Worcester Polytechnic Institute
Digital WPI

Computer Science Faculty Publications | Department of Computer Science

2-2002

A Transactional Approach to Parallel Data Warehouse Maintenance

Bin Liu
Worcester Polytechnic Institute, binliu@cs.wpi.edu

Songting Chen
Worcester Polytechnic Institute, chenst@cs.wpi.edu

Elke A. Rundensteiner
Worcester Polytechnic Institute, rundenst@cs.wpi.edu

Follow this and additional works at: https://digitalcommons.wpi.edu/computerscience-pubs

Part of the Computer Sciences Commons

Suggested Citation

Liu, Bin , Chen, Songting , Rundensteiner, Elke A. (2002). A Transactional Approach to Parallel Data Warehouse Maintenance. . Retrieved from: https://digitalcommons.wpi.edu/computerscience-pubs/116

This Other is brought to you for free and open access by the Department of Computer Science at Digital WPI. It has been accepted for inclusion in Computer Science Faculty Publications by an authorized administrator of Digital WPI. For more information, please contact digitalwpi@wpi.edu.

WPI-CS-TR-02-08 Feb 2002

A Transactional Approach to Parallel Data Warehouse Maintenance

by

Bin Liu
Songting Chen and Elke A. Rundensteiner

Computer Science Technical Report Series

WORCESTER POLYTECHNIC INSTITUTE

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

A Transactional Approach to Parallel Data Warehouse Maintenance

Bin Liu, Songting Chen, Elke A. Rundensteiner

Department of Computer Science
Worcester Polytechnic Institute
Worcester, MA 01609–2280
{binliu|chenst|rundenst}@cs.wpi.edu

Abstract

A Data Warehouse Management System (DWMS) extracts data from several distributed data sources, incorporates it into derived views in the data warehouse and maintains the views under source changes. Given the dynamic nature of modern distributed environments such as the WWW, both source data and schema changes are likely to occur autonomously and even concurrently in different information sources. We have thus developed a comprehensive solution approach, called TxnWrap, that successfully maintains the warehouse views under any type of concurrent source updates. Unlike most current solutions in the literature that apply compensation-query based strategies (and are restricted to handling data updates only), TxnWrap illustrates the application of transactional principles for solving data warehouse maintenance under both concurrent data updates and schema changes. However, TxnWrap has the restriction that the maintenance is processed one by one for each update, which limits the performance and thus delays the refresh of the DW.

In this paper, we illustrate that TxnWrap's design decision has many advantages in developing a parallel DW maintenance solution. In particular,we exploit the transactional approach that TxnWrap takes toward distributed data warehouse maintenance. For this, we first identify the read/write conflicts among the different warehouse maintenance processes. We then propose a parallel maintenance scheduler (PMS) that generates possible schedules that resolve these conflicts. Finally, we describe the commit problem for parallel maintenance process. We have proven our solution to be correct. PMS has been implemented and incorporated into our TxnWrap system. The experimental results confirm that our parallel maintenance scheduler significantly improves the performance of data warehouse maintenance.

Keywords: Data Warehouse Maintenance, Parallel, Data Update, Schema Change, Concurrent.

1 Introduction

1.1 Data Warehouse Environments

Data warehouses (DW) [GM95, MD96] are built by gathering data from several information sources (IS) and integrating it into one repository customized to user's needs. Data warehousing is important for many applications, especially in large-scale environments composed of distributed ISs, such as travel services, E-commerce and decision support systems. A data warehouse management system (DWMS) is the management system of such a DW which is responsible of maintaining the DW extent and schema upon changes of underlying ISs.

We distinguish three tasks related to maintaining the DW in such environments. The most popular one, incremental view maintenance (VM) [ZGMHW95, AASY97, SBCL00, ZRD01], maintains the DW extent whenever a data update occurs at an IS. The second one, view synchronization (VS) [KLZ^{+}97, LKNR99], rewrites any affected view definition in the DW whenever there is a schema change in one of the ISs rendering the current view definition undefined. The third one, referred to as view adaptation (VA) [GMR95, NR98, MD96], aims to adapt the view extent incrementally after the view definition has been modified either directly by the DW designer or indirectly by the view synchronization system.

In distributed environments, ISs are typically owned by different information providers and function independently. This implies that they will update their data or even their schema without any concern for how these changes may affect the DW, in particular, the materialized views defined upon them. TxnWrap [CR00] is the first stable DW maintenance solution that supports maintenance of a DW in such a dynamic environment where both data updates and schema changes occur independently in the underlying ISs. However, similar to most solutions in the literature (Which all are restricted to handling data changes only) [AASY97, GM95], TxnWrap also has the limitation of serially processing the maintenance of one individual update at a time when there is a set of updates waiting to be maintained. Our work in this paper overcomes this limitation by proposing a parallel maintenance scheduler to improve the performance of TxnWrap.

1.2 Maintenance Anomaly Problem

Let's first illustrate the maintenance problems in such environments via some running examples. As depicted in Figure 1, the DW view V is defined on relations R and S with the condition "R.C = S.C and R.B $<$ 7 and S.E $>$ 50". First, when a data update $DU_1 : Insert(3, 5, 5)$ into R in IS_1 occurs and is reported to the DWMS, then the DWMS will generate a maintenance query Q_1 = "Select S.C, S.D from S where S.C = 5 and S.E $>$ 50" to IS_2 to incrementally incorporate this data change into the view V. The DWMS at that point will assume that the IS_2 is in the state in which the DU_1 was committed. However, this may not necessarily be true. Below we distinguish between two cases that may occur at S respectively:

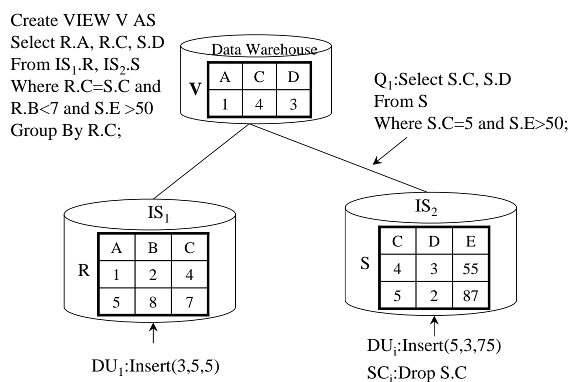

Figure 1: Explainations of Maintenance Anomaly Problems

- Case 1: Assume during the transfer time of the query Q_1 to S, S already commits a new data update, for example, $DU_i : Insert(5, 3, 75)$ into S in IS_2. This new tuple would also be captured by Q_1 and a tuple (3,5,3) would be inserted into the view. However, when the DW starts processing DU_i later, the same tuple would be inserted into the view again. A **duplication anomaly problem** appears [ZGMHW95].

- Case 2: Assume that during the transfer time of the query Q_1 to S, S undergoes the schema change SC_j: Drop S.C. Then the query Q_1 even couldn't be processed due to the inconsistency between the schema specified in the query Q_1 (S.C, S.D and S.E are required) and the schema of S (only S.D and S.E are left). A **broken query anomaly problem** arrises [ZR00].

We refer to the above DW maintenance problems that are caused by concurrent updates in dynamic environments as 'anomaly problems' . To summarize, the problem is how to execute maintenance queries while the DWMS doesn't know the current state of the underlying ISs due to the DWMS and ISs operating independently and randomaly. That is, when processing a given update message, the DWMS assumes the corresponding ISs to be still in the state in which the update message was just committed. This may however not necessarily be true because the ISs could continue to start new updates. Hence the DWMS's state is always older than the ISs' state and the maintenance queries to ISs may contain incorrect query results or may even fail to complete due to the schema changes [ZR00].

1.3 Parallel Maintenance Scheduler

TxnWrap [CR00] is a new solution that we have proposed to address these anomaly problems as illustrated in Figure 1. Unlike most recent work in DW maintenance that relies on compensation-based solutions [AASY97, ZGMW96], we introduce the concept of a DWMS_Transaction model [CR00] to formally capture the overall DW maintenance process as a transaction. Once properly cast in terms of transaction concepts, we then propose a multiversion timestamp-based concurrency control algorithm [BHG87], called ShadowWrapper, to solve the anomaly problems in DW maintenance. We've proven TxnWrap to be correct and to achieve strong consistency [1] in the data warehouse maintenance. However, like most other solutions proposed in the literature, TxnWrap applies a sequential approach towards maintaining concurrent updates. This limits its performance in a distributed environment which has overhead of network delay and IO cost etc. Here we propose to overcome this limitation by developing a parallel scheduler that is capable of maintaining a set of concurrent updates in parallel.

In this paper, we propose an efficient parallel maintenance scheduler that exhibits excellent performance for handling both concurrent data updates and schema changes. It overcomes TxnWrap's limitation in terms of performance by parallelizing the executions of different maintenance tasks. For this, we exploit the transactional approach of TxnWrap to formalize parallel DW maintenance. To achieve this, three issues must be tackled. The first one is to identify all potential conflicts among the data warehouse maintenance processes in terms of read/write of critial resources. The second is to design strategies to generate possible schedules that resolve these identified conflicts. The last is to examine the commit problem for each maintenance task

[1] We adopt the definitions of correctness and consistency levels of the DW from [ZGMHW95].

in parallel processing to make the DW consistent.

For the first issue, we formalize the DW maintenance process using the DWMS_Transaction model as well as traditional read/write operations. This allows us to analyze their relationships in a formal manner. For the second, due to the differences of the process, overhead and occurrence in the data update and schema change maintenances, we propose distinct scheduling strategies based on the different update types typical for the environment. That is, we propose an aggressive scheduler for data update only environments and both the TxnID-Order-Driven scheduler and the Dynamic-TxnID scheduler for a mixed data update and schema change environment. For the Data Warehouse commit problem, we adopt either a negative counter solution [ZRD01] or a strict commit order to make the DW consistent.

1.4 Contributions of the Work

The main contributions of this paper are:

- Identify the performance limitation of the TxnWrap system in terms of the sequential handling of a set of updates, and then characterize the research issues that must be addressed to achieve parallel maintenance.
- Formalize the constraints (in terms of read/write conflicts) of parallel scheduling the updates in a mixed schema change and data update environment.
- Propose three parallel scheduling strategies suitable for two different scenarios. That is, an aggressive scheduler for the data update only environment and the TxnID-Order-Driven and Dynamic-TxnID schedulers for the mixed data update and schema change environment. Develop solution strategies for the DW commit problems in parallel scheduling.
- Design and implement a parallel maintenance scheduler based on the above strategies and incorporate it into the existing TxnWrap system developed by the Database Systems Research Group (DSRG) at WPI.
- Conduct experimental studies comparing the sequential with the parallel schedulers that confirm the performance benefits achievable by the parallel scheduler.

