Multiversion Based View Maintenance Over Distributed Data Sources

SONGTING CHEN, BIN LIU and ELKE A. RUNDENSTEINER
Worcester Polytechnic Institute

Materialized views can be maintained by submitting maintenance queries to the data sources. However, the query results may be erroneous due to concurrent source updates. State-of-the-art maintenance strategies typically apply compensations to resolve such conflicts and assume all source schemata remain stable over time. In a loosely-coupled dynamic environment, the sources may autonomously change not only their data but also their schema or semantics. Consequently, either the maintenance or the compensation queries may be broken. Unlike compensation-based approaches found in the literature, we instead model the complete materialized view maintenance process as a view maintenance transaction (VM_Transaction). This way, the anomaly problem can be rephrased as the serializability of VM_Transactions. To achieve VM_Transaction serializability, we propose a multiversion concurrency control algorithm, called TxnWrap, which is shown to be the appropriate design for loosely-coupled environments with autonomous data sources. TxnWrap is complementary to the maintenance algorithms proposed in the literature, since it removes concurrency issues from consideration allowing the designer to focus on the maintenance logic. We show several optimizations of TxnWrap, in particular, (1) space optimizations on versioned data materialization and (2) parallel maintenance scheduling. With these optimizations, TxnWrap even outperforms state-of-the-art view maintenance solutions in terms of refresh time. Further, several design choices of TxnWrap are studied each having its respective advantages for certain environmental settings. A correctness proof based on transaction theory for TxnWrap is also provided. Lastly, we have implemented TxnWrap. The experimental results confirm that TxnWrap achieves predictable performance under a varying rate of concurrency.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*Distributed databases*; *Relational databases*; *Concurrency*

General Terms: Theory, Algorithms, Experimentation

Additional Key Words and Phrases: View Maintenance, Transaction Processing

1. INTRODUCTION

1.1 Materialized Views and Their Environment

Materialized views [Gupta and Mumick 1995; Agrawal et al. 1997] built by gathering data from possibly distributed data sources and integrating it into one repository

This work was supported in part by several grants from NSF, namely, the NSF NYI grant IRI 97–96264, the NSF CISE Instrumentation grant IRIS 97–29878, and the NSF grant IIS 9988776.
Author's address: Department of Computer Science, Worcester Polytechnic Institute, Worcester, MA, 01609, USA; email: (chenst | binliu | rundenst)@cs.wpi.edu.
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 2004 ACM 0362-5915/2004/0300-00000 $5.00

customized to users' needs are a well recognized technology for data integration, e-business, and data warehousing. One important task of a view manager is to maintain the materialized view upon source changes, since frequent data updates are common for many applications, such as stock trading or telephone call recording.

In dynamic environments like the WWW, the data sources may change their schema, semantics as well as their query capabilities. A schema change could occur for numerous reasons during the software life-cycle, including design errors, schema redesign during the early stages of database deployment, the addition of new functionalities and even new developments in the application domain, such as new tax laws or Y2K problems. Even in fairly standard business applications, rapid schema changes have been observed. In [Marche 1993], significant changes (about 59% of attributes on average) were reported for seven different applications ranging from project tracking, sales management to government administration. A similar report can also be found in [Sjoberg 1993]. Second, the emerging schema mapping techniques [Miller et al. 2000; Madhavan et al. 2001] are critical for information integration over heterogeneous data sources, which aim at mapping between heterogeneous sources are semi-automatic and depend on domain knowledge. As a result, a good mapping is thus hard to find and may evolve over time [Lee et al. 2002; Velegrakis et al. 2003].

Moreover, data sources in a distributed environment are typically owned by different information providers and function independently from one another. The relationship between materialized views and such autonomous data sources hence must be loosely-coupled [Zhuge et al. 1995]. That is, the source updates are committed without any concern of how and when the view manager will incorporate them into the view [Widom 1995; Lee et al. 2002]. This causes problems which we called *maintenance anomalies*. These are the issues we now address in this work.

1.2 Motivating Example of the Maintenance Anomaly Problems

We will now illustrate the anomaly problems via motivating examples. We first distinguish between three main maintenance tasks, namely, *view maintenance*, *view synchronization* and *view adaptation*. *View maintenance* [Zhuge et al. 1995; Agrawal et al. 1997; Salem et al. 2000] maintains the materialized view extent under source data updates. In contrast, *view synchronization* [Nica et al. 1998; Lee et al. 2002] aims at rewriting the view definition when the source schema has been changed. Thereafter, *view adaptation* [Gupta et al. 1995; Nica and Rundensteiner 1999] incrementally adapts the view extent again to match the newly changed view definition.

If there is no concurrency among source updates, namely, the current view maintenance process completes before the next source update occurs, then view maintenance incorporates each source *data update* while view synchronization and view adaptation together incorporate the source *schema change* into the view. However, the data sources are autonomous and may undergo changes concurrently, thus causing the maintenance anomaly problems illustrated below.

Example 1. *Assume we have four data sources (DS) as shown in Figure 1.*

The materialized view Asia-Customer is defined by the SQL query in Equation (1). Assume the data update " ΔC = INSERT INTO Customer VALUES

DS_1: Customer(Name, Address, Phone): Customer Info.
DS_2: Tour(TourID, TourName, Age, Type, NoDays): Tour. Info.
DS_3: Participant(Participant, TourID, StartDate, Loc): Participant. Info.
DS_4: FlightRes(Name, Age, FlightNo, Dest): Reservation. Info.

Fig. 1. Description of Data Sources.

('Ben','MA',213)" has happened at DS_1. In order to determine the delta effect on the view extent, this now requires us to send the **view maintenance query** *Q [Zhuge et al. 1995] defined in Equation (2) down to the FlightRes relation at DS_4.*

```
CREATE VIEW Asia − Customer AS
SELECT      C.Name, F.Age,
            F.FlightNo, F.Dest
FROM        Customer C, FlightRes F          (1)
WHERE       C.Name = F.Name
            AND F.Dest = 'Asia'
```

```
SELECT  'Ben' as Name, F.Age,
        F.FlightNo, F.Dest
FROM    FlightRes F                          (2)
WHERE   F.Name = 'Ben'
        AND F.Dest = 'Asia'
```

We distinguish between two kinds of anomaly problems that may arise:

—**Data Update Anomaly***: If during the transfer time of the query Q to the relation FlightRes in the DS_4, FlightRes has already committed a new* **data update** *"ΔF = INSERT INTO FlightRes VALUES ('Ben', 18,'AA43','Asia')". This new tuple would also be included in the join result of Q and the extra tuple ('Ben', 18, 'AA43','Asia') would be inserted into the view. However, later when the materialized view manager starts processing the ΔF, the same tuple would be inserted into the view again. A data update anomaly appears. Similarly, a delete operation could cause anomaly problems.*

—**Schema Change Anomaly***: If during the transfer time of the query Q to DS_4, the FlightRes relation in DS_4 has a* **schema change***, e.g., the attribute FlightRes.Age is dropped, then the query Q faces a schema conflict. In this case, the selected attribute Age required by the maintenance query Q is no longer available in the source. Hence the query Q cannot be processed by DS_4. We then say that the query Q is broken. That is, a schema change anomaly appears.*

As illustrated by the example above, sources being autonomous may commit updates that are concurrent with and hence may conflict with the view maintenance process. A concurrent data update may result in an *incorrect query result* while a concurrent schema change may result in a *broken query* that cannot be processed by the respective data source.

1.3 Contributions of this Work

A preliminary version of this paper focusing on the TxnWrap model itself has been published in the *21*st International Conference on Conceptual Modeling [Chen et al. 2002]. Our contributions there include:

(1) We illustrate that the anomaly problems as described in Section 1.2 can be reformulated as a transaction problem by modeling the complete maintenance process using a transaction model, called a *VM_Transaction*. Based on our VM_Transaction model, we demonstrate that the view maintenance anomaly problems can be mapped to the problem of the serializability of VM_Transactions.

(2) We then propose a multiversion concurrency control algorithm, called *TxnWrap*, to achieve such serializability that is appropriate for the view maintenance in a distributed environment with autonomous data sources.

(3) To the best of our knowledge, TxnWrap is the first work to integrate the existing materialized view maintenance works in the literature such as view maintenance, view synchronization and view adaptation algorithms within the context of a sound theory, the serializability theory, to remove concurrency concerns from their maintenance logic.

(4) We have implemented the TxnWrap solution and integrated it into a fully functioning view maintenance prototype system. Experiments comparing maintenance performance with and without TxnWrap enabled confirm that TxnWrap avoids the maintenance anomaly problems and achieves stable performance.

In addition, in this extended journal paper we now contain the following:

(1) We propose a new multiversion management strategy. Instead of using a global identifier as both a version number and its corresponding VM_Transaction identifier, we now propose to separate identifier design into two concepts. That is, we introduce a *local identifier* within each data source's scope for version management and a *global identifier* for VM_Transaction management where the global identifier is composed out of a vector of local identifiers. This avoids the coordination cost for generating global identifiers in distributed environments.

(2) We analyze the design choices for TxnWrap version placement. Three different wrapper design choices are discussed and their trade-offs, such as space costs and maintenance costs, are evaluated experimentally.

(3) We illustrate that the development of a sound transactional foundation for view maintenance offers the advantage of the ability to easily add additional services to the system and assure their correctness. In particular, we show how TxnWrap with minimal effort can be optimized by applying a parallel instead of sequential maintenance scheduler. This way, we achieve a better maintenance performance. An initial version of parallel view maintenance work has been published in $DaWak'$2002 [Liu et al. 2002a]. We now provide an extensive performance study of the parallel feature. Furthermore, we also present a performance study that demonstrates how the existing self-maintenance approach, such as [Quass et al. 1996] can benefit from our TxnWrap solution.

(4) We demonstrate that our TxnWrap solution can be easily extended to efficiently handle the scenarios when there are multiple tables in one data source, a rather important practical requirement not supported by previous work such as SWEEP [Agrawal et al. 1997].

(5) We introduce several space optimization strategies for versioned data materialization. Static optimization strategy is to analyze the view definition and the existing constraints to reduce the wrapper space during version creation. Dynamic optimization is a runtime strategy that aims to reduce the intermediate wrapper version space on the fly. We also conduct an experimental study to measure the impact that these space optimization strategies have on maintenance performance .

(6) Finally, a correctness proof for both sequential and parallel TxnWrap is provided utilizing concepts from transactional theory [Bernstein et al. 1987]. These correctness proofs, primarily due to taking a transactional approach towards solving this problem, are missing from most existing solutions in the literature that typically offer (hard-coded) algorithmic approaches.

The outline of the rest of the paper is as follows. The transaction model called VM_Transaction is introduced in Section 2. A multiversion concurrency control algorithm to handle the conflicts between VM_Transactions is proposed in Section 3. Section 4 discusses the architectural solutions for versioned data materialization. Wrapper optimization strategies are discussed in Section 5. Extensions of TxnWrap, in particular, parallel processing of VM_Transactions is introduced in Section 6. Section 7 presents experimental results. In Section 8 we review the related work while Section 9 concludes the paper.

2. TRANSACTIONAL MODELING OF VIEW MAINTENANCE

We first analyze the view maintenance anomaly problems. Typically, the maintenance process involves reads from individual data sources to calculate and ultimately refresh the view extent. On the other hand, the data source update process may continue to commit various changes throughout the same period. Due to the autonomy of the data sources, i.e., the loosely-coupled nature of the environment, concurrency between these two processes may occur. This maintenance anomaly problem is similar to the following process in a traditional centralized DBMS: assume a join is defined upon two source relations. Without any appropriate concurrency control, concurrent data updates or schema changes of other write transactions on these two relations during the computation of the join process would cause a concurrency problem. This is clearly similar to the view maintenance anomaly problem.

A traditional DBMS deals with this kind of problem (1) by applying a transaction model to encapsulate all operations that need to be executed atomically into one transaction, such as the two reads of the two relations and the full join computation, and (2) by using concurrency control strategies to guarantee the ACID properties of each transaction [Gray and Reuter 1992]. The ACID properties assure a consistent view of all data inside each transaction and hence solve the problem.

Similarly, in our view maintenance context, the maintenance process can be viewed as a series of read operations over sources that should be executed atomically, while the source updates are independent write operations. There are read and write conflicts since the source writes are autonomous. The above discussion leads us to the idea of applying a transactional model to solve the anomaly problem, as described in detail below.

