Optimization Strategies for Data Warehouse Maintenance in Distributed Environments

by

Bin Liu

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

by

April 2002

APPROVED:

Professor Elke A. Rundensteiner, Thesis Advisor

Professor George Heineman, Thesis Reader

Professor Micha Hofri, Head of Department

Abstract

Data warehousing is becoming an increasingly important technology for information integration and data analysis. Given the dynamic nature of modern distributed environments, both source data updates and schema changes are likely to occur autonomously and even concurrently in different data sources. Current approaches [31, 5] to maintain a data warehouse in such dynamic environments sequentially schedule maintenance processes to occur in isolation. Furthermore, each maintenance process is handling the maintenance of one single source update. This limits the performance of current data warehouse maintenance systems in a distributed environment where the maintenance of source updates endures the overhead of network delay as well as IO costs for each maintenance query.

In this thesis work, we propose two different optimization strategies which can greatly improve data warehouse maintenance performance for a set of source updates in such dynamic environments. Both strategies are able to support source data updates and schema changes. The first strategy, the parallel data warehouse maintainer, schedules multiple maintenance processes concurrently. Based on the DWMS_Transaction model, we formalize the constraints that exist in maintaining data and schema changes concurrently and propose several parallel maintenance process schedulers. The second strategy, the batch data warehouse maintainer, groups multiple source updates and then maintains them within one maintenance process. We propose a technique for compacting the initial sequence of updates, and then for generating delta changes for each source. We also propose an algorithm to

adapt/maintain the data warehouse extent using these delta changes. A further optimization of the algorithm also is applied using shared queries in the maintenance process.

We have designed and implemented both optimization strategies and incorporated them into the existing DyDa/TxnWrap system. We have conducted extensive experiments on both the parallel as well as the batch processing of a set of source updates to study the performance achievable under various system settings. Our findings include that our parallel maintenance gains around $40 \sim 50\%$ performance improvement compared to sequential processing in environments that use single-CPU machines and little network delay, i.e, without requiring any additional hardware resources. While for batch processing, an improvement of $400 \sim 500\%$ improvement compared with sequential maintenance is achieved, however at the cost of less frequent refreshes of the data warehouse content.

Acknowledgements

I have many people to thank for the help and encouragement on this work toward my master's degree. First i will thank my advisor, Professor Elke A. Rundensteiner, for her precious advice and support during this study. With her guidance and understanding i was able to finish the thesis. Great thanks are also to my thesis reader, Professor George Heineman. He gave me constructive feedback on my work which is important on my thesis.

I would also thank Songting Chen for his help and discussions on my thesis work. I would also thank to Xin, Jehad, Lily and all the other members in DSRG, who gave me a lot of help.

I would also thank WPI computer science department, which gave me a chance to study and work to earn my master's degree. And also thank to all the faculty members in the department who have helped me in this period.

A special thank to my family, my wife Xiaojie and my parents, who support me all the time.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Data Warehouse Maintenance

Data Warehouses (DW) [11, 21] are built by gathering data from data sources and integrating it into one repository customized to users' need. Data warehousing is important for many applications, especially in large-scale environments composed of distributed sources, such as travel services, E-commerce and decision support systems. A data warehouse management system (DWMS) is the management system that is responsible of maintaining the data warehouse extent and schema upon changes of the underlying sources. In distributed environments, these remote sources 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 data warehouse, in particular, the materialized views defined upon them.

When incrementally maintaining data warehouses under source updates, maintenance problems will arise due to the independence and autonomy between data sources and data warehouses.

Let's first illustrate the maintenance problems in such dynamic environments via some running examples. As depicted in Figure 1.1, the data warehouse 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 data warehouse, then the data warehouse 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 (DU_1) into the view V. The data warehouse 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 relation S respectively:

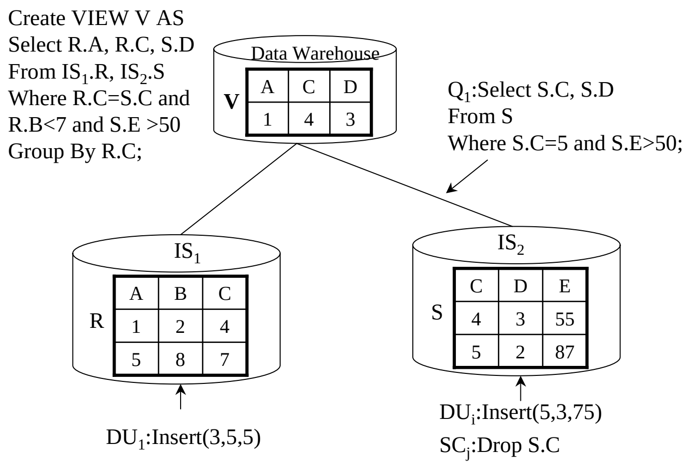


Figure 1.1: Explanations of Maintenance Anomaly Problems.

- Case 1: Assume during the transfer time of the maintenance query Q_1 to relation 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, thus an extra tuple (3,5,3) would be inserted into the view due to this maintenance query. However, when the data warehouse starts processing DU_i later, the same tuple would be inserted into the view again. A **duplication anomaly problem** appears [33].

- 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 can not even 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** arises [31].

We use term concurrent updates to describe such source updates and refer to the above data warehouse maintenance problems that caused by concurrent updates as 'anomaly problems'. To summarize, the problem is how to execute maintenance queries while the data warehouse doesn't know the current state of the underlying sources due to the data warehouse and the sources are operating independently. Thus in a dynamic environment, the maintenance queries to sources may contain incorrect query results or may even fail to complete due to schema changes [31].

1.2 State-of-the-Art in Data Warehouse Maintenance

There are three tasks related to maintaining the data warehouse in such dynamic environments. Incremental view maintenance (VM) [33, 1, 28, 32] maintains the data warehouse extent whenever a data update occurs within a source. View synchronization (VS) [15] 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. View adaptation (VA) [12, 24, 21] adapts the view extent incrementally after the view definition has been modified either directly by the data warehouse designer or indirectly by the view synchronization system.

Most work in the literature only handles data warehouse maintenance problems in a data update only environment [33, 1, 28]. DyDa [31] is the first system to handle concurrent schema change and data update maintenances. It employs a compensation query based strategy. DyDa is complex in the sense that concurrency detection and handling had to be significantly extended to support the schema changes. TxnWrap [5] is the first transactional approach to handle the concurrency for both data and schema changes. It introduces the concept of a DWMS_Transaction model [5] to formally capture the overall data warehouse maintenance process as a transaction. Once cast in terms of transaction concepts, a multiversion timestamp-based concurrency control algorithm [3], called ShadowWrapper, can solve the anomaly problems in data warehouse maintenance. However, like other solutions in the literature, both DyDa and TxnWrap apply a sequential approach towards maintaining concurrent updates. Furthermore, each maintenance process only takes care of one single source update. This limits its performance in a distributed environment where maintaining of source updates endures the overhead of network delay and IO costs in each maintenance query.

Thus it is worthwhile to investigate techniques for parallelizing the maintenance processes by running multiple maintenance processes at the same time. On the other hand, we also can try to maintain multiple source updates by one single maintenance process to reduce the total maintenance cost for a given set of source updates, along the line of batch processing. Both optimization ideas [32, 28, 7, 22] are well addressed in the literature in the context of data update only environments. The contribution of my thesis is to apply optimization strategies when both data updates and schema changes are present and also concurrent with one another. That is, in a fully dynamic data warehousing environment.

1.3 Proposed Optimization Strategies

Parallel Data Warehouse Maintenance. We propose to develop a parallel maintenance scheduler that is capable of maintaining concurrent data and schema changes in parallel. We also show that this significantly improves the performance of data warehouse maintenance. We have chosen TxnWrap [5] as the basis of our parallel processing solution because the transactional approach of TxnWrap provides a formal way to analyze conflicts (in terms of read/write of critical resources) that exist in data update maintenance and schema change maintenance. Our proposed parallel maintenance scheduler (PMS) thus overcomes TxnWrap's performance limitation by parallelizing the executions of different maintenance tasks. To achieve this, three issues must be tackled. First, we characterise all potential conflicts among the data warehouse maintenance processes in terms of read/write of critical resources. Second, we design strategies to generate possible schedules that resolve these identified conflicts. Three algorithms have been proposed that each applicable to distinct situations: an aggressive scheduler that can handle data update only environments and the TxnID-Order-Driven and Dynamic-TxnID schedulers that both can maintain updates in parallel when source data and schema changes are present. Lastly, we examine the commit problem for each maintenance task in parallel processing.

Batch Data Warehouse Maintenance. For batch data warehouse maintenance in such dynamic environments, three steps are proposed. First, group all the source updates based on the relation they come from, and analyze the relationship between data updates and schema changes from the same source. Second, evolve the view definition based on the schema changes and calculate the delta changes of each source. The third, adapt the view extent using these delta changes. One optimization strategy is proposed by making use of shared queries in a distributed environment when incrementally adapting the view extent.

Note that the target application domains of these two optimization strategies are also somewhat different. Parallel maintenance is more suitable in situations when the source updates are more distributed into different sources. For example, the view in the data warehouse is defined on 5 distributed sources. If in a certain period, 5 source updates have happened which each source has one update. Thus parallel processing of these maintenance tasks can fully make use of the processing capability of individual sources. While for the batch maintenance, the maximal performance gains are likely to be achieved when source updates are concentrated in a relative small number of sources. As described in the above example, if all these 5 updates are come from the same source, then there is only one batch maintenance task involved. Thus the total number of operations will be reduced and so will the total maintenance time.

We have designed and implemented both the parallel scheduler and the batch maintenance system based on TxnWrap [5] using Java, with Oracle8i as data server for sources and materialized views and JDBC for connection to Oracle8i. We have conducted experiments to measure the performance of parallel and batch processing under various environmental settings including the number of sources involved, view definitions in the data warehouse, network delay of the maintenance query, and so on. The experimental results confirm that both parallel and batch data warehouse maintenance achieve an excellent performance improvement compared to basic TxnWrap processing.

1.4 Assumptions and Restrictions

In this work, we have made the following assumptions on the data warehouse environments which were also made by most of previous research [33, 1] in this area.

Assumption 1 *We assume a centralized data warehouse system which means there is only one database management system that stores and maintains the materalized views.*

Assumption 2 *We assume that all source transactions are local and every data update and schema change at a source is reported to the DWMS once it is committed at the source.*

Assumption 3 *The data warehouse network environment will not have permanent unrecoverable failure and the message transfer through the network between sources and data warehouse is First-In-First-Out (FIFO).*

Assumption 4 *Each underlying source is autonomous, which means it is only responsible of answering maintenance queries from data warehouses and sending out update notifications.*

Assumption 5 *We assume that there is only one relation in each underlying data source.*

1.5 Contributions

In summary, this thesis provides the following contributions:

- For parallel data warehouse maintenance,

  - Identify the performance limitation of the TxnWrap system (and other data warehouse maintenance systems in general) 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 updates in a mixed schema change and data update environment.
  - Propose several parallel scheduling strategies suitable for different environments. Address solution strategies for the DW commit problems in parallel scheduling.
  - Design and implement the proposed parallel schedulers and incorporate them into existing TxnWrap [5] data warehousing system.
  - Conduct extensive experimental studies, with the results illustrating the performance improvements achievable due to parallel processing.

- For batch data warehouse maintenance,
  - Analyze the relationship between source data update and schema change maintenance processes.
  - Formalize the calculation of the delta changes of each source in situations when both data update and schema change are present.
  - Investigate the methods suitable for maintaining the data warehouse using delta changes, and optimize incremental view maintenance algorithm by making use of shared maintenance queries.
  - Design and implement a batch data warehouse maintenance system based on TxnWrap.

