring any global clock synchronization. For the latter, we introduce the negative counter concept to solve the problem of variant orders of committing effects of data updates to the data warehouse. We provide a proof of the correctness of PVM that guarantees that our strategy indeed generates the correct final data warehouse state. We have implemented both SWEEP and PVM in our EVE data warehousing system. Our performance study demonstrates that a manyfold performance improvement is achieved by PVM over SWEEP.

Keywords: Data warehousing – Parallel view maintenance – Concurrent data updates – Performance evaluation

1 Introduction

1.1 Background on view maintenance

A data warehouse integrates data from multiple information sources and then stores it in the form of materialized views (MVs). The information sources may be heterogeneous, distributed, and autonomous. When the data in any information source change, the materialized views at the data warehouse need to be updated accordingly. The process of updating a materialized view in response to the changes in the underlying source data is called view maintenance. The view maintenance problem has evoked great interest in the past few years.

It is popular to maintain the data warehouse incrementally by only recomputing a minimal delta to the view extent based on the particular source change [1,13,17] instead of recomputing the whole extent. Recomputation is prohibitively expensive due to the large size of data warehouses and the enormous overhead associated with the data warehouse loading process. Because the data warehouse usually needs to connect to and exchange information with multiple information sources through the network for accomplishing an incremental maintenance process, this process will be fairly time costly as well. It is hence unacceptable to block the update transaction at an information source in order for the integrator to immediately accomplish the view maintenance process. Instead, the

This work was supported in part by the NSF NYI grant IIS-979624 and NSF CISE Instrumentation grant IRIS 97-29878 and NSF grant IIS-9988776.

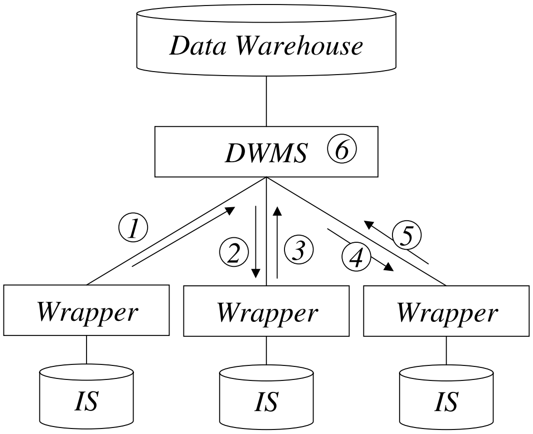


Fig. 1. Sequential view maintenance strategy

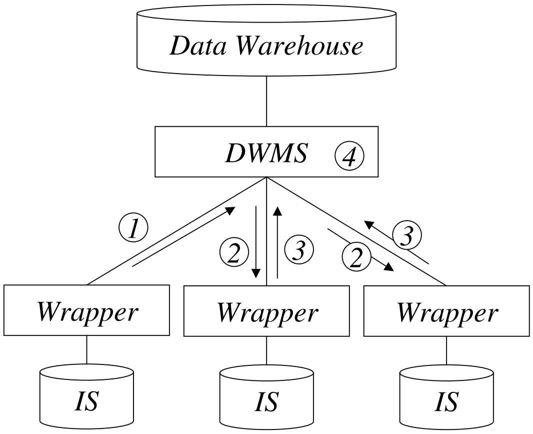


Fig. 2. Fully concurrent view maintenance strategy

maintenance is performed in a process separate from the actual source update transaction and thus called deferred view maintenance [4,12].

Since deferred view maintenance does not block the update transaction of the underlying information sources, there will be a time period during which the extent of the data warehouse will be temporarily inconsistent with that of the information sources. In recent years, there have been a number of algorithms proposed for such deferred view maintenance [5, 8,13,17,20,25,32]. In general, they fall into two strategies, namely, sequential view maintenance strategies (Fig. 1) and fully concurrent view maintenance strategies (Fig. 2).

In Fig. 1, the data update arrives at the data warehouse management system (DWMS) in step 1. Then the maintenance query is decomposed into multiple subqueries that are sequentially processed in steps 2 to 5. In this case, all the joins from the view definition are pushed down to be computed directly by the information sources to reduce the size of intermediate data. Finally, the computed result is committed to the data warehouse in step 6. This strategy significantly reduces the partial results shipped over the network as well as the load and responsibility placed upon the data warehouse server. However, this sequential maintenance processing strategy requires the data warehouse manager to wait for the processing of the database server in each information source as well as for transmissions of results or update messages over the network between data warehouse and information sources. However, most of the information sources would be idle most of the time. SWEEP [1] uses this sequential strategy for handling distributed information sources.

In Fig. 2, after the data update has arrived at the DWMS in step 1, the maintenance query again is decomposed into multiple subqueries and each subquery is sent to one of the information sources in step 2. All the partial results are returned from the sources in step 3 and joined at the DWMS in step 4. This strategy uses the various information sources in the environment at the same time, thus parallelizing their usage. However, step 4 may be very expensive due to the potentially large partial query results returned from the underlying sources. This is so because individual subqueries may not be able to exploit join conditions from the view query when retrieving data due to the fact that all such subqueries are being sent out at the same time. Second, this increased cost is due to the complexity of the joins now required to be performed at the data warehouse in order to integrate all the partial results. For this reason, the information sources may sit idle for a long time while the data warehouse manager processes the complex multiway join before the next data update can be handled. Strobe [31] basically follows this full concurrent strategy.

More recently, Posse [19] has supported different strategies for maintenance queries in order to have a better tradeoff between the size of the messages and number of queries that can be executed concurrently. Basically, it can interleave the concurrent and sequential strategies explained above into one hybrid solution so as to handle one single data update in the most effective manner.

As we can see, no matter which strategy we pick, because the data updates are handled one by one, some information sources are idle during the maintenance process. Thus, we now propose that we can further improve the overall maintenance performance by exploiting the computation power of those idling information sources. Any solution that manages to improve the performance of the view maintenance process is critical because it would reduce the time during which the data warehouse lags behind and thus is inconsistent with the information sources. Consequently, such optimization would lead to an improvement of the timeliness and hence quality of the data warehouse content important for time-critical applications, like stock market and monitoring-control applications. Optimization of data warehouse maintenance is hence the focus of our work.

1.2 Our approach – PVM

While previous work as described above focuses on the handling of one single update, in this paper we now propose an algorithm called Parallel View Maintenance, or PVM, that can efficiently process *a set of* concurrent data updates. PVM preserves all the advantages of its predecessors, in particular SWEEP [1] and POSSE [19], while overcoming their main limitation in terms of information sources staying underutilized caused by the one-by-one processing of information source updates. In particular, both SWEEP and POSSE handle data updates *one at a time*, forcing all updates to queue at the data warehouse manager until all updates received before the current one have been fully processed and incorporated into the warehouse extent.

Although a variation called Nested-SWEEP [1] handles multiple updates in a more efficient way by reusing part of the query results, it places more demands on the updates, e.g., it requires the updates to be noninterfering 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 maintenance processes of different updates. Such optimization leads to an improvement of the timeliness and hence quality of the data warehouse content, which is important for time-critical applications like stock market and monitoring-control systems.

In this work, we identify two research issues central to enabling parallel handling of the maintenance of a set of updates. 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 data warehouse, which we call the out-of-order-DW-commit order, caused by the parallel execution of the view maintenance process without blocking incoming updates. Our PVM solution proposed in this paper solves both problems.

For the first issue, we introduce two data structures for storing concurrent data updates to help PVM detect the conflicting ones. We also introduce a local timestamp mechanism to identify each data update and query result received by the warehouse manager in order to help detect concurrent data updates, which is much less restrictive compared to requiring a global timestamp mechanism. For the second issue, we extend the meaning of counters kept for tuples at the data warehouse to also include negative counts.[1] A negative counter now indicates the number of faulty deleted tuples due to the out-of-order-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. A full implementation of PVM and a popular contender in the literature (SWEEP) within our EVE [22] data warehousing testbed has been accomplished to allow for experimental evaluation of these two techniques within one uniform system. Our experimental studies show that PVM achieves a manyfold performance improvement over SWEEP based on the maximum number of threads that can be executed concurrently in the given system configuration.

1.3 Contributions

The main contributions of this work are:

- Identify the performance limitation of the state-of-the-art view maintenance solutions in the literature in terms of sequential handling of a set of updates and characterize research issues to be addressed to achieve parallel execution, in particular, the concurrent data update detection problem and the out-of-order-DW-commit problem.
- Introduce a solution strategy to solve the concurrent update detection problem in a parallel execution mode based on a data warehouse timestamp mechanism and some auxiliary data.
- Develop a solution strategy to the parallel view maintenance process without blocking the commit phase to the data warehouse by extending the range of duplicate counters to allow for negative values to keep track of and then compensate for faulty tuples generated by the out-of-order-DW-commit.

While the basic ideas of PVM were first presented in a workshop paper [29], we now offer the following additional contributions:

- Prove the correctness of the PVM solution to show that it produces the same final data warehouse extent as a complete recomputation algorithm.
- Develop a cost model and present an analytic evaluation of PVM using this model that characterizes its performance advantages over previous solutions.
- Implement both the PVM and SWEEP algorithms in a uniform environment, namely, the EVE data warehouse system [22], to conduct a comparative study of these strategies since the source code of view maintenance algorithms including SWEEP has not been made available to the community.
- Conduct a set of experimental studies that analyze the behavior of PVM and verify the performance benefits achievable by PVM under different system settings over alternate contenders.

1.4 Outline

The next section presents background material including an introduction to view maintenance basics. In Sect. 3, the PVM solution, including the two open research issues and their solutions, are presented. The different levels of data warehouse consistency in the context of distributed environments and PVM's consistency level are discussed in Sect. 4. In Sect. 5, the cost model and analytic evaluation of the performance of PVM are presented. The design and implementation of the PVM system are detailed in Sect. 6. Section 7 presents results of our experimental performance study. In Sect. 8, the related work is described, and conclusions are discussed in Sect. 9.

2 Background

2.1 Background on view maintenance

View maintenance is concerned with keeping materialized views up to date when the underlying information sources are modified over time. The straightforward approach of recomputing the complete view extent for each data update is not realistic in most practical settings. Instead, an incremental solution that only calculates the individual effect of each source update on the warehouse and then 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 happens at a base relation R_i and is received by the warehouse

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

Table 1. Notations used in this paper

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 subquery of Q_i being sent to IS_m
$SQR_{i,m}$	A subquery result of $SQ_{i,m}$
$DU_i[t]$	DU_i with unique timestamp t
ΔV_i	ΔV_i 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

manager, the incremental change is computed by the query in Eq. 1:

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

The query in Eq. 1 is computed by sending down subqueries to the respective information sources [9,10]. For simplicity, let us focus on SPJ views here, while in a later discussion section we discuss possible generalizations. The *attributes* are the columns in the view that are being projected out from the query result. The *predicates* are used to filter the query result. To explain the generation and execution order of the subqueries of Eq. 1, we introduce the notations described in Table 1.

A la SWEEP [1], the subqueries are generated in a sequence that starts to join 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:

$$SQ_{j,i-1} = \prod \sigma(R_{i-1} \bowtie \Delta R_i)$$
$$SQ_{j,i-2} = \prod \sigma(R_{i-2} \bowtie SQR_{j,i-1})$$
$$...$$
$$SQ_{j,1} = \prod \sigma(R_1 \bowtie SQR_{j,2})$$

Scan right:

$$SQ_{j,i+1} = \prod \sigma(SQR_{j,1} \bowtie R_{i+1})$$
$$SQ_{j,i+2} = \prod \sigma(SQR_{j,i+1} \bowtie R_{i+2})$$
$$...$$
$$SQ_{j,n} = \prod \sigma(SQR_{j,n-1} \bowtie R_n)$$

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 when the updates are spaced far enough in time so as to allow for completing the incremental computation of the view by executing the distributed *maintenance query* in Eq. 1 before any new update occurs. However, due to the autonomous nature of the sources participating in the data warehouse, this kind of separation in time cannot always be guaranteed. Hence we must maintain the data warehouse even under possibly concurrent data updates that are not separated far enough to accomplish the naive incremental maintenance illustrated above. Recent work by [31,1] proposes special techniques, e.g., *compensation queries*, to correct for possible errors in the query results returned by an individual source IS_i that had undergone a concurrent update.

2.2 View maintenance under concurrent data updates

Because the data updates are happening concurrently at the information sources, one data update DU_x at an information source could interfere with the processing of a query sent by the data warehouse manager to the same information source to handle another data update DU_y. We call this data update DU_x a **maintenance-concurrent** data update as it takes place concurrently with the maintenance process servicing other updates as formally defined in Definition 1.

Definition 1. *Let DU_i and DU_j denote two updates on IS_m and IS_n, respectively. The data update DU_j is called* **maintenance-concurrent** *with the update DU_i, if:*

i) DU_i is received earlier than DU_j at the middle layer, (i.e., $i < j$)

ii) DU_j is received at the data warehouse **before** *the answer of the subquery $SQ_{j,n}$ on the same IS_n generated by the data warehouse for handling DU_i.*

The data update DU_j is also called **maintenance-concurrent** *data update.*

Like previous authors [1,31], we also make the following assumption to facilitate the definition of the **maintenance-concurrent** property.

Assumption 1 *The order in which the data warehouse manager receives the messages from an information source is the same as the order in which the information source sends out the messages. It is known as FIFO Assumption.*

For example, in Fig. 3, DU_1 occurs at IS_a and DU_2 occurs at IS_b. 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 $DU_2[2]$, by the data warehouse manager. Then the data warehouse manager generates a query Q_1 in order to handle $DU_1[1]$. The query Q_1 will then be broken into subqueries to be handled by each information source. The subquery $SQ_{1,b}$ that is generated from Q_1 is sent to IS_b. Since DU_2 occurred 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 data warehouse. Then the data warehouse manager assigns $t = 3$ to $SQR_{1,b}$ and denotes it as $SQR_{1,b}[3]$. 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 $SQR_{1,b}[3]$ by the data warehouse manager. This means that DU_2 affected the subquery result $SQR_{1,b}$. By Definition 1, DU_2 is said to be a **maintenance-concurrent** data update.