2.1 Transactions in a Materialized View Maintenance Environment

In a materialized view maintenance environment, each view maintenance process is composed of the following two types of transactions:

(1) A *source update transaction* at some data source is committed, denoted as "$T_s = w(DS_i)c(s)$" where $w(DS_i)$ means the write operation on DS_i and i is the index of the data source, $c(s)$ means the commit operation.

(2) A *view maintenance transaction* that computes the delta effects caused by source updates in order to refresh the view extent. During this period, the maintenance transaction reads the view definition to generate the maintenance queries to be sent to the different sources and to compute the maintenance result. After that, it refreshes the view extent. We denote this as "$T_v = r(VD)r(DS_1)r(DS_2)\ldots r(DS_n)w(v)c(v)$", where $r(VD)$ denotes the generation of the source-specific maintenance queries based on the view definition VD, $r(DS_i)$ denotes reading from DS_i, $w(v)$ denotes refreshing the view extent and $c(v)$ denotes the commit process [1].

However, these two types of transactions are not completely independent. As suggested above, the relationship is that each source update transaction would trigger a corresponding view maintenance transaction. Furthermore, this view maintenance transaction is required to see a consistent state of all sources. That is, a view maintenance transaction cannot conflict with other source update transactions. Clearly, some coordination between these two transactions is needed, as otherwise an anomaly may occur as shown in Section 1.2.

These observations give us the hint that a high-level computational transaction model that integrates both of these two sub-transactions may be able to achieve such coordination. In addition, the ACID properties of such a high-level transaction should be sufficient to resolve the conflicts between its sub-transactions, namely source update transactions and view maintenance transactions.

2.2 A Transactional Model: VM_Transaction

We model this high-level transaction that integrates both a source update transaction and its corresponding view maintenance transaction as a **VM_Transaction**.

DEFINITION 1. *A VM_Transaction starts after a local successfully committed* source update transaction T_s *and commits when the materialized view has been successfully refreshed, i.e., the commit of the* view maintenance transaction T_v. *We denote it as "*$T = T_sT_v = w(DS_i)c(s)r(VD)r(DS_1)r(DS_2)\ldots r(DS_n)w(v)c(v)$*".*

Previous research has already hinted at this notion of a "global view maintenance transaction" [Zhuge et al. 1996]. However, [Zhuge et al. 1996] states that a global transactional approach is not feasible since the sources may not tolerate such a wait time before being allowed to commit their local updates. We now observe that the nested structure of a VM_Transaction as indicated in Definition 1 can be exploited to avoid this potential drawback. In fact, our work will show the benefits and feasibility of such a transactional approach towards view maintenance. Here, we adopt this idea of taking a transactional approach for view maintenance and put forth the following supporting arguments:

(1) The VM_Transaction model is a **conceptual** rather than a real transaction model. It encapsulates two types of transactions, namely, source update transactions and their corresponding view maintenance transactions. We observe

[1] In schema change maintenance transactions, we may need to update the view definition (VD) for any schema change that affects the view schema before we can generate the maintenance queries. More discussion on this issue can be found in Section 6.2.

that the source update transaction can in fact commit autonomously as a sub-transaction within the complete VM_Transaction. Thus sources never need to wait.

(2) With the help of the VM_Transaction concept, the view maintenance anomaly problem [Zhuge et al. 1995] can be rephrased as the well-known "read dirty data" transaction problem. With an appropriate concurrency control solution, the conflicts between the VM_Transactions, in other words, the view maintenance anomaly problems can naturally be solved.

(3) Lastly, this formal transaction-based solution lays a solid foundation for many other maintenance functions and services including handling multiple relations per source or parallel view maintenance, as we will demonstrate later.

2.3 Start, Commit, Abort and Rollback of VM_Transactions

Since our notion of a VM_Transaction is a conceptual transaction model, it does not have an automatic rollback or abort mechanism. A VM_Transaction is started when a source update has been committed. After that, we must have this source update incorporated into the view to keep the view consistent [2]. The reason is that the source update transaction is out of the control of the view manager. Thus the rollback of this update cannot be achieved. Hence whenever an error happens during the VM_Transaction, we have to redo the maintenance again until it is successfully committed.

2.4 VM_Transaction Scheduling

Two types of VM_Transaction schedulers are possible, namely, sequential and parallel ones. A sequential scheduler processes one view maintenance transaction at a time, and continues on the next one only after the current one has been completed. Most view maintenance algorithms in the literature [Zhuge et al. 1995; Agrawal et al. 1997] implicitly employ such sequential scheduling for view maintenance though without utilizing this transaction terminology. Parallel scheduling is more challenging, since out-of-order view maintenance transaction processing introduces additional conflicts [Zhang et al. 2004]. In this paper, we will first introduce our solution assuming a sequential scheduler to keep the description simple. Later in Section 6 we demonstrate that our transactional model can be easily extended to support such a parallel scheduler. This is in effect one of the advantages of adopting a transactional model for view maintenance. That is, many services can be easily added to this solid transactional framework.

2.5 Conflicts between VM_Transactions

We now examine the operations encapsulated within VM_Transactions to identify potential conflicts between them. From Definition 1, we know that there are four basic read and write operations within a VM_Transaction, namely, $w(DS_i)$, $r(VD)$, $r(DS_i)$ and $w(v)$. Hence, three types of conflicts may arise between these

[2] We adopt the view consistency notions from [Zhuge et al. 1995; Zhang et al. 2004]. More details will be discussed in Section 6.3.

operations [3]. First, the write-write conflict can occur when some write operation $w_1(DS_i)$ in VM_Transaction T_1 conflicts with $w_2(DS_i)$ in another VM_Transaction T_2. Second, some write operation $w_1(v)$ in VM_Transaction T_1 conflicts with $w_2(v)$ in another VM_Transaction T_2. Obviously such write-write conflicts would naturally be taken care of by the local data manager of the sources or the view manager respectively. The third conflict is the read-write conflict, namely, one read operation $r_1(DS_i)$ of VM_Transaction T_1 may read some inconsistent query result written by $w_2(DS_i)$ of another VM_Transaction T_2.

We can see that the view maintenance anomaly problems illustrated in Section 1.2 arise due to the read-write conflicts of VM_Transactions. That is, we rephase the view maintenance anomaly problem in terms of the read-write conflicts of VM_Transactions. Hence, as long as we can resolve such read-write conflicts, we solve the view maintenance anomaly problem.

2.6 Serializability of VM_Transaction

Above discussions lead us to a concurrency control strategy for our VM_Transaction model to resolve the conflicts. In other words, we only have to guarantee the serializability of VM_Transactions.

DEFINITION 2. *A history of VM_Transactions S is serializable iff it is equivalent to some serial schedule S_1 of these VM_Transactions.*

Note that a serial schedule of VM_Transactions S_1 assumes source update occurs after the previous maintenance process. Obviously no view maintenance anomaly would occur in this case. Consequently, no anomaly would happen in its equivalent history S either. Hence, Definition 2 describes the correctness criterion of the execution of VM_Transactions. In the next section, we will propose a concurrency control strategy, called TxnWrap, to achieve such serializability for VM_Transactions.

3. MULTIVERSION CONCURRENCY CONTROL ALGORITHM

Generally, two basic kinds of concurrency control algorithms [Bernstein et al. 1987], namely, **lock-based** or **multiversion-based**, can be used to address the serializability of traditional transactions. Below we examine which of these two strategies may be appropriate to achieve the serializability of the VM_Transactions in a loosely-coupled view maintenance context.

The traditional **lock-based** concurrency control algorithms [Bernstein et al. 1987] work as follows: For every write, an exclusive write lock is obtained and released at the end of the transaction; and for every read, a read lock is obtained, after the read operation completes, the read lock can be released. This is however too restrictive for the schedule of VM_Transactions. By Definition 1, we know that VM_Transaction will read from all sources. Hence a read lock over all sources is required. However, the sources in a distributed environment are often autonomous and would not be willing to accept locks from external agents. Hence locking algorithms are not suitable in such environments.

[3] A read-write view definition (VD) conflict may arise between VM_Transactions if we schedule view maintenance transactions in parallel. We investigate this type of conflict more closely in Section 6.

In the traditional **multiversion** concurrency control algorithms [Bernstein et al. 1987], a writer writes on a version that is different from the one that the reader reads. That is, the reader will read an older version than the one that the writer is currently writing to, thus avoiding the conflicts. In particular, we further differentiate between two categories of multiversion algorithms, i.e., the finite version [Mohan et al. 1992; Quass and Widom 1997] and the unrestricted version [Chan and Gray 1985; Agrawal and Sengupta 1989] algorithms. Finite version algorithms pre-set a fixed number of maximal versions (often 2 or some constant n). There is a so called "version expiration" [Mohan et al. 1992; Quass and Widom 1997] phase when the number of versions exceeds the allowed maximum. In that case, we have to switch the reader to a newer existing version to allow for the writers to start making a new version. To achieve this, we require that either the writer synchronizes with all readers on the old version or aborts those readers. In comparison, in unrestrictive multiversion algorithms, the writers can always create a new version and the readers can find whatever the most appropriate versions that they need at any time. No blocking between the reader and writer would ever occur.

Based on above observations, we find that an unrestrictive multiversion algorithm is the appropriate design choice in loosely-coupled environments. First, the version that the reader (view maintenance transaction) needs should never expire before the maintenance process has been completed, otherwise the materialized view would be inconsistent. Second, the autonomous writer (source update transaction) is unlikely to synchronize with the reader. For this reason, we favor the choice of an unrestricted multiversion algorithm. Also note that since the VM_Transactions will read a monotonically increasing version of data instead of some random old ones, the version cleanup can be efficiently carried out for old versions. As an added benefit, we can easily extend this concurrency control strategy to also work with a parallel scheduler as we will show in Section 6.

We now introduce an unrestricted multiversion algorithm based on the above discussion, called **TxnWrap**. The main idea is to extend the functionality of the wrapper for each data source beyond just communicating between sources and the view manager [4]. That is, the wrapper is responsible for storing the versions generated due to the source updates and thus answering any maintenance query using this versioned information. Below, we describe our TxnWrap multiversion algorithm, in particular, first the wrapper materialization of sources at initialization time and then the version management under both update notifications and query requests.

3.1 Wrapper Schema Setup

We propose to extend the wrapper of each source to contain version relations of source data, called *wrapper relations*. A simple full replication of the source data for the wrapper relations would guarantee that we can answer a maintenance query at the wrapper without any further source access. However we can further improve upon this by various optimization techniques to filter out unnecessary data, as

[4] Note that this is just one of the three architectural choices for where to keep the versioned data. See Section 4 for an elaboration on these choices. Without loss of generality, here we select one of these designs to describe the main idea.

we will discuss in detail in Section 5. Here, for the purpose of understanding the concurrency control strategy, we will first assume that each version relation corresponds to a full copy of the source relation.

We further propose to add two more attributes *V_Start* and *V_End* into the wrapper relation to denote the life time of each tuple. *V_Start* denotes the start version of the tuple (by insertion), and *V_End* denotes the version when the lifetime of the tuple ends (by deletion). We initialize the tuples in the wrapper relation by setting their *V_Start* to 0 and *V_End* to ∞. This way we guarantee that all initial tuples are visible to all VM_transactions.

We also build a *meta relation* at the wrapper for versioning of the source meta data. It contains six attributes, namely, *Rel*, *Attr*, *Rel′*, *Attr′*, *V_Start* and *V_End*, respectively. The pair of attributes *Rel* and *Attr* describes the source meta data. *V_Start* and *V_End* denote the life span of the attributes, i.e., the version number when it's been created, renamed or dropped. The pair of attributes *Rel′* and *Attr′* indicates the new name of a particular attribute or of a relation. The initial source wrapper status for the two sources from Example 1 are now shown in Figures 2 and 3, respectively.