– Provide a cost model to measure the performance of batch processing and conduct extensive experimental studies to measure batch maintenance performance.

1.6 Outline of the Thesis

The remainder of this thesis is organized as follows. Chapter 2 provides the necessary background related to data warehouse maintenance tasks in situations when both data update and schema changes are present, followed by a brief description of the TxnWrap system which represents the foundation of our proposed work. We present our parallel maintenance scheduler strategies and also its design and implementation issues in Chapter 3. The experimental studies of parallel maintenance are also presented in this chapter. Chapter 4 describes our solution strategies for batch data warehouse maintenance and the corresponding implementation issues, followed by cost model and performance studies. Finally, related work is given in Chapter 6, and conclusions and future work are summarized in Chapter 7.

Chapter 2

Background

2.1 Data Warehouse Maintenance Processes

Below we briefly introduce View Maintenance (VM), View Synchronization (VS) and View Adaptation (VA) strategies as needed for the remainder of this paper.

2.1.1 View Maintenance (VM)

View maintenance (VM) aims to incrementally maintain the view extent under a source data update (DU). The basic idea is to send a *maintenance query* based on the data update to calculate the delta change of this update on the view extent. A lot of work in the literature [33, 34, 1, 28] has addressed the conflicts between a *maintenance query* and concurrent data updates using either a *compensation* or *multi-version* [3] strategy. However, these works assume the schema of all relations remains static throughout the maintenance process.

2.1.2 View Synchronization (VS)

View Synchronization (VS) [23, 18] drops this assumption in that it aims at rewriting the view definition when the schema of the source relation has been changed. We distinguish between two primitive types of source schema changes (SCs) that may affect the view defined upon them: The SCs that rename attributes or relations at sources; and the SCs that delete attributes or relations. Note that since adding a relation or attributes will not affect the existing view definition, the VS will not take them into consideration.

[23, 18] propose some name mapping strategies of renaming the corresponding view meta data to handle source rename operations, which is relatively straightforward. For drop operations, since they would invalidate the view defined upon, they proposed two strategies. Here we briefly describe them by two examples and then formalize the methods.

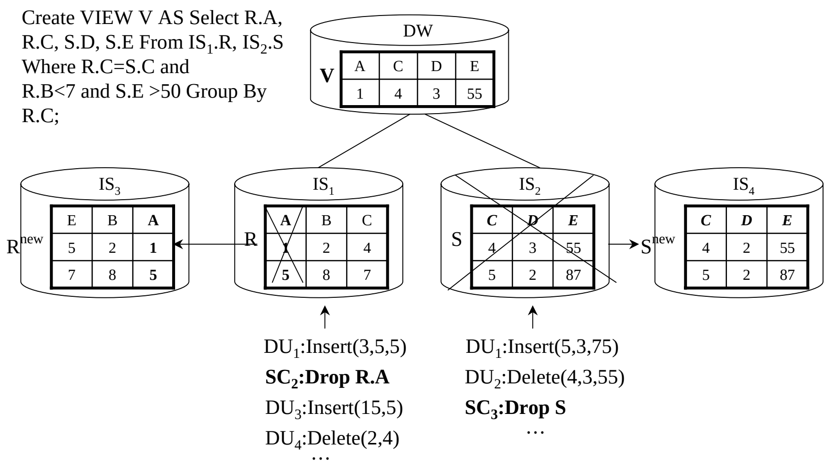


Figure 2.1: Example of View Synchronization.

- **Drop Relation:** In Figure 2.1, relation S in IS_2 is dropped in update SC_3. VS will try to find an alternative relation for replacement, in this case, say the relation S^{new} in IS_4, to rewrite the view definition. Thus the new view definition after this operation will be "Create VIEW V' as Select R.A, R.C, $S^{new}.D$, $S^{new}.E$ From IS_1.R, $IS_4.S^{new}$ Where $R.C = S^{new}.C$ and R.B < 7 and $S^{new}.C > 50$ Group By R.C".

We now generalize this algorithm using the notations in Table 2.1. For simplicity, we assume that there is only one view in the DW and each data source has exactly one relation.

Notation	**Meaning**
V	Old data warehouse view state, defined as $R_1 \bowtie R_2 \bowtie ... \bowtie R_n$
V'	New data warehouse view state, defined as $R_1^{new} \bowtie R_2^{new} \bowtie ... \bowtie R_n^{new}$
R_i	Old state of source relation i.
R_i'	New state of relation R_i after several updates.
R_i^k	kth ($k \geq 1$) replacement found by VS for view rewriting.
R_i^{new}	Relation R_i is replaced by R_i^{new} after view rewriting.

Table 2.1: Notations of View Definition.

Assume a relation R_i is dropped, VS will find a replacement R_i^{new} for the dropped relation R_i using the strategies proposed in [18], which are omitted here due to the limited space. The view V is rewritten as $V' = R_1 ... \bowtie R_{i-1} \bowtie R_i^{new} \bowtie R_{i+1} ... \bowtie R_n$.

- **Drop Attribute:** Similarly, the VS will also try to locate an alternative attribute for replacement when an attribute is dropped. Note that a join is often necessary for this step. In Figure 2.1, the attribute A of relation R in IS_1 is dropped in update SC_2. Then VS locates the relation R^{new} for replacement. The new view definition is thus rewritten as follows, "Create VIEW V' as Select $R^{new}.A$, $R'.C$, $S.D$ From $IS_3.R^{new}$, $IS_1.R'$, $IS_2.S$ Where $R.C = S.C$ and R.B < 7 and $S.C > 50$ and $IS_1.R'.B = IS_3.R^{new}.B$ Group By R.C". Here, the relation R in the old view definition is replaced by $\Pi_{A,B,C}(R' \bowtie_{R'.B=R^{new}.B} R^{new})$ forming a new view definition.

We thus generalize that if there are several attributes of R_i that are dropped, the relation R_i in the old view definition V is replaced by $R_i^{new} = \Pi_{R_i}(R_i' \bowtie R_i^1 \bowtie R_i^2 \bowtie ... \bowtie R_i^m)$, where m is the number of dropped attributes, R_i^k (k $\geq$ 1) is the replacement found by VS for the kth *drop attribute* operation and R_i' is the new state of R_i after several attributes have been dropped. Correspondingly, the new view V' is defined as $V' = R_1 ... \bowtie R_{i-1} \bowtie R_i^{new} \bowtie R_{i+1} ... \bowtie R_n$.

Note that the rewriting view may not be equivalent to the old one, although in VS there are mechanisms designed to bring the view as close as possible to the old one [18].

2.1.3 View Adaptation (VA)

View Adaptation (VA) [12, 25] incrementally adapts the view extent after the rewriting of a view definition. Since a rename operation won't affect the view extent, here we just briefly describe the incremental view adaptation after a drop operation. The basic idea is to determine the delta changes between the old relation and the new replaced one.

- **Drop Relation:** After the view rewriting for *Drop Relation* as shown in previous section, VA incrementally adapts the extent as follows. Depending on how the data warehouse system is designed, the dropped relation S can be restored either from the view in the data warehouse using $\Pi_{C,D,E}(V)$ [1] or by making use of versioned data if the data warehouse system using a multi-version strategy. Then $\Delta S = S^{new} - S$. Finally we calculate the view delta

[1] Assumptions for this include all attributes in conditions must also be selected. For details, please refer to [25]. In this case, if the assumptions are not held, we would resolve to perform view recomputation based on the new view definition.

changes by sending the query: $\Delta S \bowtie R$, which we call **view adaptation query** instead of **view maintenance query**. This **view adaptation query** aims to incorporate the effects of a source schema change into the materialized view extent. We now give a brief formalization of the algorithm above. First we calculate the difference between the old relation and the new one, i.e., $\Delta R_i = R_i^{new} - R_i$. Then we compute the view delta change by calculating $\Delta V = \Delta R_i \bowtie R_1 ... \bowtie R_{i-1} \bowtie R_{i+1} ... \bowtie R_n$.

- **Drop Attribute:** Similarly, after view rewriting for the *Drop Attribute* as shown in Section 2.1.2, we need to determine the delta between the old relation and the new joined relation, i.e.,
  $\Delta R = \Pi_{A,B,C}(R' \bowtie_{R'.B=R^{new}.B} R^{new}) - R$. Then we calculate the view delta change using the **view adaptation query**: $\Delta R \bowtie S$.

  To generalize the procedure, we first need to calculate the delta between the original R_i and the replaced R_i^{new} and join the delta with other relations to get the view change. In other words, we first get $\Delta R_i = \Pi_{R_i}(R_i' \bowtie R_i^1 \bowtie R_i^2 \bowtie ... \bowtie R_i^m) - R_i$, and we adapt the view delta change $\Delta V = \Delta R_i \bowtie R_1 ... \bowtie R_{i-1} \bowtie R_{i+1} ... \bowtie R_n$.

2.2 TxnWrap Revisited

In this section, we briefly review the TxnWrap solution which represents the foundation for our proposed parallel scheduling strategies and batch data warehouse maintenance algorithms.

2.2.1 The DW Maintenance Transaction Model

In a typical DW environment where one DW is designed over several independent sources (IS), 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) represents 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, denoted as "$r(VD)r(IS_1)r(IS_2)...r(IS_n)$". Here VD represents the view definition in the DW and r(VD) stands for the operations that generate the maintenance queries for individual ISs based on the view definition. $r(IS_i)$ is a read over IS_i which represents the maintenance query to IS_i and its corresponding results 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 update the DW extent and C_{DW} is the commit of w(DW) to the DW.

TxnWrap introduces the concept of a global transaction, called DWMS_Transaction, to encapsulate the above four DW maintenance steps within the context of the overall data warehouse environment.

Definition 1 *A DWMS_Transaction is a transaction model that encapsulates the four maintenance steps (IS_Update, Report, Propagation, 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.*

A *DWMS_Transaction* will be created only after C_{IS} of the corresponding IS update transaction has successfully been committed at the IS, and the commit of a DWMS_Transaction is right after the C_{DW} in the *Refresh* step. 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 to the DW. Thus, 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. So, for brevity, we remove the "C_{DW}" and "C_{IS}" operations and denote a DWMS_Transaction as "$w(IS_i)r(VD)r(IS_1)r(IS_2)...r(IS_n)w(DW)$". Furthermore, we refer to the *Propagation* and *Refresh* steps in one DWMS_Transaction ("$r(VD)r(IS_1)r(IS_2)...r(IS_n)w(DW)$") as the DWMS_Transaction maintenance process, since these two correspond to the actual maintenance steps in the DWMS.

Based on this model, we can rephrase the DW anomaly problem as a concurrency control problem. 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 *Propagation* phase may read some inconsistent query results written by the *IS_Update* phase of the maintenance process. Here we assume all other conflicts can be solved simply by the respective local DBMS at the IS or the DW. See [5] for further information.

2.2.2 Concurrency Control Strategy in TxnWrap

It is well known that read/write conflicts of transactions can be dealt with by either a locking or by a version-based [3] strategy. Locking of source data is not feasible in our environment due to the autonomy of data sources. Hence, TxnWrap has designed a multiversion concurrency control algorithm [3] (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 performs 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.

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

[2]We illustrate a running example of ShadowWrapper version management in Section 3.1.1.

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

Here $w(Wrapper_i)$ generates the versioned data in the wrapper indexed by its *global id* i, 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.