The **maintenance-concurrent** DU will cause an anomaly in the compensation query. Assume we have a view defined as $V = R_1 \bowtie R_2 \bowtie R_3$. We have two data updates – DU_1 for R_1 and DU_2 for R_3. Assume R_1 after DU_1 is denoted as R_1', and R_3 after DU_2 as R_3'. In order to maintain the view V, we need to send two compensation queries down, one for each update. The first compensation query is $Q_1 = DU_1 \bowtie R_2 \bowtie R_3$, and

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

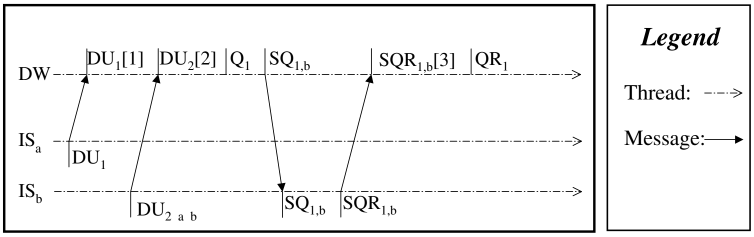


Fig. 3. Definition of a **maintenance-concurrent** data dp-date

the second compensation query is $Q_2 = R_1' \bowtie R_2 \bowtie DU_2$. The Q_1 is broken down into two subqueries to R_2 and R_3. The subquery $SQ_{1,2}$ for R_2 is $DU_1 \bowtie R_2$, and the subquery $SQ_{1,3}$ for R_3 is $SQR_{1,2} \bowtie R_3$. However, when $SQ_{1,3}$ arrives at R_3, the DU_2 has already been applied to R_3. So now the $SQ_{1,3}$ receives the result $SQR_{1,2} \bowtie R_3' = SQR_{1,2} \bowtie (R_3 + DU_2)$, instead of $SQR_{1,2} \bowtie R_3$. Hence the $SQR_{1,2} \bowtie DU_2$ is the anomaly of this subquery $SQ_{1,3}$ result.

All previous work including Strobe [31], SWEEP [1], and Posse [19] can handle data warehouse maintenance for **maintenance-concurrent** data updates. For this, both [1] and [19] introduce a local compensation strategy to remove such anomalies locally at the data warehouse. For each subquery affected by the concurrent data update, a local compensation query will be composed as the concurrent data update joins with the previous partial query result. In the previous example, a local compensation query for $SQ_{1,3}$ will be constructed as $SQR_{1,2} \bowtie DU_2$. This is exactly the same as the anomaly and can be computed locally because $SQR_{1,2}$ and DU_2 are both known values. Our solution PVM can be applied to extend any of these existing systems independently of the particulars of how they handle **maintenance-concurrent** data updates.

2.3 Assumptions

Below are the assumptions held by the previous view maintenance solutions in the literature that we continue to assume for our work on parallel view maintenance.

Assumption 2 *All information sources are independent of each other, in the sense that a data update at one information source will not propagate into other information sources.*

This assumption also holds in any of the prior work that we are aware of [1,30,31], even though it may not typically be stated explicitly.

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.*

Besides the assumptions explicitly defined in SWEEP [1], we note that the following assumptions hold for SWEEP as well as for PVM.

Assumption 4 *We assume an information source IS will send a notification message to the data warehouse manager only after the data update has been committed at that IS or the query result has been generated.*

Assumption 5 *The view definition will not be changed during the view maintenance process, and the information sources will only undergo data changes but no schema changes.*

Table 2. Schema definition of example of indirect effect of a data update

```
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))
```

Some studies [18,28] drop this assumption to evolve views in a dynamic environment.

Assumption 6 *Every information source will report all 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 information source.*

For example, we assume there are two relations R_1 and R_2 at IS1 as defined in Table 2. As one can see, R_1 has a foreign key referring to R_2. When there is a data update DU_1 that deletes tuple T1 from R_2, the indirect effect of DU_1 would be the deletion of these tuples from R_1 that have referred to T1.

2.4 Information source and data warehouse states

Definition 2. *A* **legal information source state** *from the data warehouse point of view is defined iteratively as follows:*

a. The initial state of an information source space, IIS_0, is a legal information source state.

b. For a sequence of actual updates at one IS_i for some i, denoted by $DU_{i,1}$, $DU_{i,2}$, ..., $DU_{i,n}$, an information source space state generated by applying any contiguous subsequence of $DU_{i,1}$, $DU_{i,2}$, ..., $DU_{i,k}$ with $1 \leq k \leq n$ to this IS_i is a legal information source space state.

c. For any pair of data updates DU_i and DU_j from different information sources IS_i and IS_j with $i \neq j$, the information source space state generated by applying DU_i to a legal information source space state and the information source space state generated by applying DU_j to a legal information source space state are both considered to be legal.

We consider a data warehouse state to be **legal** *if the data warehouse state can be generated from a legal information source state by execution of the data warehouse query.*

For example, if two data updates DU_1 and DU_2 occur at two different information sources, then the system could be in three legal information source states. That is, we only have committed DU_1, we only have committed DU_2, or we have already committed both.

Definition 3. *A data warehouse state is called* **quiet** *if there is no unhandled update queued in the data warehouse.* **Quiescence** *of information sources refers to a period of time when no new data updates occur at any information source until the data warehouse has handled all the reported data updates.*

3 PVM: a parallel view maintenance solution

A data warehouse maintenance algorithm for **maintenance-concurrent** updates must address the following four tasks: 1) execution of a distributed maintenance query, 2) detection of **maintenance-concurrent** data updates, 3) handling of **maintenance-concurrent** data updates, and 4) committing of maintenance query result to the data warehouse. First, the algorithm will generate the maintenance query for each update submitted from an information source. It needs to execute such a distributed maintenance query over all related ISs (thus indeed making a decision on the distributed query plan). Second, during the processing of the maintenance query, the algorithm needs to be able to detect any **maintenance-concurrent** data updates that happened concurrently with the maintenance process. Third, given such a detection, the algorithm must provide a mechanism to handle such concurrency, usually referred to as compensation queries [1,19,30,31]. Lastly, the corrected maintenance query result will be committed to the data warehouse. The first and third tasks vary from maintenance algorithm to maintenance algorithm [1,19,30,31], while we now propose that the second and fourth tasks can be generalized.

Given this observation, the PVM system separates the maintenance query execution and **maintenance-concurrent** data update handling tasks (tasks 1 and 3) from the **maintenance-concurrent** data update detection and maintenance query result committing tasks (tasks 2 and 4). PVM provides mechanisms solving the latter two tasks. This allows PVM to eventually plug in different maintenance algorithms with their own maintenance query execution and **maintenance-concurrent** data update handling strategies.

3.1 PVM architecture

The architecture of PVM is depicted in Fig. 4. Here the data warehouse manager is located at the data warehouse side, and the wrappers are located at the information source side connected by a FIFO network. Each information source wrapper contains two modules, i.e., *ProcessQuery* and *SendUpdate*. *SendUpdate* will report all the updates that happened at the information source to the data warehouse manager, and *ProcessQuery* will process the queries from the data warehouse manager and return the result.

The data warehouse manager of PVM employs five processes, namely, *Maintenance Manager*, *Commit Manager*, *Concurrency Detection*, *Assign Timestamp*, and *ViewChange*. The *Maintenance Manager* process monitors the updates received in the update message queue (UMQ) and spawns an instance of the *ViewChange* subprocess to handle an individual update if the *parallel processing update set* (PPU) is not full. The *Commit Manager* process will commit the view changes computed by the *ViewChange* process into the data warehouse. The *Concurrency Detection* process will check each received query result to determine if it is affected by incoming data updates by using three structures, i.e., *update message queue* (UMQ), *parallel processing update set* (PPU), and *related processed update set* (RPU). The *Assign TimeStamp* process gives a unique timestamp to each incoming message including update messages and query results. This is essential for detection of concurrency, as discussed in a later section. The *ViewChange* process is responsible for calculating the effect of every data update on the data warehouse by choosing a specific query plan. It is responsible for the following tasks: (1) generating the distributed query plan for each update, (2) sending remote subqueries to information sources, and (3) handling these remote queries. Handling of the remote subqueries of a maintenance task could be done either one by one in a sequential order [1,31] or, in recent work, possibly also allowing processing of the remote queries for one update to be run in parallel [19] and then joined locally at the data warehouse. Clearly, the *ViewChange* process is expensive. Hence, most of the time the *ViewChange* process is idle waiting for either messages to be shipped across the FIFO network or information sources to compute query requests. We propose that this is the process in the system that must be replicated to handle more than one data update at the same time. Our goal here is to interleave the handling of many updates by running parallel threads of *ViewChange* processes, thus it leading to potential performance improvements.

In Fig. 5, we contrast the sequential handling with the parallel handling of updates. In the upper part of the figure regarding the sequential handling of updates, the updates must wait to be handled by the view change process one by one, while the lower part depicts the parallel handling case of updates assuming two threads of the *ViewChange* process ($VC1$ and $VC2$ in our example figure) are active in the system. Hence the overall performance of handling those four updates can be significantly improved.

3.2 Detection of maintenance-concurrent updates 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 updates. We note that as in previous data warehouse management systems, PVM also holds one data structure, called update message queue (UMQ), in the data warehouse manager. It is used to buffer all the incoming update messages. 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 while the first one is taken off and handled by the data warehouse manager of

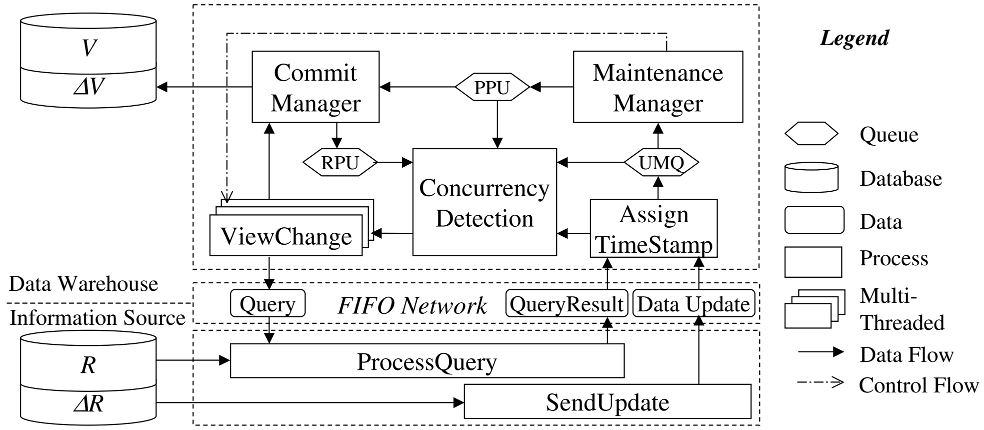


Fig. 4. PVM architecture

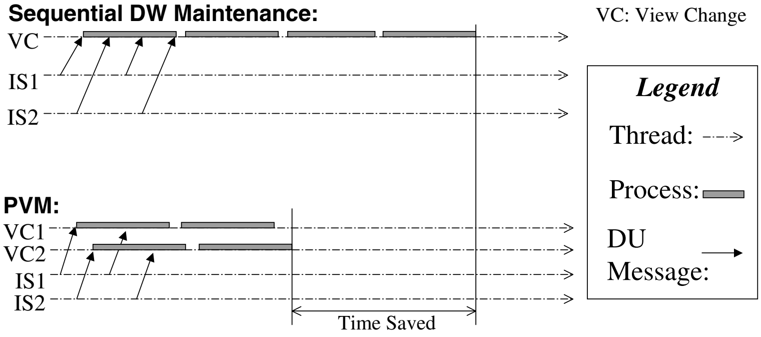


Fig. 5. Ideal execution plan of PVM

SWEEP. Given that all messages continue to wait in their arrival order in that UMQ, it is trivial to detect whether later updates arose during this process of maintaining the current update.

However, a concurrency detection that only relies on the UMQ will not work for PVM. PVM parallelizes the execution of the *ViewChange* process, i.e., it removes multiple updates from the UMQ and puts them into the *parallel processing update set* (PPU), which is a queue to store all the *parallel processing updates*, during the parallel handling process. At first glance it may appear that PVM could simply extend SWEEP's concurrent update detection scheme by checking both the UMQ and PPU data structures. We now show that this checking would not be sufficient by presenting one example illustrating that such a strategy would fail to detect concurrent updates.

3.2.1 A motivation example illustrating the maintenance-concurrent detection problem

Figure 6 shows two data updates, DU_1 and DU_2, from information sources IS_1 and IS_2, respectively. DU_1 arrived at the data warehouse 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 Fig. 6 we can see that DU_2 first waited in the UMQ to be handled. Then it is stored in the PPU while being handled. Finally, it is erased from the system by PVM 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 completely handled and thus removed from the PPU in general. Hence PVM can no longer detect the **maintenance-concurrent** data update DU_2. Thus the final state of the data warehouse after handling the DU_1 and DU_2 updates would be inconsistent with the state of the information sources.

3.2.2 Solution of detecting maintenance-concurrent updates

This problem exists because we haven't kept track of updates that, while having been completely handled already, may still be needed for the purpose of detecting prior 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 effect of the update to the data warehouse as in Strobe [1], i.e., quiescence is not required by PVM. We call these data updates *related processed updates*.

Definition 1. [Related Processed Updates]*: an update DU_i already completely handled (and committed) by the data warehouse manager is a* **related processed update** *if there exists at least one update DU_j received before DU_i by the data warehouse that has not yet been completely handled by the ViewChange process.*

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 the data warehouse. In order to solve this problem, we put a second data storage into the PVM system (Fig. 4), called the *related processed updates*

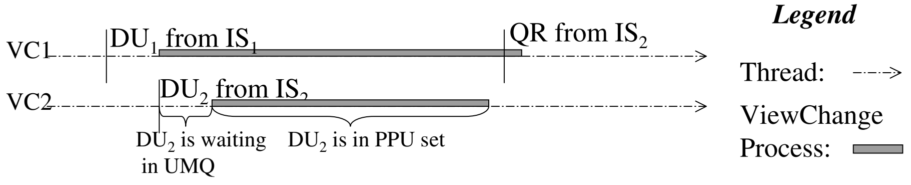


Fig. 6. Maintenance-concurrent update detection problem

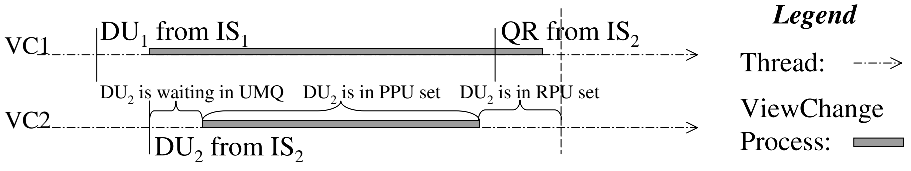


Fig. 7. Solution of **maintenance-concurrent** update detection problem

set or *RPU* for 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* as defined by Definition 1, then the *ViewChange* process will put it into the RPU.

Figure 7 illustrates the relative time intervals for the update along the timeline of the *ViewChange* process. In Fig. 7, we can see that the concurrent data update DU_2 is not immediately removed from the system after its effect is committed 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 data warehouse manager receives the query result QR_1 from IS_2, it checks UMQ, PPU, and now also RPU for concurrent data updates. This time it finds the DU_2 in the 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 that the final data warehouse state will be consistent with the information source space.

The *ViewChange* process also cleans up the RPU to remove all *unrelated* updates. This can be done by simply checking the timestamps and information source of updates in the PPU and the RPU, as explained below.

Definition 2. *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.*

From Definitions 1 and 2 we conclude Lemma 3.

