Multiple View Consistency for Data Warehousing*

Yue Zhuge, Janet L. Wiener and Hector Garcia-Molina

Computer Science Department
Stanford University
Stanford, CA 94305-2140, USA
{zhuge,wiener,hector}@cs.stanford.edu
http://db.stanford.edu/pub/zhuge/1996/mvc.ps

Abstract

A data warehouse stores integrated information from multiple distributed data sources. In effect, the warehouse stores materialized views over the source data. The problem of ensuring data consistency at the warehouse can be divided into two components: ensuring that each view reflects a consistent state of the base data, and ensuring that multiple views are mutually consistent. In this paper we study the latter problem, that of guaranteeing multiple view consistency (MVC). We identify and define formally three layers of consistency for materialized views in a distributed environment. We present a scalable architecture for consistently handling multiple views in a data warehouse, which we have implemented in the WHIPS(WareHousing Information Project at Stanford) prototype. Finally, we develop simple, scalable, algorithms for achieving MVC at a warehouse.

1 Introduction

A data warehouse stores integrated information from multiple distributed data sources. It can be used for storing cleaned, summarized, analytical data; storing historical data or backup data; or caching query results for fast query response time, e.g., in a mediation system[Wie92]. Data at the warehouse are usually read-only; they can be seen as materialized views defined over base data at the sources [LW95].

When base data at sources change, views at the warehouse need to be updated accordingly. Computing the new view states may be done by either periodical re-computation of the entire view, or by incremental maintenance of the view. With incremental maintenance, initially views are computed and stored at the warehouse. Updates on base data are reported from data sources to an integration component, the integrator. The integrator computes the corresponding changes to each view, and sends the changes to the warehouse. The warehouse then applies these changes to the views. Incremental view maintenance typically out-performs re-computation in cases where the volume of source data is large, source data is unavailable, or no "down time" is permitted at the warehouse for re-computation [ZGMHW95, SJ96].

In this paper we focus on maintaining data consistency at the warehouse as it is incrementally maintained. Intuitively, consistency means that the warehouse state "makes sense," that is, that it reflects (e.g., is a copy

*This work was partially supported by Rome Laboratories under Air Force Contract F30602-94-C-0237 and by an equipment grants from Digital Equipment Corporation and IBM Corporation.

or summary of) an actual source state at some "recent time." There are two aspects to making the warehouse consistent: making each individual view consistent, and making the views mutually consistent. In this paper we study the latter, which we call the *multiple view consistency (MVC)* problem.

Example 1: The multiple view consistency problem

Suppose there are two views at the warehouse defined as $V_1 = R \bowtie S$ and $V_2 = S \bowtie T$. The contents of the base relations and the views are shown in the following Table 1. At time t_0, relation S is empty, so both V_1 and V_2 are empty. At t_1, a tuple $[2, 3]$ is inserted into S. At t_2, changes to V_1 are computed (by joining tuple $[2, 3]$ with relation R) and the result is inserted into view V_1. At t_3, changes to V_2 are computed and the result is inserted into view V_2.

Time	R: A	B	S: B	C	T: C	D	V_1 : A	B	C	V_2 : B	C	D
t_0	1	2	-	-	3	4	-	-	-	-	-	-
t_1	1	2	2	3	3	4	-	-	-	-	-	-
t_2	1	2	2	3	3	4	1	2	3	-	-	-
t_3	1	2	2	3	3	4	1	2	3	2	3	4

Table 1: Changes of base relations and views

For both V_1 and V_2, the above view maintenance steps are correct. That is, their contents are correctly updated to reflect the change in the base relation S. However, after time t_2, V_1 reflects the new state of relation S but V_2 does not. The two views are not consistent with each other.

Notice that computing consistent values for each view, e.g., inserting tuple $[1, 2, 3]$ into V_1, is not as simple as it may first appear. For example, while we are computing this tuple, base relation R or S could be updated, leading us to insert into V_1 a tuple that never really existed in the join of R and S.

The algorithms we developed in [ZGMHW95, ZGMW96] can be used to ensure that each view does reflect consistent data. However, those algorithms only ensure the consistency of each individual view. The algorithms we develop in this paper work with single view consistency algorithms to ensure that updates to *multiple* views are applied in a mutually consistent fashion. Even though both the single and multiple view maintenance algorithms deal with consistency, as we will see, they are quite different. In particular, the multiple view algorithms are independent of the data model, while the single view ones depend on the model and particular type of views defined. □

The major contributions of this paper are:

- We identify and define formally three layers of consistency for materialized views in a distributed environment, including multiple view consistency.
- We provide a scalable architecture for consistently handling multiple views in a data warehouse. In this architecture, each view is managed by a separate concurrent process, increasing concurrency and easily allowing different algorithms for materializing each view. This architecture is implemented in our WHIPS (WareHousing Information Project at Stanford) warehouse prototype [WGL+96].

- We develop simple, scalable algorithms for achieving MVC at a warehouse. The algorithms coordinate concurrent view managers so that their warehouse updates do not violate consistency. Each algorithm can be thought of as a specialized concurrency control mechanism that exploits the application semantics (incremental view maintenance) to achieve greater concurrency.

In the rest of this introduction, (a) we argue that multiple view consistency is important for some applications, (b) we sketch the basic idea behind our architecture and MVC algorithms, and (c) we briefly comment on related work. The rest of the paper then presents the architecture and algorithms more formally.

1.1 Importance of MVC

As with traditional database systems, data consistency may or may not be important, depending on the application. Many current warehouses are used only for statistical or trend analysis, and inconsistencies may not have an impact on the final results. However, warehouses are being used more and more for other activities where consistency may be very important. For example, a warehouse may be used to handle customer inquiries, removing load from the operational systems. When the customer calls with a question, we would like to be able to read her data consistently: her checking account record, for instance, should match with her linked savings account record. If the warehouse is trying to identify *particular* customers for a special promotion (and not just general trends) based on data from multiple views, it would be useful to get the correct customers.

It is also important to note that MVC is required by some view maintenance algorithms. For example, in the multiple view maintenance problem described in [RSS96, LQA96], auxiliary views are stored in order to maintain primary views efficiently. For example, in order to maintain $V = R \bowtie S \bowtie T$, the algorithm might choose (according to some heuristic) to materialize relations $R \bowtie S$ and $S \bowtie T$ and compute V from them. The two sub-views must be consistent with each other whenever V is computed. Auxiliary views may also be stored to guarantee view self-maintainability [GJM96, QGMW96].

The key problems in keeping multiple views consistent at the warehouse are as follows.

1. One source update transaction may invoke a set of actions on multiple warehouse views, as we saw in Example 1. These actions must be applied at the warehouse as an "atomic unit." These atomic units need to be applied at the warehouse in the same (or a consistent) order as the original updates.

2. Computing those actions, or the *delta computation* of views, often takes time. It may involve queries back to the sources if base data is not cached at the warehouse. This makes it desirable to process both different updates and different views concurrently.

3. A delta computation not only takes time but it may be "intertwined" with subsequent updates. For instance, in Example 1, in between times t_1 and t_2 we computed the join of the new S tuple $[2,3]$ with R. If R is updated before we read it, we may get fewer or more tuples than what we wanted. One way

out of this dilemma is to combine both updates (the original one to S and the new one to R) into a single "atomic unit;" however, we now need the machinery to combine the updates.

The simplest solution to the MVC problem is to create a single *integrator process* that handles updates sequentially. For each update it receives, it sequentially computes the changes to all views that result from the update. (If later updates are intertwined, they must be pulled into the computation. This in itself must be done very carefully to avoid violating consistency.) After all the view changes are computed, the process submits a transaction to the warehouse, waits for it to commit, and moves on to the next update. Clearly, this does not allow for any concurrency, and given Problem 2 above, is not acceptable in a high update rate environment. However, note that this sequential approach to the MVC problem is taken (by default) by many current proposed view maintenance algorithms, e.g., [HZ96].