1.5 Outline of Paper

Section 2 introduces consistency level and the assumptions in the system. The basis of TxnWrap is presented in Section 3. In Section 4, we propose several enhancements to TxnWrap to enable us to move toward parallel scheduling. The parallel scheduling algorithms are proposed in Section 5. Section 6 discusses the design and implementation issues. Experimental studies are discussed in Section 7. Section 8 discusses related work, while conclusions are presented in Section 9.

2 Consistency Level and Assumptions

2.1 Correctness and Consistency Levels of a DW

We adopt the definitions of correctness and consistency levels of the DW from [ZGMHW95], as stated below:

- **Correctness**: Any state of the DW corresponds to one valid state of each IS.
- **Convergence**: All IS updates will be eventually incorporated into the DW resulting in a correct final state.
- **Strong Consistency**: All states of the DW are correct and the order of DW state transitions correspond to the order of the state transitions of each of the ISs.
- **Complete Consistency**: Strong consistency holds plus each state of one of the ISs is reflected by a distinct DW state.

2.2 Assumptions of the DWMS Environment

The following assumptions hold in the parallel scheduler maintenance system, as well as in most of the DWMS environments:

1. We assume a centralized data warehouse system, meaning there is only one DWMS that stores possibly several materialized views.

2. We assume all IS transactions are local and every data update and schema change at an IS is reported to the DWMS once it committed at the IS.

3. The message transfer in the network between each IS and the DW is FIFO.

4. The data warehouse network environment and information sources will not have permanent unrecoverable failures.

3 TxnWrap Revisited

In this section, we briefly review the TxnWrap solution which represents the framework based on which we develop our solution. First, we sketch the model used as basis of TxnWrap, called the DWMS_Transaction model. Second, ShadowWrapper, the multiversion concurrency control algorithm for TxnWrap, is introduced. After that, some limitations of TxnWrap are discussed.

3.1 The DW Maintenance Transaction Model

In a typical DW environment (One DW over several independent ISs), a complete DW maintenance process is composed of the following steps:

- **IS_Update**: An IS update transaction at some IS_i is committed, denoted as "w(IS_i)C_{IS}" where w(IS_i) means the write on IS_i, i is the index of the IS, and C_{IS} is the commit of this write.

- **Report**: The IS reports the update made by this transaction to the DWMS.

- **Propagation**: The DWMS computes the effect to the DW caused by this update in order to maintain the DW. We denote this step as "$r(IS_1)r(IS_2)...r(IS_n)$", where $r(IS_i)$ is a read over IS_i, because we typically need to send a maintenance query down to each IS_i and then to collect the result from each IS_i to calculate the effect on the DW.

- **Refresh**: The result calculated in the propagation step finally is refreshed into the DW, denoted as "$w(DW)C_{DW}$", where w(DW) is to write or refresh the DW extent and C_{DW} is the commit of the w(DW) to the DW.

TxnWrap introduces the concept of a global transaction to encapsulate the above four DW maintenance steps within the context of the overall data warehouse environment. Thus, we can make use of this transaction model to analyze the DW maintenance. This global transaction, denoted as a DWMS_Transaction, can be characterized as follows.

Definition 1 *A DWMS_Transaction is a transaction model that encapsulates the four maintenance steps (IS_Update, Report, Propagate, Refresh) taking care of the maintenance process for one source update. Each DWMS_Transaction starts with the processing of a local database transaction at some IS (IS_Update), and it commits when the DW database has been successfully refreshed (Refresh) reflecting the update committed in this IS_Update.*

Note that the *DWMS_Transaction* will be created only after C_{IS} of the IS update transaction has been done at the IS, and the conceptual commit of a DWMS_Transaction is right after the C_{DW} in the *Refresh* step. Thus, for briefness, we denote a DWMS_Transaction as "$w(IS_i)r(IS_1)r(IS_2)...r(IS_n)w(DW)C_{DW}$".

A DWMS_Transaction is a conceptual rather than a real transaction model. It has a nested structure and sits at a higher level above the DBMS transactions local to the IS or DW DBMS transactions local to the DWMS. In the DWMS_Transaction model, there is no automatic rollback or abort mechanism, because the local IS transaction is out of the control of the DWMS and the committed IS updates must be propagated to the DW if we want the DW to stay consistent. Thus, we can further remove the "C_{DW}" operation and simply denote a DWMS_Transaction as "$w(IS_i)r(IS_1)r(IS_2)...r(IS_n)w(DW)$".

Based on this model, we now can rephrase the DW anomaly problem as a concurrency control problem. Namely, we note that the only conflict we must consider in the context of DWMS_Transactions is the 'read dirty data' conflict. That is, one operation in the *Propagate* phase may read some inconsistent query results written by the *IS_Update* phase of the maintenance process. Here we assume all the other conflicts can be solved simply by the respective local DBMS at the IS or the DW level.

3.2 Concurrency Control Strategy in TxnWrap

Due to casting the DW maintenance as the DWMS_Transaction, (each committed IS update is referred to as one DWMS_Transaction) TxnWrap designs a multiversion concurrency control algorithm [BHG87] (called ShadowWrapper) to solve the anomaly problems in DW maintenance. That is, TxnWrap keeps versions

of both all updated tuples as well as schema meta data in a dedicated wrapper for each IS. In short, the ShadowWrapper concurrency control algorithm perform the following steps at the Wrapper [2]:

- Initialize the wrapper schema and wrapper relations according to the DW view definition and the corresponding IS's local schema and data.
- When an IS commits a local transaction and reports its update to the wrapper, the ShadowWrapper first assigns a timestamp, called a *global id* (globally unique in the DW environment), to each update. Then it generates the corresponding version data in the wrapper. After that, the ShadowWrapper reports the update (with its unique *global id*) to the DWMS.
- At any time, when a wrapper receives a maintenance query from the DWMS, then:
  - ShadowWrapper rewrites the maintenance query in terms of proper versioned data according to its *global id*, and executes this rewritten query upon the wrapper schema and data;
  - Thereafter, the ShadowWrapper returns the query result to the DWMS.
- When a DWMS_Transaction is committed in the DW, the corresponding version data will be cleaned up by the ShadowWrapper.

Thus, integrated with the ShadowWrapper, the maintenance steps for each update in TxnWrap now be characterized as follows:

- $w(IS_i)w(Wrapper_i)r(IS_1)r(IS_2)...r(IS_n)w(DW)$.

Here $w(Wrapper_i)$ generates the versioned data in the wrapper indexed by its *global id*, and $r(IS_i)$ $(1 \leq i \leq n)$ now refers to a read of the corresponding versioned data from the wrapper using its *global id* rather than directly accessing the remote (non-versioned) IS_i.

3.3 Limitations of TxnWrap

However, TxnWrap has several weaknesses. First, the space overhead of each wrapper may become excessive. For this problem, we have already undertaken some work on wrapper space optimization. That is, we can

[2] We illustrate a running example of ShadowWrapper version management in Section 4.1

only keep the attributes in the view definitions of the DW in the wrapper (Projection) and to use the local conditions of the DW view definition (Selection) to remove unnecessary tuples. These strategies reduce the space consumption. Second, like other DW maintenance algorithms in the literature, TxnWrap uses a serial processing model for the DW maintenance process described above, which restricts the system performance. That is, only after we have committed the current DWMS_Transaction maintenance process (we will refer to the *Propagate* and *Refresh* steps in the DWMS_Transaction, the steps of " $r(IS_1)r(IS_2)...r(IS_n)w(DW)$", as DWMS_Transaction maintenance process in the following sections), could we begin the handling of the next one. In the propagation step of the DWMS_Transaction, the DWMS issues maintenance queries one by one to each IS_i and collects the results. In a distributed environment, the overhead of such remote queries is typically high due to network delay though the DWMS starts sending maintenance query from the reported update to avoid the possible excessive overhead on network traffic. That is, after the DWMS receives the current result from IS_i, then it generates the maintenance query to be submitted to the IS_{i+1} [AASY97]. So, only one IS is being utilized at a time in the maintenance propagation phase having both network delay as well as the IO costs at the IS_i. Naturally, if we could interleave the execution of the different DWMS_Transaction maintenance processes, we could reduce the total network delay, and possibly also keep all ISs busy. This way, the overall performance would improve.

4 Version and Transaction Management

In this section, we extend the *global id* in TxnWrap to make the version management more independent and more flexible. As described in Section 3.2, ShadowWrapper uses a *global id* to label and then identify the appropriate version data in the wrapper. It also uses this *global id* to identify and track the corresponding DWMS_Transaction in the overall data warehousing system. Though this simplifies the control strategy used in the maintenance system, we now observe that this restricts the flexibility of scheduling DWMS_Transactions because it tightly binds the version management task in the IS wrapper with the overall maintenance task of the DWMS server. Note that *global id* has to be issued by a global id-generator in the DWMS to make the id unique in the context of overall data warehousing. We relax this binding by introducing a *local id* for version management in the wrapper and a TxnID to manage DWMS_Transactions in the DWMS.

4.1 Version Management using Local Identifier

Definition 2 *A local id is a timestamp that represents the time the update happened in the respective IS.*

Without loss of generality, we use an integer k (k $\geq$ 0) to represent the *local id*. Note that *local id* is unique within the IS scope and monotonically increasing starting from 0, thus a unique order is guaranteed in the IS scope. Compared with using the *global id* as introduced in Section 3.2, there are two gains by using *local id* instead. First, the process of id generation can be done locally in the wrapper. That is, we don't have to communicate with the DWMS during the version management. Second, we have to assume a global FIFO in the overall DW system to assume correctness of the *global id*. This is hard to coordinate in a distributed environment. The use of *local ids* for version management would relax the restriction of the global FIFO assumption [3].