Lemma 3. *A data update DU_i in PPU that is* **unrelated** *by Definition 2 can be safely removed from the RPU set while guaranteeing that no* **maintenance-concurrent** *update related to this DU_i will be missed, i.e., no potentially conflicting DU_j could ever be added to the PPU or exist in the UMQ after this removal time.*

From the definition of **maintenance-concurrent** updates (Definition 1), we know that any concurrent update would have to be between the update currently being handled and the maintenance query result. If the update DU_i is *unrelated*, then it must have been received earlier than all the other updates in PPU and UMQ. Hence, DU_i is not a **maintenance-concurrent** update, as this condition of **maintenance-concurrent** updates cannot hold. So Lemma 3 holds.

3.3 The out-of-order-DW-commit

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

3.3.1 An example illustrating the DW-commit problem

Assume we have two relations A and B with the data warehouse DW defined by $DW = A \times B$. The extents of A, B, and data warehouse DW are shown in Fig. 8. Two data updates DU_1 and DU_2 happened to B and A, respectively. DU_1 adds $< 3 >$ to B, while DU_2 deletes $< 1 >$ from A. The data warehouse manager receives DU_1 first and then DU_2. The effect of DU_1, denoted by ΔDW_1 defined by $A \times DU_1$, is shown in Fig. 8.[3] The effect of DU_2, denoted by ΔDW_2, is calculated after DU_2 is received. It is defined by $DU_2 \times B'$ as shown in Fig. 8.[4] After we commit the two ΔDW s to the data warehouse DW in the order of ΔDW_1 and ΔDW_2 as described in the *Correct Commit Order* part of Fig. 9, we get the correct data warehouse state.

If we reverse the commit order of the two ΔDW s (and the original data warehouse only counts positive tuples, as is the state of the art for current DBMS systems), we update the data warehouse in the wrong commit order, as shown at the bottom of Fig. 9. First, we would subtract ΔDW_2, which, given that $< 1, 3 >$ does not yet exist in data warehouse, would be equal to a no-op. Then after unioning ΔDW_1 to DW we get the

[3] SWEEP assumes that compensation queries are used so that each individual Q_i returns a correct query result QR_i. Thus 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 timestamp of the updates and query results, and we assume that the local compensation strategy as adapted from SWEEP. If two updates are received at the same time, we can choose any order to log them into the data warehouse. Then clearly the query for calculating the ΔDWs will depend on this selected order. However, the DW will still be updated correctly.

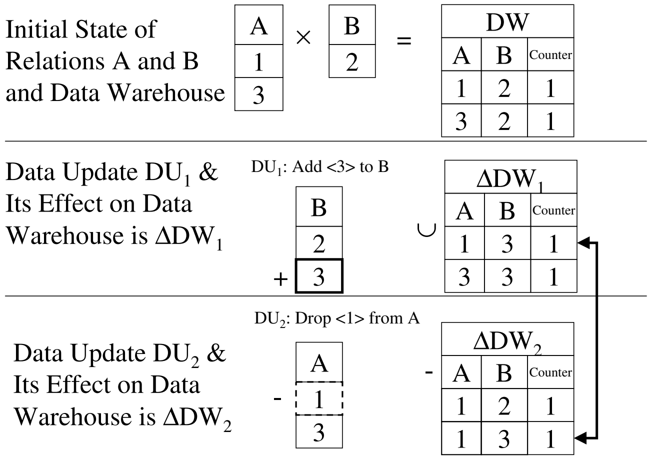

Initial State of Relations A and B and Data Warehouse
A
1
3
×
B
2
=
DW
A B Counter
1 2 1
3 2 1
Data Update DU1 & Its Effect on Data Warehouse is ΔDW1
DU1: Add <3> to B
B
2
+ 3
∪
ΔDW1
A B Counter
1 3 1
3 3 1
Data Update DU2 & Its Effect on Data Warehouse is ΔDW2
DU2: Drop <1> from A
A
- 1
3
-
ΔDW2
A B Counter
1 2 1
1 3 1

Fig. 8. Environment of an out-of-order-DW-commit example

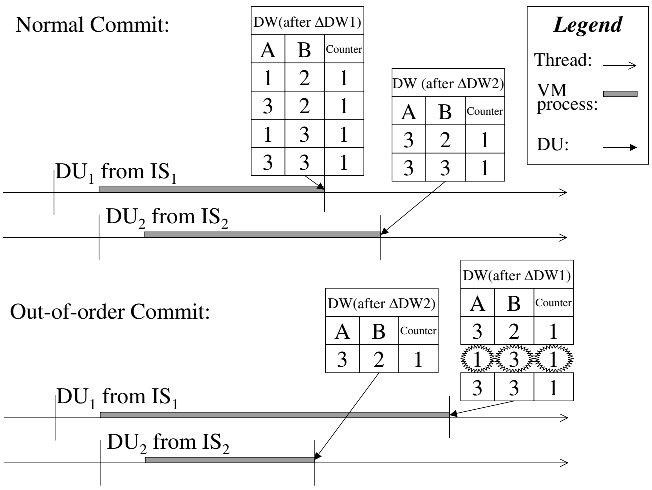


Fig. 9. Example of the out-of-order-DW-commit problem

wrong data warehouse extent depicted in Fig. 9, which now contains a faulty tuple $< 1, 3 >$ with counter 1. The correct answer instead should have been the data warehouse extent as depicted in the *Correct Commit Order* part of Fig. 9.

3.3.2 The problem caused by out-of-order-DW-commit

Assume two updates DU_1 and DU_2 happened on two different information sources IS_1 and IS_2 of the information source space with state ISS_1 and information source state transforms from ISS_1 to ISS_2 as depicted at the bottom of Fig. 10. No matter in which order these two updates are executed, the information source space will be unique, denoted by the state ISS_2.

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

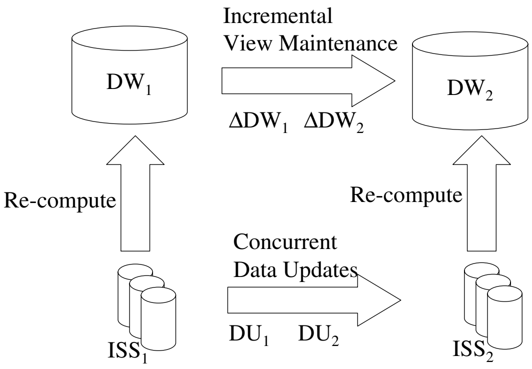


Fig. 10. Correctness criteria of extent of data warehouse

Definition 4. *The data warehouse extent of a quiet data warehouse state as defined in Definition 3 is defined as* **correct** *under multiple information source updates DU_1 and DU_2 if it is equal to the extent we get when first updating the information source space with DU_1 and DU_2 and then recomputing the data warehouse extent from scratch (Fig. 10).*

Definition 5. *We call the potential inconsistency between the final state of the data warehouse and the information source space caused by the out-of-order data warehouse commit the* **Out-of-Order-DW-Commit** *problem.*

Theorem 6. *The* **Out-of-Order-DW-Commit** *problem defined by Definition 5 will occur only when first an add-tuple and then a delete-tuple, both of which modify the same tuples in the data warehouse, are received by the data warehouse manager and both are handled in parallel by PVM.*

Proof. Please see Appendix B for the proof.

Figure 11 illustrates the **Out-of-Order-DW-Commit** problem with a time-line depiction. This problem may happen when the later handled data update DU_2 is processed 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 data warehouse prior to the effect of DU_1 being committed to the data warehouse since PVM does not require any quiescence but rather commits updates as soon as their handling is completed.

3.3.3 Negative-counter solution

We now provide a solution, called the negative-counter solution, that guarantees that the extent of the data warehouse 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 data warehouse extent as unique tuples with a counter that shows how many duplicates of each tuple exist. 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 data warehouse, 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."

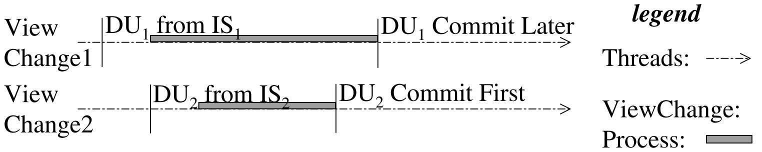


Fig. 11. Out-of-order-DW-commit problem

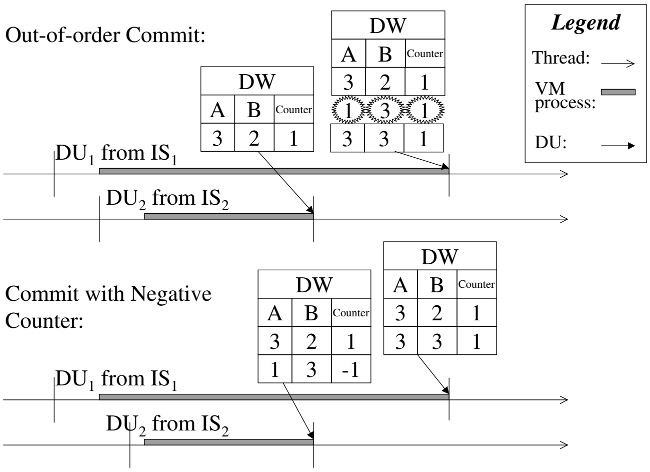


Fig. 12. Example of solution of the out-of-order-DW-commit problem

When the user accesses the data warehouse, any tuple with a counter ≤ 0 will be not visible.

We now apply the proposed solution strategy to our running example. In particular, Fig. 12 compares the extent of the data warehouse from our previous example from Sect. 3.3.2 with and without the negative counter. In the upper half of the figure, the data warehouse 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 data warehouse yet. Hence this deletion has no effect. Then, after we later commit the ΔDW_1, we now have added the faulty tuple $< 1, 3 >$ into the data warehouse, which should be deleted (however, the deletion was attempted too early).

In the lower half of Fig. 12, the data warehouse 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 data warehouse. So the tuple with a negative counter effectively remembers what tuple should have been deleted but cannot be deleted at the current time due to its not yet being in the data warehouse. For example, in the lower part, we can see that the data warehouse keeps the tuple $< 1, 3 >$ with a 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 for 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 the data warehouse. Hence the final state of the data warehouse is consistent with the state of the information sources.

Lemma 7. *Given the negative counter mechanism described in Sect. 3.3, the counters of all tuples will always be positive when the data warehouse reaches a quiescent state.*

By Definition 3, in a quiescent state, there is no more **maintenance-concurrent** update in the environment. The data warehouse will be stable and consistent with the current information source space state. Hence there can be no tuple with a negative counter in the data warehouse. The proof for Lemma 7 is shown in Appendix C.

Lemma 8. *The negative counter mechanism captured by the algorithm in Figs. 13 and 14 correctly solves the* **Out-of-Order-DW-Commit** *problem defined in Definition 5 for any two data updates handled in parallel by the data warehouse manager.*

The proof of Lemma 8 can be found in Appendix D. Lemma 8 then leads us to the more general case of correctness as given in the theorem below.

Theorem 9. *The negative-counter-based algorithm described in Figs. 13 and 14 correctly solves the* **Out-of-Order-DW-Commit** *problem for any number of* **maintenance-concurrent** *updates.*

As shown in Appendix E, the proof of this theorem can be given based on an induction on the number of concurrent tuples. This theorem confirms the correctness (as illustrated in Fig. 10) of the final state of the data warehouse after applying the effects of the data updates in any order.

3.4 The core algorithms for the PVM processes

Based on the previous description of the key features of PVM, we now can give details of the PVM processes. The update and query server components employed at each information source are given in Fig. 13. They are fairly standard, such as, for example, found in the SWEEP [1] algorithm.

Figure 14 depicts the logic of the middle layer component that is employed at the data warehouse for view maintenance. *ViewChange* is the main process invoked for every update (ΔR, i) received at the data warehouse. At initialization time, the *DataWarehouse* module will start two processes: *AssignTimeStamp* and *Maintenance Manager*. The *AssignTimeStamp* process assigns a unique local timestamp to all the messages coming into the data warehouse including data updates and query results. The *Maintenance Manager* process monitors the UMQ to check if there is any data update logged in the UMQ and if the PPU still has more room (i.e., we can create more threads to handle updates). In this case, it would move the data update from the UMQ to the PPU in one atomic operation and then create a new instance of the *ViewChange* process to handle this data update now in the PPU. The *ViewChange* process will first handle the data update and then commit the effect of the data update to the data warehouse. It will then move the data update from the PPU to the RPU if the data

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

  PROCESS SendUpdates;
  BEGIN
    LOOP
      RECEIVE $\Delta$ R from R;
      SEND ($\Delta$ R, MyIndex) TO DataWarehouse;
    FOREVER;
  END SendUpdates;

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

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

Fig. 13. Pseudocode of module Update&QueryServer of PVM

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 background process to clean up the RPU set at some fixed time interval.

3.5 Remarks on extensions of PVM

So far, we have been using select-project-join queries for our view definitions so As to focus our discussion on the core ideas of the proposed parallel mechanism. As stated in [19], the $ViewChange$ module, which handles one update for one view definition, can easily be extended to support aggregation functions and a HAVING clause in the view definition by adding additional maintenance queries. These additional maintenance queries could also be part of the query plan generated by the $ViewChange$ module. Hence they can be parallelized with other queries from different maintenance transactions.

One other expression to consider is a self-join of the same relation multiple times. The updates from one relation will appear multiple times if the relation appears multiple times in a view. Hence, if we treat those updates using separate transactions, the data warehouse may result in an inconsistent state. To prevent that, we can batch multiple updates caused by the same relation as one transaction to ensure the correctness of the data warehouse. Please refer to [16] for details of batching.

Also notice that different information sources may have different capabilities. Though we can parallelize the execution of multiple data update maintenance processes at the same time, some fast information sources might still need to wait for the slower information sources to accomplish the maintenance queries. In principle, our system could easily plug-in a $ViewChange$ module, which would support a more dynamic query plan selection based on the query-processing status of the information sources, to balance the usage of different information sources and hence to reach the optimal maintenance performance (as in [19]). Hence their work would be orthogonal to ours.

Note that the correctness of each maintenance query result has been checked by the $isConcurrent$ function of PVM to ensure its correctness once the query result arrives at the data warehouse. Their correctness is not affected by the order of the maintenance subqueries. Hence the above extension to PVM of introducing more maintenance subqueries would not affect the correctness of the PVM system in general.

4 Consistency levels and PVM

4.1 Consistency levels of the data warehouse state

Zhuge et al. [30,31] define notions of consistency of a view extent depending on how the updates are incorporated into the view at the data warehouse. The semantics of the consistency levels are somewhat different under distributed information sources compared to a single information source, as assumed in [30,31]. We hence refine the consistency levels from the data warehouse point of view.