1.2 Merge process to enforce MVC

Instead, we propose an architecture that allows as much parallelism as possible while still guaranteeing a correct solution to the MVC problem. Our architecture is shown in Figure 1. Updates on base data are reported from data sources to the integrator, and then forwarded to relevant view managers. Each view manager is a separate process that handles the delta computation for one view. View managers may reside on different machines. After computing the changes to its view, each view manager sends a list of actions to a *merge process*. The merge process collects changes to the views, holds them until all affected views can be modified together, and then forwards all of the views' changes to the warehouse in a single warehouse transaction. The merge process also keeps track of dependencies between warehouse transactions in oder to effectively control their commit order.

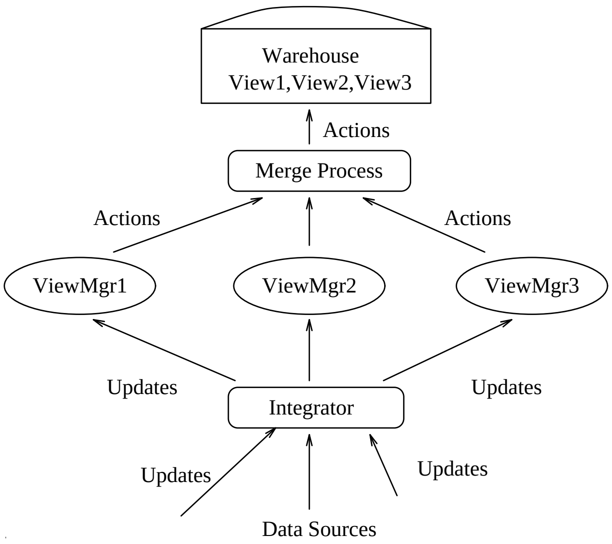

Figure 1: Data warehouse architecture

Returning to Example 1, the merge process will receive the changes to V_1 at time t_2 (from the V_1

manager), but it will hold them until it receives the corresponding changes to V_2 at time t_3. The merge process will then notify the warehouse to update both views in a single transaction. In this paper we provide algorithms for the merge process to decide when to hold and when to forward actions.

Although we split the view maintenance computation by view, there are other options to parallelize the system. For example, one could choose to use one process per update, or to use multiple threads within a process. The MVC problem arises with each of these options, and the algorithms we present in this paper can be adapted to those scenarios. However, for concreteness, in this paper we only consider the architecture of Figure 1. Incidentally, this view manager architecture does have one important advantage over other parallel architectures: Since each view is under the control of a separate process, it is very easy to use different maintenance algorithms for each view. This selection of an algorithm on a per view basis is very important since some views, e.g., aggregate views need to use different maintenance algorithms than other views, e.g., a copy of a base relation. It is also possible to design specific view managers to handle new, non-traditional types of views without affecting the existing system. For example, a specialized view manager might predict the new results of data mining tests on a view using the changes to be integrated into the view. Our architecture also makes it easy to add and delete views on the fly.

1.3 Related work and organization of this paper

We believe that MVC is an important problem that has not been addressed in previous research. In our previous papers, [ZGMHW95] and [ZGMW96], we defined single view consistency in a data warehousing environment. Hull and Zhou also defined single view consistency in [HZ96] for the Squirrel system, a data warehouse that materializes intermediate views and uses incremental view maintenance. In the Squirrel system, it is possible to achieve MVC by sequencing the propagation of each source update. Single view consistency is also discussed in [BCP96]. The definitions of materialized view consistency in the above literature are based on the database state, a global clock, timestamps, or transaction isolation properties. However, most of these definitions can be adapted to obtain the levels of consistency we define in the next section for single views. There are many view maintenance algorithms, both centralized and distributed [BLT86, GM95, GL95, SJ96]. In our system, each view manager may implement any of these existing algorithms. The merge process only needs to know the consistency level provided by each view manager so that it can run the proper merge algorithm. Finally, in a multi-database system [BGMS92, LMR90], updating the global "view" is within the same global transaction as reading source base data and therefore multiple views can be updated consistently in a single global transaction. We consider a more loosely coupled system where data sources are autonomous.

The remainder of this paper is organized as follows: In Section 2, we formally define multiple view consistency. Section 3 provides more details of our system architecture and framework. We develop two algorithms, the Simple Painting Algorithm and the Painting Algorithm, used by the merge process to achieve MVC, in Sections 4 and 5, respectively. Section 6 discusses how to distribute the algorithms and the load of the merge process to allow more parallelism. In Section 7, we discuss extensions of the algorithms. We conclude the paper in Section 8.

2 Consistency

We divide the consistency problem into three layers: source consistency, view consistency and multiple view consistency (MVC). Source consistency refers to the consistency among base data, and is enforced by source transaction protocols (if any). View consistency refers to the consistency between a view at the warehouse and its base data at the sources and is enforced by the view managers. MVC refers to the mutual consistency among views, which will be enforced by the MVC algorithms we develop in the Sections 4 and 5. Figure 2 shows where these three layers of consistency lie in the system. We discuss them each in turn.

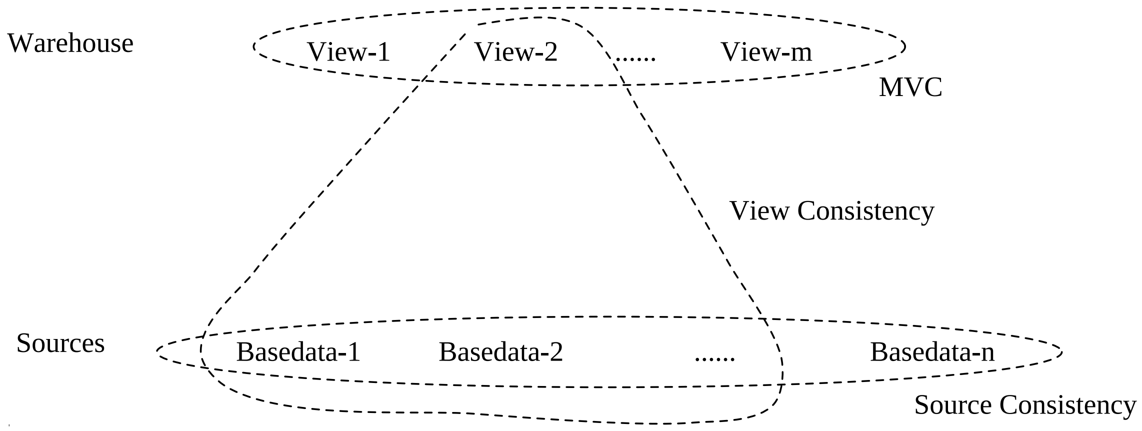

Figure 2: Three layers of consistency in a data warehouse

2.1 Source consistency

For simplicity we assume that transactions span a single source and generate a single update. (In Section 7.1 we show that the algorithms developed in this paper can be easily extended to handle source transactions involving multiple sources and updates.) We assume that the execution of source transactions is serializable and equivalent to the schedule $S = U_1, U_2, \ldots U_f$, where U_i represents the transaction that generates update U_i.

Given any serial schedule R, we define its *state sequence*, $R_{seq} = ss_0, ss_1, \ldots ss_f$ to be the base data states after each transaction commits. State ss_0 is the state before any transaction commits, and ss_f is the final state after all commit. We say that R_{seq} is a *consistent source state sequence* if R is equivalent to S defined in the previous paragraph.

2.2 View consistency

Consider one view V at the warehouse. As the view is maintained, the warehouse goes through *warehouse states*: $W_{seq} = ws_0, ws_1, \ldots, ws_q$.

Definition: We say that ws_j is consistent with source state ss_i, written $ws_j = ss_i$, if V at ws_j has the same content as $V(ss_i)$, which is the result of evaluating the expression of V at source state ss_i. □

There are many possible levels of view consistency. We defined four levels in [ZGMW96]. In this paper we focus on the two most common levels: strong consistency and completeness.

