WPI-CS-TR-00-26 December 2000

TxnWrap: A Transactional Approach to Data Warehouse Maintenance

by

Jun Chen
Songting Chen
Elke A. Rundensteiner

(Revised on Feb. 2002)

Computer Science
Technical Report
Series

WORCESTER POLYTECHNIC INSTITUTE

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

A Transactional Model for Data Warehouse Maintenance [*]

Jun Chen, Songting Chen and Elke A. Rundensteiner
Department of Computer Science
Worcester Polytechnic Institute
Worcester, MA 01609-2280
(junchen | chenst | rundenst)@cs.wpi.edu

Abstract

A Data Warehouse Management System (DWMS) maintains materialized views by issuing maintenance queries to the data sources. To address the erroneous query results caused by concurrent source updates, state-of-the-art maintenance strategies typically apply compensations to resolve the conflicts assuming all source schemata remain stable over time. However, if schema changes occur in any of the data sources, then a maintenance anomaly problem arises, namely, the maintenance or the compensation queries may be broken.

We now tackle this open problem by modeling the complete data warehouse maintenance process as a special transaction, DWMS_Transaction. The anomaly problem can now be rephrased as the serializability of DWMS_Transactions. This way allows us to apply well-established transaction theory to address the new maintenance problem. To achieve DWMS_Transaction serializability, we propose a multiversion concurrency control technique appropriate for loosely-coupled environments with autonomous data sources. TxnWrap is complementary to the diverse maintenance algorithms proposed in the literature, since it removes concurrency issues from their considerations allowing them to focus on the maintenance logic.

We have implemented TxnWrap and plugged it into an existing DWMS testbed. The experimental results confirm that TxnWrap achieves predictable performance under a varying rate of concurrency. Furthermore, we show that several optimizations of TxnWrap are easily achievable due to its footing in transaction theory, in particular, (1) space optimizations on versioned data materialization, and (2) parallel maintenance scheduling. With those optimizations, TxnWrap even outperforms state-of-the-art data warehouse maintenance solutions in terms of refresh time.

1 Introduction

1.1 Data Warehouse Environments

Data warehouses (DW) [8, 1] are built by gathering data from data sources and integrating it into one virtual repository customized to users' needs. One important task of a Data Warehouse Management System (DWMS) is to maintain the DW upon source changes, since frequent updates are common for most data sources. In addition, beyond frequent data updates, we find that schema changes are fairly common for modern applications. A schema change could occur for numerous reasons during the software life-cycle,

[*]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.

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 [13], significant changes (about 59% of attributes on the average) were reported for seven different applications ranging from project tracking, sales management to government administration. A similar report can also be found in [20].

Furthermore, in distributed environments, the data sources are typically owned by different information providers and hence function independently from one another [23]. This implies that they will update their data and schemas autonomously without any concern for how this may affect the materialized views defined upon them [21, 10], causing so called *DW maintenance anomalies.* These are exactly the problems we address in this work.

1.2 Motivating Example of the DW Maintenance Anomaly Problems

We will now illustrate the anomaly problems via motivating examples. We distinguish between three DW maintenance tasks, namely, View Maintenance (VM), View Synchronization (VS) and View Adaptation (VA). VM [23, 1, 19] maintains the DW view extent under source data updates. In contrast to VM, VS [10, 15] aims at rewriting the DW view definition when the source schema has been changed. Thereafter, View Adaptation (VA) [16, 9] incrementally adapts the view extent to again match the newly changed view definition.

If there is no concurrency among source updates, namely, the current DW maintenance completes before the next source update occurs, then VM incorporates each source *data update* (DU) while VS and VA together incorporate the source *schema change* (SC) into the data warehouse. 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 (DSs) as shown in Figure 1.*

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.

Figure 1: Description of Data Sources.

The view Asia-Customer of the data warehouse is defined by the SQL query in Equation (1). Assume the data update " ΔC *= insert ('Ben','MA',123456) into the Customer relation at DS[1]". In order to determine* *the delta effect on the DW extent, this now requires us to send the incremental view* **maintenance query** Q *[23] defined in Equation (2) down to the FlightRes relation at DS[4].*

$$\begin{array}{ll} \mathsf{CREATE\ \ VIEW} & Asia-Customer\ \mathsf{AS} \\ \mathsf{SELECT} & C.Name,\ F.Age, \\ & F.FlightNo,\ F.Dest \\ \mathsf{FROM} & Customer\ C,\ FlightRes\ F \\ \mathsf{WHERE} & C.Name\ =\ F.Name \\ & AND\ F.Dest\ =\ 'Asia' \end{array} \tag{1}$$

$$\begin{array}{ll} \mathsf{SELECT} & 'Ben'\ as\ Name,\ F.Age,\ F.FlightNo,\ F.Dest \\ \mathsf{FROM} & FlightRes\ F \\ \mathsf{WHERE} & F.Name\ =\ 'Ben' \\ & AND\ F.Dest\ =\ 'Asia' \end{array} \tag{2}$$

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

- **Duplication 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 ('Ben', 18, 'AA3456', 'Asia'). This new tuple would also be included in the join result of Q and the extra tuple ('Ben', 18, 'AA3456', 'Asia') would be inserted into the view. However, later when the DW starts processing* ΔF *= insert('Ben', 18, 'AA3456', 'Asia'), the same tuple would be inserted into the view again. A duplication anomaly appears, as has also been observed by [23].*

- **Broken Query 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, i.e., a broken query anomaly appears.*

As illustrated by the example above, sources being autonomous may commit updates that are concurrent with and hence may conflict with the DW 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, i.e., an error message is returned.