Thus, we have the formal representation of maintenance task in TxnWrap in term of typical read/write operations. In the following chapter, we first will make use of such representation to analyze the relationship (conflicts of read/write critical resource) between maintenance tasks, then propose solution strategies for parallel data warehouse maintenance.

Chapter 3

A Transactional Approach to Parallel DW Maintenance

Like other DW maintenance algorithms in the literature [33, 31], TxnWrap uses a sequential processing model for the DW maintenance process. This restricts the system performance. That is, only after the current DWMS_Transaction maintenance process has been committed, would the handling of the next one begin. Furthermore, in the propagation step of each DWMS_Transaction, the DWMS issues maintenance queries one by one to each IS_i and collects the results [1]. Thus only one IS is being utilized at a time in the maintenance propagation phase. In a distributed environment, the overhead of such remote queries is typically high involving both network delay and IO costs at the respective IS_i. If we could interleave the execution of different DWMS_Transaction maintenance processes, we would reduce the total network delay, and possibly also keep all ISs busy. This way, the overall performance would improve. We choose TxnWrap as the base system to propose our parallel schedulers. There are two reasons behind this. First, the transactional approach that TxnWrap has taken provides us with a formal way to analyze the conflicts that exist due to execution of different update maintenance processes in parallel. Second, a multi-version concurrency control strategy in TxnWrap further simplifies the processing logic of parallel maintenances. In the following sections, we first introduce the concept of *local id* to make the version and DWMS_Transaction management more flexible, then we describe our proposed solution strategies which can be applied in different situations. Finally, we provide some design and implementation issues and the corresponding experimental studies .

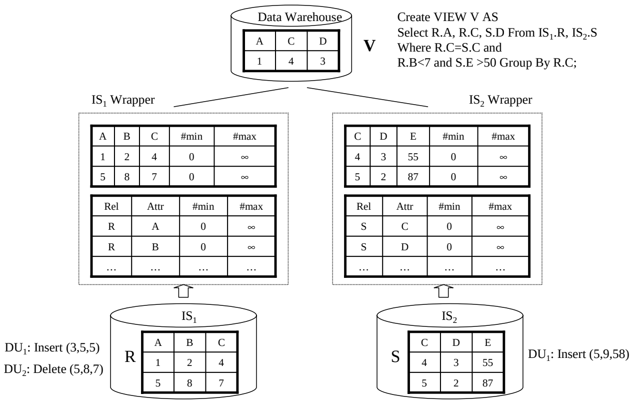
3.1 Towards Flexible DWMS_Transaction Management

Using a *global id* in TxnWrap to track IS updates restricts the flexibility of scheduling DWMS_Transactions because it tightly binds the version management in the IS wrapper with the overall maintenance task of the DWMS server. Furthermore, the *global id* would have to be issued by a global id-server in the DWMS to assure its uniqueness in the overall data warehousing system. 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, as described below.

3.1.1 Version Management using Local Identifier

We define a *local id* to be 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 and monotonically increasing starting from 0. Compared to the *global id*, there are two benefits to using *local id* instead. First, the process of id generation can be performed locally in each wrapper. Thus no longer have to communicate with the DWMS during version management. Second, we have to assume a global FIFO in the overall DW system to use the *global id*, which is too restrictive for distributed environments. Using of *local id*s would relax this restriction of the global FIFO assumption [1].

Figures 3.1 and 3.2 illustrate the version management in the wrapper using *local id*s. As an example, the IS_1 wrapper in Figure 3.1 contains the data of relation R as well as the related meta information. The IS_2 wrapper stores the same for relation S.



```
Create VIEW V AS
Select R.A, R.C, S.D From IS1.R, IS2.S
Where R.C=S.C and
R.B<7 and S.E >50 Group By R.C;
```

A	C	D
1	4	3

A	B	C	#min	#max
1	2	4	0	∞
5	8	7	0	∞

Rel	Attr	#min	#max
R	A	0	∞
R	B	0	∞
…	…	…	…

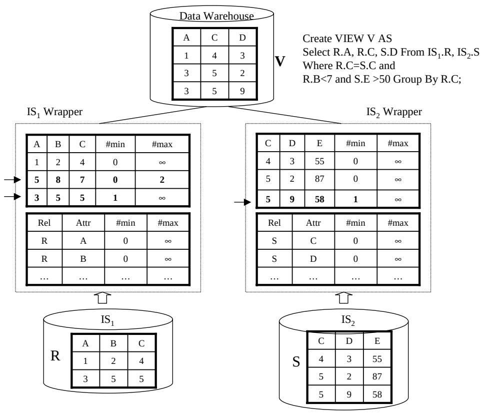
C	D	E	#min	#max
4	3	55	0	∞
5	2	87	0	∞

Rel	Attr	#min	#max
S	C	0	∞
S	D	0	∞
…	…	…	…

A	B	C
1	2	4
5	8	7

C	D	E
4	3	55
5	2	87

Figure 3.1: Wrapper Version (Before Source Updates).



```
Create VIEW V AS
Select R.A, R.C, S.D From IS1.R, IS2.S
Where R.C=S.C and
R.B<7 and S.E >50 Group By R.C;
```

A	C	D
1	4	3
3	5	2
3	5	9

A	B	C	#min	#max
1	2	4	0	∞
5	**8**	**7**	**0**	**2**
3	**5**	**5**	**1**	∞

Rel	Attr	#min	#max
R	A	0	∞
R	B	0	∞
…	…	…	…

C	D	E	#min	#max
4	3	55	0	∞
5	2	87	0	∞
5	**9**	**58**	**1**	∞

Rel	Attr	#min	#max
S	C	0	∞
S	D	0	∞
…	…	…	…

A	B	C
1	2	4
3	5	5

C	D	E
4	3	55
5	2	87
5	9	58

Figure 3.2: Wrapper Version (After Source Updates).

Two additional attributes #min and #max in the wrapper denote the lifetime 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 *Report* phase. Assume in Figure 3.1, $DU_1 : Insert(3, 5, 5)$ 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 3.2. Its [#min, #max] value is set to [1,∞]. This means that the lifetime of this tuple starts from the timestamp 1. Next,

[1]A running example of relaxing the global FIFO assumption is illustrated in Section 3.2.3.2.

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 $DU_1 : Insert(5, 9, 58)$ is committed in the IS_2. From a transaction point of view, the *local id* serves as the version write timestamp for the given IS update.

3.1.2 DWMS_Transaction Management using TxnID

In the global DWMS environment, we still need identifiers to track each DWMS_transaction, and to construct correct maintenance queries that access the appropriate versions of data in each wrapper.

Definition 2 *A TxnID τ is a vector timestamp, $\tau = [k_1 \ldots k_i \cdots k_n]$ with $\tau[i] = k_i$, that concatenates the current local id k_i of each IS_i (the largest local id that has been assigned thus far) when this TxnID is generated. n is the number of ISs and $0 \leq i \leq n$.*

While the local ids in each ISs may be the same, the TxnIDs are globally 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 3.1 and 3.2, 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.

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 3.1, 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 easily 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.

3.2 Parallel Maintenance Scheduler

3.2.1 Parallel Architecture

Figure 3.3 describes the overall architecture of the TxnWrap system, with our proposed extension for parallel scheduling (the *Parallel Scheduler* and the *Commit Controller* [2]). The maintenance process flow of each DWMS_Transaction can be described as follows. The *Parallel Scheduler* keeps fetching updates from the UMQ (a queue containing updates reported from ISs waiting to be maintained), checks whether system resources are available for maintenance processing and then verifies what conflicts related to this update exist to decide when to start the maintenance. More specifically, if this is a data update, then it is sent to the View Maintenance (VM) module, which is responsible for generating maintenance queries, sending

[2]The detailed descriptions of the TxnWrap structure can be found in [5].

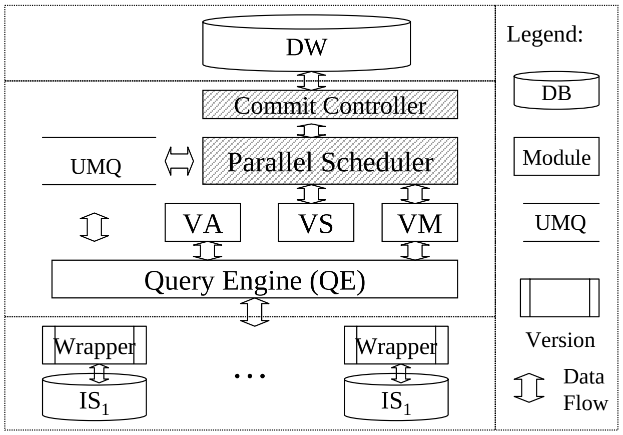


Figure 3.3: Architecture of TxnWrap Extended with Parallel Scheduler.

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 any schema element in the view definition that has been deleted. 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 submit the final result to the *Commit Controller* module. Based on the commit strategy, the *Commit Controller* will push the result to the DW. The *Parallel Scheduler* also analyzes waiting DWMS_Transactions in UMQ to identify possible constraints (in terms of read/write critical resources) and then generate a serializable schedule for a set of DWMS_Transactions.

3.2.2 Aggressive Scheduler for Data Update Only Environments

First, we study a direct extension of the serial scheduler and find that this is only suitable for data update only environments. As we stated in Section 2.2.2, one data update DWMS_Transaction maintenance process can be represented as

$r(VD)r(IS_1)r(IS_2)\cdots r(IS_n)w(DW)$. The ShadowWrapper algorithm guarantees that each r(IS_i) operation can easily identify the right versioned data using the *local ids* recorded in the TxnID for that respective IS. So, there will be no read block between DWMS_Transaction maintenance processes assuming that the versioned 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 which recorded the update in the form of properly versioned data in the respective wrapper.

Borrowing traditional concurrency control concepts [3], an aggressive scheduler for data update only environments is straightforward. That is, we can start the DWMS_Transaction maintenance processes for each data update almost at the same time as long as sufficient computational resources are available in the DWMS server [3] because there is no read/write conflicts in the *Propagation* step of DWMS_Transactions.

3.2.3 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 [17, 6]. There are three steps for maintaining a schema change:

- Determine which views in the DW are affected by the change. [(r(VD)]
- For each affected view:

[3]Detailed explanation and its correctness proof are presented in [19].

- Find the suitable replacement for schema elements removed from the view definitions via operations such as drop relation or drop attribute and rewrite the view definition in the DW if needed. [w(VD)]
- 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 and adapt the DW by committing these delta changes to the DW using the VA algorithm. $[r(VD)r(IS_1)r(IS_2)\cdots r(IS_n)]$

As can be seen, reading the VD to calculate the affected views occurs in step 1, while rewriting the VD may be required in step 2. In step 3, we read the VD again to generate the maintenance queries. Thus, the view definition (VD) of the DW represents a critical resource and the following sequence of operations occurs during SC maintenance: r(VD)w(VD)r(VD).

Putting it all together, one schema change DWMS_Transaction maintenance process can be represented as **r(VD)w(VD)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 these different transactions. In theory, it is possible that we could maintain schema changes in parallel, for example, using either a lock-based or multi-version [3] algorithm to control the concurrency of the view definition (VD), 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 don't occur that frequently. Thus, for simplicity of the control strategy, we propose to schedule schema changes sequentially.

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 r(VD). While for schema change maintenance, the following sequence occurs, $r(VD)w(VD)r(VD)$. Thus, all combinations of the read/write conflicts of VD exist in updates between two schema changes, and also between one schema change and many data updates. Hence, our parallel maintenance scheduler has to consider all these constraints.