Definition: A warehouse state sequence W_{seq} is *strongly consistent* if there exists a consistent source state sequence $S_{seq} = ss_0, ss_1, \ldots, ss_f$ and a mapping m from warehouse to source states, such that:

- Each warehouse state reflects some consistent source state. That is, for all ws_j, $ws_j = m(ws_j)$.
- The order of warehouse states matches the order of corresponding source states. That is, if $ws_j < ws_k$, then $m(ws_j) < m(ws_k)$.
- Eventually, the view reflects the final state of the source base data. That is, $ws_q = ss_f$. □

Definition: A warehouse state sequence W_{seq} is *complete* if it is strongly consistent, and for every source state ss_i, there exists a ws_j such that $m(ws_j) = ss_i$. That is, every source state is reflected in order at the warehouse. □

A view manager is responsible for bringing a view into a new consistent state after the source changes, as we saw in Example 1. Different view managers can be designed to maintain a view with different consistency levels. A *complete view manager* generates a complete warehouse state sequence. It processes one update U_j at a time and generates the warehouse view that is consistent with the source state after U_j executed. A *strongly consistent view manager* generates strongly consistent warehouse state sequences. It can batch multiple updates, U_i through U_{i+k}, bringing the warehouse from a state consistent with the sources before U_i to a state consistent with the sources after U_{i+k}. Because a strongly consistent view manager can batch intertwined updates, it is often more desirable in practice.

2.3 Multiple view consistency (MVC)

When there are multiple views at the warehouse, a *warehouse state ws* is a vector with one element for the state of each view. Each warehouse view maintenance transaction updates one or more views. The warehouse state advances after each warehouse transaction and the sequence of warehouse maintenance transactions yields a sequence of warehouse states $W_{seq} = ws_0, ws_1, \ldots, ws_q$.

Definition: We say that ws_j is multiple view consistent with source state ss_i, written $ws_j \equiv ss_i$, if for each view V at the warehouse state ws_j, its content is the same as $V(ss_i)$. □

The definitions for multiple view consistency (MVC) are very similar to that for single view consistency. All we need to do is replace "=" by "≡" in our previous definitions. For example, a warehouse system is strongly consistent if its warehouse state sequence is strongly consistent, as stated by our earlier definition with "=" replaced by "≡." With such a system, each materialized view will be strongly consistent, and for each warehouse state, all of the views will be mutually consistent. The two algorithms we present in Sections 4 and 5 guarantee MVC completeness and strong consistency, respectively.

3 System Framework

In this section we present the basic framework for our view maintenance algorithms. In particular, we describe how the processes of Figure 1 interact.

3.1 Data model

In all of the discussions and algorithms in this paper, the data at the sources and at the warehouse need not be relational. (Note that the consistency definitions of Section 2 are also independent of the data model.) However, for simplicity, in our examples we use a relational model with simple project-select-join views. Also, for our examples we assume that each update is a single tuple insert, delete, or modification.

3.2 Integrator process

The integrator receives updates from each source. Recall that each update records the changes made by a transaction and that transactions span a single source (Section 2). We assume that updates from the same source arrive at the integrator in the order they committed.

When the integrator receives an update, it performs the following steps.

1. The integrator numbers the updates by arrival order. For example, U_5 is the fifth update received. Let U_i be the update that just arrived.

2. The integrator determines the *relevant* views for U_i, REL_i. A view is relevant to U_i if it needs to be modified because of U_i. For example, for a relational model, the integrator can determine the source relation R that was modified by U_i. Then it can include in REL_i all views that use R in their definition. We could be more discerning by using selection conditions in the view definitions to rule out irrelevant updates[JBL86]. Other optimizations can also be done but are not discussed here.

3. The integrator sends REL_i to the merge process. The subscript i is implicitly sent, so that the merge process knows to which update REL_i refers. The set REL_i will be used by the merge process to know from which view managers to expect actions.

4. The integrator sends a copy of U_i to each view manager responsible for a view in REL_i. Again, the subscript i is implicitly included.

Notice that it is not necessary for the integrator to send a relevant view set REL_i directly to the merge process. An alternative could be for the integrator to send REL_i to one or more of the view managers in that set, together with U_i. The view managers could then forward REL_i to the merge process when they deliver their action lists (see below). This reduces the number of messages and may be more efficient than the scheme we adopt here. However, it is easier to describe the merge process when REL_i arrives directly, so this is what we use. The changes to the merge process to handle the alternate scheme are straightforward.

3.3 View managers

A view manager VM_x receives (in the order sent by the integrator) a sub-sequence of updates $U_{i_1}, U_{i_2}, \dots U_{i_f}$. If VM_x is complete, then for each update U_j it will generate an *action list* AL_j^x. This list contains the operations necessary to make view V_x (handled by VM_x) consistent with the source state existing after U_j was performed at its source. The view manager then forwards AL_j^x to the merge process, in order by j subscript. The view manager identifier, x, and the update identifier, j, are implicitly included. If an action list AL_j^x happens to be empty, it is still sent to the merge process. (This is not essential; it just simplifies our merge algorithm.)

A strongly consistent view manager VM_x is similar, except that several updates $U_{i_k}, \dots U_{i_{k+n}}$ can generate a single action list $AL_{i_{k+n}}^x$. Notice that the subscript of the action list identifies the last update that is included in the batch. Thus, again, AL_j^x contains the actions needed to make the view consistent with the source state existing after U_j executed.

In our solution, all actions pass through the merge process where they are coordinated and later forwarded to the warehouse. An alternative is to let the integrator in Figure 1 run a traditional distributed transaction processing protocol involving all relevant view managers and the warehouse. When update information is received, the integrator then starts a transaction within which each relevant view manager executes actions on its view directly at the warehouse. The transaction ensures that the set of actions executed is serialized with respect to actions belonging to other transactions. However, the merge process (running the algorithms we will present here) is still necessary in order to coordinate transaction commits. This alternative approach also binds one warehouse transaction to each update, while our framework allows each warehouse transaction to correspond to multiple intertwined source updates. Further, the alternative approach is only possible if different processes may contribute actions to the same transaction and this is difficult to do in some commercial database systems.

4 Simple Painting Algorithm

The Simple Painting Algorithm (SPA) is used by the merge process to maintain MVC at the warehouse when all view managers are complete. As we will see in Section 4.4, SPA guarantees complete warehouse states. SPA receives action lists from view managers and REL_i sets from the integrator, and generates transactions for the warehouse database system. We make no restrictions on message arrival order, except that messages from the same process must arrive in the order sent. This means that the merge process may receive a list AL_j^x without having received REL_j. This situation is taken care of by the algorithm.

4.1 Data Structures

Through an example, we introduce a data structure called the (*ViewUpdateTable*) (*VUT*) that we will use in the Simple Painting Algorithm.

Example 2: ViewUpdateTable (VUT)

Suppose we have three views: $V_1 = R \bowtie S$, $V_2 = S \bowtie T \bowtie Q$ and $V_3 = Q$. There are two source updates: U_1 on S and U_2 on Q. When the merge process receives REL_1 and REL_2, it builds the following (*ViewUpdateTable*) (*VUT*). The *VUT* is a two dimensional table. $VUT[i, x]$ corresponds to update U_i and view V_x.

VUT:

	$V_1(R, S)$	$V_2(S, T, Q)$	$V_3(Q)$
$U_1(S)$	(white)	(white)	(black)
$U_2(Q)$	(black)	(white)	(white)

In SPA, each entry $VUT[i, x]$ contains a single *color* field. $VUT[i, x].color$ indicates whether an update is related to a view, and whether the corresponding action list has been received. SPA initializes the color for each update U_i and view V_x when it first receives REL_i. There are four possible colors:

- *white* (w for short): waiting for the corresponding action list for this entry;
- *yellow* (y for short): the corresponding action list has been received, however, the merge process is waiting for other actions before applying it;
- *gray* (g for short) : the corresponding action list has just been applied;
- *black* (b for short): the entry need not be examined.

We know that a complete view manager sends one AL per relevant update. The merge process waits for one AL for each entry in the *VUT* whose color is *white*. As the action lists are received, they are saved in an array WT until they can be applied. To illustrate, suppose that the merge process receives AL_1^2 from VM_2 (which handles V_2). The merge process saves this action in WT_1 and sets the color of the corresponding entry:

	V_1	V_2	V_3	WT_i
U_1	w	w	b	$\emptyset$
U_2	b	w	w	$\emptyset$

$\Longrightarrow$

	V_1	V_2	V_3	WT_i
U_1	w	y	b	$\{AL_1^2\}$
U_2	b	w	w	$\emptyset$

The merge process at this time cannot apply actions in AL_1^2 to the warehouse because it knows from the VUT that U_1 also affects V_1 and it needs to wait for the corresponding actions from VM_1. Only after the merge process receives AL_1^1 can it apply both action lists together and update both views. When the warehouse applies an action list, the corresponding entry in the VUT is set to *gray*. A row in the VUT can be purged when all actions for the row have been applied to the views. □

Notice that collecting the action lists and applying them together may not always be as trivial as in the above example. For example, the receipt of an action list in row i may enable action lists in later rows to be applied. Also, some actions corresponding to later updates may be applied before actions for earlier ones, provided that those updates do not affect the same views. SPA handles all of these cases, some of which we will illustrate in Example 3.