1.3 Contributions of this Work

Our contributions in this work are:

1. We illustrate that the anomaly problems in Section 1.2 can be reformulated as a transaction problem by modeling the complete maintenance process as a transaction model, *DWMS_Transaction*. Based on our

DWMS_Transaction model, we demonstrate that the data warehouse maintenance anomaly problems can be mapped to the problem of guaranteeing the serializability property of *DWMS_Transactions*.

2. We then introduce a multiversion concurrency control algorithm to achieve such serializability within our data warehouse context of autonomous data sources. We also prove the correctness of this algorithm.

3. To the best of our knowledge, *TxnWrap* is the first work to put the existing data warehouse maintenance works in the literature such as view maintenance (VM), synchronization (VS) and adaptation (VA) algorithms into the context of a sound theory, i.e., serializability theory, to remove concurrency concerns from them.

4. We have implemented the *TxnWrap* solution and integrated it into a fully functional data warehousing system. Experiments comparing maintenance performance with and without *TxnWrap* enabled confirm that *TxnWrap* avoids the maintenance anomaly problems and hence achieves a more stable performance.

5. We also explore several optimizations of TxnWrap easily achievable due to our transaction model, namely, wrapper materialization optimizations and parallel maintenance processing. Together these achieve improvements of the system performance beyond that of previous state-of-the-art DW maintenance solutions.

The outline of the rest of the paper is as follows. The transaction model called *DWMS_Transaction* is introduced in Section 2. A multiversion concurrency control algorithm to handle the conflicts between *DWMS_Transactions* is proposed in Section 3. Section 4 discusses the architectural solutions for versioned data materialization. Section 5 presents experimental results. In Section 6 we review the related work while Section 7 concludes the paper.

2 DWMS_Transaction: Transactional Modeling of DW Maintenance

We first analyze the DW maintenance anomaly problems. Typically, the DW maintenance process involves reads from individual data sources to calculate and ultimately refresh the DW 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 prcess would cause a concurrency problem - similar to the DW 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 [7]. The ACID properties assure a consistent view of all data inside each transaction and hence solve the problem.

Similarly, in our DW context, the DW 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 clearly 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 Data Warehouse Environment

In a DW environment, each *DW maintenance process* is composed of the following two 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 on DS_i and i is the index of the data source, c(s) means the commit.

2. A *DW maintenance transaction* that computes the delta effects caused by source updates in order to refresh the DW. During this period, it may need to send maintenance queries down to different sources to get additional information. After that, the DWMS refreshes the DW extent according to the maintenance result. We denote this as "$T_{DW} = r(DS_1)r(DS_2)...r(DS_n)w(DW)c(DW)$", where $r(DS_i)$ denotes reading from DS_i, w(DW) denotes writing or refreshing the DW extent and c(DW) denotes the commit.

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 *DW maintenance transaction*. Furthermore, this *DW maintenance transaction* is required to see a consistent state of all sources, so not to conflict with other *source update transactions*. Otherwise the anomaly as shown in Section 1.2 may occur.

These observations give us the hint that a high-level transaction that integrates both of these two sub-transactions may be needed. The ACID properties of this high-level transaction could resolve the conflicts between its sub-transactions, namely *DW maintenance transactions* and *source update transactions.*

2.2 A Transactional Model: DWMS_Transaction

Below we model this high-level transaction that integrates both a *source update transaction* and its corresponding *DW maintenance transaction* into what we will call a **DWMS_Transaction**.

Definition 1 *We model the complete DW maintenance process in Section 2.1 as a DWMS_Transaction. Each DWMS_Transaction T starts after a local successfully committed* source update transaction T_s*, and commits when the data warehouse database has been successfully refreshed, i.e., the commit of the* DW maintenance transaction T_{DW}*. We denote it as "*$T = T_s T_{DW} = w(DS_i)c(s)r(DS_1)r(DS_2)...r(DS_n)w(DW)c(DW)$*".*

Previous research has already hinted at this notion of a "global view of data warehouse transactions" [24]. However, [24] 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 note that the nested structure of a *DWMS_Transaction* as indicated in Definition 1 can be exploited to avoid such a wait. We adopt this idea of taking a transactional approach towards data warehousing and put forth the following supporting arguments:

1. The *DWMS_Transaction* model is a **conceptual** rather than a real transaction model. It encapsulates two types of transactions, namely, *source update transaction* and its corresponding *DW maintenance transaction.* We observe that the *source update transaction* can in fact commit autonomously as a sub-transaction within the complete *DWMS_Transaction*, and thus sources never need to wait.

2. With the help of the *DWMS_Transaction* concept, the DW maintenance anomaly problem [23] can be rephrased as the well-known "read dirty data" transaction problem. With an appropriate concurrency control solution to achieve the ACID properties of such a *DWMS_Transaction*, the conflicts between the *DWMS_Transactions*, namely, the *DW maintenance transactions* and *source update transactions* can naturally be solved (and thus the DW maintenance anomaly problems are solved).

3. Lastly, this formal transaction-based solution lays a solid foundation for the future treatment of as of now unsolved problems including recovery or parallel processing of DW maintenance.