Figures 2 and 3 illustrate the version management in the wrapper from Section 3.2 now using *local ids*. As an example, the IS_1 wrapper in Figure 2 contains the data of relation R as well as the related meta information (the lower part). The IS_2 wrapper stores the same for relation S. Two additional attributes

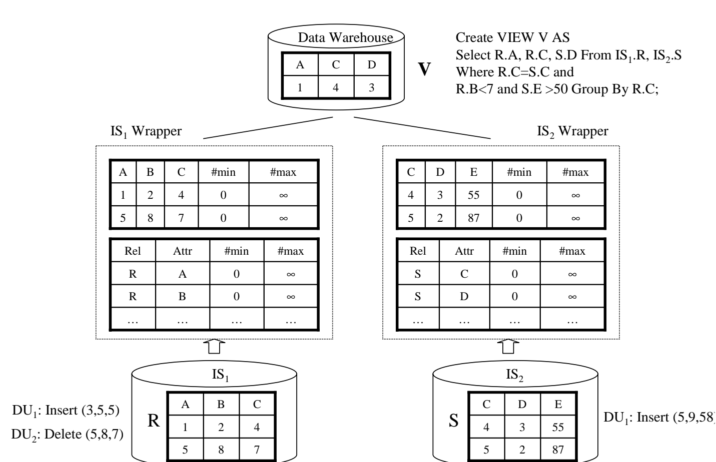

Figure 2: Wrapper Version before Updates

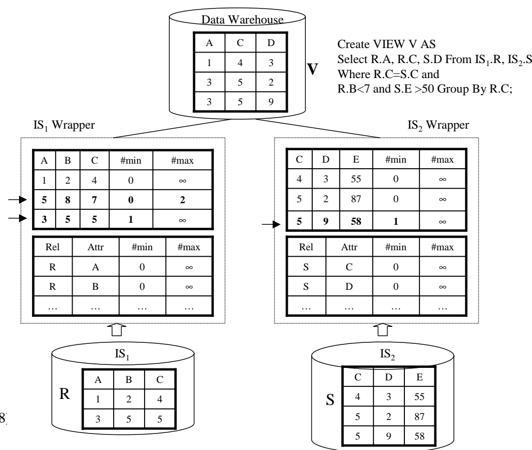

Figure 3: Wrapper Version after Updates

#min and #max in the wrapper denote the life time of each tuple. #min denotes the beginning of the life of the tuple (by insertion) while #max denotes the end of the life of the tuple (by deletion). The value of #min and of #max of an updated tuple are set by the corresponding DWMS_Transaction using its local id in the

[3]A running example of relaxing global FIFO assumption is illustrated in Section 5.3.2

Report phase. Assume as in Figure 2, $DU_1 : Insert(3,5,5)$ and $DU_2 : Delete(5,8,7)$ happened in IS_1. Then in the IS_1 Wrapper, one tuple (3,5,5) is inserted, which is depicted in Figure 3. Its [#min, #max] value is set to [1,∞]. This means that the life time of this tuple starts from the timestamp 1. Next, the tuple (5,8,7) is deleted. Its [#min, #max] value is thus changed from [0,∞] to [0,2]. This means that this tuple becomes invisible after timestamp 2. A similar process happens to the IS_2 Wrapper when $DU_1 : Insert(5,9,28)$ is committed in the IS_2. From another point of view, the *local id* serves as the version write timestamp for the given update.

4.2 DWMS_Transaction Management using TxnID

In the global DWMS environment, we still need a unique id to identify and track each DWMS_transaction, and to construct correct maintenance queries that access the appropriate versions of data in each IS wrapper. To accomplish this, we assign a unique identification number to each IS. For example, if IS_i is one of the ISs in the environment, then we use the integer i for the identification of this IS_i. Based on the static IS identification and the *local id*s of the different IS updates, we define a globally unique DWMS_Transaction identification for each IS update, denoted as TxnID, as follows.

Definition 3 *A TxnID, denoted as τ, is a vector timestamp that concatenates the current local id (the largest local id that has been assigned thus far) of each IS in the system when this TxnID is generated. For each element of τ, we have $\tau[i] = k_i$ with $0 \leq i \leq n$, n the number of ISs, k_i the current local id of IS_i.*

In the Definition 3, We call n is the size of TxnID and $\tau[i]$ the ith value of the TxnID τ. We use an integer vector of length n, say $\tau = [k_1,\ldots,k_i,\cdots,k_n]$, to explicitly represent the TxnID τ with k_i (1 ≤ i ≤ n) the current maximal *local id* of its corresponding ISs. Figure 4 illustrates how the TxnID is generated in the TxnWrap system. As in Figure 4, the jth element in the TxnID of update U_i in IS_j is k_j, which is the current *local id* of IS_j at the time U_i arrives at the DWMS (other entries of this TxnID vector respectively contain the latest local ids of corresponding ISs). It's easy to see that though the local ids in different ISs may be the same, the TxnIDs are all unique. From the view point of the DWMS, each entry of the TxnID vector records the current state of each IS on arrival of the IS update . As an example, assume three updates happened in the two ISs depicted in Figure 2, IS_1:DU_1, IS_1:DU_2 and IS_2:DU_1. Suppose they arrive at the DWMS in the following order, IS_1:DU_1,IS_2:DU_1, and then IS_1:DU_2, then their TxnIDs will be [1,0], [1,1] and [2,1] respectively. We assume that the initial local ids are all 0 and no other updates happened before.

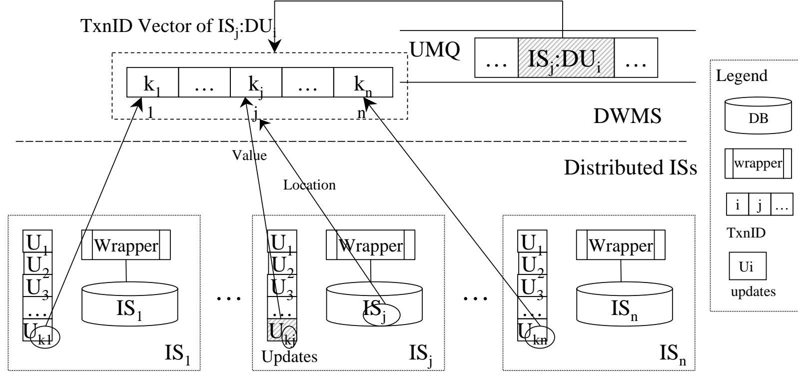

Figure 4: TxnWrap Structure and Interaction

The TxnID serves a dual purpose: one is to uniquely identify each DWMS_Transaction in the global environment and the other is to record the underlying ISs' states in terms of timestamps when this update is reported to the DWMS. We know that the maintenance queries are all IS specific. Thus, it is now possible to identify the right versioned data in the wrapper with the help of its TxnID. The following is a simple example illustrating the use of TxnID in maintenance query generation. Assume as in Figure 2, a data update IS_1:DU_1 "Insert(3,5,5)" in IS_1 is reported to the DWMS first. Then we assign TxnID [1,0] to IS_1:DU_1. To maintain this update, we will issue a maintenance query "Q_1: Select S.C, S.D From S Where S.C=5 and S.E>50" to IS_2. Based on TxnID [1,0], we know that this maintenance query should see the timestamp 0 of IS_2. Thus we rewrite the Q_1 into Q_1': "Select S.C, S.D From S Where S.C=5 and S.E>5 and (#min$\leq$ 0 and #max $>$ 0) ". Thus, even though another update DU_1:IS_2 has already happened in IS_2, its effect can still be excluded from Q_1' because of the timestamps recorded in its TxnID and the #min and #max values of each tuple in the wrapper.

5 Parallel Maintenance Scheduler

Now, we have a suitable basis to develop a parallel maintenance scheduler to maintain DWMS_Transactions. In the following sections, we will first directly extend the serial scheduler in TxnWrap to become a parallel one. This turns out to naturally only fit in a data update only environment. Then, by analyzing the read/write conflicts between data update and schema change DWMS_Transactions, we come up with our solution strategies in a fully dynamic environment.

As explained above, TxnWrap is the only DW maintenance solution for fully dynamic environments that utilizes a transaction approach. One benefit of such a transaction-based solution is that other advanced functionalities such as parallel scheduling or recovery can now more easily be added to the DW system by exploiting the principles of transaction management. In particular, we now will illustrate that with small modifications, TxnWrap can be extended to process maintenance tasks in parallel even in a highly dynamic environment. As long as we can make sure that each maintenance query can get the correct version data based on the timestamps recorded in the TxnID, then it naturally falls into place that all the read operations in the DWMS_Transaction maintenance processes could be conducted in parallel.

5.1 Parallel Architecture

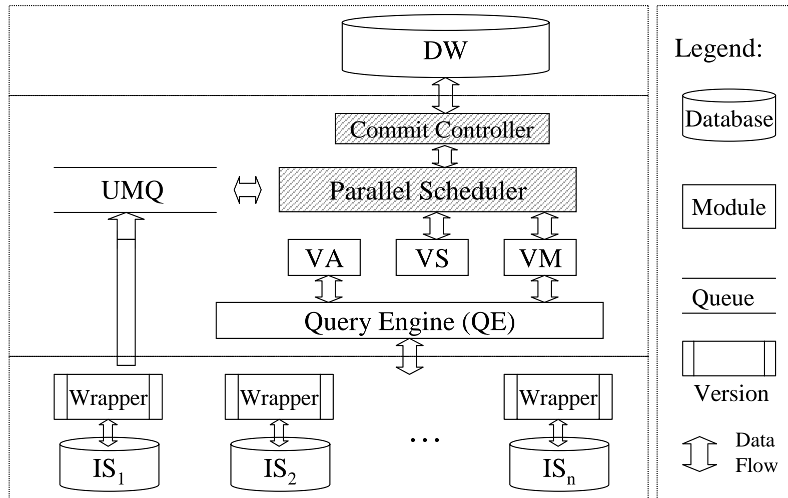

Figure 5: Architecture of TxnWrap Extended with Parallel Scheduler

Figure 5 describes the overall architecture of the TxnWrap system, with our proposed extension for parallel scheduling. Compared to the basic TxnWrap architecture, we add two new modules, one is the *Parallel Scheduler* and the other is the *Commit Controller* [4]. The maintenance process flow of each DWMS_Transaction can be described as follows. In the *Parallel Scheduler*, we keep fetching updates from the UMQ (a queue containing updates reported from ISs waiting to be maintained), check whether system resources are available for maintenance processing and then verify what conflicts (in terms of read/write critical resources) related to this update exist to decide when to start the maintenance. More specifically,

[4] The detailed descriptions of the TxnWrap structure can be found in [CR00].

if this is a data update, then it is sent to the View Maintenance (VM) module, which is responsible for generating maintenance queries, sending these queries to the wrapper, and collecting maintenance results. If this is a schema change, then it first is sent to the View Synchronization (VS) module to find a suitable replacement for an operation such as drop relation or drop attribute that affects the DW. Then VS updates the affected view definitions. If necessary, the View Adaptation (VA) module calculates the delta changes to the DW extent. After that, VM or VA submits the final result to the *Commit Controller* module. Based on the commit strategy, the *Commit Controller* will push the result to the DW.