4.2 Algorithm SPA

The family of MVC algorithms are called Painting Algorithms because applying the action lists is like painting a grid wall from top to bottom, following "guidelines." For example, one guideline is that a grid row can only be painted gray when all of its entries are either yellow or black. This ensures that a view is only updated to a consistent state, when all relevant action lists have arrived.

SPA is driven by two kinds of events: the receipt of REL_i from the integrator and the receipt of AL_i^x from view manager VM_x. As stated earlier, it is possible that the merge process receives AL_i^x before it receives REL_i. In this scenario, the merge process needs to delay the processing of AL_i^x until after REL_i arrives.

Procedure $ProcessRow(i)$ checks whether all action lists in row i and earlier have arrived and can be applied to yield a new consistent warehouse state. If so, it applies all actions in row i to the views in a single warehouse transaction. Then it recursively calls itself to check whether subsequent rows of actions can now be applied. We further explain some lines in $ProcessRow(i)$ after presenting the algorithm.

Let VM be the set of all view managers. In SPA, $nextYellow(i, x)$ is a shorthand for the row number of the next yellow entry below $VUT[i, x]$. It is the number of the AL received after AL_i^x from view manager x. If AL_i^x is currently the last AL the merge process has received from view manager x, then $nextYellow(i, x) = 0$.

Algorithm 1: Simple Painting Algorithm
Initially, $WT_i = \emptyset$ for all i.
$\triangleright$ When the merge process receives REL_i
— Allocate a new row i in the VUT. $VUT[i,x]$ refers to U_i and $V_x \in VM$.
— For all $V_x \in REL_i$ set $VUT[i,x].color = white$; otherwise set $VUT[i,x].color = black$.
— For all $AL_i^x \in WT_i$, call $ProcessAction(AL_i^x)$.
$\triangleright$ When the merge process receives action list AL_i^x:
— Let $WT_i = WT_i \cup AL_i^x$.
— If REL_i has arrived, call $ProcessAction(AL_i^x)$.

Procedure $ProcessAction(AL_i^x)$
Let $VUT[i,x].color = yellow$, and call $ProcessRow(i)$.
End Procedure

Procedure $ProcessRow(i)$
Line 1: If $\exists x$, $VUT[i,x] = white$, return.
Line 2: If $\exists x$, $\exists i' < i$, $VUT[i,x] = yellow$ and $VUT[i',x] = yellow$, return.
Line 3: For any $x \in VM$, if $VUT[i,x].color = yellow$, then let $VUT[i,x].color = gray$.
Line 4: Apply all actions in WT_i as a single warehouse transaction [See Section 4.3].
Line 5: For all $VUT[i,x] = gray$, if $nextYellow(i,x) \neq 0$, then
call $ProcessRow(nextYellow(i,x))$.
Line 6: Purge row i from the VUT. Return.
End Procedure
End Algorithm 1

In $ProcessRow(i)$, Line 1 tests whether any action in this row has not yet arrived; if so, the corresponding view can not be brought to state i (i.e., other ALs in row i cannot be applied). Line 2 checks another condition that must be satisfied before we apply the ALs in row i: for each received AL_i^x, previous lists from VM_x must have been already applied. This ensures that lists from the same view manager are applied to the warehouse in the order they are generated by the manager. If row i passes both tests, actions in this row can be applied to views stored at the warehouse. Note that applying actions in one row may trigger the application of actions in later rows. Line 5 finds all such later rows.

Example 3: Simple Painting Algorithm

This example shows how SPA handles ALs and guarantees warehouse view maintenance with *complete* consistency. Suppose there are three views: $V_1 = R \bowtie S$, $V_2 = S \bowtie T$, and $V_3 = Q$. Notice that V_3 is disjoint with V_1 and V_2. Let there be three source updates: U_1 on S, U_2 on Q and U_3 on T. Assume the merge process receives $REL_1, AL_1^2, REL_2, REL_3, AL_2^3, AL_3^2, AL_1^1$, in that order. We show the changes of the VUT and explain them below.

Time t_0: REL_1 arrived

	V_1	V_2	V_3	WT_i
U_1	w	w	b	$\emptyset$

$\Longrightarrow$

Time t_1: AL_1^2 arrived

	V_1	V_2	V_3	WT_i
U_1	w	y	b	$\{AL_1^2\}$

$\Longrightarrow$

Time t_2: REL_2 arrived

	V_1	V_2	V_3	WT_i
U_1	w	y	b	$\{AL_1^2\}$
U_2	b	b	w	$\emptyset$

$\Longrightarrow$ Time t_3: REL_3 arrived

	V_1	V_2	V_3	WT_i
U_1	w	y	b	$\{AL_1^2\}$
U_2	b	b	w	$\emptyset$
U_3	b	w	b	$\emptyset$

$\Longrightarrow$ Time t_4: AL_2^3 arrived

	V_1	V_2	V_3	WT_i
U_1	w	y	b	$\{AL_1^2\}$
U_2	b	b	y	$\{AL_2^3\}$
U_3	b	w	b	$\emptyset$

$\Longrightarrow$ Time t_5: WT_2 applied

	V_1	V_2	V_3	WT_i
U_1	w	y	b	$\{AL_1^2\}$
U_2	b	b	g	$\emptyset$
U_3	b	w	b	$\emptyset$

$\Longrightarrow$ Time t_6: Row 2 purged

	V_1	V_2	V_3	WT_i
U_1	w	y	b	$\{AL_1^2\}$
U_3	b	w	b	$\emptyset$

$\Longrightarrow$ Time t_7: AL_3^2 arrived

	V_1	V_2	V_3	WT_i
U_1	w	y	b	$\{AL_1^2\}$
U_3	b	y	b	$\{AL_3^2\}$

$\Longrightarrow$ Time t_8: AL_1^1 arrived

	V_1	V_2	V_3	WT_i
U_1	y	y	b	$\{AL_1^1, AL_1^2\}$
U_3	b	y	b	$\{AL_3^2\}$

$\Longrightarrow$ Time t_9: WT_1 applied

	V_1	V_2	V_3	WT_i
U_1	g	g	b	$\emptyset$
U_3	b	y	b	$\{AL_3^2\}$

$\Longrightarrow$ Time t_{10}: Row 1 purged

	V_1	V_2	V_3	WT_i
U_3	b	y	b	$\{AL_3^2\}$

$\Longrightarrow$ Time t_{11}: WT_3 applied

	V_1	V_2	V_3	WT_i
U_3	b	g	b	$\emptyset$

We use time t_i to refer to the changes in the VUT (rather than to changes in the views). In this example, times t_0, t_2, and t_3 correspond to the receipt of update information and addition of one row to the VUT. Times t_1, t_4, t_7, and t_8 correspond to the receipt of ALs that are simply added to the proper WTs.

At time t_5, $WT_2 = \{AL_2^3\}$ is applied to V_3 because all expected ALs in row 2 have been received. Notice that at this time, actions in the first row have not yet been applied. However, AL_2^3 can be applied because the merge process knows from the VUT that all entries prior to row 2 in the third column of the VUT are *black*, meaning that the first update is irrelevant to V_3. At times t_9 and t_{11}, WT_1 and WT_3 are applied respectively.

At times t_6 and t_{10}, rows 2 and 1 are purged from the table, respectively. We also purge all corresponding ALs after they are applied to the warehouse views. Often, the warehouse only needs to work with a small VUT as a result of purging. Although theoretically, the total number of rows in the VUT could be as many as the total number of updates, the actual number is small in a system where no view manager is a bottleneck. When all ALs of all source updates received by the merge process have been processed, all VUT entries will be empty. □

4.3 Submitting view maintenance transactions to the warehouse

There is still one remaining problem with algorithm SPA as just presented. In Example 3, WT_1 is applied at time t_9 and WT_3 is applied at time t_{11}. When the transactions are submitted to the warehouse, it is possible that the warehouse DBMS will commit WT_3 before WT_1. If so, the state of view V_2 will be invalid because it will reflect update U_3 but not U_1.