3.2.3.1 TxnID-Order-Driven Scheduler

In a DWMS_Transaction environment, 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 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, $\tau_j < \tau_k \iff \forall i, 1 \leq i \leq n$ (n is the size of TxnID vector) $\tau_j[i] < \tau_k[i]$.

Observation 1 *Once we have assigned TxnIDs to updates in a mixed data updates and schema changes environment, then parallel scheduling of these updates needs to keep their TxnID order.*

The following example illustrates this ordering restriction.

Example 1 *As depicted in Figure 3.4, 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 we assign TxnIDs $[1,0,0]$ and $[1,0,1]$ to these two updates when they come*

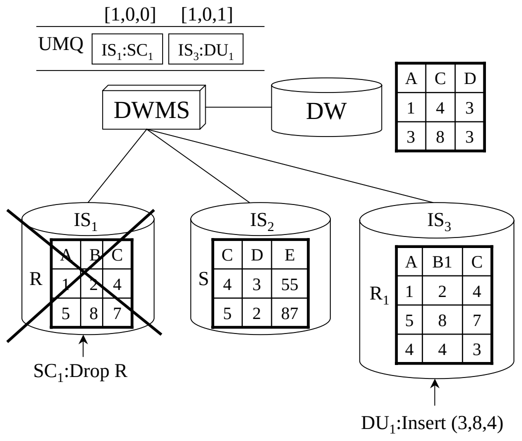


Figure 3.4: Example of Scheduling Order Restriction.

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.

Based on the above analysis, we propose the following basic TxnID-Order-Driven scheduler:

1. Start DWMS_Transaction maintenance processes based on their TxnID order.
2. Start schema change maintenance only if all the previous data updates and schema changes maintenance processes have been committed.
3. Block all the subsequent schema changes and data updates once a schema change is being processed.

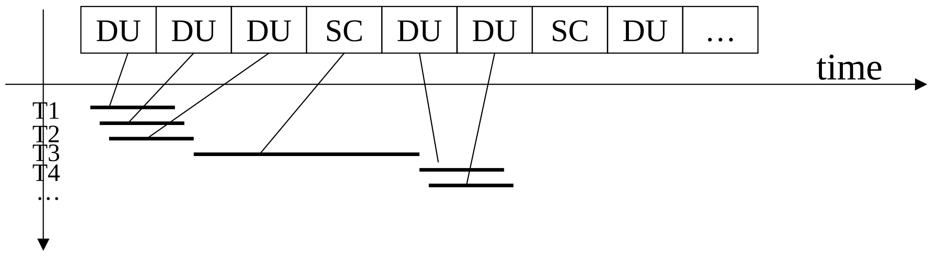


Figure 3.5: Scheduling Example of Basic TxnID-Order-Driven Algorithm.

A sample execution plan is depicted in Figure 3.5. DU and SC each stands for their corresponding DWMS_Transaction maintenance process. For space limitations, the detailed control procedures of this scheduler are omitted.

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 3.6 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. Below, we thus develop a dynamic scheduler that relaxes this ordering constraint.

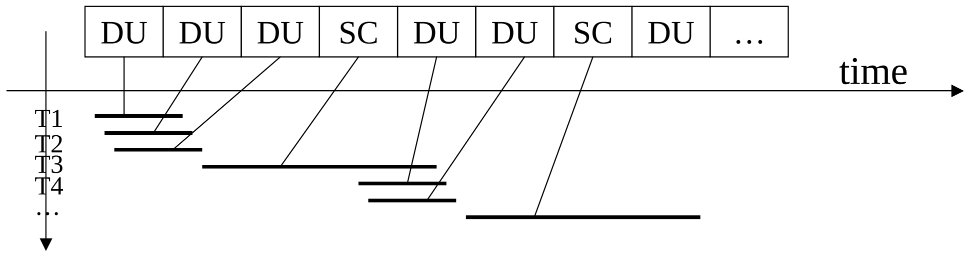


Figure 3.6: Scheduling Example of Improved TxnID-Order-Driven Algorithm.

3.2.3.2 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 $\triangle \mathrm{V} = \triangle A_i \bowtie A_j \bowtie A + (A_i + \triangle A_i) \bowtie \triangle A_j \bowtie A$.

- 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 3.4, 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 3.7 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.

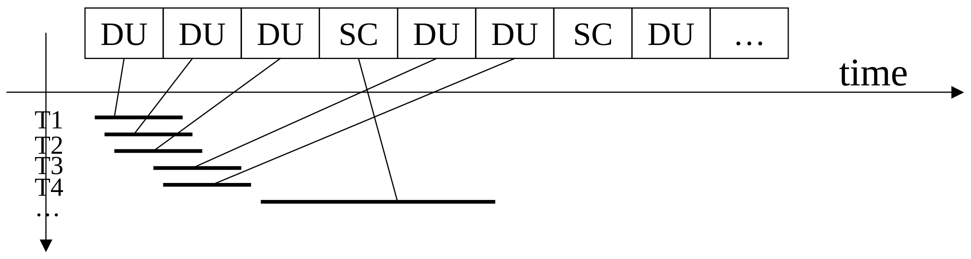


Figure 3.7: Scheduling Example of Dynamic TxnID Scheduler.

3.2.4 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 [32]. 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 [32].*
- *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 [32] 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. If every DWMS_Transaction contains only one IS transaction, then this solution will achieve complete consistency [34].

3.3 Design and Implementation Issues

We implemented 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 3.8 shows the abstract view of the parallel scheduler in the TxnWrap system extracted from Figure 3.3.

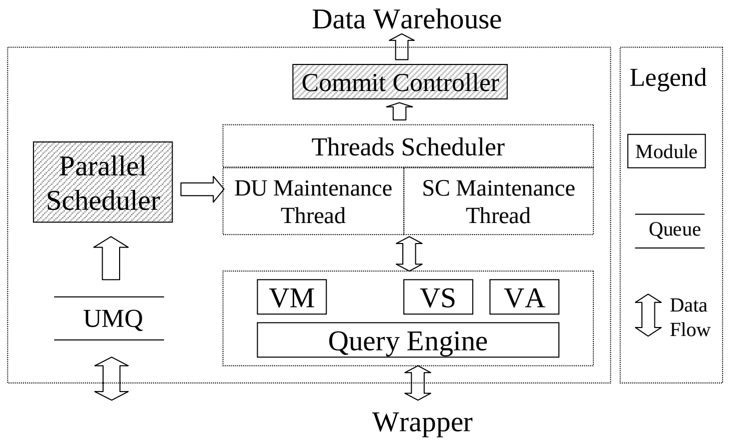


Figure 3.8: Implementation View of Parallel TxnWrap.

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 TxnID-Order-Driven 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 3.2.3.1.

- *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 3.9 depicts the control strategies implemented in the improved Fix-Order Scheduler.

Main Control

```
While threads available {
    1. Get an update from UMQ and assign TxnID
    2. If it is SC, then assign a SC_id
    3. Build a thread for this update, and start it
} Or wait until threads are available.
```

Other Related Controls

```
[Latest_started_id]
If it is a DU, then set latest_started_id = this global_id after we start to read VD.
If it is a SC, then set latest_started_id = this global_id after we start to update the VD
[latest_completedSC_id]
Set latest_completedSC_id = this SC_id after we finish VS part (in theory)
[ongoing_SC_id]
Set ongoing_SC_id = this global_id when we start SC maintenance;
Set ongoing_SC_id = -1 after we finish VS part.
```

Inside Thread

```
If it is a DU {
    1. Check latest_started_id
    2. Wait until there is no update still waiting before this.
    3. Check ongoing_SC_id
    4. Wait until global_id < ongoing_SC_id or ongoing_SC_id = -1  (no SC is running now)
    5. Start DU maintenance.
}
If it is a SC {
    1. Check latest_completedSC_id
    2. Wait until no SC is still waiting  before this
    3. Check latest_commited_id
    4. Wait until no update is still waiting for committing before this.
    5. Start SC maintenance.
}
```

Figure 3.9: Control Strategies in Improved TxnID-Order-Driven Scheduler.

3.4 Performance Studies

3.4.1 Experimentatl 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 and the fourth as the DWMS server. Each DB server contains several IS relations. Typically, there is one materialized join view defined in the DWMS machine over all IS relations.

We measure the performance of DWMS_Transaction maintenance for a set of updates. The processing time is composed of three categories. **Query-Time** describes the time span between the DB server receiving the query request and the DB server returning the desired results. **Network-Delay** is the time span between the DWMS sending out the maintenance query and the underlying IS (the DB server) receiving the query request. This is fairly small in a local network such as ours. **CPU-Time** is the time spent in the DWMS to generate maintenance queries, collect results and other miscellaneous scheduling overhead. In our experimental environment, the CPU overhead can't be reduced much because we use single CPU machines.

3.4.2 Aggressive Scheduler Experiments

The goal of this first experiment is to measure the effect of changing the number of threads on the total processing time (Figure 3.10). We set up nine 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 x-axis denotes the number of parallel threads in the system, with S denoting the serial scheduler while the y-axis represents the total time of processing 100 concurrent data updates.

In Figure 3.10, 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 be stable. This can be explained by possible additional

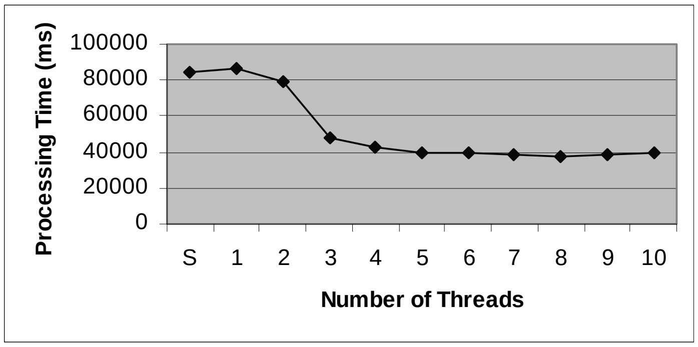


Figure 3.10: Change the Number of Threads.

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 in this scenario is around 53%. We note that the CPU overhead can't be fully reduced by our multi-threading solution because we use a one-CPU DWMS server. 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 in our environments. Thus, the Query-Time and Network Delay, which are the two tasks that are parallelized, are too small in the total processing time. For this reason, an improvement linear in the thread number is not achieved.

For the second 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. We set up six sources while the other settings are the same with Experiment 1. We change the number of tuples in each IS from 1000 to 50000.

From Figure 3.11, we see that if we increase the number of tuples in each IS, the total processing time increases. 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

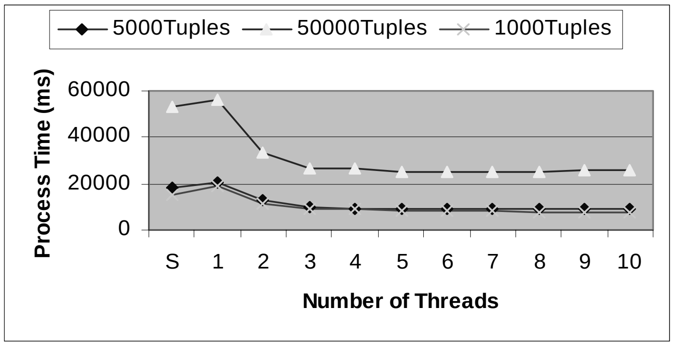


Figure 3.11: Change the Number of Tuples in Each IS.

time also increased, which in turn is partly parallelized. So the performance gain increases correspondingly.