Wrapper for DS_1

Relation C′

Name	Add.	Phone	V_Start	V_End
Tom	MA	123	0	∞
Bob	CA	146	0	∞

Meta Relation

Rel	Attr	Rel′	Attr′	V_Start	V_End
C′	Name	-	-	0	∞
C′	Add.	-	-	0	∞
C′	Phone	-	-	0	∞

Fig. 2. Initial Extent of Wrapper of DS_1

Wrapper for DS_4

Relation F′

Name	Age	FlightNo	Dest	V_Start	V_End
Tom	17	AA4399	Asia	0	∞

Meta Relation

Rel	Attr	Rel′	Attr′	V_Start	V_End
F′	Name	-	-	0	∞
F′	Age	-	-	0	∞
F′	FlightNo	-	-	0	∞
F′	Dest	-	-	0	∞

Fig. 3. Initial Extent of Wrapper of DS_4

3.2 Version and VM_Transaction Identifiers

A version number is needed in a multi-version management system to identify each version. Similarly, a VM_Transaction identifier is also necessary to track and identify each view maintenance transaction. One simplified strategy is to introduce a *global id* [Chen et al. 2002]. This identifier is issued by some global identifier server in the materialized view manager to assure its global uniqueness. In such a scenario, a global id would serve both as a version number and a VM_Transaction id. Thus, whenever a source update is reported to the wrapper, the wrapper must first get a global id from the view manager and then use that id to create versions. This however introduces coordination cost between wrappers and view manager, which may be non-trivial in distributed environments. For this reason, we now introduce two levels of identifiers, a *local id* within the scope of a data source to identify versions in the wrapper and a *TxnID* within the scope of the whole view maintenance context to manage VM_Transactions. This avoids the extra coordination cost between wrappers and view manager and is thus beneficial for distributed environments.

3.2.1 *Local Identifier in the Wrapper.* We define a *local id* to be a timestamp that represents the time the update happened at each individual data source. Without loss of generality, we use an integer k ($k \geq 0$) to represent the local id. Each local id is assigned by the corresponding wrapper with each source update. For example, assume three source updates as shown in Figure 4 happened on relations as depicted in Figure 1, then the wrapper will attach a local id 1 to the update "*INSERT INTO Customer VALUES ('Ben','CA','213')*" because this is the first update to the relation *Customer*. While for the update *"RENAME Customer.Name TO Customer.First-Last"*, its local id will be 2 because this is the second update in the relation *Customer*. The same rule also applies to the update on source relation *FlightRes*. Its local id in the wrapper will be 1 because this is the first update in this relation.

3.2.2 *VM_Transaction Identifier: TxnID.* We also need identifiers to track each global VM_Transaction and help to construct correct maintenance queries that access the appropriate versions of data. A TxnID τ is defined to be a vector $\tau = [k_1, k_2, \ldots, k_n]$ with $\tau[i] = k_i$ ($1 \leq i \leq n$ and n is the number of data sources). Each $\tau[i]$ records the current local id of each DS_i at the time this TxnID is being generated, i.e., the largest local id that has been reported to the view manager thus far. We describe the rules for generating TxnIDs as follows:

—As initial value, the view manager has a vector T with each $T[i]$=0 ($1 \leq i \leq n$). Each entry $T[i]$ will hold the local id that has been reported from the source DS_i.

—Whenever an update from source DS_i ($1 \leq i \leq n$) with a local id k is reported to the view manager, the view manager generates a TxnID τ for this source update using the following two steps: [5]
  —Set $T[i] = k$.
  —Copy T to τ (Set $\tau[j] = T[j]$ for all $1 \leq j \leq n$).
  —Return τ as the VM_Transaction identifier.

It's easy to see that though the local ids in different data sources may be the same, the TxnIDs are globally unique for each update. From the view point of the view manager, each entry of the TxnID vector records the current state of each data source on arrival of the source update to be maintained.

For example, the view *Asia-Customer* as shown in Example 1 is defined on two data sources, one is *Customer* and the other is *FlightRes*. We thus build a TxnID vector T with two entries, with each entry containing the current local id of its corresponding data source. Without loss of generality, we use $T[1]$ to hold the local id from relation *Customer* and $T[2]$ to hold the local id from the relation *FlightRes*. Initially, we have $T[i] = 0$ ($1 \leq i \leq 2$). For the three updates depicted in Figure 4, if we also assume that they arrive at the view manager in the order as we have defined them, then the TxnIDs for these updates will be $[1, 0]$, $[1, 1]$ and $[2, 1]$ respectively. That is, when the first data update (the VM_Transaction1) with a local id 1 arrives at the view manager, we set $T[1] = 1$ and return $[1, 0]$ as its TxnID. While for

[5] A FIFO transferring of updates from data souce to view manager for updates from the same data sources is required, which is also assumed by most of the literature in the view maintenance area.

update VM_Transaction2, we set $T[2] = 1$ and get $[1, 1]$ as the TxnID, and so on. As can be seen, each entry in TxnID records the right versioned data that needs to be accessed from the corresponding wrapper in its view maintenance transaction.

3.3 Version Operations

Basically, there are three classes of version operations that the wrapper must perform. First, the wrapper needs to create versions whenever an update is reported by a source. Second, the wrapper needs to read the appropriate versions to answer each maintenance query issued by the view manager. Third, any versions that are no longer required by any future VM_Transactions need to be cleaned up.

3.3.1 *Version Creation in the Wrapper.* The updates reported by a source can be categorized as data updates, which are *insert*, *delete* and *update*, or schema changes, which are *add*, *drop* and *rename* an attribute or *create*, *drop* and *rename* a table. To create versions, a local id i will be assigned as a version number by the wrapper to create a new version for each source update. The corresponding version creation rules are shown below:

(1) Delete a tuple t: update $t[V_End]$ to be i.
(2) Insert a tuple t: insert this tuple, set $t[V_Start]$ as i and set $t[V_End]$ ∞.
(3) Update a tuple t: treat it as a delete followed by an insert.
(4) Drop an attribute: update the attribute's *V_End* to be i in the meta-relation.
(5) Add an attribute: insert it into the meta relation with *V_Start* to be i and *V_End* to be ∞.
(6) Rename an attribute: In meta relation, set *Rel′* and *Attr′* of the old attribute to its new names, update *V_End* to i. Then add the renamed attribute to meta table.
(7) Drop a table: treat it as drop of all its attributes.
(8) Add a table: Add all its attributes to the meta-relation.
(9) Rename a table: Method similar to rename an attribute, but operate on all attributes.

As an example, consider the four source updates shown in Figure 4. After the corresponding version creation step, the wrapper relation and metadata relation are shown in Figures 5 and 6.

VM_Transaction1 (local id =1, TxnID=[1,0])
INSERT INTO Customer VALUES ('Ben', 'CA', '213');

VM_Transaction2 (local id =1, TxnID=[1,1])
INSERT INTO FlightRes VALUES ('Ben', '18', 'A34', 'Asia');

VM_Transaction3 (local id =2, TxnID=[2,1])
RENAME Customer.Name TO Customer.First-Last;

VM_Transaction4 (local id =2, TxnID=[2,2])
INSERT INTO FlightRes VALUES ('Sue', '20', 'A34', 'Asia');

Fig. 4. Four Source Updates

Wrapper for DS_1

Relation C′

Name	Add.	Phone	V_Start	V_End
Tom	MA	123	0	∞
Bob	CA	146	0	∞
Ben	CA	213	1	∞

Meta Relation

Rel	Attr	Rel′	Attr′	V_Start	V_End
C′	Name	C′	First-Last	0	2
C′	Add.	-	-	0	∞
C′	Phone	-	-	0	∞
C′	First-Last	-	-	2	∞

Fig. 5. DS_1 Wrapper Content after Updates

3.3.2 *Version Read in the Wrapper.* The read version operation happens when a VM_Transaction wants to query a source. Rather than being processed by the source, the queries are interpreted by the wrapper and executed by reading the appropriate versions from the wrapper data and metadata relations. Notice that the VM_Transaction requires to see a consistent state of all sources. Here a consistent state means both source data and metadata.

To achieve this, the first step is to rewrite the original maintenance query by augmenting it with the version conditions *"AND V_Start* $<=$ $\tau[i]$ *AND* $\tau[i] <$ *V_End"*. This rewritten maintenance query will be evaluated over the wrapper data relation. For example, Figure 7 depicts the rewritten versioned maintenance query for the three source data updates from Figure 4. The third rename schema change in Figure 4 results in the rewriting of view definition without any maintenance queries.

Wrapper for DS_4

Relation F′

Name	Age	FlightNo	Dest	V_Start	V_End
Tom	17	AA43	Asia	0	∞
Ben	18	A34	Asia	1	∞
Sue	20	A34	Asia	2	∞

Meta Relation

Rel	Attr	Rel′	Attr′	V_Start	V_End
F′	Name	-	-	0	∞
F′	Age	-	-	0	∞
F′	FlightNo	-	-	0	∞
F′	Dest	-	-	0	∞

Fig. 6. DS_4 Wrapper Content after Updates

VM_Transaction with TxnID=[1,0]
SELECT Name, Age, FlightNo, Dest FROM F′
WHERE Name='Ben' AND V_Start <=0 AND V_End >0

VM_Transaction with TxnID=[1,1]
SELECT Name FROM C′WHERE
Name='Ben' AND V_Start <=1 AND V_End >1

VM_Transaction with TxnID=[2,2]
SELECT First-Last FROM C′WHERE
Name='Sue' AND V_Start <=2 AND V_End >2

Fig. 7. Version Queries at Wrapper

The second step is to make sure the metadata specified in the maintenance query is consistent with the wrapper data relation. If not, we need to evolve the schema of the wrapper data relation. More precisely, if the version number of the maintenance query "$\tau[i]$" is larger or equivalent than any "*V_End*" in the wrapper metadata relation, then we know that the corresponding wrapper data relation's schema is out-dated. For example, to maintain the fourth update in Figure 4, the maintenance query in Figure 7 needs to access the 'First-Last' column, which is not available yet in the wrapper data relation. Hence the corresponding wrapper data relation has to be evolved by (1) deleting the corresponding tuples in the metadata relation and (2) renaming/dropping the corresponding columns/tables in the wrapper data relations. The above has to be done as one atomic action. In this example, we delete the (C', Name,...) tuple from metadata relation and rename the column 'Name' to 'First-Last'. After this metadata consistency checking, we can process the maintenance query. Note that we can keep those metadata tuples with $V_End <> \infty$ (there are typically only few such tuples) in memory to speed up this step.

3.3.3 *Version Cleanup in the Wrapper.* Finally, we clean up versions that are no longer needed by any maintenance transaction. The primary purpose of this operation is to reduce the size of the wrapper relations in order to speed up the maintenance queries and to evolve the wrapper schema. For a sequential scheduler, the cleanup is straightforward. That is, after the completion of a VM_Transaction

with TxnID τ, for all the local ids τ[i] ($1 \leq i \leq$ n) recorded in TxnID τ, the data with V_End less than τ[i] are no longer required by any later maintenance transaction. Hence they can be deleted from the corresponding wrapper relation specific to DS_i. For the wrapper data relation, we simply delete the tuples with $V_End <= \tau[i]$. For wrapper metadata relation, we atomically execute the update as *"DELETE * FROM meta-table WHERE V_End <= τ[i]"* and perform the corresponding schema change operations on the wrapper data relation, e.g., drop or rename attributes. For a parallel scheduler (Section 6), intuitively, we have to assure that all VM_Transactions that contain a local id less than τ[i] are completed before that respective version removal can proceed.

3.4 TxnWrap: A Multiversion Algorithm

We are now ready to introduce our TxnWrap concurrency control algorithm (TxnWrap) for materialized view maintenance.

(1) First, the TxnWrap initializes the wrapper relations as defined in Section 3.1.

(2) When a source commits a local update transaction and reports its updates to the wrapper (or extracted by the wrapper), the TxnWrap first assigns a local id as a version id and uses it to create versions in the wrapper. After that, the wrapper notifies the view manager about these updates. The view manager builds a TxnID by collecting and concatenating all the current *local id*s of each data source for this VM_Transaction.

(3) When the wrapper receives a maintenance query from the view manager, then the TxnWrap rewrites the maintenance query according to the corresponding local id recorded in the VM_Transaction identifier (TxnID) and executes it upon the wrapper relations.

(4) When a VM_Transaction is committed, version cleanup starts based on the type of scheduler, i.e., sequential or parallel. The cleanup action can be done by a background daemon process.

The correctness criterion of TxnWrap algorithm is characterized by Theorem 1.

THEOREM 1. *A history of VM_Transactions scheduled by the TxnWrap is serializable.*

We prove by contradiction in Appendix A that each TxnWrap multiversion (MV) history is serializable by showing its multiversion serial graph (MVSG) to be acyclic based on transaction theory [Bernstein et al. 1987]. By Theorem 1 and Definition 2, we know that TxnWrap maintains the views correctly. This means that the extent of the view after maintenance reflects the right states of the data sources.