Let $VS(WT_i)$ be the set of views that WT_i will update; it can be obtained directly from the VUT. We say that transaction WT_j *depends* on WT_i if $j > i$ and $VS(WT_j) \cap VS(WT_i) \neq \emptyset$. To satisfy MVC, two dependent warehouse transactions need to commit in the order in which they were submitted.

There are a few solutions to this problem; each may be appropriate in different scenarios. The most straightforward way is to execute the warehouse transactions sequentially — only submit one to the warehouse after the previously transaction has committed. This solution is good when the warehouse transactions are small and the transaction overheads are low. The merge process can also choose to only sequence dependent transactions instead of all transactions, that is, only delay a transaction when some transaction it depends on has not committed. If the warehouse DBMS can provide transaction dependency capabilities, an alternative solution is for the merge process to submit transactions with dependency information and let the warehouse DBMS handle the execution sequence.

When transaction overhead is high, the merge process can batch several WT_is and submit them to the warehouse as one batched warehouse transaction (BWT). A BWT contains all ALs in a set of WTs that are ready to be submitted to the warehouse. The merge process needs to make sure that if WT_j depends on WT_i, then all ALs in WT_i appear before all ALs in WT_j when they are batched together. A parallel DBMS at the warehouse may be able to execute some of the actions in a BWT in parallel, as long as the execution is equivalent to a serial one.

Dependencies exist and need to be handled between BWTs, just as for WTs. Therefore, on one hand, batching eliminates the dependency problems between WTs within a batch; on the other hand, batching may effectively create dependencies between WTs that were independent. For example, suppose the merge process is ready to send WT_1, WT_2, WT_3 in that order, and both WT_2 and WT_3 depend on WT_1. By batching WT_1, WT_2 into BWT_1, the merge process need not handle the dependencies between WT_1 and WT_2. On the other hand, now WT_3 depends on BWT_1 (which contains WT_1), so WT_3 cannot commit before BWT_1. Thus, now WT_3 and WT_2 cannot be executed in parallel, unless the warehouse can handle transaction dependencies. Choosing the right size and the right transactions to batch together are optimization problems that we do not address here. Finally, note that batching only yields strong consistency at the warehouse rather than complete consistency, because each BWT may advance the warehouse state by more than one.

4.4 Properties of Simple Painting Algorithm

Theorem 4.1 *The Simple Painting Algorithm is complete under MVC.*

Proof. An algorithm is complete if, when applied to an initially consistent warehouse state, it generates a complete warehouse state sequence.

Consider a set of f source updates that are processed by the view managers and coordinated by a merge process running SPA. The source updates are numbered by the integrator as $U_1, U_2 \dots U_f = SS$. SS is a serial source schedule. The merge process changes the warehouse state by sending warehouse transactions. Using SPA, a warehouse transaction WT_i corresponds to one row in the VUT, and thus corresponds to one source update. So the total number of WTs is f. Suppose the sequence of warehouse transactions executed is $W = WT_{i_1}, \dots, WT_{i_f}$, the corresponding warehouse state sequence is $W_{seq} = ws_{i_0}, ws_{i_1}, \dots, ws_{i_f}$, where ws_{i_0} is the initial warehouse state. (Note that the sequence $i_1, i_2, \dots$ may be different from the sequence $1, 2, \dots$ because the merge process can reorder actions.) Corresponding to W, we construct a source execution schedule $S = U_{i_1}, U_{i_2}, \ldots, U_{i_f}$ and source state sequence $S_{seq} = ss_{i_0}, ss_{i_1}, \ldots, ss_{i_f}$, where ss_{i_0} is the initial source state.

We prove the completeness of SPA in two steps. First, we show that S_{seq} is a *consistent* source state sequence. Then we prove that there exists a mapping m from W_{seq} to S_{seq}, where m satisfies all of the requirements of the mapping in the definition of completeness in Section 2.

We prove that S_{seq} is a consistent source state sequence by showing that schedules S and SS are equivalent. First, the two schedules involve the same set of updates. Second, let U_i and U_j be any two conflicting updates. U_i and U_j must be on the same base relation and hence affect the same view(s). If U_i precedes U_j in SS, then U_i must precede U_j in S, because WT_j will not be applied before WT_i (line 2 in $ProcessRow(i)$). Since the orders of all conflicting updates are the same in S and SS, the two schedules are equivalent[BHG87]. Therefore, S_{seq} is a consistent source state sequence.

We now define a mapping m from W_{seq} to S_{seq} as $m(ws_{i_k}) = ss_{i_k}$ for all $0 \leq k \leq f$. We show that m satisfies the mapping properties defined in Section 2. Initially, the warehouse state $ws_{i_0} \equiv ss_{i_0}$ by assumption. Because each view manager is complete and its actions applied in order, for each view V, the content of V at warehouse state ws_{i_k} is equivalent to $V(ss_{i_k})$. Therefore, we have $ws_{i_k} \equiv ss_{i_k}$ for all $0 \leq k \leq f$. It is easy to see that if $ws_i < ws_j$, then $m(ws_i) = ss_i < ss_j = m(ws_j)$. Also, for each source state ss_{i_k}, there exists ws_{i_k} such that $m(ws_{i_k}) = ss_{i_k}$. That is, the warehouse state sequence W_{seq} is complete. Since W_{seq} is any state sequence generated by a merge process running SPA, algorithm SPA is complete. $\square$

Another important property of the SPA algorithm is that it applies action lists *promptly*, i.e., it does not delay actions unnecessarily. Not all MVC complete algorithms have this property. For instance, we could devise an algorithm that waits until all actions about all source updates (U_1 to U_f) arrive, then applies $WT_1, \ldots WT_f$ to the warehouse in that order. This algorithm is also complete under MVC, but is clearly not a desirable one because it unnecessarily delays actions. From the way we constructed algorithm SPA, it is easy to see that it is *prompt* and that no other algorithm can deliver updates earlier than SPA without possibly causing inconsistencies.

5 Painting Algorithm

In this section we develop a merge algorithm for underlying view managers that are strongly consistent but not necessarily complete (e.g., Strobe view managers [ZGMW96]). We showed in [ZGMHW95], [ZGMW96] that strongly consistent view managers are usually more efficient and easier to implement than complete view managers in a distributed warehouse environment. The resulting Painting Algorithm (PA) is strongly consistent and prompt.

5.1 Algorithm PA

As mentioned in Section 3, a strongly consistent view manager sends a sequence of ALs: $AL^x_{i_1}, AL^x_{i_2}, \ldots, AL^x_{i_f}$ to the merge process. $AL^x_{i_j}$ brings view V_x from state i_{j-1} to state i_j. Since one AL can represent several intertwined source updates, SPA breaks down, as shown in the following example.

Example 4: Strongly consistent view management

As in Example 2, suppose there are three views $V_1 = R \bowtie S$, $V_2 = S \bowtie T \bowtie Q$, and $V_3 = Q$. Let there be three source updates: U_1 on S, U_2 on Q and U_3 on S. Assume the merge process first receives REL_i for $i = 1, 2, 3$. Then it receives AL^1_3 to update V_1 for both U_1 and U_3, i.e., there is no separate AL^1_1. At this point, SPA makes $VUT[3,1].color$ equal to yellow. If we do not make $VUT[1,1].color$ yellow too, then we will never apply row 1, so let us assume we do make $VUT[1,1].color$ yellow. Later, the merge process receives all other ALs corresponding to U_1 and U_2, as shown in the table entries below. At this time, SPA would apply rows 1 and 2 because all entries are either yellow or black. However, this would be incorrect: since AL^2_3 has not arrived yet, we cannot apply row 3, and since AL^1_3 and AL^1_1 are combined, we cannot apply row 1 either. And if we do not apply row 1, then we cannot apply row 2. Thus, we see that with intertwined updates, the consistency algorithm must be more careful.