2.3 Start, Commit, Abort and Rollback of DWMS_Transactions

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

2.4 DWMS_Transaction Scheduling

Two types of DWMS_transaction schedulers are possible, namely, sequential and parallel ones. A sequential scheduler allows no other *DW maintenance transactions* to be processed until the current one has been completed. Most DW maintenance algorithms in the literature [23, 1] employ such sequential scheduling for DW maintenance even if they do not utilize the transaction terminology. Parallel scheduling is more challenging, since out-of-order DW maintenance processing introduces additional conflicts. In this paper, we will first introduce our solution assuming a sequential scheduler to keep the description simple, and later we demonstrate that our transactional model can be easily extended to support such a parallel scheduler. This is in effect one advantage of adopting a transactional model for data warehouse maintenance.

2.5 Conflicts between DWMS_Transactions

We now examine the operations encapsulated within a *DWMS_Transactions* to identify potential conflicts between them.

Observation 1 *From Definition 1, we know that there are three basic read and write operations within a* DWMS_Transaction, *namely, $w(DS_i)$, $r(DS_i)$ and $w(DW)$. There are three types of conflicts between these operations. First, some write operation $w_1(DS_i)$ in DWMS_Transaction T_1 conflicts with $w_2(DS_i)$ in another DWMS_Transaction T_2. Second, some write operation $w_1(DW)$ in DWMS_Transaction T_1 conflicts with $w_2(DW)$ in another DWMS_Transaction T_2. Obviously such write-write conflicts would naturally be taken care of by the local DBMS of the sources and DW, respectively. The third conflict is the read-write conflict, namely, one read operation $r_1(DS_i)$ of DWMS_Transaction T_1 may read some inconsistent query result written by $w_2(DS_i)$ of another DWMS_Transaction T_2.*

Lemma 1 *The DW maintenance anomaly problems illustrated in Section 1.2 arise due to the read-write conflicts of DWMS_Transactions.*

The proof of this lemma is straightforward. Hence, with Lemma 1, we rephase the DW maintenance anomaly problem in terms of the read-write conflicts of DWMS_Transactions. By this lemma, as long as we can resolve such read-write conflicts, we solve the DW maintenance anomaly problem.

2.6 Serializability of DWMS_Transaction

Lemma 1 leads us to a solution approach to resolve the conflicts, namely, we have to guarantee the serializability of DWMS_Transactions.

Theorem 1 *A history of DWMS_Transactions S is serializable iff it is equivalent to some serial schedule S_1 of the same DWMS_Transactions.*

Note that a *serial* schedule of DWMS_Transactions S_1 assumes the source update and maintenance process occurs one after another. Obviously no DW maintenance anomaly would occur in this case. Consequently, by Theorem 1, no anomaly would happen in its equivalent history S either. Hence, Theorem 1 describes the correctness criterion of the execution of *DWMS_Transaction*s. In the next section, we will propose a concurrency control strategy to achieve such serializability for *DWMS_Transaction*s.

3 Multiversion Concurrency Control Algorithm for DWMS_Transactions

For the serializability for traditional transactions, generally, two basic kinds of concurrency control algorithms [3], namely, **lock-based** or **multiversion-based**, can be used. Below we examine which of these two strategies may be appropriate to achieve the serializability of the *DWMS_Transactions* in the loosely-coupled DW context.

The traditional **lock-based** concurrency control algorithms [3] work as follows: For every write, an exclusive write lock is obtained and released at the end of transaction; and for every read, a read lock is obtained, and after the read operation completes, the read lock can be released. This is too restrictive for the schedule of *DWMS_Transactions*. By Definition 1, we know that DWMS_Transaction will read over all sources hence a read lock over all sources is required. However, the sources in the DW environment are autonomous and would not be willing to accept locks from external agents as stated in Section 1.1. Hence locking algorithms are not suitable in the DW environment.

In the traditional **multiversion** concurrency control algorithms [3], the reader will read an older version than the version that the writer is currently writing to, thus avoiding the conflicts. In particular, we further differentiate between two categories of multiversion algorithms, i.e., finite version [18, 14] and unrestricted version [4, 2] 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" [18, 14] 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 appropriate versions 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 for DW maintenance. First, the version that the reader (DW maintenance transaction) needs should never expire before the maintenance process has been completed, otherwise the DW view would be inconsistent. Second, the autonomous writer (source update) is unlikely to synchronize with the reader (DW maintenance process). For this reason, we favor the choice of an unrestricted multiversion algorithm. Also note that, since the *DWMS_Transaction*s 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 later.

Based on the above discussion, we now introduce an unrestricted multiversion algorithm, called *ShadowWrapper*. The main idea is to extend the wrapper functionality beyond just communicating between sources and DW [1]. That is, the wrapper is responsible for storing the versions generated due to the source updates and thus answering any DW maintenance query using this versioned information. Below, we describe our multiversion algorithm called *ShadowWrapper*, in particular, first the wrapper materialization of sources at initialization time and then version management under both update notifications and query requests.

[1]Note that, this is just one of the three architectural choices for where to keep the versioned data (see Section 4). Without loss of generality, here we select one of them to describe the main idea.

3.1 ShadowWrapper: 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 more source access. However we can further improve upon this by utilizing the view definitions of the DW to filter out unnecessary data. For example, we can simply "push down" the Select or Project operations in the DW view definitions to filter out unnecessary tuples or attributes (see Section 5.4.).

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 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 DWMS_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. The pair of attributes "Rel'" and "$Attr'$" indicate the new name of a particular attribute or of a relation. Here at the meta data relation, "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. Two source wrappers related to the view in Example 1 are shown in Figures 2 and 3, respectively.