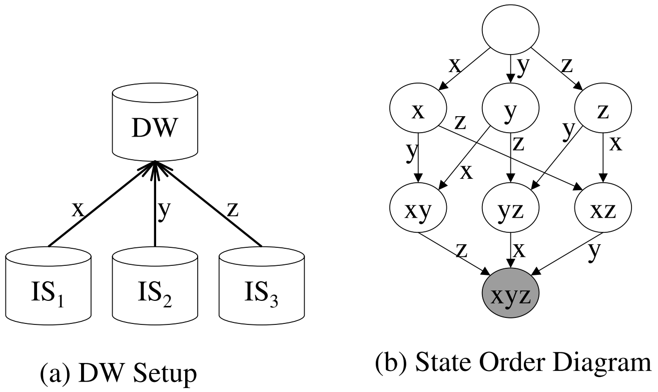
Definition 1. *A* **state order diagram** *for a given information space I and a set D of data updates DU_i with i = 1, ..., k, applied to information sources in the space I is defined to be a rooted acyclic directed diagram where each node represents a legal information source space state by Definition 2 and each directed edge E from a node ISS_j to a node ISS_k labeled with the data update DU_l indicates that information source space state ISS_k can be derived from the information source space state ISS_j by applying the data update DU_l. The root of the diagram is the initial state of the information source space I (or the quiet information source space state from which all updates in D started). The leaves of the diagram are quiet information source states reachable by ISS_0 of I by applying all updates in D in some order as long as they generate legal information source space states.*

For example, Fig. 15a depicts an environment for a data warehouse that has three updates (x, y, z) from three information sources. Figure 15b depicts the *state order diagram* for the three updates. The root node shows that no update has arrived at the data warehouse yet. The second level shows three possible states of the information source space, i.e., update x applied, update y applied, or update z applied. The third level shows the next three possible states that can be reached from the previous states. The fourth level shows the leaf node.

Definition 2. *Every directed path from the root to a leaf node of a state order diagram as defined in Definition 1 is a* **legal order of information source space states**.

```
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($\Delta R$: RELATION; Update-
Source: INTEGER; TimeStamp: INTEGER): RELATION
  VAR
    $\Delta V$, TempView: RELATION;
    j: INTEGER;
  BEGIN
    $\Delta V$ = $\Delta R$;
    /* Compute the left part of the incremental
    view resulting for $\Delta R$ */
    FOR (j = UpdateSource -1; j $\geq$ 1; j–) DO
      $\Delta V$ = ProcessSubQuery($\Delta V$, j, TimeStamp)
    ENDFOR;
    /* Compute the right part to the incremental
    view resulting from $\Delta R$ */
    FOR (j=UpdateSource+1; j $\leq$ n; j++) DO
      $\Delta V$ = ProcessSubQuery($\Delta V$, j, TimeStamp)
    ENDFOR;
    CommitManager($\Delta R$, $\Delta V$);
  END ViewChange;

  FUNCTION CommitManager($\Delta R$: RELATION;
    $\Delta V$: RELATION)
    V = V + ($\Delta V$);
    /* erase the unrelated Updates */
    CleanUp RelatedProcessedUpdate;
    CRITICAL AREA
      REMOVE ($\Delta R$, i, t)
        FROM ParallelProcessingUpdate;
      IF ($\Delta R$, i, t) is related
      THEN PUT ($\Delta R$, i, t)
        INTO RelatedProcessedUpdate;
    ENDAREA
  END CommitManager;

  FUNCTION isConcurrent($\Delta R$: RELATION; j: IN-
TEGER; t: INTEGER): BOOLEAN
    IF $\exists(\Delta R, j, t) \in UpdateMessageQueue$
      or $\exists(\Delta R, j, t) \in ParallelProcessingUpdate$
      or $\exists(\Delta R, j, t) \in RelatedProcessedUpdate$
      THEN return TRUE; ELSE return FALSE
    ENDIF
  END isConcurrent

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

  PROCESS AssignTimeStamp;
  VAR
    t: TIME; /* current system time at data warehouse */
  BEGIN
    LOOP
      RECEIVE Message FROM Data Source i
        as received order;
        IF Message is $\Delta$ THEN
          t= getCurrentTime();
          APPEND ($\Delta R$, i, t) TO UpdateMessageQueue;
        ELSE
          Assign getCurrentTime() to $\Delta V$
        ENDIF
    FOREVER;
  END AssignTimeStamp;

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

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

Fig. 14. Pseudocode of DataWarehouse module of PVM



(a) DW Setup

(b) State Order Diagram

Fig. 15. Example of state order diagram

As we can see in Fig. 15b, there are six possible legal orders of information source space states for our given example.

Definition 3. *Five consistency levels of the data warehouse in distributed environments from the data warehouse point of view can now be defined as follows:*

- *Convergence: the data warehouse state is legal by Definition 3 in any quiet state of the data warehouse.*
- *Weak consistency: all states of the data warehouse are legal by Definition 3 at all times.*
- *Consistency: weak consistency and the data warehouse states correspond to* **one** *legal order of the information source space states as defined by the state order diagram given in Definition 2.*
- *Strong consistency: consistency and convergence.*
- *Complete consistency: strong consistency and all the states in a legal order of the state order diagram as defined by Definition 2 have corresponding legal data warehouse states.*

Figure 16 illustrates the basic idea behind the five consistency levels defined in Definition 3. IS space states are depicted by circles, while data warehouse states are represented by boxes. A shaded circle represents a quiet information source space state. A shaded box represents a quiet state of the data warehouse. We denote the correspondence between a data warehouse state and its associated information source space state by a line connecting them. By Definition 2, each data warehouse state with an associated information source space state is called legal.

Hence in Fig. 16a for convergence, we notice all quiet states of the data warehouse state are legal, but there is one data warehouse state in the middle that is not legal because there is no corresponding information source space state. According to Theorem 9, PVM at least ensures convergence. In Fig. 16b for weak consistency, we see all the data warehouse states have corresponding information source states, but the order of DW states may not correspond to the order of information source states. In Fig. 16c for consistency, we see that this time all data warehouse states are legal and also the corresponding information source space states are in the legal order. However, there is no quiet data warehouse state that matches the final quiet information source space state.[5] Hence convergence is not guaranteed. In Fig. 16d for strong consistency, all the data warehouse states are legal and also in the legal order. However, there may be information source states that have no corresponding data warehouse states. In Fig. 16e for complete consistency, all the data warehouse states are legal and in the legal order. In addition, all the information source space states on exactly one legal path also have a corresponding data warehouse state.

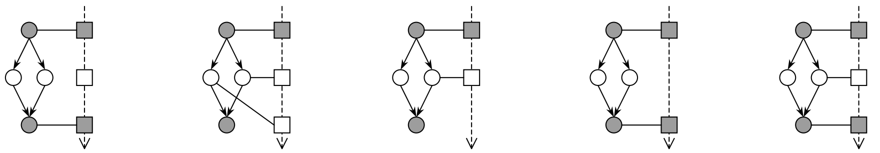
4.2 Extension to PVM to achieve complete consistency

Based on the algorithm described in the previous section, PVM ensures the convergence level of consistency. If we force the effects of data updates to be committed in the order in which they have arrived at the data warehouse, then PVM achieves the complete consistency level as explained below. PVM handles updates in parallel and commits them as soon as ΔV is ready. It could happen that one DU_i arrives at the data warehouse earlier than another DU_j but is committed later. For example, DU_i is received earlier than DU_j at the data warehouse manager (here $i < j$), and both DU_i and DU_j are handled in parallel. Let us assume the thread handles DU_j fast and thus the ΔV_j is committed to the data warehouse before ΔV_i. It is very straightforward to achieve complete consistency, if so desired. We simply would need to force ΔVs to be committed to the data warehouse in the same order in which they arrive. For the example above, we simply don't commit ΔV_j until ΔV_i has been committed.

We now discuss what mechanism is needed so that we can enhance the core engine of PVM to control the proper commit order. For this we need to address the following issues: first, how do we determine when and when not to commit a DU_i; second, how long do we have to wait in the worst case to commit? To solve the first problem, a data storage, called $CommitQueue$, is added into the PVM system. This queue stores all the ΔV that have been handled but not yet committed to the data warehouse. We also modify the $ViewChange$ and $AssignTimeStamp$ processes. We modify the $CommitManager$ to commit ΔV by using the $CommitQueue$. Whenever the data warehouse receives an update (say DU_j), we initialize an item in the $CommitQueue$ in the $AssignTimeStamp$ process. After the DU_j has been handled by PVM, the result ΔV_j is associated with DU_j in the $CommitQueue$. Then the function $CommitManager$ checks whether all the items before it in the $CommitQueue$ have been committed. After all the ΔV_i $(i < j)$ before DU_j have been committed, then we commit this ΔV_j and remove it from the $CommitQueue$. Figure 17 presents pseudocode for the new modules of $ViewChange$ and $AssignTimeStamp$ of PVM system with the major changes highlighted.

Given the modification to the PVM system in Fig. 17, now called **PVM CC**, the data warehouse system will commit data updates in the same order in which they arrive at the data warehouse. PVM CC will calculate the effect of every data update that changes the information source state from one legal information source state to another legal information source state and commit the effect of that information source state change on the data warehouse state. Hence all legal states of the information source space will be reflected as states of the data warehouse as well. According to Definition 3 of consistency

[5] For example, this would happen if the data warehouse requires the next data update to compensate the previous maintenance error.



Convergence (a)

Weak Consistency (b)

Consistency (c)

Strong Consistency (d)

Complete Consistency (e)

Fig. 16. Consistency level illustration

```
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;
    CommitQueue: QUEUE initially 0;

  PROCESS ViewChange($\Delta R$: RELATION; UpdateSource:
    INTEGER; TimeStamp: INTEGER): RELATION
  VAR
    $\Delta V$, TempView: RELATION;
    j: INTEGER;
  BEGIN
    $\Delta V$ = $\Delta R$;
    /* Compute the left part of the incremental
    view resulting for $\Delta R$ */
    FOR (j = UpdateSource -1; j $\geq$ 1; j–) DO
      $\Delta V$ = ProcessSubQuery($\Delta V$, j, TimeStamp)
    ENDFOR;
    /* Compute the right part to the incremental
    view resulting from $\Delta R$ */
    FOR (j=UpdateSource+1; j $\leq$ n; j++) DO
      $\Delta V$ = ProcessSubQuery($\Delta V$, j, TimeStamp)
    ENDFOR;
    CommitManager($\Delta R$, $\Delta V$);
  END ViewChange;

  FUNCTION CommitManager($\Delta R$: RELATION;
    $\Delta V$: RELATION)
    FIND($\Delta R$) IN CommitQueue;
    PUT($\Delta V$) TO CommitQueue;
    WHILE( $\Delta V$ in first item of Queue is not empty)
      DEQUEUE($\Delta R$, $\Delta V$);
    V = V + ($\Delta V$);
    /* erase the unrelated Updates */
      CleanUp RelatedProcessedUpdate;
      CRITICAL AREA
        REMOVE ($\Delta R$, i, t)
          FROM ParallelProcessingUpdate;
        IF ($\Delta R$, i, t) is related
        THEN PUT ($\Delta R$, i, t)
          INTO RelatedProcessedUpdate;
      ENDAREA
    END WHILE
    END CommitManager

  PROCESS AssignTimeStamp;
  VAR
    t: TIME; /* current system time at the data warehouse */
  BEGIN
    LOOP
      RECEIVE Message FROM Data Source i
        as received order;
        IF Message is $\Delta$ THEN
          t= getCurrentTime();
          APPEND ($\Delta R$, i, t) TO UpdateMessageQueue;
          APPEND ($\Delta R$, i, t, empty $\Delta V$) TO CommitQueue;
        ELSE
          Assign getCurrentTime() to $\Delta V$
        ENDIF
    FOREVER;
  END AssignTimeStamp;

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

Fig. 17. PVM CC: pseudocode of *DataWarehouse* module achieving complete consistency

in Sect. 4.1, PVM CC now achieves the complete consistency level.

Note that PVM CC still handles updates in parallel and has all the advantages of the algorithm described in Sect. 3.3.3. The wait we would possibly incur is miniscule as it only concerns the “actual commit" time that it takes right after the process we were waiting for has committed. The wait is not an “infinite wait" as we do not need to wait for a “quiet time" for updates. We just need to wait a Δt time while parallel threads finish processing updates that need to be committed for reasons of complete consistency before the current update is actually to be committed. Any complete consistent algorithm would have to commit and hence wait for the completion of the processing of these earlier updates. Hence it would have the same or worse performance than PVM, i.e., we cannot do better if we do require this level of consistency.

5 Cost model of PVM

In this section, we present a cost model we have designed to compare the performance of PVM to the performance of SWEEP in terms of number of messages and total processing time.

5.1 Factors and measurements

The factors we used in this analysis are described in Table 3. Note that the factor t captures the complete processing time for one data update. This time includes the plan generation, network delay for transferring queries and results, and processing time at the information source. The information sources may be loaded with their respective OLTP transactions. Hence the processing time at the information sources may be affected by the locking of the OLTP transactions and other local operations. However, for the purpose of evaluating PVM, it is sufficient to abstract all factors affecting the cost of processing a query on the source by one parameter t.

For simplicity, we assume we have m information sources and n updates, and $n \bmod m$ is 0. The view is defined over all of m information sources and is assumed to be affected by all n updates. We also assume that the updates are evenly distributed, and all updates happened before any processing of the updates. So they are all concurrent with one another. For simplicity, we label the information sources in increasing order from 1 to m and number the updates from 1 to n as follows:

$$\begin{array}{lllll} \multicolumn{5}{c}{IS\ index\ (j) \longrightarrow} \\ IS_1 & IS_2 & IS_3 & IS_4 & \cdots IS_m \\ \hline DU_1 & DU_2 & DU_3 & DU_4 & \cdots DU_m \\ DU_{m+1} & DU_{m+2} & DU_{m+3} & DU_{m+4} & \cdots DU_{2m} \\ \vdots & \vdots & \vdots & \vdots & \vdots\ \vdots \\ DU_{n-m+1} & DU_{n-m+2} & DU_{n-m+3} & DU_{n-m+4} & \cdots DU_n \end{array} \tag{2}$$

5.2 Comparison in number of messages

The cost models for both SWEEP and PVM are the same in terms of the total number of messages M. The reason is that PVM simply parallelizes the execution of SWEEP instead of changing the way the view maintenance for each data update is handled. There are two kinds of messages sent by the SWEEP (or PVM) algorithms: remote queries and local queries. The number of 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 updates and the number of information sources as shown below:

$$\begin{array}{llr|lllll} & & & \multicolumn{5}{c}{IS\ index\ (j) \longrightarrow} \\ & & & 1 & 2 & 3 & \cdots & m \\ \hline No. & & 1 & 0 & \frac{n}{m} & \frac{n}{m} & \cdots & \frac{n}{m} \\ of & & 2 & \frac{n}{m}-1 & 0 & \frac{n}{m} & \cdots & \frac{n}{m} \\ DU & & 3 & \frac{n}{m}-1 & \frac{n}{m}-1 & 0 & \cdots & \frac{n}{m} \\ & & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ (i) & & & & & & & \\ \downarrow & & m+1 & 0 & \frac{n}{m}-1 & \frac{n}{m}-1 & \cdots & \frac{n}{m}-1 \\ & & m+2 & \frac{n}{m}-2 & 0 & \frac{n}{m}-1 & \cdots & \frac{n}{m}-1 \\ & & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ & & n-1 & 0 & 0 & 0 & \cdots & 1 \\ & & n & 0 & 0 & 0 & \cdots & 0 \end{array} \tag{3}$$