VUT:

	$V_1(R,S)$	$V_2(S,T,Q)$	$V_3(Q)$	WT_i
$U_1(S)$	y	y	b	$\{AL^2_1\}$
$U_2(Q)$	b	y	y	$\{AL^2_2, AL^3_2\}$
$U_3(S)$	y	w	b	$\{AL^1_3\}$

□

There are two major differences between PA and SPA. First, in PA, receiving an action list in row i may cause actions in the previous rows to be applied. In the above example, the receipt of AL^2_3 will cause actions in the previous two rows to be applied. Second, it is not guaranteed that each view will go through each state, i.e., PA is strongly consistent but not complete. In Example 4, all three views will be brought into state 3 directly, skipping states 1 and 2. The strongly consistent property of the underlying view manager algorithms makes it impossible to develop an MVC algorithm that guarantees completeness.

We also use the $ViewUpdateTable$ (VUT) in the Painting Algorithm. In addition to the $color$ field, we add a second field $VUT[i,j].state$ to indicate the next state of a view after applying the related action lists. In the above Example 4, when the merge process receives AL^1_3, it fills in the state of entry $VUT[1,1].state = 3$. This shows that U_1 is intertwined with U_3. At this time, even if all other actions corresponding to U_1 have been received, they cannot be applied unless all actions in WT_3 can be applied as well. Procedure $ProcessRow(i)$ now has a boolean return value, indicating whether ALs in row i and all related rows can be applied to warehouse views. An integer list $ApplyRows$ is used to remember the possible rows whose actions will be applied.

Algorithm 2: Painting Algorithm
The main body of PA is the same as SPA, except that when REL_i is received, the merge process also sets $VUT[i,x].state = 0$ for all $V_x \in VM$. The two procedures are defined as follows.

Procedure $ProcessAction(AL_i^x)$
For all $i' \leq i$ such that $VUT[i',x].color = white$, let $VUT[i',x].color = yellow$ and $VUT[i',x].state = i$.
Let $ApplyRows = \emptyset$ and call $ProcessRow(i)$.
End Procedure

Procedure $ProcessRow(i)$: boolean
Line 1: If $i \in ApplyRows$, return $true$.
Line 2: If $\exists x,\ VUT[i,x].color = white$, then return false.
Line 3: Add row i to $ApplyRows$.
Line 4: For all x such that $VUT[i,x].color = yellow$, for all $i' < i$ such that $VUT[i',x].color = yellow$, call $ProcessRow(i')$. If any of the return values is $false$, then return $false$.
Line 5: For all x such that $VUT[i,x].state > i$, call $ProcessRow(VUT[i,x].state)$. If any of the return values is $false$, then return $false$.
Line 6: For each $j \in ApplyRows$, for each $x \in VM$, if $VUT[j,x].color = yellow$, then let $VUT[j,x].color = gray$.
Line 7: For all rows $j \in ApplyRows$, apply all actions in WT_j together as a single warehouse transaction.
Line 8: Let $ApplyRows = \emptyset$.
Line 9: For all i,x such that $VUT[i,x] = gray$, if $nextYellow(i,x) \neq 0$, then call $ProcessRow(nextYellow(i,x))$.
Line 10: Purge rows whose entries are all either $black$ or $gray$ from the VUT, and return $true$.
End Procedure
End Algorithm 2

Procedure $ProcessRow(i)$ may recursively call itself to process rows both prior to and after i. It is therefore possible that $ProcessRow(i)$ is called again for the same i. Line 1 and Line 3 in the procedure make sure that the recursive calls terminate. For example, at time t_6 in Example 5 below, $ProcessRow(3)$ calls $ProcessRow(2)$ which again calls $ProcessRow(3)$. The last call realizes that it is a repeat call because 3 is already in $ApplyRows$, and returns $true$ in Line 1. If $ProcessRow(i)$ fails, actions in row i will not be applied and $ApplyRows$ will be set to empty before the next time the procedure is called.

As in $ProcessRow(i)$ in SPA, Lines 2 and 4 in the procedure check whether any actions in row i have not yet arrived or any actions in previous, relevant, rows can not yet be applied. Line 5 says that if some entries in row i must be brought to a later state j directly, then all actions in row j must be applied together with the actions in row i. If any actions in row j cannot be applied, neither can the actions in row i. In Line 6, all actions in all rows in $ApplyRows$ are applied to the warehouse in a single transaction. After the actions are applied, Line 9 checks whether any actions in later rows can be applied as a consequence of applying actions related to row i.

Transaction dependency problems also arise in a merge process running PA, and the solutions we described for SPA can be used for PA as well.

Example 5: Painting algorithm

This example shows how the Painting Algorithm works. We have the same three views as in the last example. There are three source updates: U_1 on S, U_2 on Q, and U_3 on Q. Assume the merge process receives the following information in order: $REL_1, REL_2, REL_3, AL_1^2, AL_3^2, AL_2^3$, AL_1^1, AL_3^3. Here we show the changes in the VUT entries.

Time t_0: RELs received

	$V_1(R,S)$	$V_2(S,T,Q)$	$V_3(Q)$	WT_i
$U_1(S)$	(w,0)	(w,0)	(b,0)	$\emptyset$
$U_2(Q)$	(b,0)	(w,0)	(w,0)	$\emptyset$
$U_3(Q)$	(b,0)	(w,0)	(w,0)	$\emptyset$

$\Longrightarrow$ Time t_1: AL_1^2 arrived

	V_1	V_2	V_3	WT_i
U_1	(w,0)	(y,1)	(b,0)	$\{ AL_1^2 \}$
U_2	(b,0)	(w,0)	(w,0)	$\emptyset$
U_3	(b,0)	(w,0)	(w,0)	$\emptyset$

$\Longrightarrow$ Time t_2: AL_3^2 arrived

	V_1	V_2	V_3	WT_i
U_1	(w,0)	(y,1)	(b,0)	$\{ AL_1^2 \}$
U_2	(b,0)	(y,3)	(w,0)	$\emptyset$
U_3	(b,0)	(y,3)	(w,0)	$\{ AL_3^2 \}$

$\Longrightarrow$ Time t_3: AL_2^3 arrived

	V_1	V_2	V_3	WT_i
U_1	(w,0)	(y,1)	(b,0)	$\{ AL_1^2 \}$
U_2	(b,0)	(y,3)	(y,2)	$\{ AL_2^3 \}$
U_3	(b,0)	(y,3)	(w,0)	$\{ AL_3^2 \}$

$\Longrightarrow$ Time t_4: AL_1^1 arrived

	V_1	V_2	V_3	WT_i
U_1	(y,1)	(y,1)	(b,0)	$\{ AL_1^2, AL_1^1 \}$
U_2	(b,0)	(y,3)	(y,2)	$\{ AL_2^3 \}$
U_3	(b,0)	(y,3)	(w,0)	$\{ AL_3^2 \}$

$\Longrightarrow$ Time t_5: Row 1 applied

	V_1	V_2	V_3	WT_i
U_1	(g,1)	(g,1)	(b,0)	$\emptyset$
U_2	(b,0)	(y,3)	(y,0)	$\{ AL_2^3 \}$
U_3	(b,0)	(y,3)	(w,0)	$\{ AL_3^2 \}$

$\Longrightarrow$ Time t_6: AL_3^3 arrived

	V_1	V_2	V_3	WT_i
U_2	(b,0)	(y,3)	(y,0)	$\{ AL_2^3 \}$
U_3	(b,0)	(y,3)	(y,3)	$\{ AL_3^2, AL_3^3 \}$

$\Longrightarrow$ Time t_7: Rows 2,3 applied

	V_1	V_2	V_3	WT_i
U_2	(b,0)	(g,3)	(g,2)	$\emptyset$
U_3	(b,0)	(g,3)	(g,3)	$\emptyset$

The information about all three updates has been received and the VUT is initialized at time t_0. At time t_1, AL_1^2 is received. Since $VUT[1,1].color = white$, $ProcessRow(1)$ returns $false$. No views can be updated at this time. At time t_2, $ProcessRow(3)$ returns $false$ and no views can be updated. At time t_3, in order to evaluate $ProcessRow(2)$, the warehouse needs to evaluate $ProcessRow(1)$(Line 4 in the procedure). The latter returns $false$. At time t_4, $ProcessRow(1) = true$ so WT_1 is applied. Line 9 causes $ProcessRow(3)$ to be called but it returns $false$. Row 1 is purged from the table. At time t_6, the merge process invokes $ProcessRow(3)$ and then $ProcessRow(2)$(Line 4). The latter returns $true$. Notice that here it recursively calls $ProcessRow(3)$ again but gets a $true$ return value since row 3 has already been added to $ApplyRows$. Actions in both WT_2 and WT_3 are now applied together as a single transaction. □

The Painting algorithm also has properties that guarantee its correctness and efficiency.