3.5 Handle Multiple Relations in One Data Source

The discussions on the VM_Transaction model and its multiversion concurrency control algorithm (TxnWrap) so far assume each data source only has one relation. Such assumption, while has also been made by the other approaches in the literature thus far [Agrawal et al. 1997; Zhang and Rundensteiner 2000], can be easily lifted within our model as shown below. The VM_Transaction model naturally fits if we define read and write operations on relations instead of data

sources. For example, assume a view is defined upon two data sources involving five relations, e.g., $DS_1(R_1, R_2, R_3)$ and $DS_2(R_4, R_5)$ [6]. A VM_Transaction that models a source update to relation R_2 at DS_1 can be denoted as "$T = w(R_2)c(R_2)r(VD)r(R_1)r(R_3)r(R_4)r(R_5)w(v)c(v)$". Correspondingly, the TxnID will also be generated based on the local ids of source relations instead of data sources. Hence, in TxnWrap, we have the same number of wrapper data relations in the source wrapper as the relations in the data source.

For generating maintenance queries when one data source contains multiple relations, the view manager could send *one* combined maintenance query that involves all the relations in that data source instead of multiple maintenance queries to each relation. Correspondingly, the wrapper should also append all the version conditions of these relations.

For example, a data update ΔR_2 at relation R_2 with TxnID τ can be maintained using the following two queries: 1) A query "$R_1 \bowtie \Delta R_2 \bowtie R_3$" to DS_1's wrapper. Here we assume the result of this query is ΔDS_2^1, which denotes the effects of ΔR_2 on data source DS_1. 2) A query "$\Delta DS_2^1 \bowtie R_4 \bowtie R_5$" to DS_2's wrapper. Note that the source wrapper would append all related version conditions when receiving a combined maintenance query. For example, the version conditions for the first query will be "AND $R_1_Wrapper.V_Start \leq \tau[1]$ AND $\tau[1] < R_1_Wrapper.V_End$ AND $R_3_Wrapper.V_Start \leq \tau[3]$ AND $\tau[3] < R_3_Wrapper.V_End$". Here R_1_Wrapper and R_3_Wrapper are the corresponding wrapper relations for R_1 and R_3.

Hence the TxnWrap algorithm can be easily extended to handle multiple source relations. This also demonstrates an additional benefit of adopting the TxnWrap approach. Prior work such as SWEEP [Agrawal et al. 1997] has to send maintenance queries to each source relation even if they exist in the same database. While in a separate work, [Varde and Rundensteiner 2002] has tried to address this using a schema of two-level compensations.

4. TXNWRAP VERSION DESIGN STRATEGIES

4.1 Version Placement Design Choices

In the previous section, our concurrency control strategy, TxnWrap, will materialize versioned data in a special-purpose source wrapper. Clearly this is not the only design choice for the versioned data. Alternative choices include to either (1) add extra version information directly into the source data store or (2) to materialize the versioned data on the view server itself. We now discuss the tradeoffs between these version placement design options.

First, we consider the case of adding version information directly into each source. This would imply that two version attributes as described in Section 3.1 are directly added to each of the source relations. Clearly, this option reduces the space cost compared to storing versioned source copies at wrappers. However, the source update transactions would now be affected since they also have to deal with the version information. Such version information serves the sole purpose of external view maintenance support. This design choice thus may not be acceptable in

[6]Relations in different data sources could have same name as long as they are globally unique, i.e., the concatenation of data source name with the relation name. For simplicity, we use different names in this example to illustrate the technique.

a loosely-coupled environment since the sources may not want to degrade their transaction performance.

The second design choice would be to materialize the versioned relations inside the view manager. One potential benefit is that instead of decomposing the view maintenance query into individual remote source queries to each source, we now could use one single combined maintenance query. The reason is that all the relations are within the same view site database. This design choice is similar to the view self-maintenance approach, such as [Quass et al. 1996], where the source data is selectively replicated in the view site for maintenance.

However, in this case, the view manager would have to maintain both the materialized view and the versioned copied data as well as answer user queries. This may become a heavy burden for the view manager. Furthermore, since all the maintenance queries are against the data on the same view site, we may not be able to improve the parallel maintenance performance (an additional functionality gained by taking a multiversion-based approach as demonstrated in Section 6). This is because a single view server may not have enough system resources to handle a large number of maintenance queries in parallel. Another potential drawback is that if we have multiple warehouses, such as data marts, that share the same source tables, then the source data has to be copied into each view site resulting in a significant overall space cost for the entire environment.

In comparison, placing the versioned data into a dedicated wrapper achieves the independence and simplicity of both the sources and the view manager by moving the complex version management responsibility into a third dedicated place, as we have illustrated in Section 3. First, we can reduce the overall space cost for applications such as data marts since one wrapper can support multiple warehouses. Second, the performance gain for parallel processing can be effectively explored since the maintenance query workload is distributed to each wrapper. Our experimental studies in Sections 7.6 and 7.7 show indeed significant parallel maintenance performance improvements gained by this design choice.

4.2 Comparison with Self-Maintenance Solutions

Next, we compare our solution to the self-maintenance approach such as [Quass et al. 1996]. In [Quass et al. 1996], the authors propose to selectively materialize the source data at the warehouse so that the view maintenance can be achieved without accessing the sources. Note that this approach also has to make multiple copies for data mart applications due to its data materialization at the warehouse. Since the self-maintenance approach [Quass et al. 1996] does not explicitly assume any concurrency control strategies, we now investigate how the concurrency issues could be tackled in the context of this approach.

Since the source copies are kept at the warehouse, the updating of source copies and the updating of views can be combined into a single maintenance transaction. However, in this case, deadlocks may occur. Here is a simple example. Assume the view is $R_1 \bowtie R_2$ and there are two source updates ΔR_1 and ΔR_2. R'_1 and R'_2 are the corresponding source copies materialized at the data warehouse. $\Delta R'_1$ and $\Delta R'_2$ are the changes to the source copies based on ΔR_1 and ΔR_2. The maintenance transactions for these two updates are $w(R'_1)$ followed by $\Delta R'_1 \bowtie R'_2$ and $w(R'_2)$ followed by $\Delta R'_2 \bowtie R'_1$, respectively. If the write operations acquire table-level

locks, then the deadlocks may occur. The reason is that the first transaction reads R'_2 table and writes R'_1 table, while the second transaction reads R'_1 and writes R'_2 table. There are cyclic read-write conflicts on R'_1 and R'_2 tables between these two transactions, resulting in deadlocks. Even if fine-granularity locking mechanisms, such as low-level locks are used, the deadlocks still may occur. For example, when $\Delta R'_1$ and $\Delta R'_2$ contain any tuples that are joinable to each other, the deadlock occurs on those tuples. Such deadlocks may introduce significant extra cost. In comparison, our TxnWrap approach does not have such deadlock problems.

An alternative transaction processing strategy for self-maintenance approaches is to separate the action of updating source copies and the updating of views as two independent transactions. In this case, although these two transactions can be executed in parallel, the same maintenance anomalies may occur as in Section 1.2. We then must apply either compensation-based or multiversion-based solutions to solve the anomalies as discussed in this paper. A multiversion-based solution outperforms the compensation-based one in that it achieves a more steady performance and allows parallel maintenance as shown via experiments in Sections 7.2, 7.3 and 7.5. Hence, based on the above analysis, the existing self-maintenance approaches can be enhanced by our TxnWrap solution to achieve better maintenance performance. In short, self-maintenance approaches themselves could benefit from the TxnWrap solution.

5. WRAPPER OPTIMIZATION STRATEGIES

5.1 Static Reduction Optimization

Version management assuming a full duplication of a source relation requires significant storage space. For this reason, several optimization strategies can be applied to reduce the storage. First, we can optimize the storage by analyzing the view definition. *Selection Optimization* extracts the local condition in a view definition and then pushes it down to the wrapper relation definition to reduce the wrapper size. For example, in Example 1, by pushing the local condition of the view definition, i.e., "$F.Dest = Asia$" into the wrapper relation definition, we can reduce the wrapper space for source relation *FlightRes*. This results in the additional advantage that some data updates with "$F.Dest \neq Asia$" can now be filtered early on as being irrelevant. They thus would not even be reported to the view manager. *Projection Optimization* projects any attribute of the source not involved in the view definition from the wrapper. Thus, for source relation *Customer* of DS_1 in Example 1, the attributes *Address* and *Phone* can be safely dropped from the DS_1 wrapper.

Second, we can explore existing constraints to reduce the space cost. For example, the key and foreign key constraints can also be used to reduce the versions. Assume two tables A and B with column $A.a$ referencing primary key $B.b$. Assume the materialized view defined as $A \bowtie B$ with join condition $A.a = B.b$. In this case, we can avoid to make any version copy of table A since the maintenance does not need to access table A at all. Similar ideas are proposed for the self-maintenance approach such as [Quass et al. 1996]. In fact, all space optimizations strategies developed for self-maintenance approaches should also be applicable here.

5.2 Dynamic Version Optimization

While the above strategies reduce the wrapper space statically, we now consider how to optimize the wrapper space dynamically. Here we measure the versioning cost by comparing the number of tuples of the wrapper relations to that of the source relations. We find that eventually after all the updates have been maintained and all the versions have been cleaned up, the wrapper relation contains the same number of tuples as the sources with just two extra version attributes. We thus consider this to be the minimum version cost. However, the intermediate state of the wrapper relation may also contain more tuples than the sources.

The first case is when we have source updates that result in both inserts and deletes of versions. For example, the update of all '$Price$' columns to '$Price$'*0.9 will double the size of the wrapper data relation. Although such tuple explosion is temporary and will be cleaned up, it may still significantly degrade the performance.

To solve this problem, if the wrapper identifies some update that potentially affects many of the current tuples (by analyzing both statistics and the update statement itself), then we propose to hold updating the wrapper data relation and just send the view manager a special update message. When the view manager starts to maintain this update message, the wrapper will be notified to directly update the wrapper data relation without any tuple duplication. Note that by this strategy we need to update the modified tuples' V_Start using the new local id. This is a different versioning strategy from what we initially would do based on Section 3.3.1. Also any subsequent changes to this relation have to be recorded in a special manner at the wrapper due to semantic issues. They can be either recorded as statements (such as update and delete statements) or deltas (such as inserts). This strategy avoids the tuple explosion problems with minimal extra cost, i.e., logging an update statement is a lot cheaper than duplicating a huge chunk of the relation.

The second case is when we delete a large number of tuples from one relation. Instead of updating these tuples in the wrapper, we can apply similar techniques as above to delete them directly thus avoiding the large intermediate version space.

6. PARALLEL MAINTENANCE SCHEDULER

6.1 Parallel Scheduling in a Data Update Only Environment

One advantage of using such a principled sound approach towards view maintenance, namely a transactional approach, is that other transactional-based features can be gotten for "free". As one example of this, we discuss in this section how TxnWrap can be extended to support *parallel view maintenance*. First, we note that a direct extension of the sequential TxnWrap scheduler would achieve provably correct parallel maintenance for view maintenance transactions in a data update only environment. As described in Section 2.1, one *data update view maintenance transaction* is represented as $r(VD)r(DS_1)r(DS_2)\cdots r(DS_n)w(v)$. Thus, there will be no read block between view maintenance transactions in TxnWrap if we apply a multi-version concurrency control strategy [7]. Borrowing from traditional

[7] We consider the w(v) operation in view maintenance transactions separately in the commit control discussion (Section 6.3).

concurrency control concepts [Bernstein et al. 1987], an *aggressive scheduler* can thus straightforwardly be applied. That is, we can start maintaining data update view maintenance transactions concurrently as long as sufficient computational resources are available in the view manager because there are no read/write conflicts in these transactions due to the multiversion concurrency control strategy employed by TxnWrap.

6.2 Parallel Scheduling in a Data and Schema Change Environment

However, more issues must be dealt with if we take source schema changes into consideration. To illustrate this, let us first briefly review how schema changes are maintained [Lee et al. 2002; Chen et al. 2001]. There are three steps for maintaining a source schema change:

—Determine which view is affected by the source schema change: $[r(VD)]$

—Find the suitable replacement for schema elements removed from the view definition and rewrite the view definition if needed: $[w(VD)]$

—Calculate the delta changes in term of tuples to be added or to be deleted due to the replacement between the old and the new view definition and adapt the view extent by committing these delta changes: $[r(VD)r(DS_1)r(DS_2)\cdots r(DS_n)w(v)]$

Thus, one *schema change view maintenance transaction* can be represented as $\mathbf{r(VD)w(VD)}r(VD)r(DS_1)r(DS_2)\cdots r(DS_n)w(v)$. Clearly, the view definition (VD) represents a critical resource among the different view maintenance transactions. Hence, when scheduling the maintenance for mixed data updates and schema changes, we have to consider the potential $r(VD)/w(VD)$ conflicts besides the anomaly problems (the read-write conflicts between the *source update transactions* and the *view maintenance transactions*).