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

$$M_{local} = \begin{cases} \frac{n}{m} - k & j \leq (i-2) \bmod (m+1) \\ \frac{n}{m} - (k-1) & j > (i-2) \bmod (m+1) \\ 0 & j = i \bmod m \\ 0 & j \leq (i-2) \bmod (m+1)\ and\ \frac{n}{m} < k \\ 0 & j > (i-2) \bmod (m+1)\ and\ \frac{n}{m} < k-1 \end{cases} \tag{4}$$

with j denoting the index of information sources and i the index of updates.

$$k = \begin{cases} \frac{(i-2)}{m} + 1 & i > 1 \\ 0 & i = 1 \end{cases} \tag{5}$$

$$M_{local} = \sum_{i=1}^{n-1} i - m \sum_{i=0}^{\frac{n}{m}-1} n = (\frac{m-1}{2m})n^2 \tag{6}$$

And from [1] we know that

$$M_{remote} = n \times (m-1) = (m-1)n \tag{7}$$

5.3 Comparison in total execution time

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

$$\begin{aligned} T_{SWEEP} &= M_{remote} \times C_{remote} + M_{local} \times C_{local} \\ &= (m-1)n \times C_{remote} + (\tfrac{m-1}{2m})n^2 \times C_{local} \end{aligned} \tag{8}$$

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

$$T_{SWEEP} = (nm - n) \times C_{remote} \approx O(n \times m \times C_{remote}) \tag{9}$$

This shows that the time performance of SWEEP is linear in the product of the number of information sources and the number of updates. Hence for one update, the estimated time is:

$$t_{SWEEP} = (m-1) \times C_{remote} \tag{10}$$

Table 3. Parameters of the analysis

	Factors of the cost model
n	Number of concurrent updates that happened before the middle layer starts to handle them
m	Number of information sources
p	Size of PPU (maximum number of parallel threads)
q	Number of queries that could be processed by the information source space at one time
c	Number of consecutive updates from same information source (the smaller c, the more even updates distribution)
i	Average time intervals between two consecutive updates
	Measurement of the cost model
M	Number of messages
T	Elapsed time of handling a set of updates
t	Elapsed time of handling one data update

We are going to use t instead of t_{SWEEP} for the following discussion.

Then, for SWEEP we have:

$$T_{SWEEP} = nt \tag{11}$$

For PVM the number of instances of parallel handling (i.e., size of the PPU set), denoted by p, as well as the query capability q of the information source space will affect the performance of PVM. q denotes how many queries can be processed at the same time at the information source space. We denote the overhead of generating a new thread and waiting for locks to access the critical area as 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 data sources have much larger (infinite) query processing capabilities. Then for every update in the UMQ, we can assess how long it will wait to be handled. The first 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. In general, $DU_p = (p-1)a$. 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. Thus, we get the following wait sequence:

$$\begin{array}{l|lllllllllll} DU_i & 1 & 2 & 3 & \cdots & p-1 & p & p+1 & \cdots & 2p & 2p+1 & \cdots \\ \hline Wait & 0 & a & 2a & \cdots & (p-2)a & (p-1)a & t & \cdots & t+(p-1)a & 2t & \cdots \end{array} \tag{12}$$

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 information sources 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 outweigh its overhead in terms of system resources. This case is similar to case 1. The waiting time $Wait(i)$ is: $Wait(i) = \lfloor \frac{i-1}{q} \rfloor t + [(i-1) \bmod q]a$.

Based on the discussion, the general waiting time for DU_i is:

$$Wait(i) = \lfloor \frac{i-1}{min(p,q)} \rfloor t + [(i-1) \bmod min(p,q)]a \tag{13}$$

Then the total execution time of PVM is:

$$T_{PVM} = \max_{i=1\ldots n} (Wait(i)) + t. \tag{14}$$

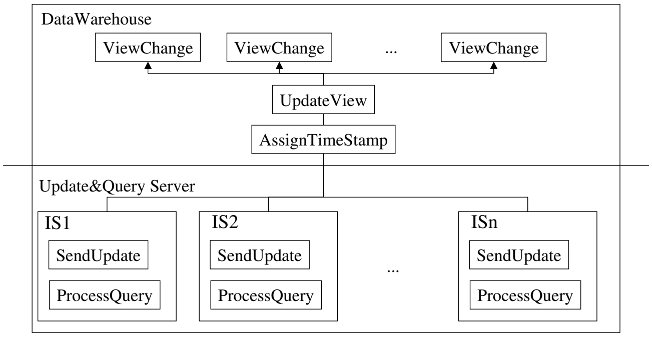


Fig. 18. PVM algorithm thread overview

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

$$T_{PVM} = \left(\frac{n-1}{k} + 1 \right) t. \tag{15}$$

Comparing the performance of T_{SWEEP} and T_{PVM} we observe that PVM has approximately a best-case scenario of k-fold better performance over SWEEP with $k = min(p,q)$.[6] This implies, for example, that if your DW implementation platform comfortably supports ten concurrent threads over ten data sources, then up to a tenfold performance improvement may be achievable.

The previous analysis is based entirely on the assumption that updates are evenly distributed over the information sources. If the updates are not evenly distributed, for example all the d updates of IS_1 will be handled after the updates of the IS_2, and so on, the overall performance of PVM will be d times lower unless a proportional query capacity also happens to exist at this bottleneck of IS_1.

There is a tradeoff between C_{local} and C_{remote}. From the previous discussion we know that for each update, the handling time is $t = (m-1) \times C_{remote} + (\frac{m-1}{2m})n \times C_{local}$. Let us denote the first part as t_{remote} and the second part as t_{local}. The ratio between t_{local} and t_{remote} is:

$$\frac{t_{local}}{t_{remote}} = \frac{n}{2m} \times \frac{C_{local}}{C_{remote}} \tag{16}$$

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

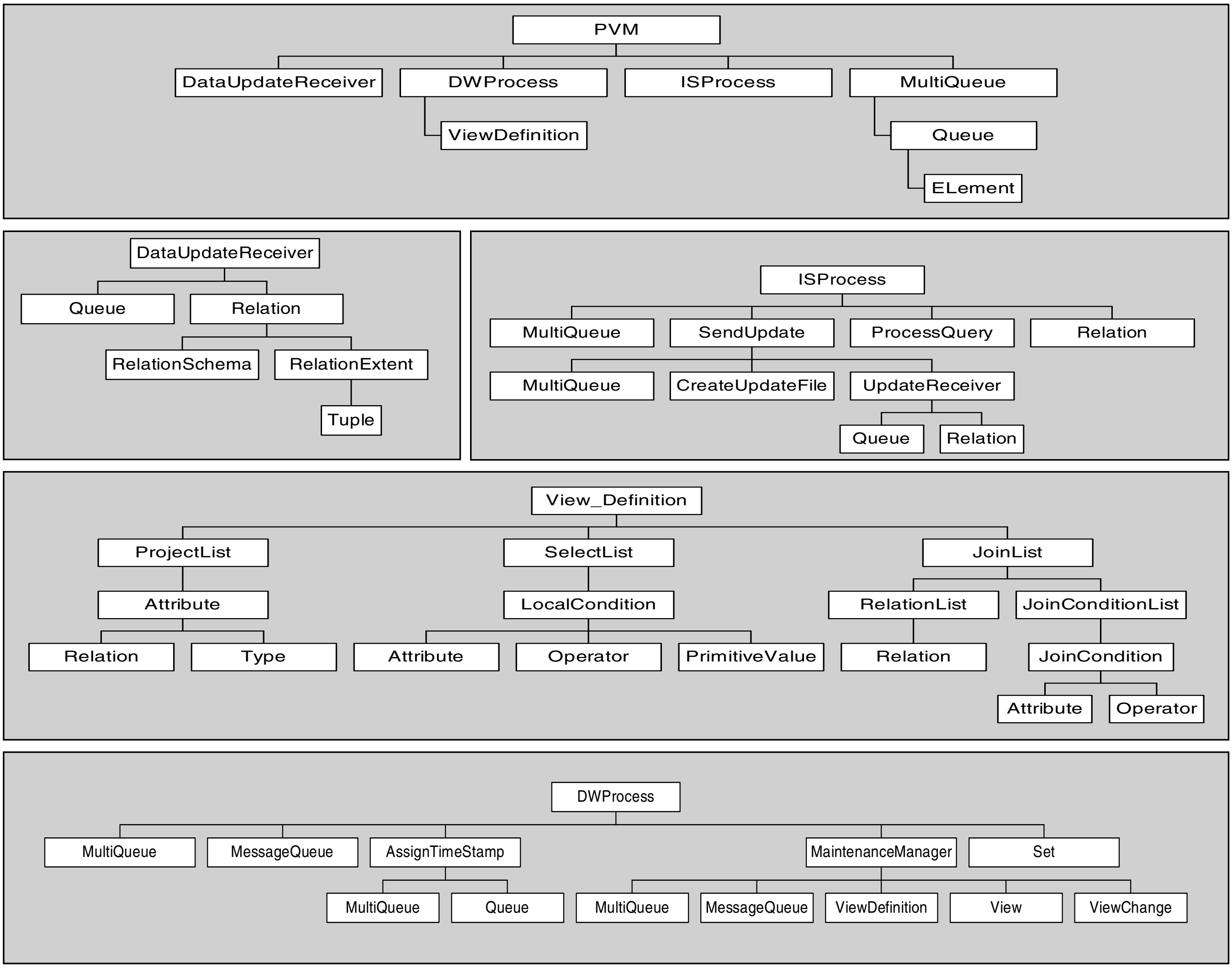


Fig. 19. PVM implementation class relation

If we know $\frac{C_{local}}{C_{remote}}$ is 0.01, we have ten information sources, and we want $\frac{t_{local}}{t_{remote}}$ to be less than 0.1 in order to ignore the cost of local join queries. Then the maximum number of **maintenance-concurrent** updates we can have is 200. If we have more than 200 **maintenance-concurrent** updates, the t_{local} time can no longer be neglected.

6 Design and implementation of the PVM system

The PVM system is implemented within the EVE data warehousing [16] environment implemented in JAVA. Currently, the system connects to Oracle and ODBC MS Access servers using the Oracle JDBC driver and the JDBC-ODBC bridge, respectively.

Figure 18 provides an overview of the PVM system in terms of the types of different threads. For every information source there are two types of threads running. The thread called *SendUpdate* will keep sending the data updates of this information source to the data warehouse manager. The thread called *ProcessQuery* will process the queries sent from the data warehouse manager and return the query result back to the data warehouse manager. Within the data warehouse manager, there are three kinds of threads. The *AssignTimeStamp* thread will assign a timestamp to all the incoming data updates and query results. The *MaintenanceManager* thread will monitor the data updates and create one subthread of type *ViewChange* for each data update to be handled concurrently.

For example, assume we set up a system that can handle two data updates in parallel over three information sources. Then at system setup, two threads of *SendUpdate* and *ProcessQuery* will be created at each information source. In addition, there will be two threads of *AssignTimeStamp* and *MaintenanceManager* at the data warehouse. At run time, if multiple data updates occurred, one additional thread of *ViewChange* will be created at the data warehouse for each of the data updates to be handled concurrently, while all the other data updates will wait in the UMQ for processing. In general, the maximum number of threads in our system can be computed as in Eq. 17, where p and m are defined in Table 3.

$$\begin{aligned} <\#\,of\ threads\ in\ DW> &= p+2 \\ <\#\,of\ threads\ in\ ISs> &= m\times 2 \\ <\#\,of\ threads\ in\ PVM> &= <\#\,of\ threads\ in\ DW> \\ &\quad + <\#\,of\ threads\ in\ ISs> \\ &= m\times 2+p+2 \end{aligned} \tag{17}$$

Figure 19 depicts an object-oriented design of the PVM system, focusing in particular on the aggregation relationships of the PVM classes. The *PVM* component contains four classes. *DataUpdateReceiver* is used to detect data updates, *DWProcess* represents the data warehouse manager, *ISProcess* implements the wrapper of the information sources, and *MultiQueue* "manages" the FIFO network connection between *ISProcess* and *DWProcess*. The *ISProcess* class contains the reference to the *MultiQueue*, the *SendUpdate* class the reference to the *SendUpdate* module, the *ProcessQuery* class the reference to the *ProcessQuery* module, and the *Relation* class the reference to capture the schema of the relation stored at this information source. The *DWProcess* class also contains the reference to the *MultiQueue*. The *MessageQueue* class is used to implement the UMQ, the *AssignTimeStamp* class implements the module *AssignTimeStamp*, the *MaintenanceManager* class implements the module *MaintenanceManager*, and the *ViewChange* class implements the module *ViewChange*.

7 Performance studies

7.1 System setup

We have implemented the SWEEP and PVM algorithms in JAVA within our prototype EVE data warehousing system, a demonstration of which has was at ACM SIGMOD'99 [22]. 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. Every IS$_i$ (on Oracle) has its own index number from 1 to m. Each information source has one relation. The relation of each IS_i is denoted by R_i, with i the index of IS_i. The schema of every relation is the following:

```
CREATE TABLE $R_i$ (A as integer,
                    B as integer); ($1 \leq i \leq m$)
```

The extents of all R_i are randomly generated and all are approximately the same size, i.e., ranging up to several 5000-tuple clusters per information source for different experiments. The data updates were randomly generated, with the number of data updates changing as indicated from experiment to experiment. In general, each data update was counted (as assumed in our model) only if it actually affected the extent of the information source (i.e., it was not a duplicate for an insert or a nonexisting tuple for a delete). In general, the join selectivity between a data update at one information source with the extent of another information source was kept to assure that at least one "join" tuple was returned from each maintenance query, i.e., that the maintenance query results were not empty.

The data warehouse is defined over all m information sources as given in Fig. 20.

```
CREATE VIEW  V AS
SELECT       $R_1$.A, $R_2$.A, $R_3$.A, ..., $R_m$.A
FROM         $R_1$, $R_2$, $R_3$, ..., $R_m$
WHERE        $R_1$.A = $R_2$.B, $R_2$.A = $R_3$.B,...
                  ... $R_{m-1}$.A = $R_m$.B
```

Fig. 20. Data warehouse view definition

Each of the experiments described below have been rerun approximately ten times, and the reported measurement values represent the average over all runs.

7.2 Changing the maximum number of concurrent maintenance threads (PPU size p)

This experiment has been designed to study the effect of varying the number of threads that can be run in parallel under PVM with the total execution time of handling a set of concurrent updates. For this, we change the maximum number of threads (PPU size p, where p is defined in Table 3, from 1 to 10 on the x-axis), while measuring the total execution time T (in wall clock seconds on the y-axis). In particular, fixed settings for this experiment are: number of updates n is 60, number of information sources m is 4, and time interval i is 0.