Wrapper for DS_1

Relation C'

Name	V_Start	V_End
Tom	0	∞
Bob	0	∞

Meta Relation

Rel	Attr	Rel'	Attr'	V_Start	V_End
C'	Name	-	-	0	∞

Figure 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	∞

Figure 3: Initial Extent of Wrapper of DS_4

3.2 Version Operations of the ShadowWrapper

Basically, there are three classes of version operations the wrapper must perform. First, the *ShadowWrapper* needs to create versions whenever an update is reported by a source. Second, the *ShadowWrapper* needs to read the appropriate versions to answer each maintenance query issued from the DWMS. Third, the *ShadowWrapper* needs to clean up any versions that are no longer required by any *DWMS_Transaction*.

3.2.1 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, each *DWMS_Transaction* will be assigned a version number for creation of new version based on its corresponding source update transaction. The corresponding version creations 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.

For example, assume there are three source updates as shown in Figure 4 on the relations depicted in Figure 1. After the version creation step shown above, the wrapper relation and meta relation are shown in Figures 5 and 6.

DS Transaction1 (DWMS_Transaction Version id = 1)

1. INSERT INTO Customer('Ben', 'CA', '2137461928');

DS Transaction2 (DWMS_Transaction Version id = 2)

1. INSERT INTO FlightRes('Ben', 18, 'AA3456', 'Asia');

DS Transaction3 (DWMS_Transaction Version id = 3)

1. Rename Customer.Name to Customer.First-Last;

Figure 4: Three Source Updates

Wrapper for DS_1

Relation C'

Name	V_Start	V_End
Tom	0	∞
Bob	0	∞
Ben	1	∞

Meta Relation

Rel	Attr	Rel'	Attr'	V_Start	V_End
C'	Name	C'	First-Last	0	3
C'	First-Last	-	-	3	∞

Figure 5: Wrapper Content for DS_1 after Updates

3.2.2 Version Read in the Wrapper

The read version operation happens when a DWMS_Transaction wants to query a source. Rather than being processed by the source, the queries are actually rewritten by the wrapper and executed by reading appropriate versions of the wrapper schema and relations. Notice that the DWMS_Transaction requires to see a consistent state of all sources. Hence the versions created afterwards should be invisible to this DWMS_Transaction as suggested by Section 1.2. To achieve this, the original maintenance query is rewriting by augmenting it with the version conditions *"AND V_Start <= i AND i < V_End"* and it will be evaluated on the *wrapper relation.* Here i is the version number of this DWMS_Transaction. For example, Figure 7 depicts the rewritten versioned maintenance query for the first two source updates in Figure 4.

Wrapper for DS_4

Relation F'

Name	Age	FlightNo	Dest	V_Start	V_End
Tom	17	AA4399	Asia	0	∞
Ben	18	AA3456	Asia	2	∞

Meta Relation

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

Figure 6: Wrapper Content for DS_4 after Updates

DWMS_Transaction with id = 1:

```
SELECT Name, Age, FlightNo, Dest FROM F'
    WHERE Name = 'Ben' AND V_Start <= 1 and V_End > 1;
```

DWMS_Transaction with id = 2:

```
SELECT Name FROM C' WHERE
    Name = 'Ben' AND V_Start <= 2 and V_End > 2;
```

Figure 7: Version Queries at Wrapper

3.2.3 Version Cleanup in the Wrapper

Finally, we need to clean up versions that are no longer needed by any maintenance transactions. For a sequential scheduler, the cleanup is straightforward. That is, after the completion of a DWMS_Transaction with version number "i", all the data with V_End less than "i" will be deleted since they are no longer visible nor required by any later maintenance transaction. To achieve this, in the meta relation, we execute the update as *"delete * from meta-table where V_End <= i"* and perform corresponding operations on these attributes in the wrapper relation. Then we also delete the tuples in the wrapper relations with $V\_End < i$. For a parallel scheduler, intuitively, we have to assure all DWMS_Transactions with version number less than "i" are completed before version removal (see [11]).

3.3 ShadowWrapper: A Multiversion Algorithm for DW Maintenance

We are now ready to introduce our ShadowWrapper concurrency control algorithm for DW maintenance.

1. First, the ShadowWrapper initializes the wrapper schema and relations as defined in Section 3.1.

2. When a source commits a local update transaction and reports its updates to the wrapper, the *ShadowWrapper* first gets a unique global *DWMS_Transaction* version id from the DWMS and uses it to create versions in the wrapper. After that, the wrapper notifies the DWMS about these updates.

3. When the wrapper receives a maintenance query from the DWMS, then ShadowWrapper rewrites the maintenance query according to the version id of the corresponding DWMS_Transaction and executes it upon the wrapper relation.

4. When a *DWMS_Transaction* is committed in the DW, 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.

Theorem 2 *A history of DWMS_Transactions scheduled by the ShadowWrapper is serializable.*

The proof of Theorem 2 is presented in Appendix. By Theorem 1, we know that the DW maintenance is correct.

3.4 Consistency Level of the DW

We adopt the definitions of correctness and consistency levels of the DW from [23].

- **Correctness:** Any state of DW corresponds to a valid source state.
- **Convergence:** Source updates will be incorporated into the DW resulting in a valid state.
- **Strong Consistency:** The states of DW are valid and the order of DW transitions corresponds to the order of the state transitions at data sources.
- **Complete Consistency:** Strong consistency holds. Each state of sources is reflected by a distinct DW state.