In a view maintenance environment, we keep the assumption of a FIFO network for updates being transmitted from the sources to the view manager which come from the same data source, otherwise certain updates wouldn't be correctly maintained. For example, assume two updates "U_1:INSERT INTO Customer VALUES ('Ben', 'CA', '213')", "U_2:DELETE FROM Customer WHERE Name = 'Ben' " happened on the same source relation *Customer* as shown in Example 1 in this given order. Then we should maintain U_1 before U_2 in the view manager. If not, it is possible that the maintenance result of U_2 couldn't be refreshed or may be incorrect because the corresponding tuple isn't in the view extent yet. Thus, we can't reorder view maintenance transactions randomly.

In addition, more ordering restrictions need to be imposed once we assign the corresponding TxnID to each update. To clarify this, we first define the order of TxnIDs as follows: given two TxnIDs τ_i and τ_j, $\tau_i < \tau_j \Leftrightarrow$ (1) $\forall h$, $1 \leq h \leq n$ (n is the size of TxnID vector), $\tau_i[h] \leq \tau_j[h]$. (2) $\exists k$, $1 \leq k \leq n$, $\tau_i[k] < \tau_j[k]$. In situations that there is a mixture of data and schema changes to be maintained, we can't randomly reorder these view maintenance transactions once their TxnIDs are assigned even if these updates come from different data sources. Otherwise the maintenance result may also be inconsistent with the data source state. We use the following example to illustrate this ordering restriction.

Example 2. *Figure 8 illustrates the view Asia-Customer first defined in Example 1. For simplicity, we use Customer $\bowtie$ FlightRes to represent the view. Assume that after a certain time period, source relation Customer becomes unavailable, i.e, a schema change "DROP TABLE Customer" has happened. Based on view synchronization [Lee et al. 2002; Nica et al. 1998] techniques, the view manager will locate a replacement (Here we use the relation Registered-User as the replacement of relation Customer) to update the view definition and its extent. After that, a data update "INSERT INTO Registered-User VALUES ('Ben', 'CA', 456)" also happened on the relation Registered-User. Assume these two updates are reported to the view manager for maintenance following the order as defined above. Assume they are assigned TxnIDs [1,0,0] and [1,0,1] respectively. After the correct maintenance, the final view definition will evolve to Registered-User $\bowtie$ FlightRes, and it will have two tuples as depicted in Figure 9.*

However, now assume we maintain the data update with TxnID [1,0,1] first. Then no changes to the view extent would be recorded because the source relation Registered-User is not in the view definition of the Asia-Customer yet. While maintaining the update [1,0,0], the view manager can't see the data update's effect because the TxnID tells us that it can only see the state '0' of relation Registered-User in this maintenance process. State '0' denotes the state of the Registered-User relation before the data update happened. Thus, the maintenance result of this data update will be lost. In short, we can't change the scheduling order of these two updates after we have assigned them the corresponding TxnIDs.

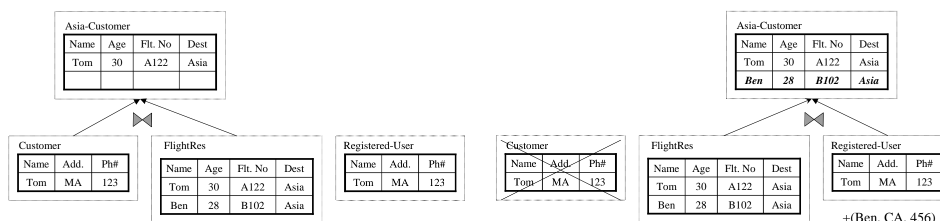

Fig. 8. View Asia-Customer Before Updates. Fig. 9. View Asia-Customer After Updates.

As can be seen, this ordering restriction is due to the concurrency conflicts on the critical resource, namely, the view definition *Asia-Customer*. That is, the view definition could be changed by some source schema changes while other view maintenance transactions still need it. As we discussed before, this concurrency can also be resolved either using **lock-based** or **multiversion** strategies. However, in this particular context, a multiversion based strategy does not have any preference because we have to keep the TxnID order to generate the right view definition version anyway. For example, for the two updates in Example 2, the data update with TxnID $[1, 0, 1]$ can't be processed until the new version of the view definition has been built, that is, after the schema change with TxnID $[1, 0, 0]$ completes at least its view synchronization (VS) portion. Thus, a **multiversion** approach doesn't offer us any benefits rather the extra overhead of versioning and version

control. Based on the above analysis, we propose the following parallel maintenance scheduler using a **lock-based** control strategy to handle the conflicts in the view definition access due to some schema changes:

—Start view maintenance transactions based on their TxnID order.

—Synchronize the $w(VD)$ operations of schema change maintenance. That is, no $r(VD)$ or $w(VD)$ operations will start before the previous $w(VD)$ has been finished.

```
/* Parallel schedule the updates need to be maintained in the Txn_Queue */
1:    Parallel_Scheduler (TXN_Queue) {
2:        /* TXN_Queue contains unmaintained updates */
3:        while (TXN_Queue != empty) {
4:            VM_TXN = get first VM_TXN from TXN_Queue;
5:            /* get first VM_Transaction */
6:            if  (VM_TXN.getType() = DU) { /* data update VM_Trans. */
7:                Build a new thread to maintain VM_TXN or wait until new thread is available;
8:            } else { /* a schema change VM_Trans. */
9:                Wait until all running maintenance threads commit;
10:               View synchronization (VM_TXN);
11:               Build a new thread to wrap view adaptation(VM_TXN);
12:               /* synchronize view definition update, while run view adaptation  process in parallel
13:                  with other processes */
14:           }
15:       } /* end of TXN_Queue equals empty */
16:   } /* end of Parallel_Scheduler */
```

Fig. 10. Parallel Scheduling Process

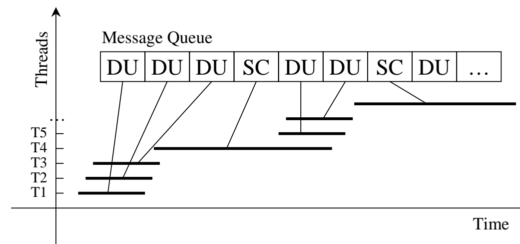

Fig. 11. Parallel Scheduling Example

Figure 10 lists the basic logic of this parallel scheduler. A sample execution plan is depicted in Figure 11. DU and SC stand for data update maintenance transaction and schema change maintenance transaction respectively. That is, each thread is responsible of one maintenance transaction. These threads can be run in parallel except each schema change maintenance will synchronize these processes.

6.3 Commit Control and Consistency

We adopt the notions of consistency of a view extent depending on how and when the source updates are incorporated into the materialized view in a distributed environment from [Zhang et al. 2004; Zhuge et al. 1995]. That is, five consistency levels (convergence, weak consistency, consistency, strong consistency and complete consistency) of the materialized view are defined based on the materialized view states according to source updates. However, in a parallel scheduling environment, even if each individual update is being maintained successfully, the final view state after committing these effects may still be inconsistent. This Variant Commit Order problem in a data update only environment has been addressed in [Zhang et al. 2004]. We can apply the same commit control strategy to our mixed data update and schema change environment. The control strategy we apply in our current parallel TxnWrap work is a strict commit order control. That is, only after we commit all the effects of all previous updates, can we begin to commit the current delta changes to the view extent. As we will show in Appendix A, it reaches at least a strong consistency level if we apply this strict commit order control strategy.

6.4 More Discussions on the Parallel Scheduler

The limitation of the above parallel scheduler is that once we assign the TxnIDs based on the arrival order of updates at the view manager, we then must stick with this order when scheduling transactions. For example, all the subsequent data updates that arrive after a schema change have to wait for the schema change to finish its view synchronization portion. This leads us to think about whether other less strict parallel scheduling solutions may be possible within the TxnWrap context. For this, we first need to determine whether it is possible to change the scheduling order of updates in the view manager while still keeping the view consistent.

Observation 1. *Any maintenance order for updates from different data sources will generate the same and correct final maintenance results.*

This observation allows us to assign the corresponding TxnIDs dynamically. For example, if we assign the data update in Example 2 the TxnID [0,0,1] as if it comes to the view manager first, while assigning the schema change TxnID [1,0,1], the final maintenance result of these two updates in this newly assigned order would also be correct. This observation gives us the hint that we should be able to exchange the scheduling order of updates in the view manager that come from different data sources as long as we assign the TxnIDs correspondingly. This way, a more flexible scheduler can be designed by freely exchanging the maintenance orders for updates that come from different data sources. We are thus able to reduce the wait of the data updates that arrived after a schema change.

7. EXPERIMENTAL EVALUATION

7.1 Experimental Testbed

Since TxnWrap by itself is not a stand-alone system, we have plugged it into the existing Dynamic Data Warehouse Maintenance (DyDa) system developed at WPI [Chen et al. 2001]. The basic DyDa system uses SWEEP [Agrawal et al. 1997], a compensation-based algorithm to handle concurrent data updates. It also has some

enhanced view adaption strategies to detect and correct a mixture of concurrent data updates and schema changes [Chen et al. 2004]. By disabling this concurrency logic and instead plugging TxnWrap into the DyDa system, we now have a transactional-based maintenance solution. The java code to handle concurrencies in the DyDa system is more than 5000 lines, while in contrast the TxnWrap module contains less than 900 lines of code. In this section, we experimentally evaluate TxnWrap by comparing these two systems.

In our experimental study, we have conducted our experiments on four Pentium III PCs with 256M memory, running Windows NT and Oracle8i. We employ six data sources with one relation each evenly distributed over three machines. Each relation has four attributes and 100,000 tuples. There is one materialized join view defined upon these six source relations residing on the fourth machine.

Currently, we set each VM_Transaction in our experiments to contain exactly one update, as to compare the performance with other algorithms in a similar configuration. The issue of batching multiple data and schema updates in one VM_Transaction is orthogonal to this paper. We thus ask readers to refer [Liu et al. 2002b] for details on this topic and corresponding experimental results.

7.2 Data Update Processing

In this experiment, we evaluate the overhead of TxnWrap for data update processing under a varying rate of concurrency compared to the SWEEP [Agrawal et al. 1997] algorithm. SWEEP applies local compensation to remove the *data update anomaly* as shown in Section 1.2. Figure 12 shows the average maintenance cost of one data update (on the y-axis) under different numbers of concurrent data updates (on the x-axis) for both systems.

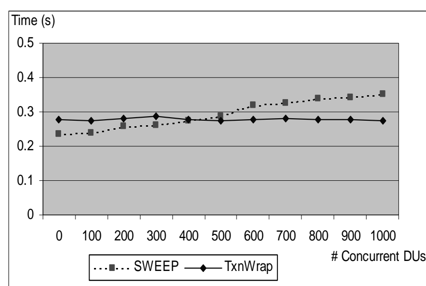

Fig. 12. Concurrent Data Update Process

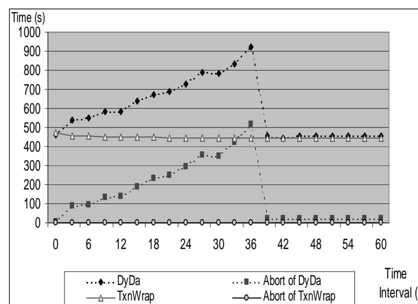

Fig. 13. Concurrent Schema Change Process

From Figure 12, we can see that initially the maintenance cost of TxnWrap is larger than that of SWEEP when the number of data updates concurrent with the one currently being processed is 0 (i.e., there is no concurrent data update). This cost is mainly caused by the versioned maintenance query, which requires additional version conditions to be evaluated. The maintenance cost of SWEEP increases with an increase in the number of concurrent data updates, because it has to use local compensation to deal with concurrent data updates. The more updates are found to be concurrent when handling one update, the more compensation has

to be undertaken. In comparison, TxnWrap exhibits an almost fixed cost for the handling of data updates independent of the number of concurrent data updates, and thus achieves a steady performance.