In Fig. 21, the left chart shows the expected value calculated based on our cost model (specifically Eq. 15 of the cost model), while the right chart shows the actual experimental measurements. Based on the model, we expect to see an up to p-fold performance improvement of PVM in the ideal case as we increase the maximum number of threads by p. As seen in the left chart in Fig. 21, we can see that the total processing time of PVM drops rapidly at the rate of $1/p$ as we increase p until the curve flattens out into a horizontal line. In our setup, the flattening occurs at roughly $p = 4$, which corresponds to the system limitation in terms of its query capabilities of the information source space, i.e., with the number of information sources m=4 and the number of queries that can be handled concurrently per information source being set to 1. The latter is so because no concurrency is supported at the information sources due to JDBC implementation limitations.

The right chart shows actual measurements we made using our PVM system; these measurements indeed closely follow the expected behavior. The initial total processing time when $p = 1$ is very similar to the expected time. And as the size of PPU p increases, the total performance of PVM is decreasing at a rate of roughly $1/p$.

The maximum percentage of performance improvement we measured is 330%, which is less than the best-case improvement of up to 400%. That the actual percentage of improvement was slightly less than expected can be explained by additional system overhead not accounted for in our simple cost model as well as by the fact that the maintenance queries processed by the information sources are blocked by each other at every information source because the query capability of each information source is only one query at a time.

Figure 22 illustrates the overhead of the potential blocking between the data update maintenance processes for 3 information sources and 3 data updates. There are 3 portions of the charts; from left to right are SWEEP, PVM when $p = 2$, and PVM when $p = 3$. Every row in the table shows one clock

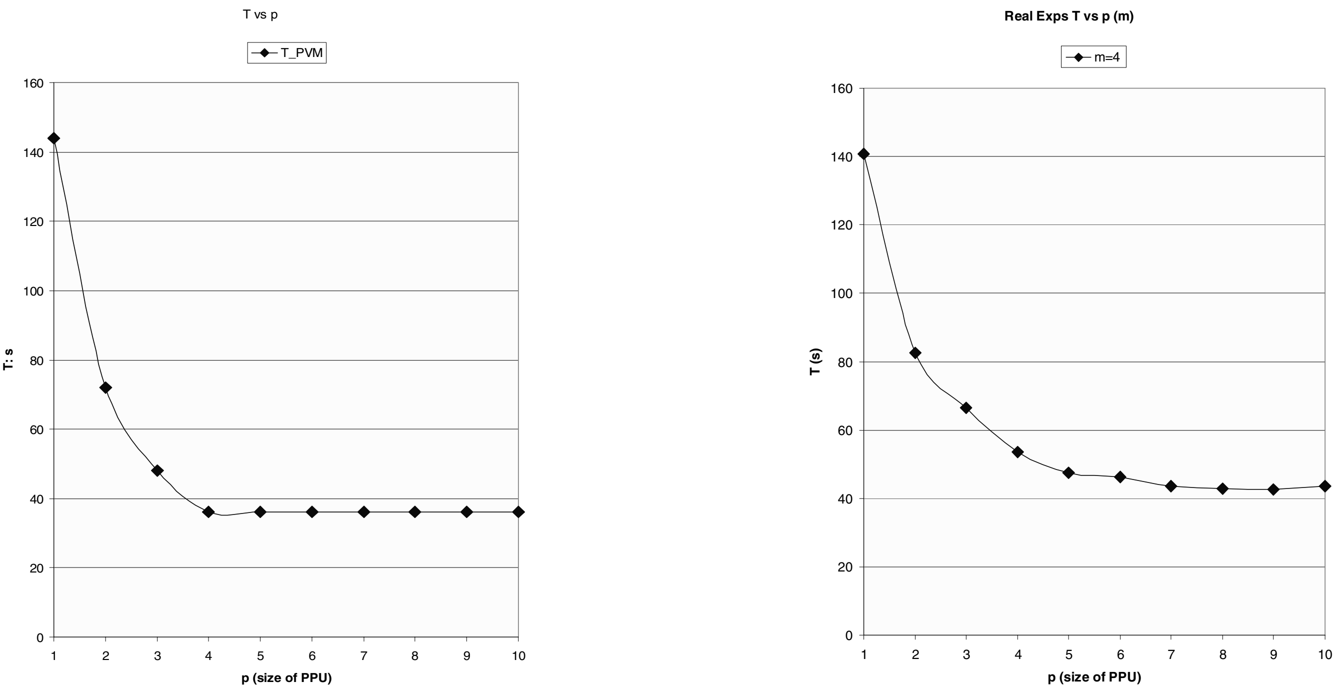


Fig. 21. Experiment 1: change maximum number of concurrent maintenance threads (size of PPU p). Left figure is cost model, right figure is experimental result

Tick	SWEEP			PVM (p=2)			PVM (p=3)		
	IS1	*IS2*	*IS3*	*IS1*	*IS2*	*IS3*	*IS1*	*IS2*	*IS3*
0	DU1	**DU2**	*DU3*	DU1	**DU2**	*DU3*	DU1	**DU2**	*DU3*
1		SQ1,2		**SQ2,1**	SQ1,2		**SQ2,1**	SQ1,2	
2		SQR1,2		**SQR2,1**	SQR1,2		**SQR2,1**	SQR1,2	
3			SQ1,3			SQ1,3		*SQ3,2*	SQ1,3
4			SQR1,3			SQR1,3		*SQR3,2*	SQR1,3
5	**SQ2,1**				*SQ3,2*	**SQ2,3**	*SQ3,1*		**SQ2,3**
6	**SQR2,1**				*SQR3,2*	**SQR2,3**	*SQR3,1*		**SQR2,3**
7			**SQ2,3**	*SQ3,1*					
8			**SQR2,3**	*SQR3,1*					
9		*SQ3,2*							
10		*SQR3,2*							
11	*SQ3,1*								
12	*SQR3,1*								

Fig. 22. Block overhead between data update maintenance processes

tick. Each cell shows the status of the information source for that clock tick. For example, at (tick 0, SWEEP) IS_1 has data update DU_1 committed, and at (tick 1, SWEEP) IS_2 processes the subquery $SQ_{1,2}$ for the process of DU_1. If the cell is empty, this means the corresponding IS is idling at that time. As we can see, SWEEP uses 12 ticks to handle 3 data updates, while PVM ($p = 2$) uses 8 ticks, and PVM ($p = 3$) uses 6 ticks. Intuitively, the performance of PVM ($p = 2$) should be twofold of that of SWEEP, which would mean 6 ticks. However as can be seen it actual takes us 8 ticks. The reason for that is the blocking on IS_3 where $QR_{2,3}$ has to wait for $QR_{1,3}$. By increasing p from 2 to 3, we can improve the overall performance by further utilizing IS_2 for $SQ_{3,2}$ while IS_3 is blocked. For a fixed query plan generated by the $ViewChange$ process, more threads can improve the utilization of the information sources and hence result in a better performance. This can be done until we exhaust the total query capability of the information sources.

Also, while rapidly improving upon the performance when increasing p from 1 to 2 and so on up to 4, the decrease of the curve continues (though at a much reduced rate) until PPU size $p = 7$. In summary, PVM improves the performance of SWEEP by several magnitudes depending on system resources.

We have compared our PVM algorithm with selecting $p = 1$ with the SWEEP algorithm and found no significant performance differences between the two. For the most part, PVM with $p = 1$ and SWEEP are identical in terms of software in our system. Thus, we will not focus specifically on SWEEP in the remainder of this study.

7.3 Changing the number of information sources (m)

In this experiment, we not only control the maximum number of parallel maintenance threads (by varying PPU size p from

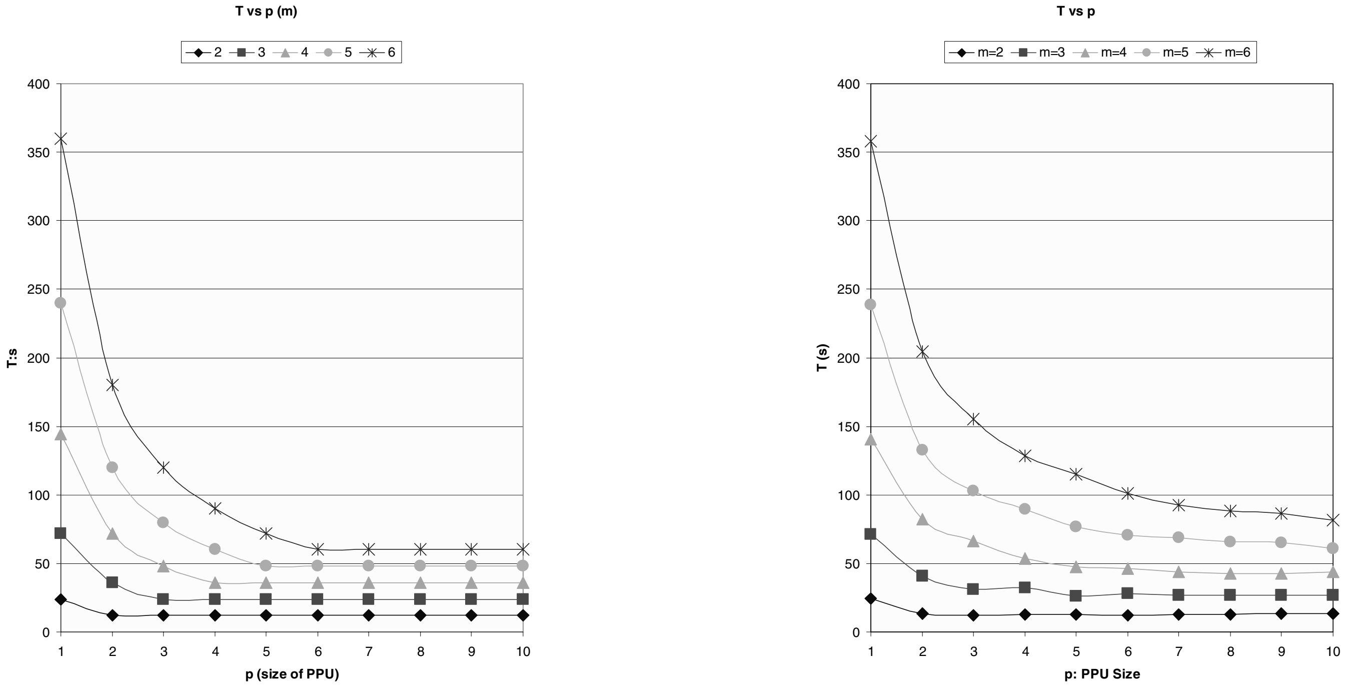


Fig. 23. Changing the number of information sources (and number of threads). Left figure is the cost model. Right figure is the experimental result

1 to 10 on the x-axis), but we also study the effect of different numbers of information sources (by plotting different curves for different values of m with m ranging from 2 to 6) on the overall performance of handling a set of data updates. In summary, for this experiment we set the number of updates per information source n/m at 10, and the time interval i at 0, while varying the number of information sources m from 2 to 6 and the size of the PPU p from 1 to 10.

The results reported in Fig. 23 again have two components, namely, the left chart in the figure is directly derived from our cost model, while the right one reports the actual experimental measurements. We observe in general that the measured trends approximately reflect the expected ones, though with some slight overhead added on.

Based on Fig. 23, we make the following observations. First, as already observed in the first experiment, an increase in the maximum number of concurrent threads (i.e., PPU size p increasing from left to right) decreases the total execution time T by $1/p$ with each additional thread, e.g., 1/2, 1/3, 1/4, etc. This dependency on the number of threads holds true regardless of the number of information sources (i.e., the different lines in the figure). Second, the overall query capability $m \times q$ of the system, with m being the number of information sources and q the number of queries an information source can handle in parallel, is a delimiting factor on the overall performance achievable in the system. Hence for each of the lines the decrease in execution time slows down and eventually reaches a certain point (we call it the "turning point") at which it completely flattens. That is, an increase in the number of threads p would no longer be effective beyond this turning point. It no longer positively affects the performance costs T as the environment's query capability and not the parallel data warehouse maintenance capability becomes the bottleneck.

In summary, the observations above show that the performance gain of PVM is based on both the query capability of the information source space and the maximum number of threads (p) of the data warehouse manager.

7.4 Change distribution of update load from information sources (c)

In this experiment, we change the distribution (but not number) of updates c, which is over a fixed number of information sources m, to examine how data update distribution affects the performance of SWEEP and PVM. Here the notion of distribution c captures rates at which update are being generated from the various information sources, assuming a total of 30 updates and 3 information sources. Then if c is equal to 1, the sequence of updates is generated as follows by the various information sources: DU_1 comes from IS1, DU_2 from IS2, DU_3 from IS3, then DU_4 comes from IS1 again, and so on. If c is equal to 5, DU_1 to DU_5 come from IS1, then DU_6 to DU_{10} come from IS2, etc. In summary, the settings are: the number of information sources m is 3, number of updates n is 30, the size of PPU p is 5, and the time interval i is 0, while varying the *number of continuous updates from the same information source* c from 1 to 10.

For this experiment, the cost model in its present form cannot predict any behavior due to c not being captured by the cost model. Intuitively we expect the performance will get worse as c gets larger, and when c is large enough, the performance of PVM will remain flat but will still be better than SWEEP. Figure 24 depicts the actual experimental results. The different distribution of continuous updates c are plotted on the x-axis (ranging from $c = 1$ to 10) and the time (in seconds) taken by PVM and SWEEP for different c values is

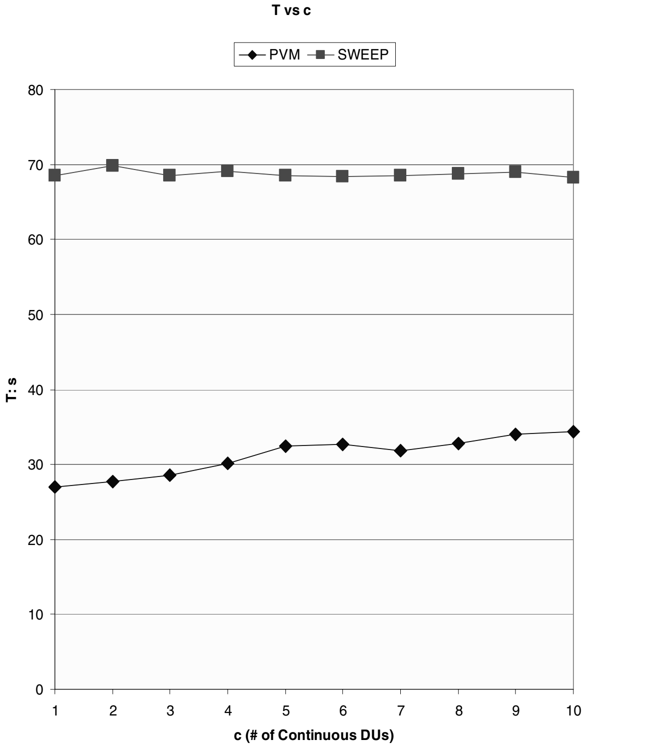


Fig. 24. Change of distribution of updates over information sources

plotted on the y-axis. The two lines correspond to PVM and SWEEP, respectively.