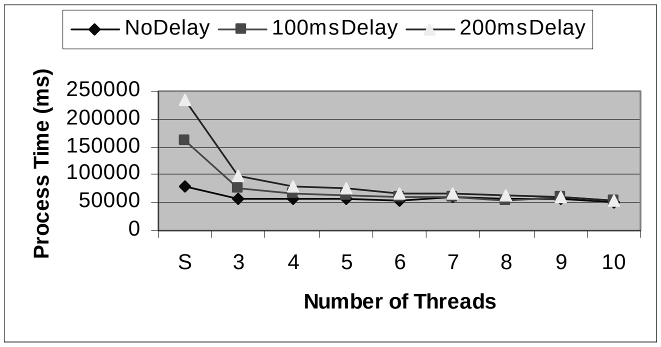


Figure 3.12: Change the Network-Delay in Each Maintenance Query.

The goal of the third 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 3.12 from no network-delay to 100ms and then 200ms. From Figure 3.12, we see that the larger the network delay, the more performance improvement is being achieved. Clearly, we can fully make use of the network delay in the parallel scheduler.

3.4.3 Experiments with Mixed Data and Schema Changes Scenarios

This experiment measures the effect of changing the number of schema changes on the performance of the improved TxnID-Order-Driven Scheduler. We use six 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.

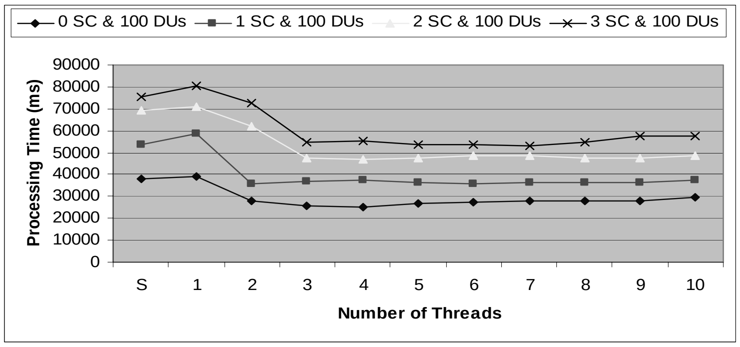


Figure 3.13: Change the Number of Schema Changes.

From Figure 3.13, 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, if we add more schema changes, the maximum improvement achieved by the scheduler will decrease because we can't fully maintain schema changes in parallel and all the subsequent data updates have to wait until the present schema change has finished its VS. If we increase the number of parallel threads, the total processing time also increases a little. This is due to the extra overhead on the commit controller caused by an increase in updates waiting to be committed.

Chapter 4

Batch Data Warehouse Maintenance

Another possible data warehouse maintenance optimization strategy is batch processing. That is, instead of processing one source update at a time each in a separate dedicated maintenance process, we can group several source updates together and try to maintain all of these updates as one data warehouse maintenance process. This way, the total maintenance performance for a given set of source updates is likely to improve.

There are two new issues of batch processing in a dynamic environment in which both concurrent data updates and schema changes are present. The first one is that we need to determine how to batch both schema changes and data updates at the same time. The second one is that we need to conduct batch processing in a concurrent update environment. In the following sections, we first provide the overall steps of our solutions and the algorithms for batch processing in such an environment. That is, given a sequence of source updates that need to be maintained, we first group these updates based on the source relation that they have occurred on.

Then for those schema changes that affect the data warehouse view definition, the necessary view evolution will be conducted. After that, view adaptation algorithms will adapt the view extent to make the data warehouse consistent.

4.1 Preprocessing Source Updates

If there is a sequence of source updates in the data warehouse that need to be maintained, we first partition these updates based on the source relations that they come from. There are two types of source updates in a dynamic environment. Thus we can further group the updates from the same source relation into two subgroups, one is for schema changes and the other is for data updates. Given a sequence of updates to one relation R_i, we define two subsequences as follows. $< SC_i >$ denotes the collection of all schema changes to R_i, while $< DU_i >$ denotes the collection of all data updates. We keep the same order in subsequences as they appear in the original one because the order of updates coming from the same source will affect the result of view maintenance. Notice that the schema changes from the same relation can sometimes be combined. For example, if *rename A to B* and followed by *rename B to C* occur to the same relation, we could simply *rename A to C*. Also note that, the data updates may be inconsistent with their schema due to some schema changes occurring in between two data updates. For example, given two tuple inserts into the same relation with a drop attribute schema change in between, then these two tuple inserts will have different schemata. Thus we have to preprocess these updates for each relation to adjust these differences and to enable us to maintain them in batch processing.

4.1.1 Combine Schema Changes

For schema changes defined in $< SC_i >$ from source R_i, we want to combine these schema changes in $< SC_i >$ as much as possible to get an equivalent sequence with the minimal number of schema changes. This would optimize our later data warehouse schema change processing, because those removed individual schema changes could then be ignored.

Table 4.1 shows all possible combinations between two SCs (SC_1, SC_2) with SC_1 the row entry and SC_2 the column entry. If the entry at the position T$[SC_1, SC_2]$ in Table 4.1 is empty, then this means that the combination of the two operations SC_1 followed by SC_2 has no effect on each other. Hence the combined result will keep both of them. Note that there is no schema changes could happen after a *"drop relation"* operation because all the schema changes are occurred on the same relation.

	$S \rightarrow T$	drop S	$b \rightarrow c$	drop b	drop $Cond_c$
$R \rightarrow S$	$R \rightarrow T$	drop R	-	-	-
$a \rightarrow b$	-	-	$a \rightarrow c$	drop a	-

Legend: *Capital letter (e.g., R, S, T) represents relation, lower case letter (e.g., a, b, c) represents attribute.*

Table 4.1: Combination Rules between Two SCs

Notice that we don't consider *add* attribute or relation here, because neither will affect the data warehouse view definition. Finally we define $< SC'_i >$ by combining the schema changes in $< SC_i >$ pairwise using the rules above.

4.1.2 Combine Data Updates

We then try to calculate the effect of data updates in subsequence $< DU_i >$. Notice that the data updates before a schema change may have different schemata with the data updates after due to the schema change in between. Thus we can't simply group them together. To group these data updates, we define $< DU'_i >= \Pi_{attr(R_i) \cap attr(R'_i)} < DU_i >$. That is, we project on the common attributes of both the original relation R_i and its new state R'_i which is R_i after incorporating all these updates. These common attributes are actually the attributes of R_i minus the dropped ones.

The purpose of this projection is to remove all attributes that are not related to the final data warehouse view at the end of the batch and thus to make the data updates in $< DU_i >$ all schemata consistent. We justify this below.

Lemma 1 $\Pi_{attr(R'_i) \cap attr(R_i)}$ *contains all the attributes related to view definition.*

This is obvious since either dropped or added attributes will not appear in the new view definition.

Lemma 2 *Suppose a view* $V = R_i \bowtie R_A$*, where* R_i *is a single relation and* R_A *is an abbreviation of the join from* $R_1 \bowtie \cdots \bowtie R_{i-1} \bowtie R_{i+1} \bowtie \cdots \bowtie R_n$*. We have* $\Pi_{attr(R'_i) \cap attr(R_i)} < DU'_i > \bowtie R_A = \Pi_{attr(R'_i) \cap attr(R_i)} < DU_i > \bowtie R_A$.

Proof:

$\Pi_{attr(R'_i) \cap attr(R_i)} < DU'_i > \bowtie R_A = \Pi_{attr(R'_i) \cap attr(R_i)} (\Pi_{attr(R_i) \cap attr(R'_i)} < DU_i >) \bowtie R_A$
$= \Pi_{attr(R'_i) \cap attr(R_i)} < DU_i > \bowtie R_A$.

This lemma proves that the projected data updates have the same effect on DW view maintenance as the original ones. Thus the projection is correct in terms of DW maintenance.

Lemma 3 *All* $< DU'_i >$ *have the same schemata.*

Proof: We prove this by contradiction. Suppose that one tuple r contains one more attribute than another tuple s. This extra attribute must be either an added attribute in r or a dropped attribute in s. Note that the added attribute would only appear in the new state of relation R'_i, while the dropped attribute will only appear in the old state of relation R_i. Thus such attributes will not appear in $attr(R'_i) \cap attr(R_i)$. That is, the tuples in $< DU'_i >$ will not contain such attributes. Thus the assumed case will never happen. Hence all tuples in $< DU'_i >$ must have the same attributes.

By the above lemmas, we can see that the projection operation is as expected.

Example 3 *Assume a view V(A,B,C) defined as* $R_1(A,B) \bowtie R_2(A,C)$. *Suppose relation* $R_2(A,C)$ *has updates:* DU_2 *= +(3,4), add attribute D, +(4,5,6), drop attribute C, and -(5,7). We have* R_2*(A,C) and* R'_2*(A,D) and* $attr(R_2) \cap attr(R'_2) = \{A\}$. *We get* $< DU'_2 >= \Pi_{attr(R_2) \cap attr(R'_2)} < DU_2 >= \{+(3), +(4), -(5)\}$, *which are schemata consistent.*

Now let's examine the new view definition V'*, a possible rewriting could be* $V'(A,B,C) = R_1(A,B) \bowtie \Pi_{attr(R_2)}(R'_2(A,D) \bowtie R_3(A,C))$. *Since only attribute A of* R'_2 *is involved in the view definition* V'*, we confirm that* $< DU'_2 >$ *is sufficient for view maintenance.*

After preprocessing, we end up with two subsequences of updates $< SC'_i >$ and $< DU'_i >$ for each source relation R_i. Note that if there is a schema change *"drops relation* R_i*"* in $< SC'_i >$, then $< DU'_i >$ would thus become empty because $attr(R'_i)$ is empty. In other words, if a relation is dropped, we won't consider any previous data updates from it. If there is no drop operation in $< SC_i >$, then there would be no change on $< DU_i >$ since $attr(R_i)$ and $attr(R'_i)$ are the same [1]. We describe the relationship between $< SC'_i >$ and $< DU'_i >$ as follows:

- If $< SC'_i >$ contains *"Drop Relation* R_i*"*, then $< DU'_i >= \emptyset$ and

[1]No view adaptation is necessary for rename operation, we thus think the schema of R_i and R'_i are the same if there are only rename schema changes existed.

$< SC'_i > =$ Drop Relation R_i.

- If $< SC'_i >$ contains a *“Drop Attribute”* operation, then both $< SC'_i >$ and $< DU'_i >$ might not be empty.
- If $< SC'_i >$ contains no *drop* operation of either kind, $< DU'_i > = < DU_i >$.

4.2 Evolve View Definition

For schema changes that affected the view definition in the data warehouse, view evolution is likely to occur. View Synchronization [23, 18] is responsible of rewriting the view definition. From preprocessing steps described above, we know that all the schema changes are collected in $< SC'_i >$. Thus, next we rewrite the view definition using VS techniques for all schema changes in $< SC'_i >$. Here we denote the old view definition as $V = R_1 \bowtie R_2 \bowtie ... \bowtie R_n$ and the new view definition as $V' = R_1^{new} \bowtie R_2^{new} \bowtie ... \bowtie R_n^{new}$.

Based on the VS description in Section 2.1.2, we have the following possible rewritings for each relation R_i. If the updates on R_i contain *drop relation*, then from the previous section we know that $< SC'_i >$ contains only the *drop relation* operation. Thus the rewriting [23, 18] is just to find an alternative relation. If the updates contain *drop attributes*, alternative attributes and additional joins are needed as described in Section 2.1.2. If the updates don't contain any *drop operations*, then it is exactly the same with R'_i. In summary, we have each new source relation as:

$$R_i^{new} = \begin{cases} \Pi_{attr(R_i)} R_i^1 & : \; Drop-Rel \\ \Pi_{attr(R_i)} (R'_i \bowtie R_i^1 \bowtie R_i^2 ... \bowtie R_i^m) & : \; Drop-Attr \\ \Pi_{attr(R_i)} R'_i & : \; No-DropSC \end{cases} \tag{4.1}$$

The meanings of R_i^{new}, R_i' and R_i^k are described in Table 2.1 in Section 2.1.2. Here we assume a valid view rewriting exists. Otherwise the view would become invalid and there would be no need to maintain it.