In short, the *Parallel Scheduler* must analyze waiting DWMS_Transactions to identify possible constraints (in terms of read/write critical resources) and then to generate a serializable schedule for a set of DWMS_Transactions. As stated before, TxnWrap is capable of maintaining the DW in a concurrent data update and schema change environment. Due to the differences between the data update and schema change maintenance in terms of process, maintenance overhead and the frequency of occurrence in the DW environment, below we identify the read/write constraints in two different situations, namely, the data update only environment and the mixed data update and schema change environment. Then we propose suitable parallel schedulers for these two situations. More precisely, we propose three parallel schedulers below. That is, the aggressive scheduler works in a data update only environment, which is the natural extension of the serial scheduler in TxnWrap, while the TxnID-Order-Driven scheduler and the Dynamic-TxnID scheduler both work in a mixed data update and schema change environment.

5.2 Aggressive Scheduler for Data Update Only Environments

First, we formalize the data update maintenance process using a transaction-like notation. Incorporated with the DWMS_Transaction model described in Section 3, one data update DWMS_Transaction T_{DU_i}, with DU_i happened in IS_j, can be formally represented using read/write operations as follows:

- $T_{DU_i} = w[IS_j], w[Wrapper_j], r[VD], r[IS_1], r[IS_2], \cdots, r[IS_n], w[DW]$.

Here VD represents the view definition in the DW. r[VD] stands for the operation that breaks down the maintenance query for individual ISs based on the view definition VD. Thus, one data update DWMS_Transaction maintenance process can be represented as "$r[VD], r[IS_1], r[IS_2], \cdots, r[IS_n], w[DW]$".

In the DWMS_Transaction module, each local DBMS controls the concurrency of w[IS],w[Wrapper] and w[DW] operations respectively. As reviewed in previous sections, *local id*, TxnID and versioned data are three key points of TxnWrap. The ShadowWrapper algorithm guarantees that each r[IS_i] operation can easily identify the right version data using the *local id*s recorded in the TxnID for that respective IS. So, there will be no read block between DWMS_Transaction maintenance processes under the condition that the version data is generated before any of the corresponding version read operations. In TxnWrap, this condition is always true because a DWMS_Transaction is created only after the local DBMS transaction has been committed and reported to the DWMS. The later is clearly after the w[Wrapper] operation has recorded the update in the form of properly versioned data in the respective wrapper.

Borrowing traditional concurrency control concepts [BHG87], we now propose a scheduling algorithm, while we call aggressive scheduler, for data update only environments. The main control of the scheduler and its corresponding execution plan (the execution schedules of the DWMS_Transaction maintenance processes) for a set of updates is depicted in Figure 6. In the execution plan, each DU denotes one data update DWMS_Transaction maintenance, while T_i (i $\geq$ 1) denotes the maintenance process related to corresponding DUs. That is, we can start the data updates' DWMS_Transaction maintenance processes almost at the same time as long as sufficient computational resources are available in the DWMS server [5].

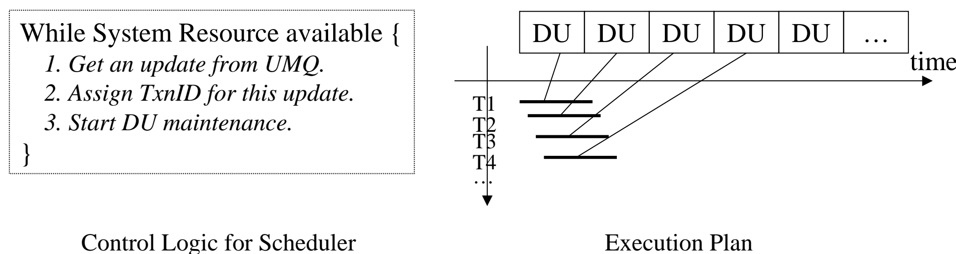

Figure 6: Control Logic of Aggressive Scheduler

5.3 Scheduling in a Mixed Data Update and Schema Change Environment

However, more issues must be dealt with if we take schema changes into consideration. First, we briefly review how schema changes are maintained [RLN97, CZC$^+$01, DZR99]. There are roughly five steps for

[5] Its correctness proof is presented in [LR02].

maintaining a schema change:

- Determine which views in the DW are affected by the change (step 1)
- For each affected view (step 2):
  - Find the suitable replacement for schema elements removed from the view definitions via operations such as drop relation or drop attribute. (step 2.1)
  - Rewrite the view definition in the DW if needed. (step 2.2)
  - Calculate the delta changes in term of data tuples to be added or to be deleted due to the replacement between the old and the new view definition. (step 2.3)
  - Adapt the DW by committing these delta changes to the DW using the VA algorithm. (step 2.4)

The VS module listed in Figure 5 is responsible for the steps 2.1 and 2.2 while the VA module corresponds to steps 2.3 and 2.4.

As can be seen by the above steps, the view definition (VD) of the DW represents a critical resource. The following sequence of operations occurs during SC maintenance:

- $r_1[VD], \cdots, w_{2.2}[VD], \cdots, r_{2.4}[VD]$

Subscripts 1, 2.2 and 2.4 are the step numbers according to the SC maintenance steps. As is stated above, reading the VD to calculate the affected views occurs in step 1, while rewriting the VD may be required in step 2.2. In step 2.4, we read the VD again to generate the maintenance queries.

Putting it all together, one schema change DWMS_Transaction T_{SC_i}, with SC_i the ith update in IS_j, can be represented as follows:

- $T_{SC_i} = w[IS_j]w[Wrapper_j]\mathbf{r[VD]},\cdots,\mathbf{w[VD]},\cdots, \mathbf{r[VD]}r[IS_1]r[IS_2],\cdots,r[IS_n]w[DW]$

Similarly, one schema change DWMS_Transaction maintenance process represents the steps of "$\mathbf{r[VD]},\cdots,\mathbf{w[VD]}$
$,\cdots,\mathbf{r[VD]}$ $r[IS_1]r[IS_2],\cdots,r[IS_n]w[DW]$". Thus, if more than one schema change DWMS_Transactions exist in the UMQ, we can't interleave their executions randomly because of the r[VD]/w[VD] conflicts in the

different transactions. However, in the DWMS environment, the occurrence of schema changes is rare. Thus, for simplicity of the control strategy in the scheduler, we schedule schema changes sequentially. That is, only after the processing of the pervious schema changes have been completed and committed to the DW, which happens in step 2.4, then we begin the handling of the next schema change. In theory, it is possible that we could maintain schema changes all the way in parallel, for example, using either a lock-based or multi-version [BHG87] algorithm to control the concurrency of the view definition (VD), which here is a shared resource. But the resulting control strategy would be more complicated and the likelihood of a major performance gain is low since schema changes are not that frequent and are waiting to be maintained in the DWMS.

Now, we examine the DWMS_Transactions in the case of a mixture of data updates and schema changes. For the data update maintenance, we need to read the view definition to break down the maintenance queries, denoted by the r[VD] operation. While for schema change maintenance, the following sequence occurs, $r_1[VD], \cdots, w_{2.2}[VD], \cdots,$
$r_{2.4}[VD]$. Thus, all combinations of the read/write conflicts of VD exist in updates between two SCs, and also between one SC and many DUs. So, in a mixed data update and schema change environment, our parallel maintenance scheduler has to consider all these constraints. Below, two strategies are devised to solve these constraints.

5.3.1 TxnID-Order-Driven Scheduler

Typically, we can distinguish between two basic kinds of concurrency control algorithms, either lock-based or multi-version timestamps based, that both can guarantee a serializable schedule of transactions. First, we investigate the application of a lock-based strategy to address the conflicts arising in a mixed data update and schema change environment.

A traditional lock-based algorithm doesn't care about the order of transactions as long as the schedule is equivalent to one of the serial schedules. For example, given two transactions $T_1 = (r_1[x], w_1[x])$ and $T_2 = (r_2[x], w_2[x])$. In a lock-based scheduler, both schedules listed below are acceptable: $r_1[x], w_1[x], r_2[x], w_2[x]$ or $r_2[x], w_2[x], r_1[x], w_1[x]$. But this may not be the case in a DWMS_Transaction environment. At least, we need to keep the assumption of FIFO for updates which come from the same information source, otherwise certain updates wouldn't be correctly maintained. For example, two updates "DU_1: Insert into A(1,2,3)" and "DU_2: Delete (1,2,3) from A" happened in the same IS in this order, we should maintain DU_1 before DU_2 in

the DWMS. If not, it is possible that the maintenance result of DU_2 couldn't be refreshed in the DW because the corresponding tuple isn't in the DW yet. Thus, we can't reorder the DWMS_Transactions randomly. Secondly, once we assign the corresponding TxnID (timestamps) to each update, more ordering restrictions need to be imposed. That is, we can't randomly reorder these DWMS_Transactions in the scheduler even if these updates come from different ISs, otherwise the maintenance result may also be inconsistent with the IS state. To explain this, we first define the TxnID order as follows. Assume two TxnIDs τ_j and τ_k, $\tau_j < \tau_k$ $\iff \forall i, 1 \le i \le n$ (n is the size of TxnID vector) $\tau_j[i] < \tau_k[i]$.

Observation 1 *Once we have assigned TxnIDs to updates (that could be mixed data updates and schema changes), then parallel scheduling of these updates needs to keep the TxnID order.*

The following example illustrates why we have to keep TxnID order in such a mixed data update and schema change environment.