7.3 Schema Change Processing

In this experiment, we examine the performance of TxnWrap for schema change processing, which includes both view synchronization and view adaptation. We measure the maintenance cost for twenty schema changes while varying the time intervals between them. As described in Section 1.2, if there are some concurrent schema changes, the query would be broken and the current maintenance process may be aborted in the DyDa system. This is a significant cost. Figure 13 presents the maintenance as well as the abort cost (as part of the maintenance cost) of both systems (on the y-axis). From the figure, we see that the DyDa system has a varying abort cost under different system loads. The explanation of the curves of the DyDa system can be found in [Chen et al. 2004]. Eventually when the time interval is larger than the maintenance time, the updates will not conflict with each other. In this case, both systems have the same maintenance cost. In comparison, TxnWrap, employing a multiversion algorithm, avoids any abort and hence again achieves a steady performance.

7.4 Impact of Version Management Optimizations

In this experiment, we study the performance achieved when applying the wrapper optimization strategies described in Section 5. Since various optimization strategies all have the same goal, namely, to reduce the size of wrapper relations, we study how such size reduction affects the maintenance performance.

We measure the average maintenance cost of one data update or one schema change under different size reduction factors for wrapper optimization and compare TxnWrap to the non-multiversion solution DyDa. As shown in Figures 14 and 15, the x-axis is the size of the wrapper relations after optimization over that before optimization. Clearly, we find that the smaller the wrapper relations, the better maintenance performance.

We note that in Figures 14 and 15, there are still several differences between data update maintenance and schema change maintenance. There is a crossover point for data update maintenance. The reason is that the extra version attributes and conditions complicate the maintenance queries and are thus initially more costly to evaluate. In comparison, the maintenance of a schema change is much more expensive [Lee et al. 2002]. The performance largely depends on the number of tuples of the relations. The extra version attributes and conditions have a relatively small overhead. Hence, the performance improvement is more significant for schema change maintenance.

7.5 Parallel View Maintenance

7.5.1 *Basic Parallel View Maintenance Performance.* We have experimentally evaluated TxnWrap when extended with a parallel view maintenance scheduler. We use the same system setup as described above and use a workload of 500 data updates and vary the number of maintenance threads, i.e., the maximum number of source updates that can be maintained by the system at the same time. The

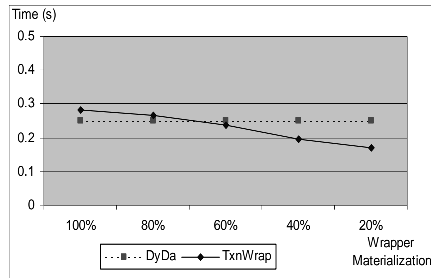

Fig. 14. Data Update Processing with Wrapper Materialization Optimization

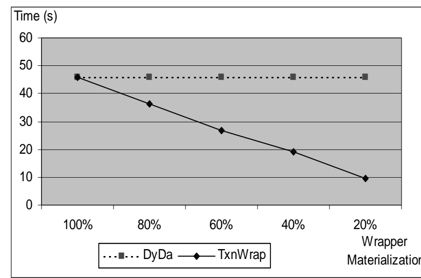

Fig. 15. Schema Change Processing with Wrapper Materialization Optimization

total maintenance time of these 500 data updates is depicted in Figure 16. The x-axis denotes the number of parallel threads in the system with S denoting the serial scheduler. The y-axis represents the total processing time.

If we only use one thread, then the total processing time is very similar to the serial one. There is some fairly small overhead for the parallel maintenance scheduler and thread management. Given our system setup, the total processing time reaches its minimum with five parallel threads. A maximum percentage of performance improvement of 150% is observed. When we further increase the thread number, the processing time achieves only some small extra improvement. The main reason is that the parallel processing power of wrapper databases is limited. They are not able to effectively process a large number of maintenance queries concurrently due to resource contention.

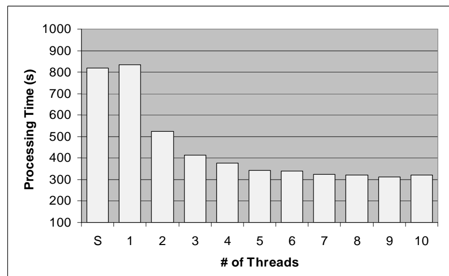

Fig. 16. Parallel Data Updates Maintenance

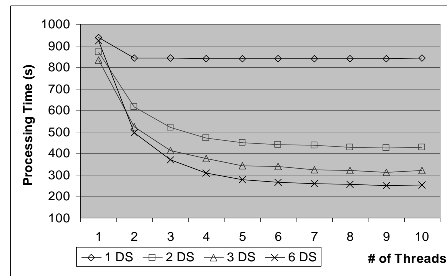

Fig. 17. Impact of Source Rel. Distribution

7.5.2 *Impact of Source Relation Distribution.* Figure 17 shows the changes of the parallel maintenance performance due to the distribution of source relations. Here we assume that each data source has a similar query processing capability. We vary the distributions of the six source relations from one data source to six data sources. The more data sources, the smaller the number of source relations located at each data source. Seen from Figure 17, if we distribute the source relations to more data sources, then a higher rate of performance improvement can be achieved. For example, if the six source relations are evenly distributed over the six data sources, then the maximal percentage of performance improvement in our

setting reaches around 250% when using 5 parallel threads. This is as expected because more maintenance queries can be answered in parallel by all these data sources.

7.5.3 *Impact of Network Delay.* In Figure 18, we measure the effect of changing the network delay in each maintenance query. A network delay is added when evaluating each maintenance query based on the average time to transfer a tuple across the network. For example, if we assume that the average time to transfer a tuple with 20 bytes is ℓ, then it takes k*2*ℓ to transfer k tuples with 40 bytes each. The performance changes from no network-delay to 'ℓ = 10ms' and then 'ℓ = 30m' in each maintenance query are listed in Figure 18. From the figure, we can see that while the high network delay has a negative effect on the serial TxnWrap performance, such effects are quickly offset by using our parallel TxnWrap scheme. The reason is that although the network transfer is slow, we can still run many other tasks in parallel, which improves the overall throughput. This is an especially useful feature since the wrappers and view manager are distributed over the network in our framework.

7.5.4 *Impact of Schema Changes.* We measure the parallel maintenance performance when schema changes and data updates are mixed. We set up a workload of 400 data updates and 2 schema changes. The two schema changes are evenly scattered among these 400 data updates. Figure 19 depicts changes of the total maintenance cost in terms of processing time when we vary the number of parallel threads. Similar with what we have measured for data updates only environments (Figure 16), the total processing time reaches its minimal around thread number five. The maximal percentage of performance improvement we measured in this case is around 45%. This is because (1) schema change maintenance is much more time consuming than that of a data update and (2) we cannot fully maintain schema changes in parallel.

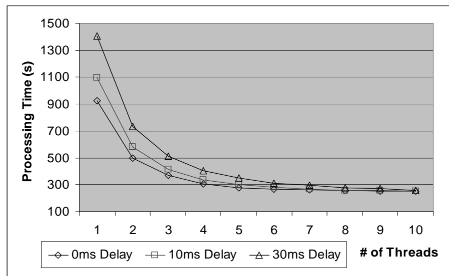

Fig. 18. Impact of Network Delay on Parallel Maintenance

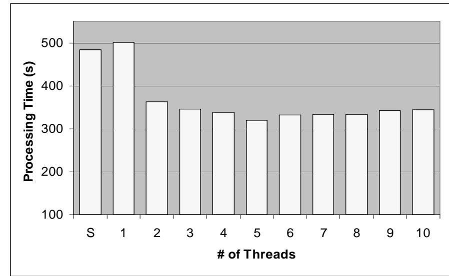

Fig. 19. Parallel Maintenance of both Data Updates and Schema Changes

7.6 Wrapper Placement Evaluation

As we have discussed in Section 4.1, distributing the wrapper relations as done in our proposed TxnWrap architecture is likely beneficial for maximally exploiting the parallel maintenance capability. We experimentally evaluate this hypothesis in

this section. Figure 20 depicts the maintenance performance under two extreme experimental settings. The "DW-Wrapper" case has six wrapper relations placed in the same database engine where the materialized view resides, while the "6DS-Wrapper" case has the wrapper relations placed in six separate database engines. We vary the number of parallel maintenance threads (x-axis). We use again the workload of 500 data updates as in Section 7.5.

From Figure 20, we can see that the maintenance performance of the "DW-Wrapper" approach could hardly be improved by adding more maintenance threads due to the high resource contention when handling a large amount of maintenance queries in parallel (the database server has a single CPU and a single disk in our experimental setting). In comparison, the "6DS-Wrapper" approach achieves better performance when adding more maintenance threads since the workload is distributed over several servers with less resource contention.

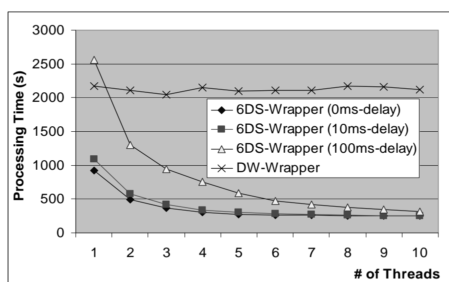

Fig. 20. Impact of Wrapper Placement

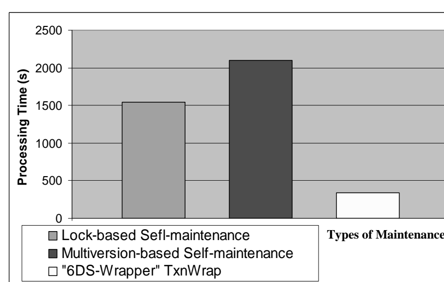

Fig. 21. TxnWrap vs. Self-Maintenance

However, the "6DS-Wrapper" approach may suffer from high network delay since each maintenance query will be evaluated across the network. Seen from Figure 20, given the network delay factor "$\ell = 100ms$" in this experimental setting, the total maintenance time of the "6DS-Wrapper" approach is worse than that of the "DW-Wrapper" in the case of sequential maintenance. We also find that this parallel maintenance strategy makes TxnWrap less sensitive to slow network environments. This is especially useful since the wrappers and view manager are distributed over the network. The reason is that although the network transfer is slow, we can still run many other tasks in parallel. This may improve the overall throughput.

7.7 TxnWrap vs. Self-Maintenance

In our system, we implement the self-maintenance approach [Quass et al. 1996] by replicating source relations in the DW server and issuing one combined maintenance query to this local DW server to maintain a source update. We compare TxnWrap to a lock-based self-maintenance solution and to a multiversion-based self-maintenance solution as described in Section 4.2.

Figure 21 compares the total maintenance time between the lock-based self-maintenance approach, the multiversion-based self-maintenance approach (i.e., the parallel "DW-Wrapper" TxnWrap approach) and the parallel "6DS-Wrapper" TxnWrap approach. The experimental setting is the same as in Section 7.6. Seen from Figure 21, the total processing time of the lock-based self-maintenance is less than

the multiversion-based self-maintenance approach since the latter has extra version management overhead. Both approaches are limited in terms of parallelism performance due to the high resource contention because of the data all residing on the same machine. In comparison, distributing the replicated data under the TxnWrap approach (labelled as "6DS-Wrapper" in Figure 21) gains much more parallelism performance. While the lock-based approach may also exploit more potential parallelism when the replicated data is distributed, one problem is that the lock-based approach integrates the source updates and the maintenance process into one single transaction. Managing such a global transaction in a distributed context is nontrivial. The potential deadlocks worsen this problem. In summary, we conclude that our proposed parallel maintenance with wrapper data materialization effectively improves the maintenance performance over prior approaches while offering the extra functionality to handle concurrent schema changes.

8. RELATED WORK

Maintaining materialized views under source updates is one of the important issues of data integration [Widom 1995; Zhuge et al. 1995]. Initially, some research has studied incremental view maintenance assuming no concurrency. Such algorithms for maintaining a view under source data updates are called view maintenance algorithms [Lu et al. 1995; Colby et al. 1996]. There has also been some work on rewriting view definitions under source schema changes [Nica et al. 1998; Lee et al. 2002], and on adapting the view extent under source schema changes [Gupta et al. 1995; Nica and Rundensteiner 1999].