Theorem 5.1 *The Painting Algorithm is strongly consistent under MVC.*

The proof that PA guarantees strong consistency is similar to the proof that SPA is complete, and is omitted here. Similar to SPA, PA also guarantees promptness.

6 Distributing the merge process

The merge process (*MP*) in the previous sections may become a bottleneck as the system scales up — when the number of views defined in the system increases, or when the base relation update frequency increases. In this section we propose techniques that distribute the merge computations. To determine which proposed technique is best, i.e., to determine whether the merging should be distributed and how to distribute it, is a sophisticated optimization problem. The factors that affect the decision include the number of views in the system, the update frequency and update patterns, the cost of message passing, the cost of temporarily storing the action lists, and the overhead of running a merge process. In the following discussion we give examples to show different merging alternatives.

Basically, a merge process can be split into several ones. The most straightforward way of splitting is to first partition view managers into groups such that base relations used in the views of one group are disjoint with those used in the views of other groups. Then each group of views is assigned one merge process. Example 6 illustrates the partitioning and gives a graphical representation of the partitioned system.

Example 6: Partitioning view managers into disjoint groups

Consider a system with the three views in Example 3: $V_1 = R \bowtie S$, $V_2 = S \bowtie T$ and $V_3 = Q$. Actions related to updates on S need to be coordinated between VM_1 and VM_2, but actions related to Q only involve VM_3. Thus, the three view managers can be partitioned into two groups: $G_1 = \{VM_1, VM_2\}$ and $G_2 = \{VM_3\}$. Figure 3 shows the partitioned system using a structure called the *View Merge Tree* (*VMT*). Each node in a *VMT* represents either a view manager, a merge process, or the warehouse; each edge indicates where to forward action lists from the child node. The warehouse is the root node of the tree. In Figure 3, each group of view managers uses one merge process to coordinate the actions for the group.

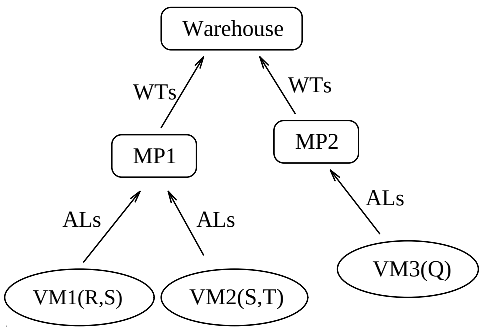

Figure 3: *VMT* after partitioning view managers into groups

Each *MP* in the generated *VMT* collects *AL*s from its children as if they were the only view managers in the system. Then the *MP* runs SPA or PA according to the type of underlying view managers, just as in the previous sections. Later it forwards the *AL*s to the warehouse in a warehouse transaction. □

One advantage of the partitioning is that action lists are coordinated only when necessary — when they are from view managers that share common base data. Another advantage is that different *MP*s can run different merge algorithms.

Although all actions from view managers in the same group need to be coordinated eventually, a *MP* may be further distributed into a hierarchy of *MP*s when necessary. In this scenario, a set of ALs may be first combined into one AL, then the combined AL is merged with other ALs. Figure 4 shows two possible distributions of the merge processes for the three views: $V_1 = R \bowtie S, V_2 = S \bowtie T$, and $V_3 = T \bowtie Q$.

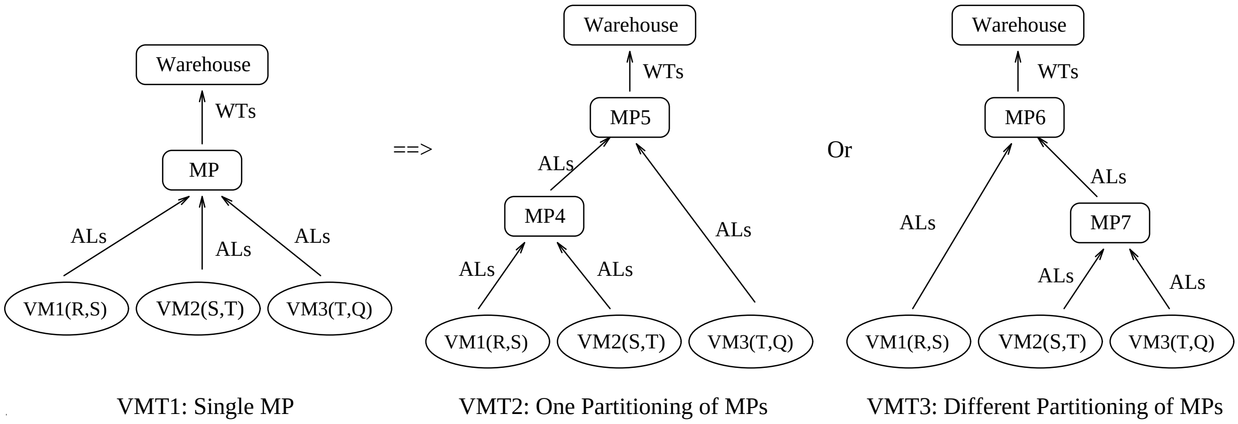

Figure 4: Distributing a merge process

In VMT_2 and VMT_3 in Figure 4, each merge process functions essentially the same as before, collecting and coordinating action lists. However, one *MP* node may be a child of another *MP* node. The child *MP*, for example, MP_4 in VMT_2 will modify its algorithm to output ALs instead of warehouse transactions(WTs). Since a *MP* may send out ALs just like a view manager, we assign each *MP* an unique id that distinguishes it from other view managers and *MP*s. In the example, MP_4 will combine all actions from VM_1 and VM_2 using SPA or PA. When it is ready to send WT_i, it will instead combine all ALs in WT_i into an action list AL_i^4 and forward AL_i^4 to MP_5. From the point of view of MP_5, MP_4 behaves as if it were an ordinary view manager "VM_4". MP_5 coordinates "VM_4" with actions from VM_3. Any *MP* whose direct parent is the warehouse, e.g., MP_5, still sends transactions to the warehouse as before.

When distributing the merging, it is important to find a good *VMT* for the system. First, this *VMT* should be correct, meaning that if two views V_i and V_j share a base relation, then ALs from VM_i and VM_j must be merged by some *MP* before being delivered to the warehouse. Then, a chosen *VMT* should yield good performance. The three *VMT*s in Figure 4 may illustrate the variation in system performance when using different *VMT*s. If the number of views is small and the traffic caused by passing ALs is low, a "flat" *VMT* such as VMT_1 works efficiently. In reality, a *MP* needs to be distributed only when the number of views in the system is so large that the *MP* can not handle all actions quickly enough to keep on with the incoming flow. When the *MP* must be distributed e.g., if the *MP* in Figure 4 can not handle ALs from more than two view managers, then for certain scenarios VMT_2 may be a better choice than VMT_3, and vice versa. For example, if the update frequency on relation Q is much higher than update frequencies on any other base relations, then VMT_2 is better because Q updates only need to go through one level of merging.

Finding an optimal or a good *VMT* is a complicated optimization problem. The first step involves deciding what comprises a "good" system: one with the minimum number of messages exchanged? one with the minimum total data transfer? or one with the minimum average time before updates are reflected in views? Next, according to the actual base relation update pattern and frequency, the load limit of a *MP* must be determined, i.e., what is the proper fan-in of a *MP* node in the *VMT* before it saturates. Then, heuristics may be used to pick a good *VMT* for the system. Since the number of all *VMT*s is exponential in the number of view managers, it may not be feasible to test all possibilities. Due to space constraints, we will not get into optimization details here.

Finally, a merge process is a logical concept. A *MP* can reside anywhere physically: with the warehouse, with a view manager or as a separate process. For example, it is possible to combine one merge process with each view manager, such that each *VM*/*MP* pair does two things: compute the changes to this view, and merge its own actions with actions it obtains from other view managers. Combining a merge process together with a view manager may save some communication, for example, some ALs may be directly passed to the warehouse without forwarding stop.