Example 1 *As depicted in Figure 7, assume two updates from two different ISs, one is the schema change "SC_1: drop table R" in IS_1, and the other is the data update "DU_1: insert into $R_1(3,8,4)$" in IS_3. We also assume that the view definition V before the updates is $\prod_{(A,C,D)}(R \bowtie S)$. After we drop relation R in IS_1, the system will find R_1 in IS_3 as a replacement and the new view V' will be $\prod_{(A,C,D)}(R_1 \bowtie S)$. So, if*

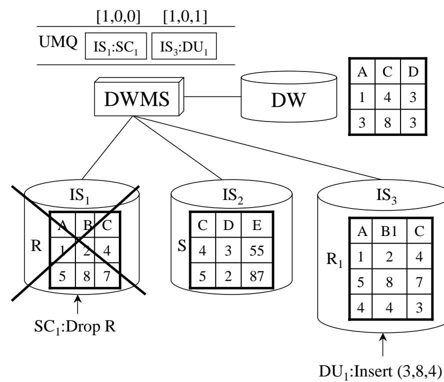

Figure 7: Example of Scheduling Order

we assign TxnIDs [1,0,0] and [1,0,1] to these two updates when they come to the UMQ. This means that the schema change "drop table R" has arrived at the DWMS first, and the data update "insert into R_1" arrived second. If these two updates are being maintained correctly, then the final schema in data warehouse will be V', and its content is $(1,4,3),(3,8,3)$. But if the parallel maintenance scheduler schedules [1,0,1] before

[1,0,0], that is, no changes to the DW extent because R_1 is not in the view definition V yet. While for update [1,0,0], the DWMS can't see the data update because the TxnID tells us that when the DWMS maintains this update, it can only see the state 0 in IS_3. State 0 is the state before DU_1 happened in IS_3. So, the final result in the data warehouse will be V', and its content will be $(1,4,3)$. That is, the maintenance result of DU_1 will be lost. In short, we can't simply change the scheduling order for these two updates.

Thus, we can't directly apply the traditional lock-based concurrency control strategy in our mixed data update and schema change DWMS_Transaction environment.

Next, we examine the application of the multi-version concurrency control strategy. That is, a new version of VD is created whenever a writer (schema change DWMS_Transaction) attempts to rewrite it. Any reader will be scheduled to read an appropriate version of VD. Here the assumption exists: $w[VD_i] \preceq r[VD_i]$. That is, any reader can read the ith version data if and only if the writer has already created it. So, there also exists this order constraint problem. As illustrated in Example 1, we need to schedule SC_1 before DU_1 if their TxnIDs are [1,0,0] and [1,0,1] respectively. Otherwise the read operations in [1,0,1] can't read the right version of VD (V' as in the Example 1) which has been created by [1,0,0].

Based on the above analysis, we advocate the following guidelines for our parallel scheduling when interleaving executions of DWMS_Transactions in a mixed data updates and schema changes environment.

- No constraints (in terms of read/write conflicts of the VD in the DWMS_Transaction) exist between any data updates if there is no schema change before these data updates in the UMQ.
- If a schema change currently is being processed by the VS algorithm (w[VD]), then the data updates in the UMQ have to wait until this schema change finishes its VS part.
- We need to keep the order of related data updates and schema changes once we have assigned TxnIDs.
- A schema change can only start its maintenance process when all the data updates with smaller TxnIDs have committed their delta changes to the DW [6].

Given these guidelines, we propose the following basic TxnID-Order-Driven scheduler.

1. Start DWMS_Transaction maintenance processes based on the TxnID order.

[6] We describe how this constraint may be relaxed in [LR02].

2. Start schema change maintenance only if all the previous data updates and schema changes maintenance processes have been committed.

3. Simply block all the subsequent schema changes and data updates once a schema change is being processed;

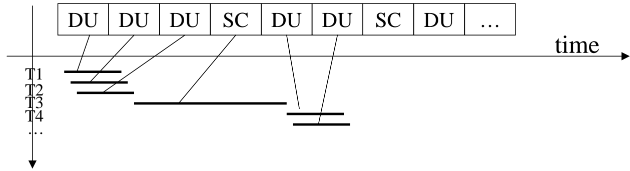

Figure 8: Scheduling Example of Basic TxnID-Order-Driven Algorithm

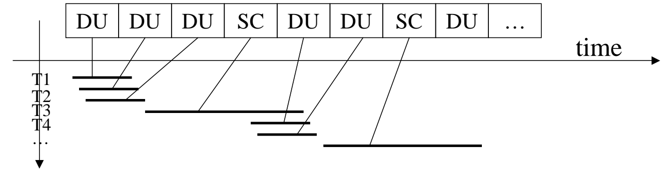

Figure 9: Scheduling Example of Improved TxnID-Order-Driven Algorithm

A sample execution plan is depicted in Figure 8. DU and SC each stands for their corresponding DWMS_Transaction maintenance process. For simplicity, we omit the detailed control procedures of this scheduler since they can easily be derived.

Additional improvements are possible. We don't have to fully block all the subsequent data updates while a schema change is being processed. That is, we could only synchronize the VS part of a schema change (the w[VD] operation), while the scheduler continues to analyze the following updates. If it is a data update, then the scheduler could start maintaining it. If it is a schema change, then the scheduler would continue to keep waiting until the previous schema change has been committed. Figure 9 depicts an example of this improved scheduling plan.

The limitation of this algorithm is that once we assign the TxnIDs based on the arrival order of updates at the DWMS, we then have to keep this order in scheduling. That is, all the following data updates in the UMQ have to wait for the previous schema change to finish its VS part. In the following section, we will relax this ordering constraint and address the corresponding issues related to this restriction by developing a dynamic scheduler.

5.3.2 Dynamic TxnID Scheduler

To have a more flexible scheduler, we first need to determine if it is possible to change the scheduling order of updates in the UMQ while still keeping the DW consistent in a mixed data update and schema change

environment.

Observation 2 *The arrival order of updates at the DWMS doesn't affect the DW maintenance correctness as long as these updates come from different ISs.*

We provide the following as an argument supporting the above observation. Without loss of generality, we define the view in the DW as V = $A_i \bowtie A_j \bowtie$ A, where A is an abbreviation of the join of possibly multiple relations. Assume there are changes $\triangle A_i$ in A_i and $\triangle A_j$ in A_j independently.

- if $\triangle A_i$ arrives at the DWMS before $\triangle A_j$, then the final change to the DW should be $\triangle$V = $\triangle A_i \bowtie A_j \bowtie A + (A_i + \triangle A_i) \bowtie \triangle A_j \bowtie A$.
- if $\triangle A_j$ arrives at the DWMS before $\triangle A_i$, then the final change to the DW should be $\triangle V' = A_i \bowtie \triangle A_j \bowtie A + \triangle A_i \bowtie (A_j + \triangle A_j) \bowtie A$.

As you can see, $\triangle V = \triangle V'$. So, the maintenance result should always be the same even if these two updates come to the DWMS in a different order. The assumption is that we can read the right version of the data during the maintenance process. Compared to the example in Figure 7, the difference is that we now delay the assignment of the TxnID to the update until the time of scheduling. This way, the DWMS can see the correct version of the data.

This observation gives us the hint that we should be able to exchange the scheduling order of updates in the UMQ that come from different ISs as long as we assign the corresponding TxnIDs dynamically. That is, if a schema change arrives, we can postpone its maintenance process, and go on maintaining the following data updates, as long as these data updates come from different ISs than the IS to which the schema change belongs to. This may give us some increased performance because less data updates would be waiting for scheduling. Also, the schema change maintenance is probably more time consuming, so it is reasonable we postpone its maintenance while letting the following data updates, which have a light overhead, be maintained first. Figure 10 is an example of the Dynamic TxnID scheduler execution plan. Here, we assume that we generate a TxnID for each update only when we are ready to schedule it.

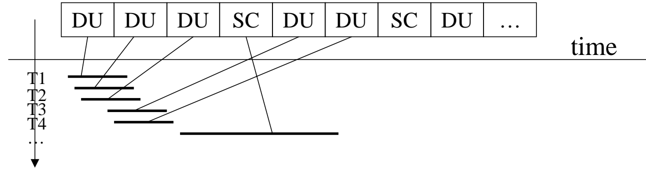

Figure 10: Scheduling Example of Dynamic TxnID Scheduler

5.4 DW Commit and Consistency

Even if each individual data update is being maintained correctly, the final DW state after committing these effects may still be inconsistent. This Variant DW Commit Order problem in a data update only environment has been addressed in [ZRD01]. Not surprisingly, this problem also exists in the mixed data update and schema change environment. Let's examine the following example to illustrate this problem.

Example 2 *Assume the following update sequence in the UMQ: DU,DU,SC,DU,DU,SC,...*

- *The commit problem between the DUs is the same as in [ZRD01].*
- *Based on our parallel scheduler, no commit problem between* $< DU, SC >$ *and* $< SC, SC >$ *sequences can arise because of the ordering constraint we add into our scheduler.*
- *For the sequence* $< SC, DU >$*, its commit problem is also the same as what has been addressed in [ZRD01] because we only start DUs after the previous SC finishes its VS part.*

That is, we can apply the same commit control strategy used in the data update only environments also to our mixed data update and schema change environments.

However, the easiest control strategy is a strict commit order control. That is, only after we commit all the previous $updates'$ effects, could we begin committing the current delta changes to the DW. It also can easily be seen that if every DWMS_Transaction contains only one IS transaction, then this solution will achieve complete consistency [ZGMW96].

Another issue in commit control is if it is possible that we can start the schema change processing before the previous data updates have been committed. Let's review the potential conflict that exists in starting schema change maintenance before all previous data updates have committed. In the commit phase, the

DU DWMS_Transaction actually includes r[VD], w[DW] operations because the data refreshed in the DW should be consistent with the DW schema (its view definition), while the SC DWMS_Transaction includes the operations r[VD],w[VD],$r[IS_1], \ldots$. If we interleave these operations, a 'dirty read' problem could arise. However, we can start the SC maintenance before previous DUs committing in some cases to improve the performance of the scheduler further. For example, the 'drop relation' operation. If the DU comes from the same IS, then we can safely drop the effects in the commit phase if the SC commits before the DU. While if the DU is from a different IS, then we can change the schema of the effects to fit into the new VD before committing and the final result will still be the same.