Figure 24 illustrates that the distribution of updates c decreases PVM's performance to a small degree while not affecting the performance of SWEEP. The reason for the former is that as c goes up, more and more updates will arrive at the same information source at the same time. Those queries will be synchronized at that site given that only one query can be processed at a time by our information sources. Hence the performance is slightly worse. If c is larger than n (total numbers of updates), then the performance will not be affected by c any more because all the updates come from one information source. However, we note that in all these cases PVM exhibits significantly better performance than SWEEP, e.g., for the given system setup SWEEP has a 100% longer execution time than PVM.

7.5 Change number of updates n

Figure 25 depicts the performance of PVM and SWEEP in terms of seconds (depicted on the y-axis) for different numbers of data updates n (depicted on the x-axis varying from 4 to 40 per information source). In all plots, the number of information sources is fixed, in this case m=2. Again, the left chart in Fig. 25 is derived from our cost model, while the right one corresponds to the actual experimental result. The settings are: number of information sources m is 2, number of continuous updates from one information source c is 1, size of PPU p (for PVM) is 5, while varying the *number of updates for each information source* ($\frac{n}{m}$) from 4 to 40.

We expect PVM will have an up to $min(p, m)$-fold better performance than SWEEP. As expected, the ratio of T between SWEEP and PVM ($min(p, m)$ = 2) is about 2, i.e., we achieve a 100% increase in performance.

7.6 Change time interval i

We change the time interval i between the occurrence of two adjacent updates. Here, the time interval i simulates at what rate new updates are generated. For example, if i is equal to 20 ms, one update happens every 20 ms. Our experimental settings are: number of information sources m is 2, number of updates n is 40, size of PPU p is 5, while varying the *time interval* i from 0 to 1600 (ms).

Figure 26 shows the behavior of PVM and SWEEP in response to the change of the time interval, with the left chart derived directly from our cost model and the right one reporting the actual experimental measurements. In Fig. 26, the time interval i is plotted on the x-axis (by varying it from 0 to 1600 ms) and the time (in seconds) taken by PVM and SWEEP is presented on the y-axis. Given that we measure both PVM and SWEEP, two lines are depicted. Our cost model (see the left chart) indicates that we expect that the larger i is, the less a performance gain PVM will achieve compared to SWEEP. When i is large enough, PVM will eventually be the same as SWEEP. In other words, the more frequently the update comes, the better the performance of parallel maintenance handling than the nonparallel one.

So when the time interval i is large enough, i.e., larger than the time required to handle of one individual data update, then the system will effectively process the updates sequentially one after the other according to SWEEP's query plan and the improved parallel handling capability of PVM is not utilized. That is, in that case, only one thread of PVM is ever used (PPU=1). Thus in a very quiet data warehouse environment with very sparse data updates, PVM cannot improve the maintenance performance further beyond the performance of the original maintenance algorithm, in this case SWEEP.

Our experimental measurements indeed confirm that increasing the time interval between continuous updates will change the performance of both algorithms. When the intervals are larger than the total time taken by the SWEEP process for all the updates, the SWEEP total execution time increases linearly. For PVM, the performance will not be affected until the intervals are larger than the total time that PVM takes to process all the updates. Then the total execution time of PVM becomes linear as well. The point where the PVM performance switches from being flat to increasing linearly is roughly $min(p, m)$-fold earlier than SWEEP.

8 Related work

Self-maintenance [7,8,11,20,24] is one way to maintain the materialized views at the data warehouse without access to the base relations by replicating all or parts of the base data at the data warehouse. As more and more data are added to the data warehouse, the space complexity increases, giving 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 data warehouse when the materialized views are not self-maintainable. The ECA [30] family of

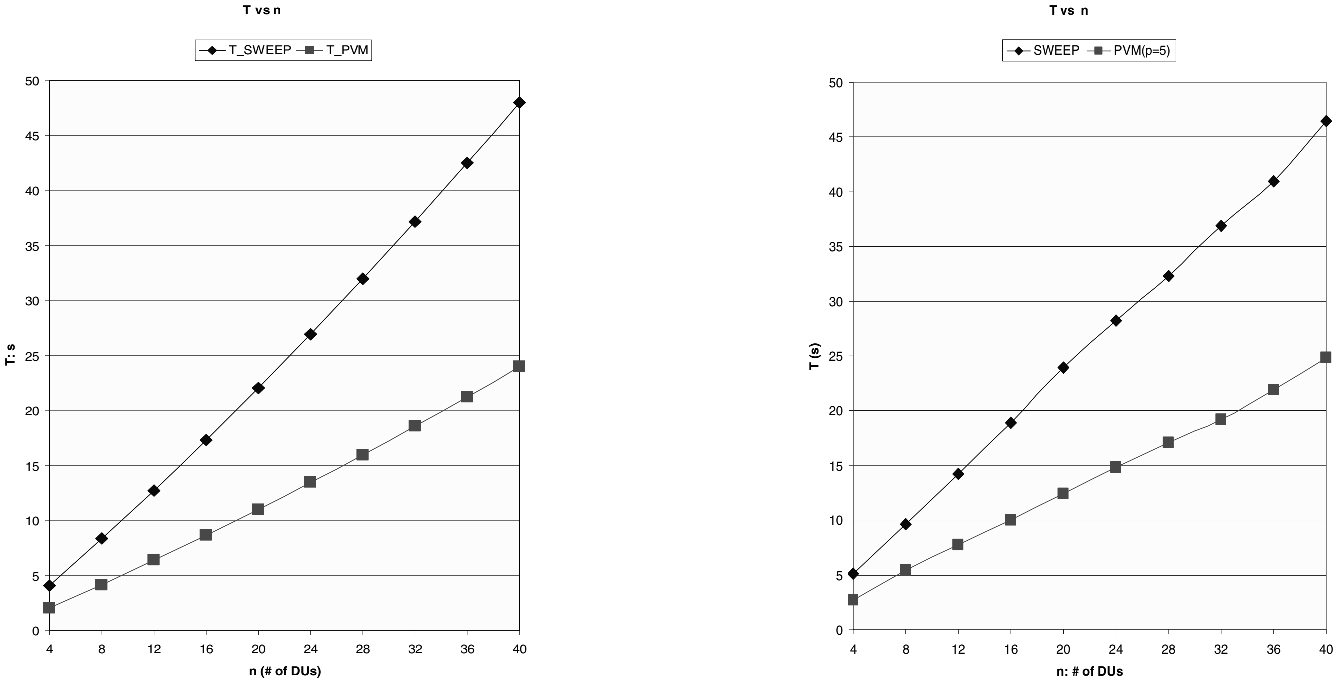


Fig. 25. Change number of updates. Left figure is the cost model. Right figure is the experimental result

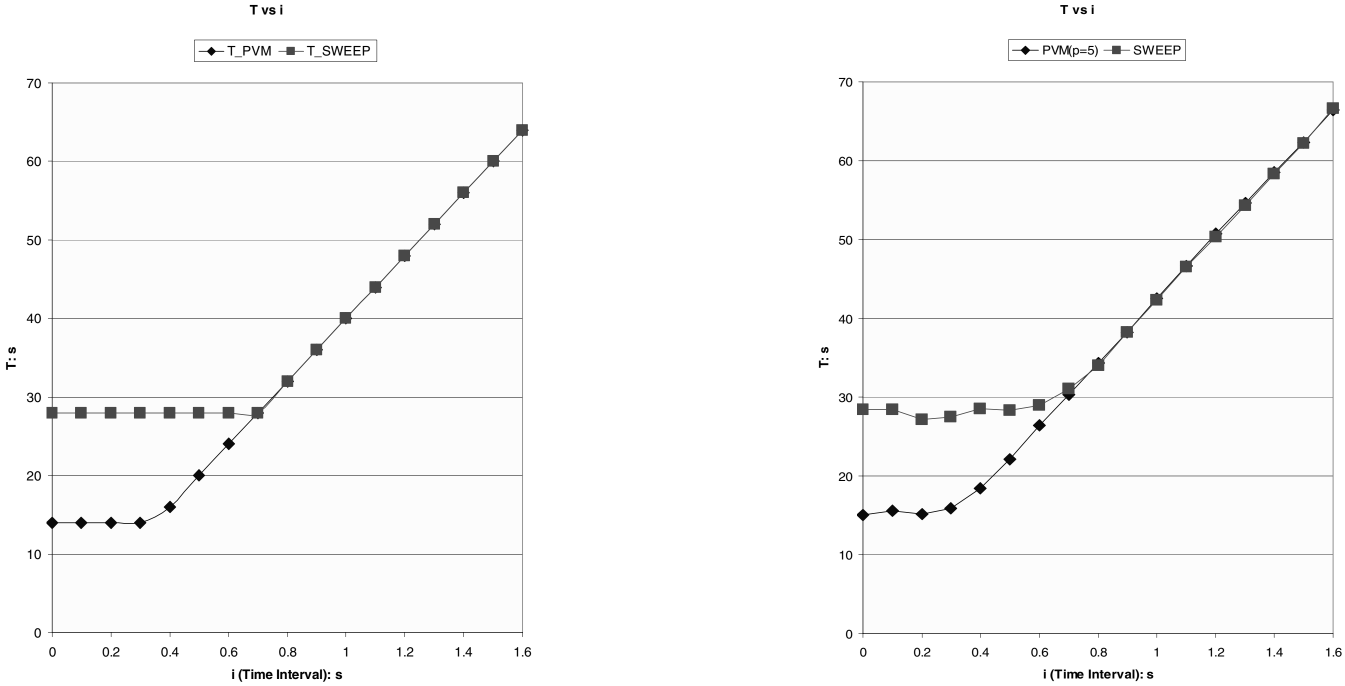


Fig. 26. Experiment 5: change time interval between two continuous data updates (i). Left figure is the cost model. Right figure is the experiment result

algorithms introduces the problem and solves it partially, i.e., for one single information source. Strobe [31] and SWEEP [1] are two view maintenance algorithms for multiple information sources both of which focus on the concurrency of data updates. The Strobe [31] algorithm introduces the concept of queuing the view updates at the data warehouse and committing the updates to the data warehouse only when the unanswered query set is empty. The algorithm solves the consistency problem but is subject to the potential threat of infinite waiting, i.e., the data warehouse extent may never get updated. Other mechanisms [2] are based on requiring the information sources to timestamp the view updates by a global clock shared by all information sources. So far, consistency maintenance methods either have the infinite waiting deficiency or need a global timestamp service, while our solution is able to make use of timestamps local to the data warehouse manager only.

Table 4. Comparison of ECA, Strobe, SWEEP, and PVM

Algorithm	Consistency	Message Cost	Quiescence	Key Points	Execution
Strobe	Strong	O(n)	Required	Unique key assumption	Multithread
C-Strobe	Complete	O(n!)	Not required	Unique key assumption	Multithread
SWEEP	Complete	O(n)	Not required	Local compensation	Single-thread
Nested SWEEP	Strong	O(n)	Not required but requires noninterference	Local compensation	Single-thread
POSSE	Complete	O(n)	Not required	Full concurrency	Multithread
PVM	Complete	O(n)	Not required	Parallel execution	Multithread

The SWEEP [1] family of algorithms eliminates the above mentioned limitations by applying local compensation techniques. SWEEP uses special detection methods for concurrent updates that do not need the global time stamp. It also requires no quiescent state before being able to update the data warehouse. Nested-SWEEP [1] is used to handle a set of updates by reusing the query results. Due to its recursive solution, it requires noninterference of the updates; otherwise, an infinite recursive call may result in a maintenance failure. PVM does not apply any recursive process optimization to share the query result as in Nested-SWEEP; rather it parallelizes the SWEEP maintenance process. So it will not have an infinite wait.

The Posse algorithm [19] proposes a data warehouse framework that supports a concurrent execution plan of the maintenance subqueries of one maintenance task, however, at the well-known cost of possibly having to bring more data back to the data warehouse manager. The algorithm can select various degrees of concurrency for the maintenance queries (or probing queries in their terms) to tradeoff between message size and processing cost. However no cost model, implementation, or experiment results are provided. Posse's work could also be used as a maintenance strategy in the $ViewChange$ module in our system where different execution plans can be plugged in for each maintenance task (by default, we assume to use SWEEP here, but any plan is possible and does not affect the correctness of our strategy.) PVM could improve the performance for their optimized algorithm if information sources remain idle exposed only with one maintenance task at a time. Some integration of the parallelism of intramaintenance query vs. the parallelism we propose across separate maintenance queries could be combined into one maximally flexible solution for maintenance. They also discuss briefly that the local compensation techniques from SWEEP [1] can be extended to support SQL view queries with aggregation and a HAVING clause over data with bag semantics. Such a solution is also applicable to our system.

A comparison of the features of popular algorithms, such as Strobe, C-Strobe, SWEEP, Nested-SWEEP, Posse [19], and PVM, is shown in Table 4 inspired by [1]. As we can see, PVM inherits all the advantages from its ancestors, e.g., complete consistency, low message complexity O(n), no quiescence requirement, and local compensation, and it is multithreadable.

Most algorithms like ECA [30], Strobe [31], SWEEP[1], and Posse[19] have been designed for a single view. Zhuge et al. [33] defined multiple views to be consistent with each other as the multiple view consistency problem. With multiple views, the maintenance algorithms discussed above could still be used to maintain the view. Also, some existing algorithms for a single view can be extended to handle multiple views [14], while new algorithms [3,5,33] have also been proposed specifically for multiple views.

Salem et. al [23] introduced an asynchronous incremental view maintenance algorithm. They keep the maintenance transaction asynchronous from the update transaction, so that the data warehouse can reach a state that is behind that of the information sources. Their algorithm can at most reach strong consistency, meaning some information source states will not be reflected in the data warehouse. However, their algorithm decreases the number of compensation queries and hence increases the overall maintenance performance. During the roll-up, the data warehouse manager still needs to wait for the remote queries to be processed. Hence parallelism techniques as proposed in our work could be applied here in order to further improve the maintenance performance.

9 Conclusions

In this paper, we have investigated the problem of parallel view maintenance. First we identified the potential performance bottleneck of the current state-of-the-art view maintenance solution called SWEEP [1] in terms of sequential handling of updates. In this work, we have identified several open issues to achieve parallel view maintenance, notable, concurrent data update detection in a parallel execution mode and the out-of-order-DW-commit problem. We then presented an integrated solution called PVM that is capable of handling both of these problems.

If we require the effects of data updates to be committed in the order in which they arrive, then PVM achieves the complete consistency of the data warehouse. If on the other hand we commit an update as soon as its successful handling has been completed, then some may be submitted before others. Hence PVM then would ensure the convergence level of consistency.

The parallel mechanism of the PVM algorithm can be applied to optimize prior view maintenance solutions from the literature that used sequential handling of maintenance tasks. In this paper, we apply PVM concepts to the SWEEP algorithm, the state-of-the-art algorithm, to improve its performance. Our cost model of SWEEP and the PVM systems demonstrates that PVM improves SWEEP's performance significantly. We have also fully implemented both SWEEP and PVM in our EVE data warehousing system [22], which is now one of the first public software tools for distributed view maintenance available in the database research community.