As first noted in [Zhuge et al. 1995], if the data sources are on remote servers from the view site and an asynchronous loosely-coupled environment is assumed, then the maintenance process of sending maintenance queries to the remote data sources may experience concurrency conflicts due to the autonomy of sources. [Zhuge et al. 1995] introduces a compensation-based algorithm, called ECA, for incremental view maintenance of concurrent source *data updates* restricted to a single source. Strobe [Zhuge et al. 1996] handles multiple data sources while still assuming the schema of all sources to be static. SWEEP [Agrawal et al. 1997] applies local compensation of maintenance over distributed sources. [Salem et al. 2000] proposes to materialize delta changes of both sources and views with timestamps, thus being able to asynchronously refresh the view extent. They introduce a propagation algorithm that reduces the number of compensation queries. Our earlier work on the DyDa project [Zhang and Rundensteiner 2000] is the first attempt to address mixed workloads of concurrent data updates and schema changes. However, these existing efforts including DyDa are all compensation-based. That is, they all require special algorithms to detect possible anomalies due to concurrent updates of sources and then to correct them thereafter. Furthermore, a formal basis for these algorithms is missing. This makes it more difficult to develop formal proofs of correctness or extensions in terms of services such as recovery or parallelism. Our new approach TxnWrap employs a multiversion solution removing concurrency concerns from view maintenance algorithms altogether, thus no longer requiring any compensation strategies. This is complimentary to any of the previous work mentioned above by isolating concurrency concerns from the maintenance logic.

Beyond view maintenance work, there are techniques in version management and concurrency control in general that relate to our approach. Multiversion concurrency control algorithms [Agrawal and Sengupta 1989; Chan and Gray 1985; Quass and Widom 1997; Mohan et al. 1992] can be categorized into two types, namely, finite version and unrestricted version algorithms. In this paper, we demonstrate that unrestricted version algorithms are the most appropriate design choice for materialized view maintenance. In addition, our proposed algorithm can also handle versioned schema meta data. [Quass and Widom 1997] proposes to utilize a two-version algorithm to resolve the concurrency between the view maintenance and its read-only sessions. Our work instead focuses on the other end of the spectrum, i.e., the concurrency between frequent source updates and the view maintenance. In the former work, the read-only sessions of the materialized view could be scheduled for any version as long as it is consistent. While in our work, the view maintenance reads will ask for a particular version of source data. To keep the view consistent, the desired version should always be available to the view maintenance process.

The TxnWrap versioning approach proposed in this paper is somewhat similar to the temporal database model [Zaniolo et al. 1997] in the literature. Our work shows the potential benefits for exploiting this temporal database model for view maintenance. Many existing temporal index schemes [Lomet and Salzberg 1989] can be applied here to further improve the maintenance query performance.

9. CONCLUSIONS

In this paper we take a fresh new look at the concurrency problems of materialized view maintenance. In contrast to previous work, we now present a solution based on transaction theory. While previous work [Zhuge et al. 1995; Salem et al. 2000] focuses on specialized compensation algorithms for view maintenance, we instead propose a transactional model that removes concurrency considerations from the maintenance logic. In particular, we encapsulate both the source update transactions and the actual view maintenance process into what we refer to as a VM_Transaction. We then propose a multiversion algorithm called TxnWrap to achieve the serializability of VM_Transactions. This way TxnWrap addresses the maintenance anomaly problem not only under concurrent data updates but also under concurrent schema changes. Our TxnWrap solution can be plugged into any existing view maintenance system as a middle layer that removes concurrency issues from the view manager. As an added benefit, we found TxnWrap very easy to implement. The experimental results reveal that our solution achieves a rather stable maintenance performance. With even simple optimizations of the wrapper, TxnWrap already outperforms previous solutions from the literature [Agrawal et al. 1997; Zhang and Rundensteiner 2000] in terms of refresh rates. Another added benefit we find about the transactional approach towards view maintenance is that parallel processing can easily be added. The experimental results confirm the performance achievable by the parallel maintenance. A formal correctness proof for both TxnWrap and parallel TxnWrap is also provided based on transaction theory [Bernstein et al. 1987].

For the future work, we are beginning to look more closely at supporting multiple views with inter-dependencies between views, since our transactional-modeling

based approach removes the concurrency issues and simplifies the overall processing logic.

ACKNOWLEDGMENTS

The authors would like to thank Xin Zhang, Jun Chen, Andreas Koeller and all DSRG members at WPI for many valuable suggestions. The authors also thank the anonymous reviewers whose detailed comments helped greatly to improve the paper.

REFERENCES

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

Agrawal, D. and Sengupta, S. 1989. Modular Synchronization in Multiversion Databases. In *Proceedings of SIGMOD*. 408–417.

Bernstein, P. A., Hadzilacos, V., and Goodman, N. 1987. *Concurrency Control and Recovery in Database System*. Addison-Wesley Pub.

Chan, A. and Gray, R. 1985. Implementing Distributed Read-Only Transactions. *IEEE Trans. on Software Engineering 11*, 205–212.

Chen, J., Chen, S., and Rundensteiner, E. A. 2002. A Transactional Model for Data Warehouse Maintenance. In *Proceedings of Conceptual Modeling*. 247–262.

Chen, J., Zhang, X., Chen, S., Andreas, K., and Rundensteiner, E. A. 2001. DyDa: Data Warehouse Maintenance under Fully Concurrent Environments. In *Proceedings of SIGMOD Demo Session*. 619.

Chen, S., Chen, J., Zhang, X., and Rundensteiner, E. A. 2004. Detection and Correction of Conflicting Source Updates for View Maintenace. In *Proceedings of ICDE*. 436–448.

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

Gray, J. and Reuter, A. 1992. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann.

Gupta, A. and Mumick, I. 1995. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Engineering Bulletin 18(2)*, 3–19.

Gupta, A., Mumick, I., and Ross, K. 1995. Adapting Materialized Views after Redefinition. In *Proceedings of SIGMOD*. 211–222.

Lee, A. M., Nica, A., and Rundensteiner, E. A. 2002. The EVE Approach: View Synchronization in Dynamic Distributed Environments. *IEEE TKDE 14,* 5, 931–954.

Liu, B., Chen, S., and Rundensteiner, E. A. 2002a. A Transactional Approach to Parallel Data Warehouse Maintenance. In *Proceedings of DaWaK*. 307–317.

Liu, B., Chen, S., and Rundensteiner, E. A. 2002b. Batch Data Warehouse Maintenance in Dynamic Environments. In *Proceedings of CIKM*. 68–75.

Lomet, D. B. and Salzberg, B. 1989. Access Methods for Multiversion Data. In *Proceedings of SIGMOD*. 315–324.

Lu, J. J., Moerkotte, G., Schue, J., and Subrahmanian, V. S. 1995. Efficient Maintenance of Materialized Mediated Views. In *Proceedings of SIGMOD*. 340–351.

Madhavan, J., Bernstein, P. A., and Rahm, E. 2001. Generic schema matching with cupid. In *Proceedings of VLDB*. 49–58.

Marche, S. 1993. Measuring the Stability of Data Models. *European Journal of Information Systems 2,* 1, 37–47.

Miller, R. J., Haas, L. M., and Hernández, M. A. 2000. Schema mapping as query discovery. In *Proceedings of VLDB*. 77–88.

Mohan, C., H.Pirahesh, and Lorie, R. 1992. Efficient and Flexible Methods for Transient Versioning of Records to Avoid Locking by Read-only Transactions. In *Proceedings of SIGMOD*. 124–133.

Nica, A., Lee, A. J., and Rundensteiner, E. A. 1998. The CVS Algorithm for View Synchronization in Evolvable Large-Scale Information Systems. In *Proceedings of EDBT*. 359–373.

Nica, A. and Rundensteiner, E. A. 1999. View Maintenance after View Synchronization. In *International Database Engineering and Applications*. 213–215.

Quass, D., Gupta, A., Mumick, I. S., and Widom, J. 1996. Making Views Self-Maintainable for Data Warehousing. In *Conference on Parallel and Distributed Information Systems*. 158–169.

Quass, D. and Widom, J. 1997. On-Line Warehouse View Maintenance. In *Proceedings of SIGMOD*. 393–400.

Salem, K., Beyer, K. S., Cochrane, R., and Lindsay, B. G. 2000. How To Roll a Join: Asynchronous Incremental View Maintenance. In *Proceedings of SIGMOD*. 129–140.

Sjoberg, D. 1993. Quantifying Schema Evolution. *Information and Software Technology 35,* 1, 35–54.

Varde, A. S. and Rundensteiner, E. A. 2002. MEDWRAP: Consistent View Maintenance over Distributed Multi-Relation Sources. In *Proceedings of DEXA*. 341–350.

Velegrakis, Y., Miller, R. J., and Popa, L. 2003. Mapping Adaptation under Evolving Schemas. In *Proceedings of VLDB*. 584–595.

Widom, J. 1995. Research Problems in Data Warehousing. In *Proceedings of CIKM*. 25–30.

Zaniolo, C., Ceri, S., Faloursos, C., Snodgrass, R. T., Subrahmanian, V. S., and Zicari, R. 1997. *Advanced Database Systems*. Morgan Kaufmann Pub.

Zhang, X., Ding, L., and Rundensteiner, E. A. 2004. Parallel Multi-Source View Maintenance. *VLDB Journal 13,* 1, 22–48.

Zhang, X. and Rundensteiner, E. A. 2000. DyDa: Dynamic Data Warehouse Maintenance in a Fully Concurrent Environment. In *Proceedings of DaWaK*. 94–103.

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

Zhuge, Y., García-Molina, H., and Wiener, J. L. 1996. The Strobe Algorithms for Multi-Source Warehouse Consistency. In *Parallel and Distributed Information Systems*. 146–157.

Received August 2003; May 2004; accepted August 2004

THIS DOCUMENT IS THE ONLINE-ONLY APPENDIX TO:

Multiversion Based View Maintenance Over Distributed Data Sources

SONGTING CHEN, BIN LIU and ELKE A. RUNDENSTEINER
Worcester Polytechnic Institute

ACM Transactions on Database Systems, Vol. x, No. x, 12 2004, Pages 0–30.

A. PROOF OF TXNWRAP MAINTENANCE

Since TxnWrap is a specific kind of multiversion concurrency control algorithm, we thus proceed to prove the correctness of this concurrency control algorithm using the multiversion serializability theory. Below we first review several definitions as defined by [Bernstein et al. 1987].

In general, we use $r_i[x]$ (or $w_i[x]$) to denote the execution of a Read (or Write) operation issued by transaction T_i on a data item x. We also use c_i and a_i to denote T_i's Commit and Abort operations respectively.

DEFINITION 3. *A transaction T_i is a partial order of operations t_i with ordering relation $<_i$ where*

(1) $T_i \supseteq \{ r_i[x], w_i[x] \mid x$ *is a data item* $\} \bigcup \{ a_i,\ c_i \}$;
(2) $a_i \in T_i$ *iff* $c_i \notin T_i$;
(3) for $t \in T_i$*: if* t *is* c_i *or* a_i *(whichever is in* T_i*), for any other operation* $p \in T_i$, $p <_i t$*; and*
(4) if $r_i[x], w_i[x] \in T_i$*, then either* $r_i[x] <_i w_i[x]$ *or* $w_i[x] <_i r_i[x]$.

A.1 Classical Serializability Theory

DEFINITION 4. *Let* $T = T_1, \ldots, T_n$ *be a set of transactions. A complete history* H *over* T *is a partial order with ordering relation* $<_H$ *where:*

(1) $H = \bigcup_{i=1}^{n} T_i$;
(2) $<_H \supseteq (\bigcup_{i=1}^{n} <_i)$*; and*
(3) for any two conflicting operations $p, q \in H$*, either* $p <_H q$ *or* $q <_H p$.

A transaction T_i is *committed* (or *aborted*) in history H if $c_i \in H$ (or $a_i \in H$). Given a history H, the *committed projection of* H, denoted $C(H)$, is the history obtained from H by deleting all operations that do not belong to transactions committed in H.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 2004 ACM 0362-5915/2004/0300-00000 $5.00

Definition 5. *Let H be a history over $T = T_1, \dots, T_n$. The serialization graph (SG) for H, denoted SG(H), is a directed graph whose nodes are the transactions in T that are committed in H and whose edges are all $T_i \rightarrow T_j$ $(i \neq j)$ such that one of T_i's operations precedes and conflicts with one of T_j's operations in H.*

Theorem 2. *(The Serializability Theorem) A history H is* ***serializable*** *iff SG(H) is acyclic.*

A.2 Multiversion Serializability Theory

Let $T = T_1, \dots, T_n$ be a set of transactions, where the operations of T_i are ordered by $<_i$ for $1 \leq i \leq n$. Suppose a multiversion scheduler translates T's operations on (single version) data items into operations on specific versions of those data items. We formalize this translation by a function h that maps each $w_i[x]$ into $w_i[x_j]$, each $r_i[x]$ into $r_i[x_j]$ for some j, here x_j represents the jth version of x, each c_i into c_i, and each a_i into a_i.