4.3 Adapt View Extent

After we finish evolving the view definition in the data warehouse based on the schema changes in each $< SC_i' >$, we also need to adapt the view extent correspondingly to make the data warehouse consistent. Typically, there are two ways to adapt the view in the data warehouse. Assume the old view definition is $V = R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n$ and the new view definition is $V' = R_1^{new} \bowtie R_2^{new} \bowtie \cdots \bowtie R_n^{new}$, V' incorporates all schema changes.

- **Recomputation.** For this, just simply recalculate the view extent based on the new source relations.
  $V' = (R_1^{new}) \bowtie (R_2^{new}) \bowtie \cdots \bowtie (R_n^{new})$. There are only n join operations (we will use join operation and maintenance query alternatively in the following sections) in recomputation, but the results of each query are likely very large. Thus, unless the underlying base relations are very small, it is an undesirable choice.

- **Incremental Maintenance.** That is, rather than completely recomputing the view extent, the changes from V to V' are computed by making use of Equation 4.2 and then applied the $\triangle V$ to the extent of V.

$$V' = (R_1 + \triangle R_1) \bowtie (R_2 + \triangle R_2) \bowtie \cdots \bowtie (R_n + \triangle R_n) \qquad (4.2)$$

Depending on how you rewrite the Equation 4.2, you can get different computation formula to calculate the delta changes.

- **Case 1:** Split the equation using all the components in it, then you will end up with 2^n factors with each factor having n-1 join operations in it. Thus the $\triangle V$ will be the summation of $2^n - 1$ factors with each factor containing n-1 joins. In each factor, there is at least one $\triangle R_i$, which will make the join cost decreased greatly if we can start the join from such relative small delta changes.

  $\triangle V = (R_1 \bowtie R_2 \bowtie \cdots \bowtie \triangle R_n) + (R_1 \bowtie R_2 \bowtie \cdots \triangle R_{n-1} \bowtie R_n) + \cdots + (\triangle R_1 \bowtie \triangle R_2 \bowtie \cdots \triangle R_{n-1} \bowtie \triangle R_n)$

- **Case 2:** Make use of the equation $R_i^{new} = R_i + \triangle R_i$ with $1 \leq i \leq n$. We can rewrite the equation as follows:

$$\begin{aligned} \triangle V &= \triangle R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n \\ &+ R_1^{new} \bowtie \triangle R_2 \bowtie R_3 \bowtie \cdots \bowtie R_n \\ &+ R_1^{new} \bowtie R_2^{new} \bowtie \cdots \bowtie \triangle R_i \cdots \bowtie R_n \\ &+ \cdots \\ &+ R_1^{new} \bowtie R_2^{new} \bowtie \cdots \bowtie \triangle R_n \end{aligned}$$

or

$$\begin{aligned} \triangle V &= \triangle R_1 \bowtie R_2^{new} \bowtie R_3^{new} \bowtie \cdots \bowtie R_n^{new} \\ &+ R_1 \bowtie \triangle R_2 \bowtie R_3^{new} \bowtie \cdots \bowtie R_n^{new} \\ &+ R_1 \bowtie R_2 \bowtie \cdots \triangle R_i \cdots \bowtie R_n^{new} \\ &+ \cdots \\ &+ R_1 \bowtie R_2 \bowtie R_3 \bowtie \cdots \bowtie \triangle R_n \end{aligned}$$

Both of these equations only have n factors with each factor having n-1 join operations. In each factor, there is exactly one $\triangle R_i$ which is enough to make the whole join operation cost of that factor small if we start the join operation from this $\triangle R_i$.

For the first case in incremental maintenance, we can compute the delta changes to the view extent using the delta changes of each relation R_i and only the old state of each relation R_i (the state that before these updates happened). While for the second case, we compute the changes using both old states R_i and the new states R_i^{new} of source relations along with their delta tables. In both cases, we divide the whole DW maintenance task into a series of join operations. Depending on what the underlying system is, the way to access R_i and R_i^{new} is different. For example, in a multi-version based system [3], these two states can be easily identified by timestamps and versions. In compensation based systems [33, 31], the old state R_i could be projected from view extent in some cases and the new state R_i^{new} could be gotten by extra conflict detection and compensation strategies.

Compared with these two cases, the second case appears more favorable in most situations because the number of factors is much less than the first case, while each factor in the second case is probably still small due to each factor containing one delta changes. Typically the size of such delta changes is much smaller compared with that of the source relation.

Thus, the last question remains for batch processing is how to calculate $\triangle R_i$ for each source relation R_i so that it incorporates the effect of $< DU_i' >$. We define the following formula to calculate the delta changes for each source:

$$\triangle R_i = \begin{cases} \prod_{attr(R_i)}(R_i^1) - R_i & : \quad Drop - Rel \\ \prod_{attr(R_i)}(R_i' \bowtie R_i^1 \bowtie \cdots \bowtie R_i^m) - R_i & : \quad Drop - Attr \\ \prod_{attr(R_i)}(R_i') - R_i & : \quad No - DropSC \end{cases}$$

Thus, if no drop schema operation appears, then the change to the original relation is the summation of data updates. That is, $\Delta R_i = \prod_{attr(R_i)}(R_i') - R_i$

$=< DU'_i >$. If there is a schema change *"drop relation"* operation, then we know that the $< DU'_i >$ is empty. This means that the data updates that happen before the 'drop relation' have no effect on the data warehouse maintenance. However, if the schema changes contain *"drop attribute"* operations, notice that $< DU'_i >$ might not be empty. We now prove that $< DU'_i >$ has already been incorporated into $\Delta R_i = \Pi_{attr(R_i)}(R'_i \bowtie R^1_i \bowtie R^2_i ... \bowtie R^m_i) - R_i$.

Lemma 4 $\Pi_{(attr(R_i) \cap attr(R'_i))}(R'_i) = \Pi_{(attr(R_i) \cap attr(R'_i))} R_i + \Pi_{(attr(R_i) \cap attr(R'_i)} < DU'_i >$.

This lemma is intuitively true. Because $< DU'_i >$ are from R_i and the final state is R'_i.

Theorem 1 $\Pi_{attr(R_i)}(R'_i \bowtie R^1_i \bowtie R^2_i ... \bowtie R^m_i)$ *has taken the* $< DU'_i >$ *into consideration.*

We have $\Pi_{(attr(R_i) \cap attr(R'_i))}(R'_i \bowtie R^1_i \bowtie R^2_i ... \bowtie R^m_i) = \Pi_{(attr(R_i) \cap attr(R'_i))}(R_i \bowtie R^1_i \bowtie R^2_i ... \bowtie R^m_i) + \Pi_{(attr(R_i) \cap attr(R'_i))} < DU'_i > \bowtie R^1_i \bowtie R^2_i ... \bowtie R^m_i$ by Lemma 4. The latter is the delta effect of $< DU'_i >$. Thus Theorem 1 is proven.

Thus, we can conclude that after we adapt the view extent using $\triangle R_i$ of each source R_i, we get the correct maintenance as well as adaptation result.

4.4 Optimize Number of Operations

From Section 4.3, we can see that a trade-off exists between the number of factors (join operations) and the operator size of each join operation in maintenance. In recomputation, there is only 1 factor with n-1 join operations, but the size of each in the join operation is typically large. This would will make it very costly. In the second case of incremental maintenance, there are n factors with each factor containing one delta change. This delta will typically decrease the result size of each factor. While in the first case of incremental maintenance, the result size of each factor could be even smaller because each factor is likely to have more than one delta changes. But the total number of factors will up to $2^n - 1$. In a distributed environment, the cost of remote queries is typically high. Thus if we can reduce the number of join operations and also try to keep the result size of each factor relatively small, good DW maintenance performance is likely to be achieved.

If we omit the join operators in the ΔV computation in Equation 4.3, the formula can be rewritten as a matrix, as depicted in Figure 4.1.

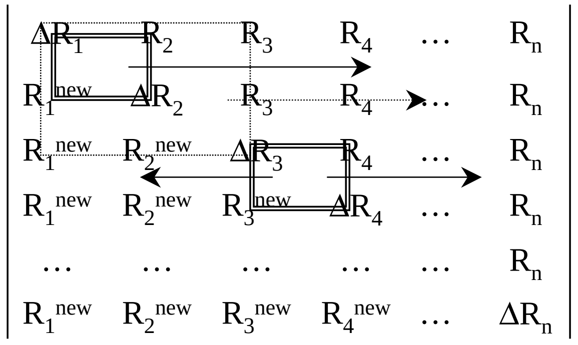


Figure 4.1: Group Delta Tables.

Seen from Figure 4.1, Row 1 and Row 2 have the common elements from R_3 to R_n, Row 1, Row 2 and Row 3 have the common elements from R_4 to R_n, and so on. So, if we can make use of theses common join operations in calculating the Equation 4.3, then we can further reduce the total number of join operations.

We call the matrix defined in Figure 4.1 the Join-Matrix with size n. A Group-Join is the summation of join operations along the diagonal of the Join-Matrix. The size of the Group-Join is the number of the summation operations. A Share-Join is the common join operations that exist in different rows in a Join-Matrix. As depicted in the figure, if we select the size of Group-Join k=2, that means we will partition all these rows by 2 [2]. So, $\triangle R_1 \bowtie R_2 + R_1^{new} \bowtie \triangle R_2$ is one of the Group-Join in the

[2]It is possible to partition these operations unevenly. For simplicity, we only analyze the

matrix. Its Share-Join is $R_3 \bowtie R_4 \bowtie \cdots \bowtie R_n$. The $\triangle R_3 \bowtie R_4 + R_3^{new} \bowtie \triangle R_4$ is the second Group-Join while $R_1^{new} \bowtie R_2^{new}$ and $R_5 \bowtie \cdots \bowtie R_n$ are its corresponding Share-Joins, and so on. If we increase Group-Join size k, then the join operation numbers of each Share-Joins will decrease.

Based on the Join-Matrix, we can roughly estimate its performance. We omit the cost differences between different relations, and simply use $\psi(n)$ to denote the total cost of calculating the join of n relations. With the Group-Join size is k, then the calculation cost will be

$$\lfloor\frac{n}{k}\rfloor * k * \psi(k-1) + \psi(n-k) + \triangle \text{ with } \triangle = [n - \lfloor\frac{n}{k}\rfloor * k] * \psi(n-1)$$

$\triangle$ includes the factors left for the n that can't be divided by k [3].

Thus, if we only optimize for the number of join operations to be calculated, then $\psi(n)$=n. That is, the cost measure will be

$$\lfloor\frac{n}{k}\rfloor * k * (k-1) + (n-k) + \triangle$$

By simple calculation, we can know when k is around $\sqrt{n}$, then the cost will reach its minimal.

However, the real calculation cost of these operations is more than the number of join operations. It also depends on the base relations, the delta tables, the previous join result and also the DW environments such as network delay in distributed environment, view definition of the DW, etc. Thus the optimized group size will differ in different situations. However, given a specific data warehouse environment, we can estimate such point via some experimental data.

situation that operations are evenly partitioned by a given k in this thesis.