Now let's examine which of the DW consistency levels our system reaches. It can easily be seen that if every *DWMS_Transaction* contains only one *source update transaction*, then our solution will achieve "complete consistency". This is so because for each *source update transaction*, there is a corresponding *DWMS_Transaction* that brings the DW into a consistent and distinct state. However, it is sometimes preferred to process the *source update transactions* in a batched fashion to improve performance [19]. Intuitively by merging updates, there would be less maintenance queries. In a batch case, several *source update transactions* would be included in one *DWMS_Transaction* and thus only "strong consistency" can be reached.

4 TxnWrap Version Placement Design Choices

In the previous section, our concurrency control strategy, *ShadowWrapper*, will materialize versioned data in a special-purpose source wrapper. Clearly it's not the only design choice for the versioned data. Alternative choices include to either (1) add extra version information directly into the sources or (2) to materialize the versioned data in the DW itself. We now briefly discuss the tradeoffs between these version placement design options.

First, let us consider the case of adding version information directly into each source, i.e., two version attributes as described in Section 3.1 are directly added to the source relations. Clearly, the space utilization of this option is optimal, because no duplication of data is needed. However, the source update transactions may now be affected in terms of performance due to having to deal with the version information. Note that the source update transactions will remain autonomous without blocking from the DWMS.

Second, we could materialize the versioned relations inside the DW. This solution has two advantages. First, instead of decomposing the DW view query to individual small queries to be submitted to each source, we could use only one combined maintenance query because these relations would now all be within the same local DW database. This way, DW self-maintenance would be possible even when some sources may be temporarily unavailable. In fact, a similar approach has been proposed by [17]. However, the space cost for versioning at DW is obviously large now. To help this, besides the basic space optimizations like pushing "Select" and "Project", [17] suggests to utilize the foreign keys or functional dependencies to further optimize the storage. Both are also appliable here. Note that the DWMS now would have to maintain both DW and these versioned data as well as answer DW user queries, which may become a heavy burden for the DWMS. In comparison, placing the versioned data into a dedicated wrapper achieves the independence and simplicity of both the sources and the DW by moving the complex version management into a third place.

5 Experimental Evaluation

5.1 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 [5]. The basic DyDa system uses SWEEP [1], 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 [22]. By disabling all 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* contains less than 900 lines of code. In this section, we experimentally evaluate *TxnWrap* by comparing these two systems.

In our experimental study, we 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 PC machines. Each relation has four attributes and 100000 tuples. There is one materialized join view defined upon these six source relations residing on a separate machine. Currently, we assume each *DWMS_Transaction* contains only one update. If we were to merge updates in a batch fashion [19], the performance would be even better since less maintenance queries would be required.

5.2 Data Update Processing

In this experiment, we evaluate the overhead of *TxnWrap* and compare it against the SWEEP [1] for data update processing under a varying rate of concurrency compared to the algorithm. SWEEP applies local compensation to remove the *duplication anomaly* as shown in Section 1.2. Figure 8 shows the average maintenance cost of one data update (on the y-axis) under a number of concurrent data updates for both systems.

From Figure 8, we can see that initially the maintenance cost of TxnWrap is larger than that of SWEEP when the number of concurrent data update is 0 (i.e., there is no concurrent data update). This is mainly caused by the versioned maintenance query, which can be attributed to the 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. In comparison, *TxnWrap* exhibits an almost fixed cost for handling of data updates independent of the number of concurrent data updates, and thus achieves more steady performance.

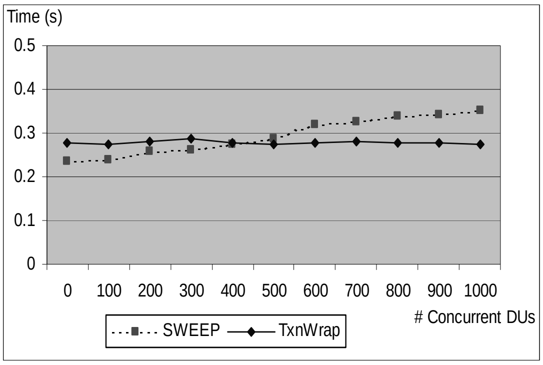

Figure 8: Concurrent Data Update Processing

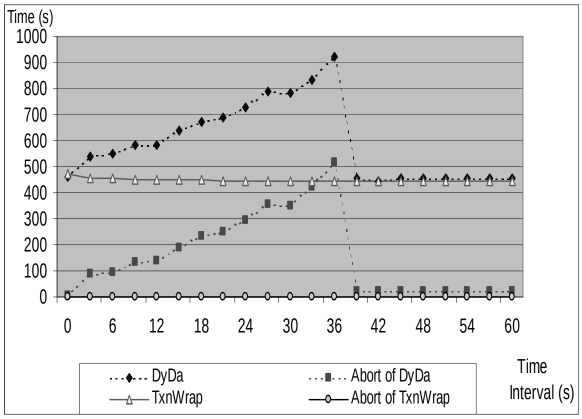

Figure 9: Concurrent Schema Change Processing

5.3 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 by employing twenty schema changes but 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 9 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 [2]. Eventually when the time interval is larger than the maintenance time, the updates won't conflict with each other thus both systems have the same maintenance cost. In comparison, *TxnWrap*, employing a multiversion algorithm, avoids any abort and hence again achieves a steady performance.