Definition 6. *A complete multiversion (MV) history H over T is a partial order with ordering relation $<_{MV}$ where:*

(1) $H = h(\bigcup_{i=1}^{n} T_i)$ for some translation function h;

(2) for each T_i and all operations p_i, q_i in T_i, if $p_i <_i q_i$, then $h(p_i) <_{MV} h(q_i)$;

(3) if $h(r_j[x]) = r_j[x_i]$, then $w_i[x_i] <_{MV} r_j[x_i]$;

(4) if $w_i[x] <_i r_i[x]$, then $h(r_i[x]) = r_i[x_i]$; and

(5) if $h(r_j[x]) = r_j[x_i]$, $i \neq j$, and $c_j \in H$, then $c_i <_{MV} c_j$.

Condition (1) states that the scheduler translates each operation submitted by a transaction into an appropriate multiversion operation. Condition (2) states that the MV history of version-specific transactions preserves all orderings stipulated by the transactions. Condition (3) states that a transaction may not read a version until it has been produced. Condition (4) states that if a transaction writes into a data item x before it reads x, then it must read the version of x that it previously created. If H satisfies condition (4), we say that it *preserves reflexive reads-from relationships*. Condition (5) says that before a transaction commits, all the transactions that produced versions it read must have already committed. If H satisfies condition (5) we say it is *recoverable*.

Definition 7. *A complete MV history is* ***serial*** *if for every two transactions T_i and T_j that appear in H, either all of T_i's operations precede all of T_j's or vice versa. A serial MV history H is* ***one-copy serial (or 1-serial)*** *if for all i, j and x: if T_i reads x from T_j, then i=j, or T_j is the last transaction preceding T_i that writes into any version of x. An MV history is* ***one-copy serializable (or 1SR)*** *if its committed projection is equivalent to a 1-serial MV history.*

To justify the value of 1SR as a correctness criterion, we need to show that the committed projection of every 1SR is also equivalent to a serial 1V history.

Theorem 3. *Let H be a MV history over T. C(H) is equivalent to a serial 1V history over T iff H is 1SR.*

Since the serial 1V histories determine the correctness of a concurrency control algorithm, we reduce the proof of a multiversion concurrency control algorithm to the proof of the problem whether all of its MV histories are 1SR.

THEOREM 4. *A multiversion concurrency control algorithm is correct iff all its MV histories are 1SR MV histories.*

In order to determine whether a MV history is 1SR, the SG needs to be extended to MVSG.

DEFINITION 8. *Given a MV history H and a data item x, a version order $\ll$ for x in H is a total order of versions of x in H. A version order for H is the union of the version orders for all data items.*

DEFINITION 9. *Given a MV history H and a version order $\ll$, the multiversion serialization graph for H and $\ll$, MVSG(H, $\ll$), is SG(H) with the following version order edges added: for each $r_k[x_j]$ and $w_i[x_i]$ in C(H) where i, j, and k are distinct, if $x_i \ll x_j$ then include $T_i \rightarrow T_j$, otherwise include $T_k \rightarrow T_i$.*

This definition says that if $r_k[x_j]$ and $w_i[x_i]$ are in C(H), then the version order edge forces $w_i[x_i]$ to either precede $w_j[x_j]$ or to follow $r_k[x_j]$ in every serial history of H.

THEOREM 5. *A MV history H is 1SR iff there exists a version order $\ll$ such that MVSG(H, $\ll$) is acyclic.*

A.3 Correctness Proof of TxnWrap

In the TxnWrap context, we assume n data sources, denoted as DS_1, DS_2,$\cdots$, DS_n. We use $DS_i(j)$ to denote the state of DS_i after having been updated by a source update transaction with *local id* j in DS_i. We refer to this as the jth version of DS_i. In this context, we annotate all initial states of data sources by the version number 0. The transaction identifier TxnID τ of a VM_Transaction T is generated by the view manager as soon as the update message arrives. We use $\tau_1, \tau_2, \cdots, \tau_m$ to denote TxnIDs. Similarly, we use V(τ_i) to denote the view state after the maintenance result has been committed to the view extent by the VM_Transaction with TxnID τ_i ($1 \leq i \leq$ n). For simplicity, we use V(0) to denote the initial state of the view extent based on the initial states of all data sources.

As in [Bernstein et al. 1987], we use $r_{\tau_i}[x]$ (or $w_{\tau_i}[x]$) to denote the execution of a Read (or Write) operation issued by transaction T_{τ_i} on a data item x. We use c_{τ_i} to denote the commit operation of T_{τ_i}. Notice that we don't have any abort operation for the VM_Transactions (See Section 2.3 for the explanation). We use x_{τ_i} to denote any version data that is being accessed by the VM_Transaction with TxnID τ_i. The order of the version data x can be defined based on the order of the TxnID, $x_{\tau_i} \ll x_{\tau_j} \Leftrightarrow \tau_i < \tau_j$, where τ_i, τ_j are TxnIDs of the corresponding VM_Transactions.

The following is an illustrative example. Assume three VM_Transactions are generated in the view manager. T_{τ_1} is for the source update in DS_h with local id 1, T_{τ_2} is for the update in DS_k with local id 1, and T_{τ_3} again is for in DS_h with local id 2, assume $1 \leq h, k \leq n$ with n the number of data sources. For simplicity, we assume the initial versions of the data source extents stored in their respective

wrappers are $DS_1(0), DS_2(0), \ldots, DS_n(0)$ and the inital view extent is V(0). These three VM_Transactions can be represented as follows:

(1) $T_{\tau_1} = w_{\tau_1}[DS_h(1)]r_{\tau_1}[DS_1(0)]\ldots r_{\tau_1}[DS_k(0)]\ldots r_{\tau_1}[DS_n(0)]w_{\tau_1}[V(\tau_1)]c_{\tau_1}$
(2) $T_{\tau_2} = w_{\tau_2}[DS_k(1)]r_{\tau_2}[DS_1(0)]\ldots r_{\tau_2}[DS_h(1)]\ldots r_{\tau_2}[DS_n(0)]w_{\tau_2}[V(\tau_2)]c_{\tau_2}$
(3) $T_{\tau_3} = w_{\tau_3}[DS_h(2)]r_{\tau_3}[DS_1(0)]\ldots r_{\tau_3}[DS_k(1)]\ldots r_{\tau_3}[DS_n(0)]w_{\tau_3}[V(\tau_3)]c_{\tau_3}$

For the sequential schedule, below is a sample history H_1 of T_{τ_1}, T_{τ_2} and T_{τ_3}:

—$H_1{=}w_{\tau_1}[DS_h(1)]w_{\tau_2}[DS_k(1)]w_{\tau_3}[DS_h(2)]$
$r_{\tau_1}[DS_1(0)]\ldots r_{\tau_1}[DS_k(0)]\ldots r_{\tau_1}[DS_n(0)]w_{\tau_1}[V(\tau_1)]c_{\tau_1}$
$r_{\tau_2}[DS_1(0)]\ldots r_{\tau_2}[DS_h(1)]\ldots r_{\tau_2}[DS_n(0)]w_{\tau_2}[V(\tau_2)]c_{\tau_2}$
$r_{\tau_3}[DS_1(0)]\ldots r_{\tau_3}[DS_k(1)]\ldots r_{\tau_3}[DS_n(0)]w_{\tau_3}[V(\tau_3)]c_{\tau_3}$

For the parallel schedule, we may also interleave the execution of the view maintenance transactions. The following is a sample history H_2 of such a schedule [8].

—$H_2{=}w_{\tau_1}[DS_h(1)]w_{\tau_2}[DS_k(1)]w_{\tau_3}[DS_h(2)]$
$r_{\tau_1}[DS_1(0)]\ldots r_{\tau_1}[DS_k(0)]\ldots r_{\tau_2}[DS_1(0)]\ldots r_{\tau_2}[DS_h(1)]r_{\tau_1}[DS_n(0)]w_{\tau_1}[V(\tau_1)]c_{\tau_1}$
$r_{\tau_3}[DS_1(0)]\ldots r_{\tau_3}[DS_k(1)]\ldots r_{\tau_3}[DS_n(0)]r_{\tau_2}[DS_n(0)]w_{\tau_2}[V(\tau_2)]c_{\tau_2}w_{\tau_3}[V(\tau_3)]c_{\tau_3}$

For both histories, we can construct the MVSG as follows. We add the edge $T_{\tau_1} \rightarrow T_{\tau_2}$ since $r_{\tau_2}[DS_h(1)]$ reads the result from $w_{\tau_1}[DS_h(1)]$; add the edge $T_{\tau_2} \rightarrow T_{\tau_3}$ since $r_{\tau 3}[DS_k(1)]$ reads the result from $w_{\tau_2}[DS_k(1)]$; add the edge $T_{\tau_1} \rightarrow T_{\tau_3}$ since $w_{\tau_1}[DS_h(1)]$ precedes $w_{\tau_3}[DS_h(2)]$. No other version order edges exist. Thus G is an acyclic MVSG graph in this example.

We now prove the correctness of both type of schedules by contradiction. We assume that the view manager keeps all arriving update messages in a queue. At any time t, each update message in this queue represents a VM_Transaction. Thus, we denote this set of VM_Transactions as T = $T_{\tau_1}, T_{\tau_2}, \ldots, T_{\tau_m}$, with $\tau_1, \tau_2, \ldots, \tau_m$ their corresponding TxnIDs.

THEOREM 6. *Given any VM_Transaction set queueing in the view manager, the multiversion serializable graph G of any history over this set of VM_Transactions is acyclic. More strictly, all version order edges in G are pointing from the VM_Transaction T_{τ_i} with TxnID τ_i towards another VM_Transaction T_{τ_j} with TxnID τ_j and $\tau_j > \tau_i$.*

Proof: We prove this by contradiction.

(1) Assumption: There is one version order edge, $T_{\tau_i} \leftarrow T_{\tau_j}$ with TxnID $\tau_i < \tau_j$.
(2) There are two possible reasons for adding this edge to the MVSG graph.
 (a) If this edge is added due to the serial graph (SG) definition [Bernstein et al. 1987], this can't be true because in TxnWrap and in parallel TxnWrap algorithm, we always assign a TxnID only to a transaction after its corresponding versions have been built.
 (b) If this edge is added by the additional MVSG definition [Bernstein et al. 1987], then there are only two cases to consider:
 —$w_{\tau_j}[x_{\tau_j}]\ldots r_{\tau_k}[x_{\tau_i}]$ and $x_{\tau_j} \ll x_{\tau_i}$. This is impossible because we assume the version order $x_{\tau_i} \ll x_{\tau_j}$ if $\tau_i < \tau_j$.

[8] Assume that we use a strict commit control strategy.

—$r_{\tau_j}[x_{\tau_k}] \ldots w_{\tau_i}[x_{\tau_i}]$ and $x_{\tau_k} \ll x_{\tau_i}$. That is, T_{τ_j} reads some data item whose version is earlier than that of the same data written by T_{τ_i}. This is also impossible in both TxnWrap and parallel TxnWrap because we always assign the latest version number (local id) to the VM_Transaction in the TxnID.

(3) Contradiction: Since all cases lead to contraditions, the assumption is not correct. Thus, there is no version order edge $T_{\tau_i} \leftarrow T_{\tau_j}$ with $\tau_i < \tau_j$. So the MVSG is acyclic.

Thus, the maintenance of a set of VM_Transactions $T_{\tau_1}, T_{\tau_2}, \ldots, T_{\tau_m}$ using both TxnWrap schedulers has the identical effect to maintaining each VM_Transaction sequentially as if no concurrency among source updates exists. That is, the current VM_Transaction completes before the next source update occurs which in turn generates the next VM_Transaction. Seen from the above discussion, TxnWrap reaches a strong consistency level in terms of the consistency levels defined in [Zhang et al. 2004; Zhuge et al. 1995]. If every VM_Transaction contains only one data source transaction, then this will even achieve complete consistency.