7 Algorithm extensions

SPA and PA can be easily adapted to handle different assumptions from the ones we have made up to now. We now discuss a few extensions of the algorithms.

7.1 Source transactions and multiple sources

A source transaction may update more than one base relation that belongs to more than one view. As a result, even if two views do not share common base data, they may still need to be maintained consistently. For example, if we have $V_1 = R$ and $V_2 = S$, and a source transaction inserts one tuple into R and one tuple into S, then the new tuples should appear in both views at the same time. In this scenario, MVC means that if sources have transactions (local or global) involving more than one update, then all updates in a transaction should be reflected in either all views or none.

We assume that all source transactions are serializable and can be sequenced as $T_1, T_2, \ldots T_n$. The execution of each source transaction thus corresponds to a consistent source state. That is, the source state after executing T_i is ss_i. In other words, all base data after the commit of T_i but before T_{i+1} comprise source state ss_i. When there is more than one source, a source state can be seen as a vector with one element for each source, representing the state of each source at a given instant in time. A more detailed description of possible transaction scenarios is given in [ZGMW96]. We still define $V(ss_i)$ as the result of evaluating the definition of V at source state ss_i.

After defining source states, the definition of view consistency and MVC are the same as in Section 2. SPA and PA are unchanged except that each occurrence of "update" in the algorithm need to be replaced by "transaction". The merge process receives information about transactions instead of updates, and one row in the VUT represents one transaction instead of one update. The relevant views for an update is re-defined to be the relevant view set for a source transaction: REL_i now includes all views that are affected by any update performed by transaction T_i.

7.2 Other types of view managers

Although we have focused on the most commonly used two types of view managers in our previous discussion, the framework we developed as well as the idea of algorithms can be adapted easily to handle other type of view managers. We give a few examples here:

- A view manager may do periodical refreshing instead of incremental maintenance. Such a view manager will appear to the MP in our system as if it were an ordinary strongly consistent view manager. The action lists from this view manager will tell the warehouse to delete the entire old view and insert tuples of the new view. The merge process still coordinates the ALs and passes them along. If the amount of data passing from the view manager to the warehouse is large, the MP can be modified to coordinate transaction commit only, instead of handling all data transfer.

- A view manager may only guarantee the *convergence* of the view it manages. That is, it only guarantees the eventual correctness of the view but not the correctness of intermediate view states. Then the MP can just pass along all ALs it received, and also guarantees the *convergence* of the warehouse views. That is, all warehouse views are consistent eventually, although some of them may go through inconsistent intermediate states.

- A view manager may be *complete-N*, that is, it may process N source updates at a time and maintain the view consistently after every N updates. In this scenario, the MP can use an algorithm that is similar to SPA, but instead it collects all ALs corresponding to every N updates, then forwards them to the warehouse. The warehouse view maintenance is *complete-N* as well.

When there is a combination of different types of view managers in the system, it is always possible to use the merge algorithm corresponding to the view manager guaranteeing the weakest level of consistency. For example, if there are both complete and strongly consistent view managers in a system, a MP can always use PA to guarantee strong consistency. There is room for other optimizations when there are mixed types of view managers, but we will not discuss them here.

7.3 View maintenance based on other views

Sometimes changes to one view can be evaluated using changes to another view. For example, in the Squirrel system [HZ96] a view can be defined on top of other views. Sometimes changes to two views can be evaluated using common information. For example, two views may share the same sub-view. Optimizing view maintenance by using other view's contents is an interesting problem but not the focus of this paper. The architecture we proposed in Section 1 could be extended easily to allow one view manager to handle a set of closely related views. From the point of view of the *MP*, a set of views to be managed by one view manager can be treated as "one view" which is defined over the union of all base relations of the set of views.

8 Conclusion

In this paper we presented algorithms to enforce multiple view consistency in a warehousing environment. Inter-view consistency ensures that warehouse applications can correctly access data from multiple views, and is as important as consistency is in conventional database systems. As far as we know, this paper is the first to address the problem of efficiently enforcing consistency across views. Our SPA and PA algorithms make it possible to distribute the incremental view maintenance work over multiple concurrently-executing view managers. Our merge process can then coordinate results from different managers and guarantee data consistency. Further, we showed that the merging work can be itself distributed over a collection of merge processes. Each process can perform part of the merge in a pre-defined but flexible manner.

The Simple Painting Algorithm and Painting Algorithm will be implemented in the WHIPS [WGL+96] system at Stanford. We plan to study the performance of the two algorithms and evaluate their performance. In particular, we plan to investigate the effect of the merging process on view freshness (recall that the merging delays the application of some *AL*s to the warehouse views), and under which update load the merge process becomes a bottleneck for the system.

Acknowledgments. We would like to thank Wilburt Labio, Dallan Quass and other members of the Stanford Database group for suggestions and discussions on the topic of this paper.

References

[BCP96] E. Baralis, S. Ceri, and S Paraboschi. Conservative timestamp revised for materialized view maintenance in a data warehouse. In *Proceedings of the Workshop on Materialized Views, Techniques and Applications*, pages 1–9, Montreal, Canada, June 1996.

[BGMS92] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of multidatabase transaction management. *VLDB Journal*, 1(2):181–239, October 1992.

[BHG87] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, Massachusetts, 1987.

[BLT86] J.A. Blakeley, P.-A. Larson, and F.W. Tompa. Efficiently updating materialized views. In *Proceedings of ACM SIGMOD Conference*, pages 61–71, Washington, D.C., June 1986.

[GJM96] A. Gupta, H. Jagadish, and I. Mumick. Data integration using self-maintainable views. In *Proceedings of the 1996 EDBT*, 1996.

[GL95] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *Proceedings of ACM SIGMOD Conference*, pages 328–339, San Jose, California, May 1995.

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

[HZ96] R. Hull and G. Zhou. A framework for supporting data integration using the materialized and virtual approaches. In *Proceedings of ACM SIGMOD Conference*, pages 481–492, Montreal, Quebec, Canada, June 1996.

[JBL86] N. Coburn J.A. Blakeley and P.-A. Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. In *Proceedings of the Twelfth VLDB Conference*, pages 457–466, Kyoto, Japan, August 1986.

[LMR90] W. Litwin, L. Mark, and N. Roussopoulos. Interoperability of multiple autonomous databases. *ACM Computing Surveys*, 22(3):267–293, September 1990.

[LQA96] W. Labio, D. Quass, and B. Adelberg. Physical database design for data warehousing. Technical report, Dept. of Computer Science, Stanford University, 1996. Available by anonymous ftp to `db.stanford.edu` in file `pub/labio/1996/view.ps`.

[LW95] D. Lomet and J. Widom, editors. *Special Issue on Materialized Views and Data Warehousing,* IEEE Data Engineering Bulletin 18(2), June 1995.

[QGMW96] D. Quass, A. Gupta, I. Mumick, and J. Widom. Making views self-maintainable for data warehousing. In *Proceedings of the Fourth PDIS Conference*, Miami Beach, Florida, December 1996.

[RSS96] K. A. Ross, D. Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. In *Proceedings of ACM SIGMOD Conference*, pages 447–458, Montreal, Quebec, Canada, June 1996.

[SJ96] M. Staudt and M. Jarke. Incremental maintenance of externally materialized views. In *Proceedings of the 22ndVLDB Conference*, Bombay, India, September 1996.

[WGL+96] J.L. Wiener, H. Gupta, W.J. Labio, Y. Zhuge, H. Garcia-Molina, and J. Widom. A system prototype for warehouse view maintenance. In *Proceedings of the Workshop on Materialized Views, Techniques and Applications*, pages 26–33, Montreal, Canada, June 1996.

[Wie92] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):38–49, March 1992.

[ZGMHW95] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proceedings of ACM SIGMOD Conference*, pages 316–327, San Jose, California, May 1995.

[ZGMW96] Y. Zhuge, H. Garcia-Molina, and J.L. Wiener. The Strobe algorithms for multi-source warehouse consistency. In *Proceedings of the Fourth PDIS Conference*, Miami Beach, Florida, December 1996.