We have conducted a set of experiments to study the performance of PVM. The experimental results show that PVM has a multifold performance improvement over SWEEP [22] under a heavy load of updates. The more frequently updates happen and the more evenly distributed they are over the information sources, the larger a performance improvement can be gained by PVM over SWEEP. Given sufficient query processing capability of the information sources, we are able to observe up to p-fold performance improvement for PVM when increasing the number of parallel threads run in our system to p.

Acknowledgements. 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 Professor Nabil Hachem for his guidance with some of the ideas incorporated into this paper.

Appendix

A Correctness proofs of the PVM algorithm

A.1 Notations

Notations used in the proofs are listed in Table 5.
We assume all the views are select-project-join (SPJ) views.

B Proof of Theorem 6

Theorem 6: The **Out-of-Order-DW-Commit** problem defined by Definition 5 will only occur when first an add-tuple and then a delete-tuple, which will modify the same tuples in the data warehouse, are received by the data warehouse manager and both are handled in parallel by PVM.

Proof. We prove this by examining 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 a delete-tuple operation at position j in **DW-Receive-Order**, where $1 \leq i, j \leq 2$.

Using this notation, we represent Theorem 6 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 will show via an example that the following may hold: $E < a_1\, d_2 > \neq E < d_2\, a_1 >$.

Let us define the DW by $DW = V(R_1, R_2)$. R_1 is modified by data update "a_1," that is, by adding one tuple. The updated R_1 is denoted by R_1'. R_2 is modified by data update "d_2," that is, by dropping one tuple. The updated R_2 is denoted by R_2'. We assume the data warehouse manager received "a_1" before "d_2." Thus, $E(a_1) = (\cup)V(a_1, R_2)$, 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 get $DW' = DW + E < a_1\, d_2 >= DW \cup V(a_1, R_2) - V(R_1', d_2)$, which is $V(R_1', R_2')$. Thus, $DW + E < a_1\, d_2 >= DW'$.

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 $V(R_1, d_2)$ in DW. Thus, $V(a_1, d_2) + V(R_1, d_2)$, which is $V(R_1', d_2)$, is in $DW \cup V(a_1, R_2)$. Hence no error occurs at the deletion of $V(R_1', d_2)$. $DW + E < a_1\, d_2 >$ indeed 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 **Out-of-Order-DW-Commit** problem. The reason for this 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 data warehouse, 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.

It turns out to be true that $E < a_1\, d_2 > \neq E < d_2\, a_1 >$.

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

There is no **Out-of-Order-DW-Commit** problem for the data warehouse 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 data warehouse 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, $DW - E(d_1)$ has no tuple with negative count. Thus $DW - E(d_1)$ 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, $DW - E(d_1) - E(d_2)$ has no 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:

Table 5. Notations

$DU_j[i]$	data update at information source IS_j and assigned timestamp i by data warehouse manager
a_i	add-tuple operation at information source
d_i	is delete-tuple operation at information source
m	A **DW-Commit-Order** order of a-s and d-s denoted by a regular expression[7]
$E(DU)$	effect of DU on the data warehouse
$E < m >$	Extent after committing a set of updates in **DW-Commit-Order** m
$V(R_1, R_2, ..., R_n)$	extent of V generated based on the extents of $R_1, R_2, ...R_n$

- Because $R'_1 \subset R_1$ and $d_2 \in R_2$, $DW - E(d_2)$ has no 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, $E < d_2\ d_1 >= V(R'_1, R'_2)$.

In conclusion, because d_1 and d_2 are both in the original data warehouse, the **Out-of-Order-DW-Commit** problem does not occur. Thus $E < d_1\ d_2 >= E < d_2\ d_1 >$.

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 data warehouse 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 $< d_1 a_2 >$, which means d_1 was received earlier than a_2. As before, we denote R'_1 (or R'_2) as an 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 $DW - E(d_1)$ has no tuples with a negative count. It equals $V(R'_1, R_2)$. And the union will not generate any tuple with a negative count either.
- $DW' = DW - E(d_1) \cup E(a_2) = V(R_1, R_2) - V(d_1, R_2) \cup V(R'_1, a_2) = V(R'_1, R_2) \cup V(R'_1, a_2) = V(R'_1, R'_2)$. 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 the 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 $(DW \cup E(a_2)) - E(d_1)$ has no 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 **Out-of-Order-DW-Commit** problem occurs because the tuples to be deleted are in the original data warehouse. Thus $E < d_1\ a_2 >= E < a_2\ d_1 >$. Q.E.D.

C Proof of Lemma 7

Lemma 7: Given the negative counter mechanism described in Sect. 3.3, the counters of all tuples will always be positive when the data warehouse reaches a quiescence state.

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

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 6, we know that only a_1 being received earlier than d_2 by the data warehouse can cause the **Out-of-Order-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 compensated by $E(a_1)$. This means the final result will be positive or an empty set. Q.E.D.

D Proof of Lemma 8

Lemma 8: The negative-counter-based mechanism described in the algorithm in Figs. 13 and 14 correctly solves the **Out-of-Order-DW-Commit** problem if any two data updates are handled in parallel by the data warehouse.

Proof. We will prove Lemma 8 based on Theorem 6 and Lemma 7. We can represent Lemma 8 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 updates, no matter how they commit to the data warehouse, the final state of the data warehouse will be consistent with the information source space if we use the negative counter technique. We break this discussion into four subcases:

- Case 1: $E < a_1\ a_2 >= E < a_2\ a_1 >$
- Case 2: $E < d_1\ d_2 >= E < d_2\ d_1 >$
- Case 3: $E < d_1\ a_2 >= E < a_2\ d_1 >$
- Case 4: $E < a_1\ d_2 >= E < d_2\ a_1 >$

Cases 1, 2, and 4 have already been shown not to produce any **Out-of-Order-DW-Commit** problem by Theorem 6. Hence we do not need to discuss them further; we will only need to explain how the negative count concept solves the **Out-of-Order-DW-Commit** problem of $E < a_1\ d_2 >= E < d_2\ a_1 >$.

Assume we have a data warehouse with two information sources that have R_1 and R_2, respectively, defined by $DW = V(R_1, R_2)$. Assume the two updates 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 not really be 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 the data warehouse. So the final result will be correct.

Thus $E < a_1\ d_2 >= E < d_2\ a_1 >$.

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 the data warehouse.

E Theorem 9 proof

Theorem 9: The negative-counter-based algorithm (Figs. 13 and 14) correctly solve the **Out-of-Order-DW-Commit** problem for any number of **maintenance-concurrent** updates.

Proof by induction.

Induction hypothesis: Theorem 9 is true for any set of updates of length n.

Induction basis:

For n = 1: It is trivial

For n = 2: It has proven by Lemma 8

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 8, we can see that, for a pair of data updates a_i, d_{i+1} with p,q denoting any prefix or suffix, 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 the same reason, we have the 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 on both sides with only one pair of data updates having their order switched will result in the same data warehouse 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, O_l always exists in S_1, which corresponds to O_{i_1} in S_2, which in turn can move to the first place of S_1 by 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 that we can convert it to the remaining sequence of S_2. Thus 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.

References

1. Agrawal D, El Abbadi A, Singh A, Yurek T (1997) Efficient view maintenance at data warehouses. In: Abstracts of ACM SIGMOD international conference on management of data, Tucson, AZ, 13–15 May 1997, pp 417–427
2. Baralis E, Ceri S, Paraboschi S (1996) Conservative timestamp revised for materialized view maintenance in a data warehouse. In: Abstracts of the workshop on materialized views: techniques and applications (VIEW 1996), Montreal, 7 June 1996, in cooperation with SIGMOD conference 1996, pp 1–9
3. Colby L, Kawaguchi A, Lieuwen D, Mumick I, Ross K (1996) Supporting multiple view maintenance policies. In: Abstracts of the ACM SIGMOD international conference on management of data, SIGMOD 1997, 13–15 May 1997, Tucson, AZ, pp 405–416
4. Colby LS, Griffin T, Libkin L, Mumick IS, Trickey H (1996) Algorithms for deferred view maintenance. In: Abstracts of ACM SIGMOD international conference on management of data, Montreal, 4–6 June 1996
5. Colby LS, Mumick IS (1996) Staggered maintenance of multiple Views. In: Abstracts from workshop on materialized views: techniques and applications, pp 119–128 (VIEW 1996), Montreal, 7 June 1996
6. Ding L, Zhang X, Rundensteiner EA (1999) The MRE wrapper approach: enabling incremental view maintenance of data warehouses defined on multi-relation information sources. In: Abstracts of the ACM 2nd international workshop on data warehousing and OLAP (DOLAP'99), 6 November 1999, Kansas City, MO, pp 30–35
7. Gupta A, Jagadish H, Mumick I (1996) Data integration using self-maintainable views. In: Abstracts of the 5th international conference on extending database technology (EDBT), Avignon, France, 25–29 March 1996, pp 140–144
8. Gupta A, Jagadish HV, Mumick IS (1997) Maintenance and self maintenance of outer-join views. In: The 3rd international workshop on next generation information technologies and systems (NGITS '97), Neve Ilan, Israel
9. Gupta A, Mumick I (1995) Maintenance of materialized views: problems, techniques, and applications. IEEE Data engineering bulletin, special issue on materialized views and warehousing, 18(2):3–19

10. Gupta A, Mumick IS, Subrahmanian VS (1993) Maintaining views Incrementally. In: Abstracts of ACM SIGMOD international conference on management of data, Washington, D.C., 26–28 May 1993, pp 157–166
11. Huyn N (1996) Efficient view self-maintenance. Abstracts of the workshop on materialized views: techniques and applications (VIEW 1996), Montreal, 7 June 1996, pp 17–25
12. Kawaguchi A, Lieuwen DF, Mumick IS, Quass D, Ross KA (1997) Concurrency control theory for deferred materialized views. Abstracts of database theory – ICDT '97, 6th international conference, Delphi, Greece, 8–10 January 1997, pp 306–320
13. Kawaguchi A, Lieuwen DF, Mumick IS, Ross KA (1997) Implementing incremental view maintenance in nested data models. In: Abstracts from the 6th international workshop on database programming languages (DBPL-6), Estes Park, CO, 18–20 August 1997, pp 202–221
14. Labio WJ, Yerneni R, García-Molina H (1999) Shrinking the warehouse updated window. In: Abstracts of ACM SIGMOD international conference on management of data, 1–3 June 1999, Philadelphia, pp 383–395
15. Lee AJ, Nica A, Rundensteiner EA (2001) The *EVE* approach: view synchronization in dynamic distributed environments. IEEE Transactions on knowledge and data engineering, 14(5):931–954
16. Liu B, Chen S, Rundensteiner EA (2002) A transactional approach for parallel data warehouse maintenance. Technical report WPI-CS-TR-02-08, Worcester Polytechnic Institute
17. Mohania MK, Konomi S, Kambayashi Y (1997) Incremental maintenance of materialized views. In: Abstracts of the 8th international conference on database and expert systems applications (DEXA '97), Toulouse, France, 1–5 September 1997, pp 551–560
18. Nica A, Lee AJ, Rundensteiner EA (1998) The CVS algorithm for view synchronization in evolvable large-scale information systems. In: Abstracts of the international conference on extending database technology (EDBT'98), Valencia, Spain, March 1998, pp 359–373
19. O'Gorman K, Agrawal D, Abbadi AE (1999) Posse: a framework for optimizing incremental view maintenance at data warehouses. In: Abstracts of the 1st international conference on data warehousing and knowledge discovery (DaWaK '99), Florence, 30 August–1 September 1999, pp 106–115
20. Quass D, Gupta A, Mumick IS, Widom J (1996) Making views self-maintainable for data warehousing. In: Abstracts of the 4th international conference on parallel and distributed information systems, 18–20 December 1996, Miami Beach, pp 158–169
21. Rundensteiner EA, Koeller A, Zhang X (2000) Maintaining data warehouses over changing information sources. Communications of the ACM, pp 57–62
22. Rundensteiner EA, Koeller A, Zhang X, Lee A, Nica A, Van-Wyk A, Li Y (1999) Evolvable view environment. In: Abstracts of ACM SIGMOD international conference on management of data, 1–3 June 1999, Philadelphia, pp 553–555
23. Salem K, Beyer KS, Cochrane R, Lindsay BG (2000) How to roll a join: asynchronous incremental view maintenance. In: Chen W, Naughton JF, Bernstein PA (eds) Proceedings of the 2000 ACM SIGMOD international conference on management of data, Dallas, 16–18 May 2000, 29(2):129–140
24. Samtani S, Kumar V (1998) Maintaining consistency in partially self-maintainable views at the data warehouse. In: Abstracts of the 9th international conference on systems applications (DEXA '98), Vienna, 24–28 August 1998, pp 206–211
25. Wiener JL, Gupta H, Labio W, Zhuge Y, Garcia-Molina H, Widom J (1996) A system prototype for warehouse view maintenance. Abstracts of the workshop on materialized views: techniques and applications (VIEW 1996), Montreal, 7 June 1996, pp 26–33
26. Zhang X, Rundensteiner EA (1999) Flexible data warehouse maintenance under concurrent schema and data updates. In: Abstracts of IEEE international conference on data engineering, special poster session, Sydney, March 1999, p 253
27. Zhang X, Rundensteiner EA (1999) The SDCC framework for integrating existing algorithms for diverse data warehouse maintenance tasks. In: Abstracts of the international database engineering and applications symposium (IDEAS 1999), 2–4 August 1999, Montreal, pp 206–214
28. Zhang X, Rundensteiner EA DyDa: dynamic data warehouse maintenance in a fully concurrent environment. In: Abstracts of data warehousing and knowledge discovery. Lecture notes in computer science. Springer, Berlin Heidelberg New York, September 2000, pp 94–103
29. Zhang X, Rundensteiner EA, Ding L (2001) PVM: parallel view maintenance under concurrent data updates of distributed sources. In: Abstracts of data warehousing and knowledge discovery, Munich, September 2001, pp 230–239
30. Zhuge Y, García-Molina H, Hammer J, Widom J (1995) View maintenance in a warehousing environment. In: Abstracts of the 1995 ACM SIGMOD international conference on management of data, San Jose, 22–25 May 1995, pp 316–327
31. Zhuge Y, García-Molina H, Wiener JL (1996) The Strobe algorithms for multi-source warehouse consistency. In: Abstracts of of the 4th international conference on parallel and distributed information systems, 18–20 December 1996, Miami Beach, pp 146–157
32. Zhuge Y, García-Molina H, Wiener JL (1998) Consistency algorithms for multi-source warehouse view maintenance. Distributed Parallel Databases 6(1):7–40
33. Zhuge Y, Wiener JL, García-Molina H (1997) Multiple view consistency for data warehousing. In: Abstracts of the 13th international conference on data engineering, 7–11 April 1997 Birmingham, UK, pp 289–300