6 Design and Implementation

We have completed the implementation of our parallel maintenance scheduler and incorporated it into the existing TxnWrap system developed by the Database Systems Research Group at WPI. We use java threads to encapsulate the DWMS_Transaction maintenance of the updates and correspondingly interleave the executions of these threads. That is, we impose no constraints on data updates' maintenance threads while we have related threads wait whenever a schema change maintenance thread is running. Figure 11 shows the abstract view of the parallel scheduler in the TxnWrap system extracted from Figure 5.

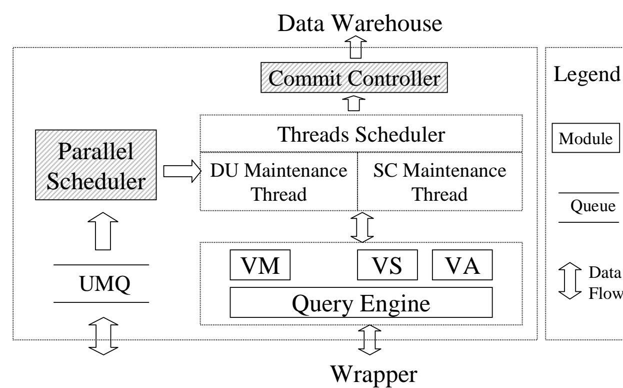

Figure 11: Implementation View of PTxnWrap

A parallel scheduler in the DU only environment is straightforward to implement because there is no constraint (no read block between DWMS_Transactions) between threads. Here we briefly introduce how the improved Fix-Order scheduler in a mixed DU and SC environment be implemented in the system. We

have four variables to control the ordering constraint we addressed in Section 5.3.1.

- *latest_started_id*. To control the start sequence of updates.
- *latest_completedSC_id*. To control the sequential order when we execute SCs.
- *ongoing_SC_id*. To control the maintenance of the DUs after one SC could be started only this SC has completed its VS part or no SC is running.
- *latest_committed_id*. To control the commit sequence of updates to make sure that SC could only starts its maintenance when all previous DUs have been committed.

Logically, we need to assign every SC an individual ID to identify the sequence of schema changes, we call it a SC_id. And for all updates, we use its TxnID as the global_id. Figure 12 depicts the control strategies implemented in the improved Fix-Order Scheduler.

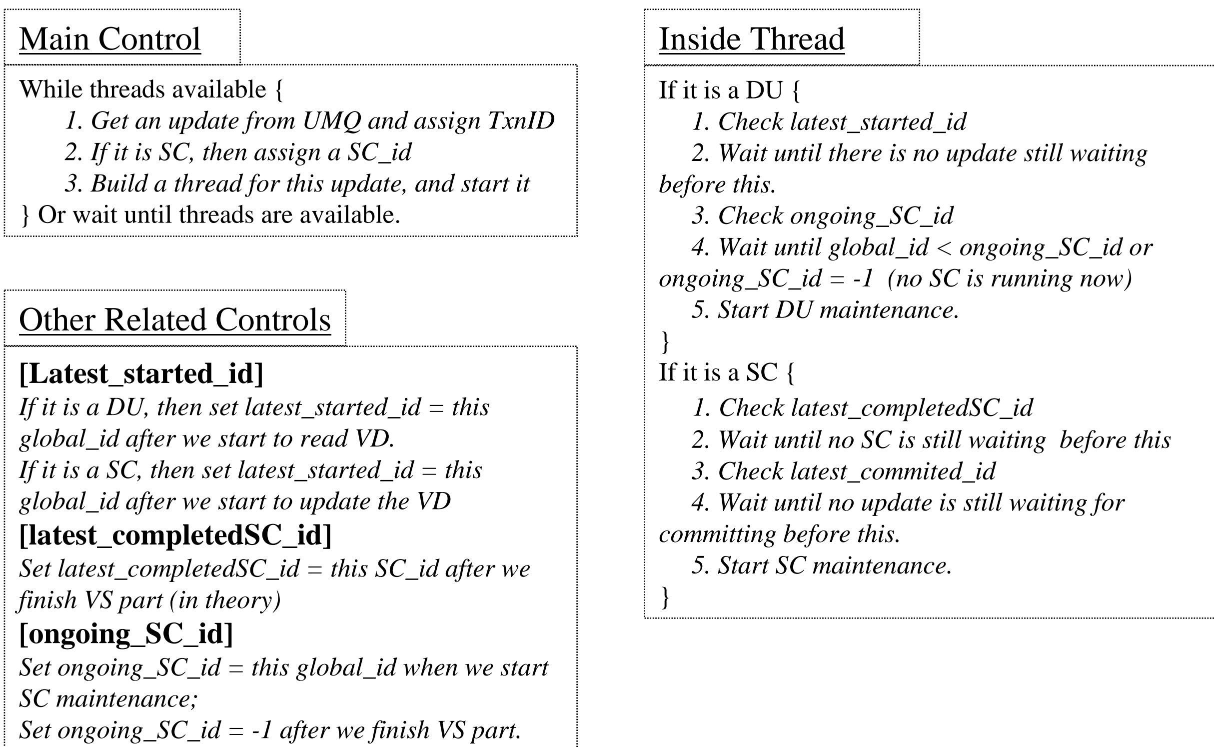

Figure 12: Control Strategies in the improved Fix-Order Scheduler

7 Performance Studies

7.1 Experimental Environment

Our experimental environment uses a local network and four machines (Pentium III PC with 256M memory, running Windows NT workstation OS and Oracle 8.1.6.0.0 Server). Three of them serve as Database (DB) servers, while the fourth one as the DWMS server. Each DB server contains several IS relations depending on the specific requirements of each experiment. Typically, there is one materialized join view defined in the DWMS machine over all IS relations.

7.2 Cost Measurements

TxnWrap focuses on removing concurrencies from the DW maintenance process, while our parallel scheduler focuses on interleaving the execution of the maintenance processes. So we measure the total cost (the processing time) of the third and the fourth steps of DWMS_Transactions (The DWMS_Transaction maintenance process) for a set of updates to reflect its performance. To make the cost measurements clearer, the processing time of a DWMS_Transaction maintenance is composed of three categories:

- **Query-Time**. The time span between the DB server receiving the query request and the DB server returning the desired results.
- **Network-Delay**. The time span between the DWMS sending out the maintenance query and the underlying IS (the DB server) receiving the query request.
- **CPU-Time**. The time spent in the DWMS to break down the maintenance queries, collect the results and all the other miscellaneous scheduling overheads.

Note that in our experimental environment, the CPU overhead can't be reduced because we use a single CPU machines. Also note that the network delay in such a local network is almost neglectable.

The main variable in our experiments is thus the number of parallel threads under a fixed number of concurrent updates and a fixed number of ISs. For the scheduler in a mixed data update and schema change environment, the distribution of schema changes and the number of schema changes will also affect the scheduler performance. We vary the number of information sources, the size of each information source,

the schema of each information source and the physical distribution of the ISs to observe the performance in such different settings.

7.3 Experiments

7.3.1 Aggressive Scheduler

Experiment 1: The goal of this experiment is to measure the effect of changing the number of threads on the corresponding total processing time. We set up are nine information sources and one view defined as a join of these nine ISs. These ISs are evenly distributed over three DB servers located on different machines. Each IS has two attributes and 1000 tuples. We use 100 concurrent data updates as our sample. The change of total processing time is depicted in Figure 13. The x-axis denotes the number of parallel threads in the system, with S denoting the serial scheduler. The y-axis represents the total time of processing these 100 concurrent data updates.

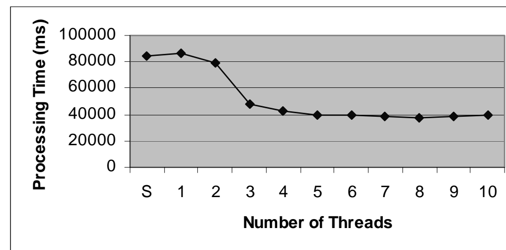

Figure 13: Change the number of Threads

In Figure 13, if we only use one thread, then the total processing time is slightly higher than the serial one. This is due to the overhead of the parallel maintenance scheduler logic and thread management. Around thread number 5, the total processing time reaches its minimal. If we further increase the thread number, the processing time will stay stable. This can be explained by possible additional system overhead such as the maintenance queries processed by ISs are blocked by each other at every IS because the query capability of each IS is limited.

The maximum percentage of performance improvement we measured in this scenario is around 53%. The reason is that in our experimental environment, the CPU overhead can't be fully reduced by our multi-threading solution because we use a one-CPU DWMS server. On the other hand, the network delay in a

local network environment is typical less than 1ms, so the total Network-Delay in the maintenance process is very small. The Query-Time is also relatively small compared to the CPU cost. Thus, the Query-Time and Network Delay, which are the two tasks that are parallelized, is too small in the total processing time. For this reason, an improvement linear in the thread number is not achieved.

Experiment 2: For this experiment, we change the number of tuples in each IS to measure the effect of increasing the Query-Time of each maintenance query on system performance. Assume there are three information sources and the view defined in the DW is the join of these three ISs. These ISs are evenly distributed over three DB servers each in a different machine. Each IS has two attributes and the number of tuples in each IS is increased from 1000 to 50000. We use 100 concurrent data updates. The change of the total processing time is depicted in Figure 14. Similar to experiment 1, the x-axis and y-axis represent the number of threads and total processing time respectively. From Figure 14, we can see that if we increase

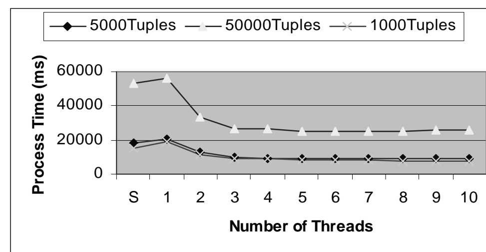

Figure 14: Change the Number of Tuples in each IS

the number of tuples in each IS, the total processing time will increase. This is as expected. Also, the improvement percentage is increased slightly, from around 50% to around 53%. This can be explained by the fact that the percentage of the Query-Time in the total processing time also increased, which in turn can be partly run in parallel. So the performance gain increases correspondingly.

Experiment 3: The goal of this experiment is to measure the effect of changing the Network-Delay. It is similar to experiment 2 except each IS has 1000 tuples. We list the performance changes in Figure 15 from no network-delay, 100ms and then 200ms delay in each maintenance query. From Figure 15, we can see that the larger the network delay, the more performance improvement is being achieved. This is so because we can fully make use of the network delay in the parallel scheduler.