5.4 Impact of Version Management Optimizations

The version management assuming a full duplication of a source relation requires significant storage space. We have explored several optimization strategies. *Selection Optimization* extracts the local condition in a DW view definition, pushes it down to wrapper relation definition and hence filters tuples. Thus it reduces the wrapper size. For example, in Example 1, by taking the local condition of view definition, i.e., "F.Dest = 'Asia'" into consideration, we can reduce the wrapper space. This results in the additional advantage that some data updates with F.Dest<> 'Asia' will not even be reported to the DW, thus avoiding some

[2] The explanation of the curves of the DyDa system could be found in [22].

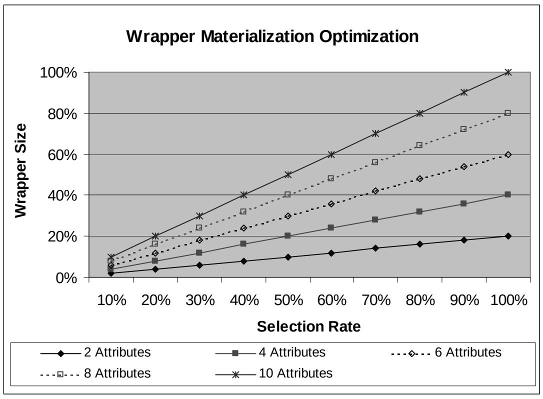

Figure 10: Wrapper Materialization Optimization

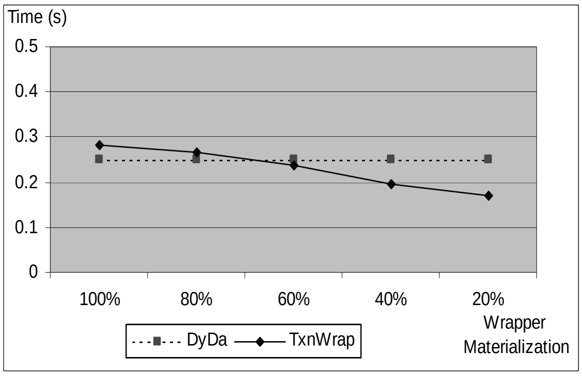

Figure 11: Data Update Processing with Wrapper Materialization Optimization

unnecessary maintenance work. *Projection Optimization* projects any attribute of the source not involved in the view definition from the wrapper. We can combine the above two approaches to achieve even better space utilization. Assume we have a source relation containing 10 attributes with the same length. We project several of them from the wrapper. We also apply the local condition of the view definition to select a subset of the tuples. We find that the combination of both optimizations can greatly reduce the wrapper size as depicted in Figure 10.

We now evaluate the average maintenance cost of one data update or one schema change when applying the above optimizations. As shown in Figures 11 and 12, after space optimization, the size of the wrapper is smaller than that of the source. Consequently, the maintenance query cost decreases.

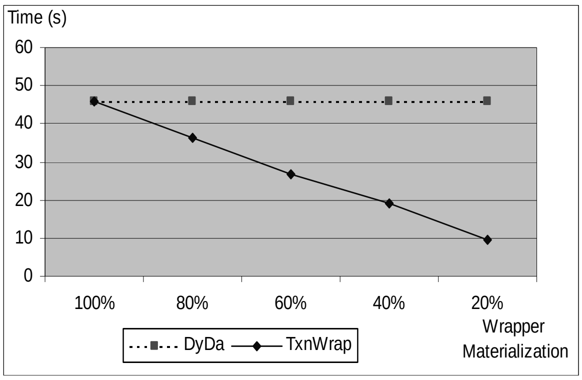

Figure 12: Schema Change Processing with Wrapper Materialization Optimization

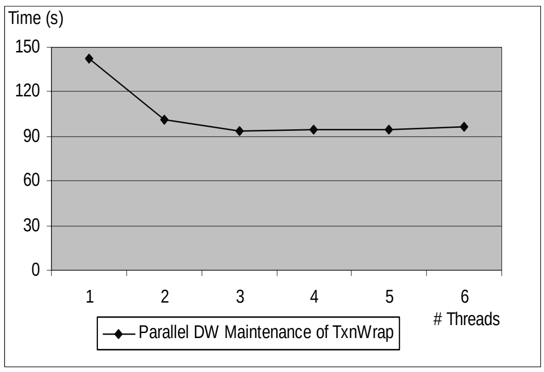

Figure 13: Parallel DW Maintenance

5.5 Parallel DW Maintenace

In this section, we present initial results of extending *TxnWrap* with a parallel DW maintenance scheduler [11]. The intuition is that since *TxnWrap* employs an unrestricted multiversion concurrency control algorithm, the maintenance queries as well as source updates won't affect each other when running in parallel. In this experiment, we use 500 data updates and vary the number of DW maintenance threads, i.e., the maximum number of source updates that could be maintained by the system at the same time. The total maintenance time is depicted in Figure 13.

From the figure, we see that when the number of threads increases, the total cost of processing time decreases. When the number reaches three, the total processing time almost reaches its optimal. We observe that after a certain number, an increase in the number of threads won't improve the performance anymore. One reason for this is that the ability of sources for parallelly answering maintenance queries is limited. Another reason is that we use a one-CPU PC for DWMS server. We have also developed algorithms for parallel maintenance of source schema changes. Due to the space limitations, please refer to [11] for details.