[3]We can recursively apply the Group-Join to reduce the cost in calculating the leftovers

4.5 Implemented Batch DW Maintenance Architecture

The proposed batch processing can be applied to different DW maintenance systems. Here we choose TxnWrap [5] as our base system to implement the batch DW maintenance system. The reason is that TxnWrap uses a multiversion concurrency strategy, thus we can easily access the old state of each relation R_i before all updates have happened and the new state R_i^{new} which exactly incorporates the effect of these updates from the versioned data. Furthermore the transactional approach that TxnWrap system used would make the concurrency control strategy of batch maintenance easier. We could apply the same processing logic to a compensation-based view maintenance system such as DyDa [31]. However, the corresponding concurrent update detection and handling strategies will be more complex. Figure 4.2 gives a high-level view of the batch maintenance system architecture.

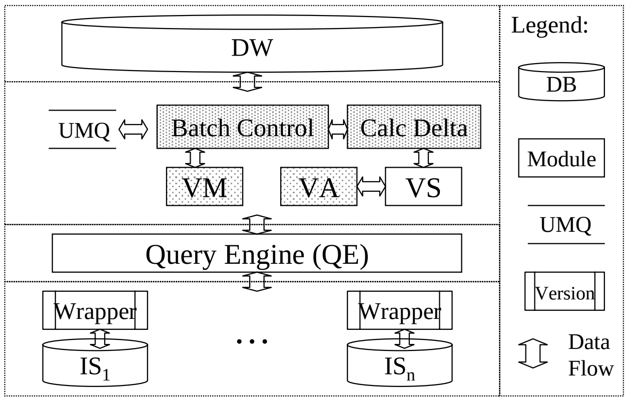


Figure 4.2: Batch DW Maintenance Architecture.

We plug two modules into TxnWrap framework. One is *Batch Control*, which is responsible for controlling the overall batch maintenance process. The other is *Calculate Delta Changes*, which takes care of grouping updates together and generating delta table for each corresponding relation. We also extend some of the exist modules such as VS and VA by plugging some new functions to support the batch processing requirements. The whole batch maintenance process flow can be described as follows. The *Batch Control* module keeps fetching the updates from the UMQ (which contains all the updates that have been reported from ISs). If a certain criteria is reached, such as the time has been elapsed since the last batch processing or a number of updates have been accumulated. Then *Batch Control* module triggers the batch maintenance processing by sending all these updates to the *Calculate Delta Changes* module. Three processes are involved in this module. First, it groups updates based on the relations where these updates happened. Second, it evolves the view definition of the DW if there are related schema changes by calling functions in the VS module. Third, it calculates delta changes for each IS using the proposed algorithms. After that, *Batch Control* sends these delta tables to VM/VA to maintain or adapt the DW extent using the maintenance algorithms we proposed in Section 4.3.

4.6 Performance Studies

We have implemented our batch DW maintenance system based on the TxnWrap. We compare the total processing time of maintaining a certain number of source updates using batch processing against the basic (non-batch) TxnWrap to measure the respective system performance.

4.6.1 Experimental Environment

Our experimental environment consists 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 and the fourth as the DWMS server. Each DB server contains serveral IS relations. Typically, there is one materialized join view defined in the DWMS machine over all IS relations.

4.6.2 Cost Measurement

Given a number of source updates, the total maintenance cost of TxnWrap can be roughly represented as:

$$T_{txnwrap} = \sum_{i=1}^{n} \sum_{j=1}^{s-1} C_{f(1)} \tag{4.3}$$

Here n is the number of source updates in the data warehouse that are to be maintained in one batch processing, s is the number of source relations, while $C_{f(1)}$ represents the average cost of issuing and answering a maintenance query composed of a single source update. As described in Section 4.3, the corresponding batch maintenance cost can be represented as:

$$T_{batch} = \sum_{i=1}^{s} \sum_{j=1}^{s-1} C_{f(n)}, \text{ with } \sum^{s} f(n) = n \tag{4.4}$$

with $C_{f(n)}$ represents the cost of issuing a maintenance query for grouped f(n) source updates. Note that the actual value of function f(n) depends on the distribution of these updates on the underlying sources. We use the term granularity of maintenance query to describe the size f(n). To simplify the discussion below, we minimize the difference between maintenance queries with the same granularity, and assume that total number of updates are evenly distributed in different sources. Thus the maintenance cost can be simplified as $T_{txnwrap}$ = n(s-1)C_1 and T_{batch} = s(s-1)$C_{n/s}$.

Note that the maintenance cost of $C_{n/s}$ (n/s > 1) will be different depending on how we implement issuing the large size maintenance query. If the cost of $C_{n/s}$ is a linear function on the variable n, then when:

$$\frac{nC_1}{sC_{n/s}} > 1$$

we can always use batching processing whenever it is possible and acceptable to users. However, if the cost of $C_{n/s}$ is not a linear function, then we instead to find an optimized number k (k >1), which maximaizes:

$$\frac{kC_1}{sC_k} > 1$$

Then the total maintenance cost for batching n ($n/s \geq$ k) updates can be written as Equation 4.5. We can see that total cost will also reach its minimal with the optimized k.

$$T_{batch} = s(s-1)(\lfloor\frac{n}{ks}\rfloor C_k + C_{n-\lfloor\frac{n}{ks}\rfloor k}) \tag{4.5}$$

In our environment, we assume that the remote sources are totally autonomous. Thus we can not require any support toward maintenance from the source relations such as building temporary tables, locking source relations, etc. This is typical in a web based application environment. Thus, in our implementation of issuing a maintenance query with granularity n, we have to use a SQL query to get data from the sources and to maintain and adapt the view extent in the data warehouse. Based on such an implementation, the maintenance cost of C_k with k > 1 is not of linear complexity.

In the following experiments, we will change the granularity of each maintenance query (number of updates, distributions of updates, etc), the view definition and the environments such as network delay to observe the performance changes of the batch processing.

4.6.3 Experiments

Change the Number of Updates being Processed. The goal of this experiment is to measure the effect of changing the granularity of maintenance queries (the number of updates being grouped) on the total processing time of batch processing and also compare the performance with that of TxnWrap system (Figures 4.3 and 4.4). We set up 6 sources and one view defined as a join of these sources. Theses sources are evenly distributed over three DB servers located on different machines. Each source has 100,000 tuples. We vary the number of data updates from 10 to 150 and all these updates come from the same source. The x-axis denotes the number of updates while the y-axis represents the total processing time of these data updates.

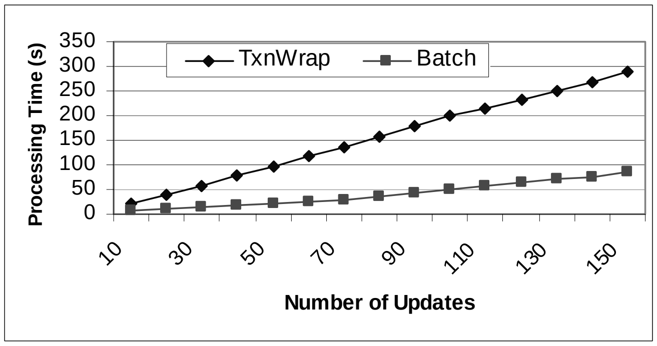


Figure 4.3: Maintenance Cost of TxnWrap and Batch.

From Figure 4.3, we see that if we increase the total number of updates, the TxnWrap processing time increases steadily of the same rate because in the above experimental environments, each update maintenance query cost of C_1 is almost the same. While in batch maintenance, the total processing time also increases slowly because the cost of C_n increases when we enlarge the number of grouped updates. We can see that the increase of the maintenance query cost C_n for a small batch size of n is much less than the total cost increase for the same number of queries when done individually. The total cost of batch maintenance in above environments is around 4 times lower than that of TxnWrap maintenance.

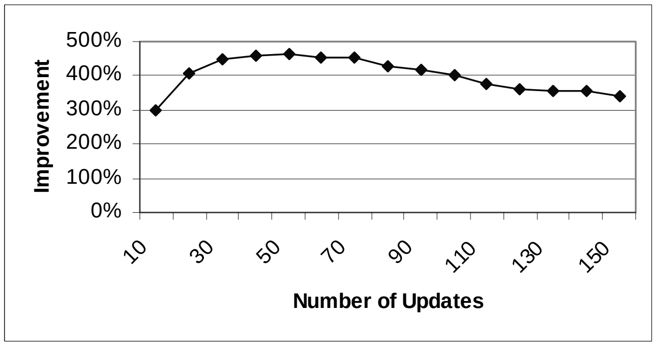


Figure 4.4: Performance Improvement of Batch Processing.

Figure 4.4 measures the ratio between TxnWrap and Batch processing by comparing their total processing time depicted in Figure 4.3. The higher the ratio, the more the performance improvement. We can see that in our current environmental setting, the cost of C_n is not a linear function of n. The result shows that when the total number of updates is around 50, the batch maintenance processing is the most efficient. Its cost is almost 500% less than that of sequential TxnWrap processing. Note that this empirically determined optimal batch size is specially to our experimental settings, and a new optimal number would need to be determined for a new experiment.

We can see that the larger the number of grouped source updates, the higher the increase of the query cost C_n. While for TxnWrap, the ratio of increase is almost fixed. Thus, for a batch maintaining a large number of updates, we can make use of maximal txnwrap/batch ratio to divide this large C_n into smaller subbatch queries of size k, thus $\sum C_k$ with k < n will also smaller than $\sum C_n$.

In Figure 4.5, we can see that if keep on increasing the number of source updates, and still try to incorporate all these updates into one single maintenance query, then even the cost of such batch maintenance will become worse than that of sequential

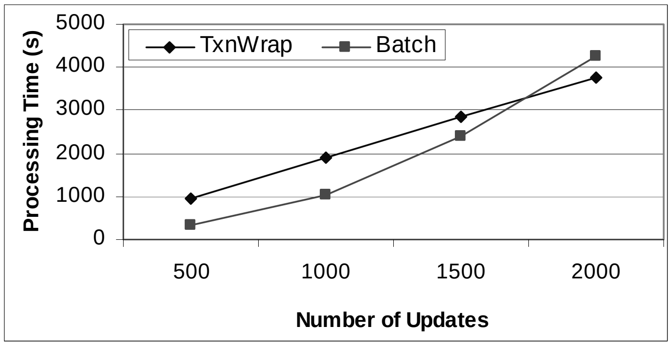


Figure 4.5: Batch Large Number of Updates in a Single Query.

processing. This is because the increase of the cost C_n for a large number n will be much larger than the increase of cost for sequential processing. In Figure 4.6, we use different k, which is around the optimization number we found in Figure 4.4. We see that when k is also around 50, the total batch processing cost is the best.

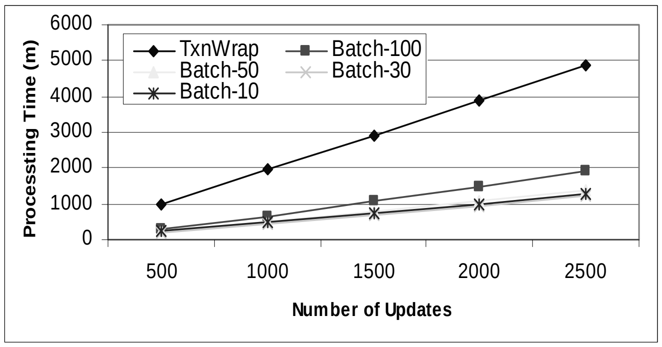


Figure 4.6: Batch using Different Query-size.