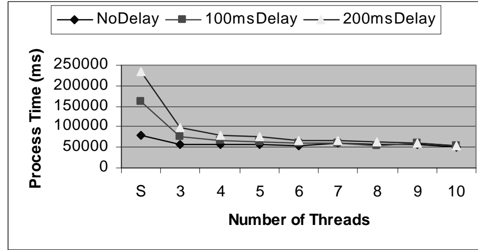

Figure 15: Change the Network-Delay in each Maintenance Query

7.3.2 Mixed Data Updates and Schema Changes Scenarios

Experiment 4: We now repeat experiment 1 in Section 7.3.1 in a mixed data update and schema change environment using the improved TxnID-Order-Driven scheduler, allowing us to compare the performance of these two different schedulers. In this experiment, we assume there are six information sources and the view defined in the DW is the join of the three of these six ISs. These ISs are evenly distributed over three DB Servers on different machines. Each IS has two attributes and 1000 tuples. We use 100 concurrent data updates and two schema changes [7]. The change in total processing time is depicted in Figures 16 and 17.

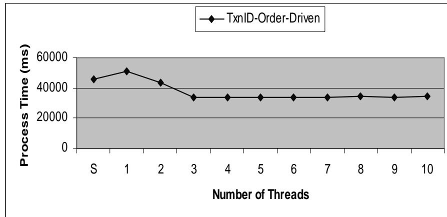

Figure 16: Change the Number of Threads in the TxnID-Order-Driven Scheduler

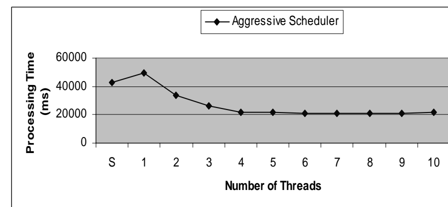

Figure 17: Change the Number of Threads in the Aggressive Scheduler

From Figure 16, we can see that the maximum improvement achieved by the improved TxnID-Order-Driven scheduler reaches around 30%. Compared with the aggressive scheduler under a similar (the difference is the extra two schema changes in the TxnID-Order-Driven scheduler) scenario, the performance improvement of this scheduler is somewhat decreased. This is because of the ordering constraint in the TxnID-Order-Driven scheduler and the extra overhead in commit control.

Experiment 5: The goal of this experiment is to measure the effect of changing the number of schema

[7]The different distribution of schema changes and data updates will have a little impact on the performance. Here we just randomly pick one of the scenarios.

changes on the performance of improved TxnID-Order-Driven Scheduler. We use six information sources and the view is defined on three of them. Each IS has two attributes and 10000 tuples. We use 100 concurrent data updates and change the number of schema changes from 0 to 3. The change in the total processing time is depicted in Figure 18.

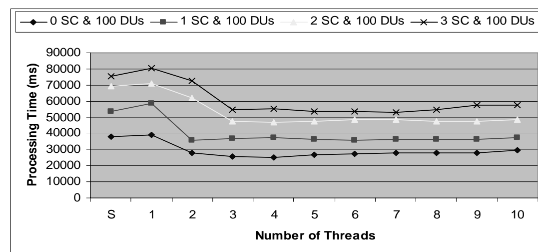

Figure 18: Change the Number of Schema Changes

From Figure 18, we see that the total processing time increases if we add more schema changes because a schema change maintenance is much more time consuming than that of a data update. Furthermore, we also observe that if we add more schema changes, the maximum improvement achieved by the scheduler will decrease. This is because we can't fully maintain schema changes in parallel and all the subsequent data updates have to wait when a schema change is being maintained. Another point about this scheduler is that if we go on increasing the number of parallel threads, the total processing time will also increase a little. This is due to the extra overhead on the commit controller caused by an increase in updates waiting to be committed.

8 Related Work

Maintaining materialized views under source updates in a data warehouse environment is one of the important issues of data warehousing [GM95, Wid95, WB97, GMLWZ98]. 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 [BLT86, GMS93, CGL$^+$96, LMSS95]. There has also been some work on maintaining materialized view definitions under IS schema changes [KLZ$^+$97, LKNR99], and on adapting its view extent under IS schema changes [GMR95, MD96, NR99, ZR00].

Self-Maintenance [AJR97, GJM96, QGMW96] make views self-maintainable by storing auxiliary data

at the data warehouse so that the warehouse data can be maintained without accessing any source data. Recently, [YW00] also have extended this concept to temporal views. [Lab99] summarize limited source access approaches.

In approaches that need to send maintenance queries down to the IS space, concurrency problems can arise. [ZGMHW95] introduced the ECA algorithm for incremental view maintenance under concurrent IS data updates restricted to a single IS. Strobe [ZGMW96] handles multiple ISs while still assuming the schemata of all ISs to be static. SWEEP [AASY97] ensures the consistency of the data warehouse in a larger number of cases compared to Strobe [ZGMW96]. As a compromise between self-maintenance and view maintenance via compensation, [Huy97] proposes an intermediate approach to maintenance without using all the base relations but with requiring additional views to facilitate maintenance.

DyDa [ZR00] addresses concurrent SCs by integrating VS, VA and VM into one system to deal with data update and schema change maintenance. DyDa is a heavy-weight solution in the sense that it adds complexity of concurrency detection and handling not only in each VS, VA and VM maintenance module but also requires special strategies of coordination between these modules to achieve overall correct compensation.

TxnWrap [CR00] is the first transactional approach to handle the concurrency for both DUs and SCs. It succeeds in requiring little cooperation from the IS space, while still being able to deal with both DUs and SCs. Its beauty lies in its simplicity. It exploits basic techniques from the established field of transaction and concurrency control theory. A general discussion of classical multiversion concurrency control algorithms can be found in [BHG87, Bha99]. TxnWrap encapsulates each maintenance process in a DWMS_Transaction and uses a multiversion concurrency control algorithm to guarantee a consistent view of data inside each DWMS_Transaction.

In the context of maintaining a DW in parallel, PVM [ZRD01] addresses the problem of concurrent data update detection in a parallel execution mode and the variant DW commit order problem. However, PVM works in a data update only environment. Extension of this approach when considering schema changes would be rather complex, requiring extra coordination algorithms.

9 Conclusions

In this paper, we identify the performance limitation of TxnWrap in terms of sequential scheduling of all DW maintenance tasks. Based on the DWMS_Transaction model, we formalize the constraints that exist in maintaining data updates and schema changes concurrently. We propose different parallel schedulers. We also implement a parallel DW maintenance system based on TxnWrap. The experimental results reveal that parallel maintenance scheduler exhibits an excellent performance compared to TxnWrap in maintaining a set of concurrent updates in a dynamic environment.

References

[AASY97] D. Agrawal, A. El Abbadi, A. Singh, and T. Yurek. Efficient View Maintenance at Data Warehouses. In *Proceedings of SIGMOD*, pages 417–427, 1997.

[AJR97] P. Ammann, S. Jajodia, and I. Ray. Applying Formal Methods to Semantic-Based Decomposition of Transactions. *ACM Transactions on Database Systems (TODS)*, 22(2):215–254, June 1997.

[Bha99] B. Bhargava. Concurrency control in database systems. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 11(1):3–16, January 1999.

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

[BLT86] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently Updating Materialized Views. In C. Zaniolo, editor, *Proceedings of SIGMOD*, pages 61–71, Washington, DC, May 1986. acm.

[CGL+96] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for Deferred View Maintenance. In *Proceedings of SIGMOD*, pages 469–480, 1996.

[CR00] J. Chen and E. A. Rundensteiner. Txnwrap: A transactional approach to data warehouse maintenance. Technical report, Worcester Polytechnic Institute, November 2000.

[CZC+01] 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, Santa Barbara, CA, May 2001.

[DZR99] L. Ding, X. Zhang, and E. A. Rundensteiner. The MRE Wrapper Approach: Enabling Incremental View Maintenance of Data Warehouses Defined On Multi-Relation Information Sources. In *Proceedings of the ACM First International Workshop on Data Warehousing and OLAP (DOLAP'99)*, pages 30–35, November 1999.

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

[GM95] A. Gupta and I.S. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Engineering Bulletin, Special Issue on Materialized Views and Warehousing*, 18(2):3–19, 1995.

[GMLWZ98] Héctor García-Molina, Wilburt L., Janet L. Wiener, and Yue Zhuge. Distributed and Par-

allel Computing Issues in Data Warehousing . In *Symposium on Principles of Distributed Computing*, page 7, 1998. Abstract.

[GMR95] A. Gupta, I.S. Mumick, and K.A. Ross. Adapting Materialized Views after Redefinition. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 211–222, 1995.

[GMS93] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. In *Proceedings of SIGMOD*, pages 157–166, 1993.

[Huy97] N. (Pierre) Huyn. *Maintaining Data Warehouse Under Limited Source Access*. PhD thesis, Stanford University, August 1997.

[KLZ+97] A. Koeller, Y. Li, X. Zhang, A.J. Lee, A.Nica, and E.A. Rundensteiner. *Evolvable View Environment (EVE): Maintaining Views over Dynamic Distributed Information Sources*. Centre for Advanced Studies Conference, November 1997.

[Lab99] W. J. Labio. *Efficient Maintenance and Recovery of Data Warehouse*. PhD thesis, Stanford University, August 1999.

[LKNR99] A. J. Lee, A. Koeller, A. Nica, and E. A. Rundensteiner. Non-Equivalent Query Rewritings. In *International Database Conference*, Hong Kong, July 1999.

[LMSS95] James J. Lu, Guido Moerkotte, Joachim Schue, and V. S. Subrahmanian. Efficient Maintenance of Materialized Mediated Views. In *Proceedings of SIGMOD*, pages 340–351, San Jose, California, May 1995.

[LR02] Bin Liu and E. A. Rundensteiner. PTxnWrap: Parallel Data Warehouse Maintenances. Technical Report inproceeding, Worcester Polytechnic Institute, Dept. of Computer Science, 2002.

[MD96] M. Mohania and G. Dong. Algorithms for Adapting Materialized Views in Data Warehouses. *International Symposium on Cooperative Database Systems for Advanced Applications*, pages 353–354, December 1996.