6 Related Work

Maintaining materialized views under source updates in a DW environment is one of the important issues of data warehousing [21, 23]. Initially, some research has studied incremental view maintenance assuming no concurrency. Such algorithms for maintaining a data warehouse under source data updates are called view maintenance algorithms [6, 12]. There has also been some work on rewriting view definitions under source schema changes [10], and on adapting the view extent under source schema changes [9, 16].

In approaches that need to send maintenance queries to the data sources, concurrency problems can arise. [23] introduces a compensation-based algorithm, called ECA, for incremental view maintenance under concurrent source *data updates* restricted to a single source. Strobe [24] handles multiple data sources while still assuming the schema of all sources to be static. SWEEP [1] applies local compensation of maintenance over distributed sources. [19] 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. Like DyDa [22], these efforts are all compensation-based. However, unlike DyDa, none of them can handle the concurrent source schema changes. Our new approach TxnWrap employs a multiversion solution removing concurrency concerns from DW maintenance algorithms thus no longer requiring any compensation strategies.

Our earlier work on the DyDa project [22] is the first attempt to address mixed workloads of concurrent data updates and schema changes. In DyDa, we had to enhance all maintenance modules (VM,VA,VS) to detect and correct various types of concurrency problems using compensation strategies. This puts much complexity into these modules. Also a formal basis for these algorithms is missing. For this reason, we now propose a novel transactional model to handle the DW maintenance concurrency problem.

Multiversion concurrency control algorithms [2, 4, 18, 14] 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 DW maintenance. In addition, our proposed algorithm can also handle versioned schema meta data. [18] proposes to utilize a two-version algorithm to resolve the concurrency between the DW maintenance and DW read-only sessions. Our work instead focuses on the other end of the spectrum, i.e., the concurrency between frequent source updates and DW maintenance. In the former work, the DW read-only sessions could be scheduled for any DW version as long as it's consistent. While in our work, the DW 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 DW maintenance process.

7 Conclusions

In this paper we take a fresh new look at the concurrency problems of data warehouse maintenance and in contrast to previous work we now present a solution based on a transaction theory. While previous work [23, 19] focuses on specific compensation algorithms for DW maintenance, we instead propose a transactional model that removes concurrency considerations from the maintenance logic. In particular, we encapsulate both source update transactions and DW maintenance process into one *DWMS_Transaction*. We then propose a multiversion algorithm called *ShadowWrapper* to achieve the serializability of *DWMS_Transaction*s. 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 data warehouse system as a middle layer that removes concurrency issues from the DWMS. As an added benefit, we found *TxnWrap* very easy to implement. The experimental results reveal that our solution achieves a stable DW maintenance performance. With proper optimizations, *TxnWrap* even outperforms previous solutions from the literature [1, 22] in terms of refresh rates. We are begining to look more closely at parallel maintenance processing, since our transactional modeling based approach could be naturally extended for that purpose [11]. We are also studying the issues such as DW maintenance recovery and batch DW maintenance.

References

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

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

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

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

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

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

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

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

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

[10] A. M. Lee, A. Nica, and E. A. Rundensteiner. The EVE Approach: View Synchronization in Dynamic Distributed Environments. *IEEE TKDE*, 2002. to appear.

[11] B. Liu, S. Chen, and E. A. Rundensteiner. A Transactional Approach for Parallel Data Warehouse Maintenance. Technical Report WPI-CS-TR-02-08, WPI, 2002.

[12] J. J. Lu, G. Moerkotte, J. Schue, and V. S. Subrahmanian. Efficient Maintenance of Materialized Mediated Views. In *Proceedings of SIGMOD*, pages 340–351, San Jose, California, May 1995.

[13] S. Marche. Measuring the Stability of Data Models. *European Journal of Information Systems*, 2(1):37–47, 1993.

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

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

[16] A. Nica and E. A. Rundensteiner. View Maintenance after View Synchronization. In *International Database Engineering and Applications Symposium (IDEAS'99)*, pages 213–215, 1999.

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

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

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

[20] D. Sjoberg. Quantifying Schema Evolution. *Information and Software Technology*, 35(1):35–54, 1993.

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

[22] X. Zhang and E. A. Rundensteiner. DyDa: Dynamic Data Warehouse Maintenance in a Fully Concurrent Environment. In *Data Warehousing and Knowledge Discovery*, pages 94–103, 2000.

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

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

A Proof of ShadowWrapper

As we know, the DW maintenance consists of several maintenance queries. Thus as long as these queries reflect a consistent state of sources, the DW would also stay consistent assuming all DW maintenance modules are correct. Thus we focus on the correctness of the core strategy of the *TxnWrap* solution, namely the *ShadowWrapper* technique. Theorem 3 summarizes the correctness criterion of the classical multiversion serializability theory described in [3]. Below we will make use of notations and definitions from [3] for transaction operations, histories, and serial graph.

Theorem 3 *A multiversion concurrency control algorithm is correct iff each MV history is 1SR, i.e., there exists a version order such that the multiversion serial graph (MVSG) is acyclic. [3].*

Assume there are k data sources, denoted $DS_1, DS_2, ..., DS_k$. We use $DS_i(j)$ to denote the result state of DS_i after having been updated by (the first-step of) a *DWMS_Transaction* with id j. We refer to this as the *jth* version of DS_i. In this context, we denote all DSs' initial states to have version number 0. Similarly, we use $DW(j)$ to denote the *jth* version of the DW extent with version number j, that is, the result written by the *DWMS_Transaction* with id j. Also we use $DW(0)$ to denote its initial state. The id of a *DWMS_Transaction* is assigned by the DWMS as soon as the update message arrives in the DWMS. As mentioned in Sections

4 and 5, this id can be used as a monotonically increasing timestamp as well as the version number of data items. We use the terms transaction id, timestamp and version number exchangeably in the following proof below.