[NR98] A. Nica and E. A. Rundensteiner. Using Containment Information for View Evolution in Dynamic Distributed Environments. In *Proceedings of International Workshop on Data Warehouse Design and OLAP Technology (DWDOT'98)*, Vienna, Austria, August 1998.

[NR99] A. Nica and E. A. Rundensteiner. View Maintenance after View Synchronization. In *International Database Engineering and Applications Symposium (IDEAS'99)*, pages 213–215, August, Montreal, Canada 1999.

[QGMW96] D. Quass, A. Gupta, I. S. Mumick, and J. Widom. Making Views Self-Maintainable for Data Warehousing. In *Conference on Parallel and Distributed Information Systems*, pages 158–169, 1996.

[RLN97] E. A. Rundensteiner, A. J. Lee, and A. Nica. On Preserving Views in Evolving Environments. In *Proceedings of 4th Int. Workshop on Knowledge Representation Meets Databases (KRDB'97): Intelligent Access to Heterogeneous Information*, pages 13.1–13.11, Athens, Greece, August 1997.

[SBCL00] 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.

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

[Wid95] J. Widom. Research Problems in Data Warehousing. In *Proceedings of International Conference on Information and Knowledge Management*, pages 25–30, 1995.

[YW00] J. Yang and J. Widom. Temporal view self-maintenance in a warehousing environment.

Technical report, Stanford University, Computer Science Department, February 2000.

[ZGMHW95] Y. Zhuge, Héctor García-Molina, J. Hammer, and J. Widom. View Maintenance in a Warehousing Environment. In *Proceedings of SIGMOD*, pages 316–327, May 1995.

[ZGMW96] Y. Zhuge, Héctor García-Molina, and J. L. Wiener. The Strobe Algorithms for Multi-Source Warehouse Consistency. In *International Conference on Parallel and Distributed Information Systems*, pages 146–157, December 1996.

[ZR00] X. Zhang and E. A. Rundensteiner. DyDa: Dynamic Data Warehouse Maintenance in a Fully Concurrent Environment. In *Data Warehousing and Knowledge Discovery, Proceedings*, pages 94–103. Lecture Notes in Computer Science (LNCS) by Springer Verlag, September 2000.

[ZRD01] X. Zhang, E. A. Rundensteiner, and L. Ding. PVM: Parallel View Maintenance Under Concurrent Data Updates of Distributed Sources. In *Data Warehousing and Knowledge Discovery, Proceedings*, September, Munich, Germany 2001.

A Proof of PTxnWrap

Theorem 1 *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. [BHG87]*

Assume there are k ISs, denoted as IS_1, IS_2,..., IS_k. We use $IS_i(j)$ to denote the result state of IS_i after having been updated by the first-step of a DWMS_Transaction with *local id* j in IS_i. We refer to this as the jth version of IS_i. In this context, we denote all ISs' initial states to have version number 0. TxnID of a DWMS_Transaction is generated by the DWMS as soon as the update message arrives in the DWMS. As mentioned in previous sections, TxnID is the vector timestamps which record the latest version number of each IS when the update comes to the DWMS. We use $\tau_1, \tau_2, \cdots, \tau_i$ to denote TxnIDs. Similarly, we use DW(τ_i) to denote the result commit to the DW by the DWMS_Transaction with TxnID τ_i. And also we use DW(0) to denote its initial state.

As in [BHG87], we use $r_{\tau_i}[x]$ (or $w_{\tau_i}[x]$) to denote the execution of a Read (or Write) issued by transaction T_{τ_i} on a data item x. We also use c_{τ_i} to denote $T'_{\tau_i}s$ commit operation. Notice that we don't have abort for the DWMS_Transactions. We use x_{τ_i} to denote the version data that used by DWMS_Transaction with TxnID τ_i. The order of version data can be defined as follows.

- $x_{\tau_i} < x_{\tau_j} \iff \tau_i < \tau_j$, where τ_i, τ_j are TxnIDs of corresponding DWMS_Transactions.

The following is an illustrative example. Assume three DWMS_Transactions arrive in the DWMS. T_{τ_1} from IS_m with *local id* 1, T_{τ_2} from IS_n with *local id* 1, and T_{τ_3} again from IS_m with *local id* 2,

assume $1 \leq m, n \leq k$ with k is the number of ISs. The initial versions of the IS extents presented by their respective wrappers are $IS_1(0), IS_2(0), ..., IS_k(0)$ and the inital DW extent is DW(0). The these three DWMS_Transactions can be represented by:

1. $T_{\tau_1} = w_{\tau_1}[IS_m(1)]r_{\tau_1}[IS_1(0)]\ldots r_{\tau_1}[IS_m(1)]\ldots r_{\tau_1}\ [IS_n(0)]\ldots r_{\tau_1}[IS_k(0)]w_{\tau_1}[DW(\tau_1)]c_{\tau_1}$
2. $T_{\tau_2} = w_{\tau_2}[IS_n(1)]r_{\tau_2}[IS_1(0)]\ldots r_{\tau_2}[IS_m(1)]\ldots r_{\tau_2}\ [IS_n(1)]\ldots r_{\tau_2}[IS_k(0)]w_{\tau_2}[DW(\tau_2)]c_{\tau_2}$
3. $T_{\tau_3} = w_{\tau_3}[IS_m(2)]r_{\tau_3}[IS_1(0)]\ldots r_{\tau_3}[IS_m(2)]\ldots r_{\tau_3}\ [IS_n(1)]\ldots r_{\tau_3}[IS_k(0)]w_{\tau_3}[DW(\tau_3)]c_{\tau_3}$

For the serial schedule, the following is a sample history (H_1) of T_{τ_1}, T_{τ_2} and T_{τ_3}:

- H_1=$w_{\tau_1}[IS_m(1)]w_{\tau_2}[IS_n(1)]w_{\tau_3}[IS_m(2)]r_{\tau_1}[IS_1(0)]\ldots r_{\tau_1}[IS_m(1)]\ldots$
  $r_{\tau_1}[IS_n(0)]\ldots r_{\tau_1}[IS_k(0)]w_{\tau_1}[DW(\tau_1)]c_{\tau_1}r_{\tau_2}[IS_1(0)]\ldots r_{\tau_2}[IS_m(1)]\ldots r_{\tau_2}[IS_n(1)]\ldots$
  $r_{\tau_2}[IS_k(0)]w_{\tau_2}[DW(\tau_2)]c_{\tau_2}r_{\tau_3}[IS_1(0)]\ldots r_{\tau_3}[IS_m(2)]\ldots r_{\tau_3}[IS_n(1)]\ldots r_{\tau_3}[IS_k(0)]w_{\tau_3}[DW(\tau_3)]c_{\tau_3}$

For the parallel schedule, we interleave the execution of the third-step of DWMS_Transactions. The following (H_2) is a sample history of such schedule.

- H_2=$w_{\tau_1}[IS_m(1)]w_{\tau_2}[IS_n(1)]w_{\tau_3}[IS_m(2)]\ r_{\tau_1}[IS_1(0)]\ldots r_{\tau_1}[IS_m(1)]\ldots r_{\tau_1}[IS_n(0)]\ \ldots$
  $r_{\tau_2}[IS_1(0)]r_{\tau_2}[IS_m(1)]r_{\tau_1}[IS_k(0)]w_{\tau_1}[DW(\tau_1)]c_{\tau_1}\ r_{\tau_3}[IS_1(0)]\ldots\ r_{\tau_3}[IS_m(2)]\ldots r_{\tau_3}[IS_n(1)]\ldots$
  $r_{\tau_3}[IS_k(0)]\ r_{\tau_2}[IS_n(1)]\ldots r_{\tau_2}[IS_k(0)]\ w_{\tau_2}[DW(\tau_2)]c_{\tau_2}w_{\tau_3}[DW(\tau_3)]c_{\tau_3}$

For both histories, we can construct the MVSG according to its definition [BHG87]. That is, we add the edge $T_{\tau_1} \rightarrow T_{\tau_2}$ since $r_{\tau_2}[IS_m(1)]$ reads the result from $w_{\tau_1}[IS_m(1)]$; add edge $T_{\tau_2} \rightarrow T_{\tau_3}$ since $r_{\tau 3}[IS_n(1)]$ reads the result from $w_{\tau_2}[IS_n(1)]$; add edge $T_{\tau_1} \rightarrow T_{\tau_3}$ since $w_{\tau_1}[IS_m(1)]$ precedes $w_{\tau_3}[IS_m(2)]$. There are no more version order edges. Thus G is an acyclic MVSG graph in this example.

We now prove both schedules by contradiction. The DWMS keeps all arriving update messages in a queue. At any time t, each update message in this queue represents a DWMS_Transaction. Thus, let's denote this set of DWMS_Transactions as T=$T_{\tau_1}, T_{\tau_2}, \ldots, T_{\tau_k}$, which $\tau_1, \tau_2, \ldots, \tau_k$ is its corresponding TxnID.

Theorem 2 *Given any DWMS_Transaction set queuing in the DW, the multiversion serializable graph G of any history over this set of DWMS_Transactions is acyclic. More strictly, all version order edges in G*

are pointing from the DWMS_Transaction T_{τ_i} with small TxnID 'τ_i' toward another DWMS_Transaction T_{τ_j} with larger TxnID 'τ_j'.

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 two possible reasons to add this edge to the MVSG graph.
   (a) If this edge is added by the serial graph(SG) definition [BHG87], This can't be true because in TxnWrap and PTxnWrap algorithm, we always assign TxnID only if its corresponding versions have been built.
   (b) If this edge is added by additional MVSG definition [BHG87], then there only have two cases to consider:
      - $w_{\tau_j}[x_{\tau_j}] \ldots r_{\tau_k}[x_{\tau_i}]$ and $x_{\tau_j} < x_{\tau_i}$. This is impossible because we assume the version order $x_{\tau_i} < x_{\tau_j}$ if $\tau_i < \tau_j$.
      - $r_{\tau_j}[x_{\tau_k}] \ldots w_{\tau_i}[x_{\tau_i}]$ and $x_{\tau_k} < x_{\tau_i}$. 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 TxnWrap and PTxnWrap because we always assign the latest version number (local id) to DWMS_Transaction in the TxnID.
3. Contradition: 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 TxnID $\tau_i < \tau_j$. So the MVSG is acyclic.