As in [3], we use $r_i[x]$ (or $w_i[x]$) to denote the execution of a Read (or Write) issued by transaction T_i on a data item x. We also use c_i to denote T_i's commit operation. Notice that we do not have aborts for the $DWMS_Transaction$s.

We now introduce an illustrative example. Assume three $DWMS\_Transactions$ arrive in the DWMS. T_1 from DS_m with timestamp 1, T_2 from DS_n with timestamp 2, and T_3 again from DS_m with timestamp 3, assume $1 \leq m, n \leq k$ with k the number of DSs. The initial versions of the source extents presented by their respective wrappers are $DS_1(0), DS_2(0), ..., DS_k(0)$ and the initial DW extent is $DW(0)$. Then these three $DWMS_Transaction$s can be represented by:

1. $T_1 = w_1[DS_m(1)]\ r_1[DS_1(0)]...r_1[DS_m(1)]...r_1[DS_n(0)]...r_1[DS_k(0)]\ w_1[DW(1)]\ c_1$.
2. $T_2 = w_2[DS_n(2)]\ r_2[DS_1(0)]...r_2[DS_m(1)]...r_2[DS_n(2)]...r_2[DS_k(0)]\ w_2[DW(2)]\ c_2$.
3. $T_3 = w_3[DS_m(3)]\ r_3[DS_1(0)]...r_3[DS_m(3)]...r_3[DS_n(2)]...r_3[DS_k(0)]\ w_3[DW(3)]\ c_3$.

Here is a sample history H_1 of T_1, T_2 and T_3:

- $H_1 = w_1[DS_m(1)]\ w_2[DS_n(2)]\ w_3[DS_m(3)]\ r_1[DS_1(0)]...r_1[DS_m(1)]...r_1[DS_n(0)]r_1[DS_k(0)]\ w_1[DW(1)]$
  c_1
  $r_2[DS_1(0)]...r_2[DS_m(1)]...r_2[DS_n(2)]...r_2[DS_k(0)]\ w_2[DW(2)]\ c_2$
  $r_3[DS_1(0)]...r_3[DS_m(3)]...r_3[DS_n(2)]...r_3[DS_k(0)]\ w_3[DW(3)]\ c_3$.

Thus we can construct the multiversion serializable graph (MVSG) G according to its definition [3]. That is, we add the edge $T_1 \to T_2$ since $r_2[DS_m(1)]$ reads the result from $w_1[DS_m(1)]$; add edge $T_2 \to T_3$ since $r_3[DS_n(2)]$ reads the result from $w_2[IS_n(2)]$; add edge $T_1 \to T_3$ since $w_1[DS_m(1)]$ precedes $w_3[DS_m(3)]$. There are no more version order edges. Thus G is an acyclic MVSG graph in this example.

We now prove ShadowWrapper formally by contradiction. The DWMS keeps all arriving update messages in a FIFO message queue before processing. At any time t, each update message in this queue represents a $DWMS\_Transaction$. Thus let's denote this set of $DWMS_Transaction$s as $T = T_1, T_2, ..., T_k$ with T_1 located at the head of queue and T_k at the tail. The ids of T_1 through T_k are monotonic increasing transaction ids.

Our sequential *DWMS_Transaction* scheduler in the DWMS will execute them in exactly the same order. Thus this sequence represents one serial history of all these *DWMS_Transactions*.

Theorem 4 *Given any DWMS_Transaction set queuing in the DW, the multiversion serializable graph G of any history over of this set of DWMS_Transactions is acyclic. More strictly, all version order edges in G are pointing from the DWMS_Transaction T_i with smaller transaction id "i" toward another DWMS_Transaction T_j with larger transaction id "j".*

Proof: We prove Theorem 4 by contradiction.

(1) Assumption: There is one version order edge, $T_i \leftarrow T_j$ (with T_i's id $<$ T_j's id).

(2) Proof: There are two possible reasons to add this edge to the MVSG graph.

(2.1) If this edge is added by the serial graph (SG) definition [3], either T_i reads some data written by T_j, or there are conflict operations but T_j precedes T_i. This cannot be true.

(2.2) If this edge is added by the additional MVSG definition [3], then there are two cases to consider:

a. $w_j[x_j]$ ... $r_k[x_i]$ and $x_j \ll x_i$, that is, T_j writes some data earlier than T_i. According to Assumption in Section 3.2, all IS transactions are local. Thus for T_i and T_j to write to the same data would mean that they are from the same IS. Since T_i has a smaller id than T_j, T_i must commit before T_j at that IS. Thus it is impossible for T_j to write to some common data before T_i.

b. $r_j[x_k]$ ... $w_i[x_i]$ and $x_k \ll x_i$, that is, T_j reads a data item whose version is earlier than that of the same data item written by T_i. This is also impossible because we are using a sequential scheduler and thus T_i has already committed before the read operation of T_j can take place. Put differently, $w_i[x_i]$ would not be scheduled after $r_j[x_k]$.

(3) Contradiction: Since all cases lead to contradictions, the assumption is not correct. That is, there is no version order edge $T_i \leftarrow T_j$ with T_i's id $<$ T_j's id. Thus the MVSG graph is acyclic.

Finally by Theorems 4 and 3, we prove the *ShadowWrapper* algorithm to be correct.