Batch Pure Schema Changes. In this next experiment, we try to measure the effect of pure source schema changes on the performance of batch system (Figures 4.7 and 4.8). We use the same experimental environment as in the previous experiment and vary the number of schema changes. There are two cases here. One, if the schema changes are happening in different sources, thus we can't do too much to optimize them. Thus the total processing time will be almost the same with TxnWrap system. This is depicted in Figure 4.7. We increase the number of schema changes (drop relation operations, which can't be combined) and the total processing time of batch maintaining increases correspondingly.

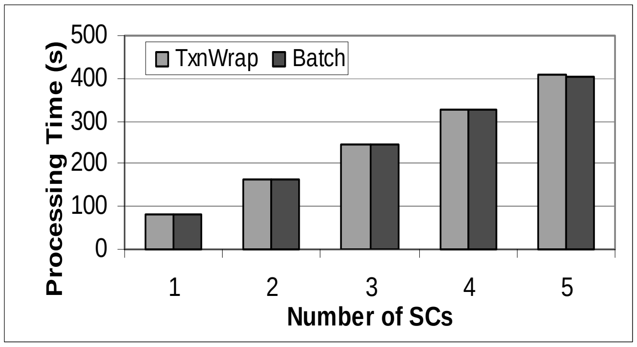


Figure 4.7: Schema Changes from Different Sources.

Figure 4.8 shows the case when we can combine the schema changes from the same sources. Then the total processing time will be decreased dramatically as expected. We only illustrate two cases here. In first experiment, five rename relation operations are then followed by one drop relation operation. The another experiment is six rename relation operations.

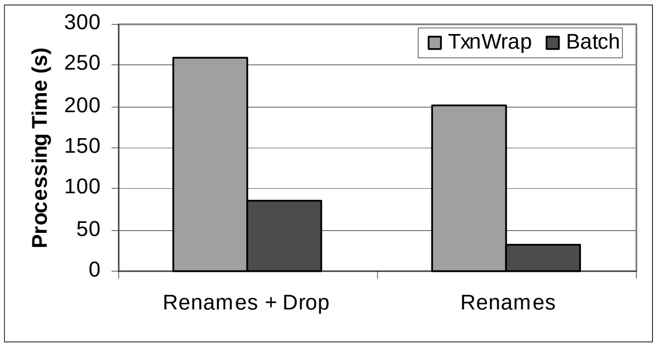


Figure 4.8: Schema Changes from the Same Source.

Batch in Both Data Update and Schema Change Environments. This experiment measures the performance of batch maintenance in an environment in which both data updates and schema changes are present. We use the same experimental environments as above. Note that the location of the drop schema change in a group of updates (the source updates pattern) will affect the batch maintenance processing because all the data updates before the drop operation can be safely dropped.

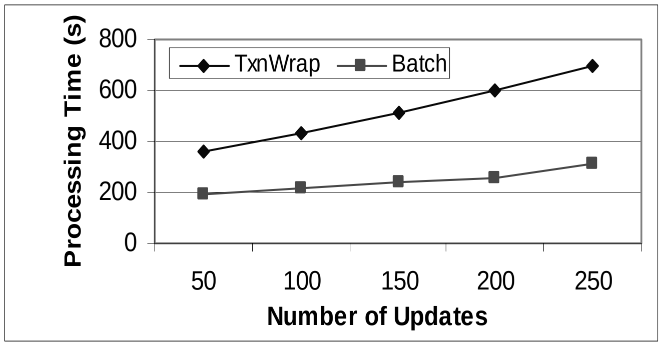


Figure 4.9: Maintenance Cost of Both Data Updates and Schema Changes.

In Figure 4.9, we measure the total processing time when we fix the location of schema change and increase the number of updates. In all cases, the first 25 data updates can be dropped due the drop relation schema change. Both processing times increase due to the increase in the number of updates, but the increase of the batch maintenance is much slower than that of TxnWrap. The reasons are similar to those of the first experiment. Compared with the experiment that has pure data updates, the performance increase is a little bit smaller (Figure 4.10). There are two reasons for this. One, a schema change processing is much more time consuming compared to one data update maintenance processing. Two, the cost of one schema change processing in batch processing is almost the same as in TxnWrap's (see the second experiment).

Figure 4.11 shows the effect of changing the location of the drop schema change in a fixed number of updates in the performance of batching. In this experiment, the number of data updates that can be dropped are from 25 up to 85 out of 100 source updates. The total TxnWrap processing time increases steadily due to

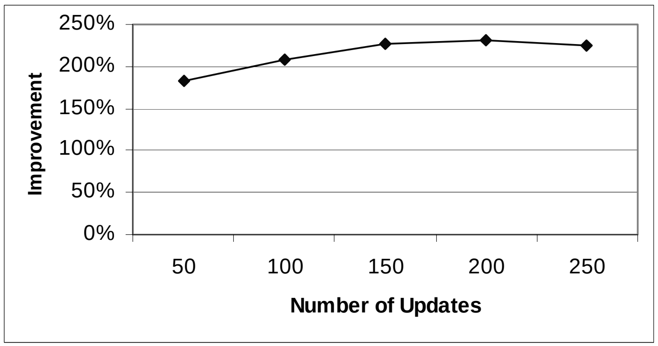


Figure 4.10: Performance Improvement of Batch Processing.

View Adaptation processing time increasing because the delta changes between the original source and the replacement expends (the time of calculating delta changes and issuing and answering view adaptation queries). The cost of batch processing decreases a little bit. This is because the more dropped data updates, the smaller the delta changes for each source. Thus the cost of each maintenance query also decreases.

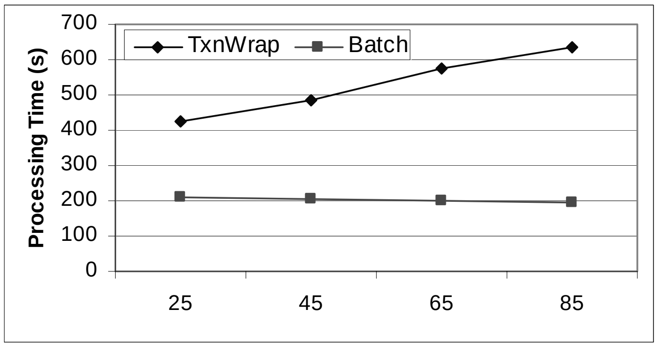


Figure 4.11: Change the Number of Dropped Updates.

Change View Definitions. The goal of this experiment is to determine the effect of the view definition itself on the performance of batch processing. Figure 4.12 shows the result of executing 100 data updates on different views defined on 4 sources up to 10 sources. We can see that the more complex the view definition, the more expensive cost of both batch processing and TxnWrap become for maintenance. It is as expected because the performance can be measured by $n(s-1)C_1$ for TxnWrap and $s(s-1)C_{n/s}$ for batch processing with s the number of sources that the view is defined upon. Thus, if we increase the number s, the total processing time of both systems will increase. The performance improvement can be measured by $nC_1/sC_{n/s}$. Thus improvement will decrease with the increase of s. This is depicted in Figure 4.13

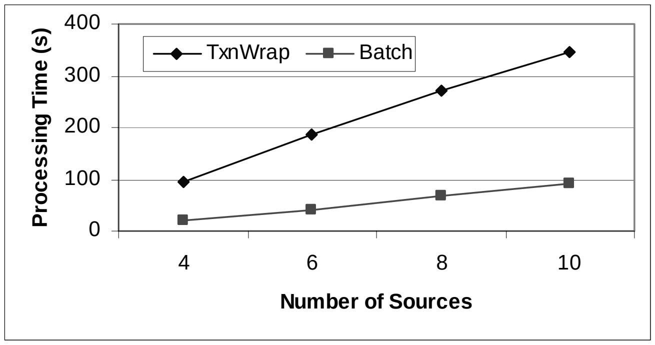


Figure 4.12: Maintenance Cost of Changing View Definition.

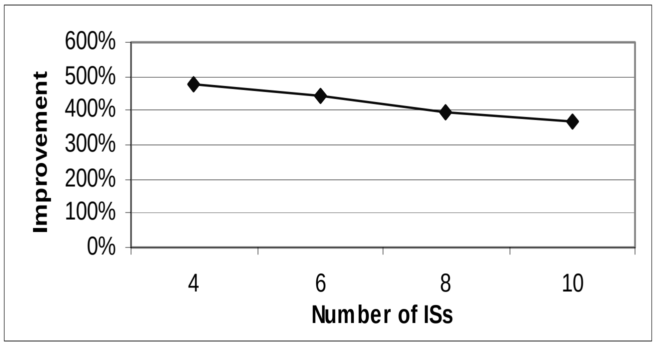


Figure 4.13: Performance Improvement of Batch Processing.

Chapter 5

Related Work

Maintaining materalized views under source updates in a data warehouse environment is one of the important issues of data warehousing [29, 8]. 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 [7, 9, 20]. There has also been some work on rewriting view definitions under IS schema changes [17], and on adapting the view extent under IS schema changes [12, 25, 31].

Self-Maintenance [2, 10, 26] 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, [30] also have extended this concept to temporal views. [16] summarize limited source access approaches.

In approaches that need to send maintenance queries down to the IS space, concurrency problems can arise [33]. They introduce the compensation-based algorithm ECA for incremental view maintenance under concurrent IS *data updates* restricted to a single IS. Strobe [34] handles multiple ISs while still assuming the schema of all ISs to be static. SWEEP [1] ensures the consistency of the data warehouse in a larger number of cases compared to Strobe [34]. As a compromise between self-maintenance and view maintenance via compensation, [14] proposes an intermediate approach to maintenance without using all the base relations but with requiring additional views to facilitate maintenance.

DyDa [31] is the first system to also handle concurrent schema changes which still employing a compensation query based strategy, like all prior approaches. DyDa is a heavy-weight solution in the sense that it adds complexity of concurrency detection and handling not only into each VS, VA and VM maintenance module but it also requires special strategies of coordination between these modules to achieve overall correct compensation.

TxnWrap [5] is the first transactional approach to handle the concurrency for both data and schema changes. TxnWrap encapsulates each maintenance process in a DWMS_Transaction and uses a multiversion concurrency control algorithm [3, 4] to guarantee a consistent view of data inside each DWMS_Transaction.

In the context of maintaining a DW in parallel, PVM [32] 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 complex given that it is a compensation based approach.

There are also many works on maintaining data warehouse using delta changes to the source relations. In [7], an incremental deferred view maintenance algorithm is introduced. It proposes the decomposition of the view maintenance problem into two separate propagate and refresh phases. It makes use of auxiliary tables that contain information recorded since the last view refresh to maintain view extent. [13] introduces several algorithms to use view definition to produce rules that compute the changes to the view using the delta changes to the source relations. [22] presented a incremental maintenance of relational views that involved aggregation using summary-delta table. A summary-delta table records the net change to an aggregate over a particular time window. [27] describes an algorithm called *2VNL* which makes use of multi-version approach to reduce contention between materialized view updates maintenance and the corresponding data warehouse read applications.

[28] is a closely related work to our batch view maintenance. It proposed a compensation-based technique for asynchronous incremental view maintenance. It also provides explicit control over the granularity of the view maintenance transactions. Compared with our approach in this paper, it can't handle the schema changes in the IS updates. Also no further optimization of the view maintenance algorithm due to the complexity of their compensation-based approach.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

Data warehouse maintenance techniques are becoming important because of increasingly use of data warehousing applications. Given the dynamic nature of modern distributed environments, both source data updates and schema changes are likely to occur autonomously and even concurrently in different sources. Current approaches [33, 1, 31, 5] to maintain a data warehouse in such dynamic environments apply sequential processing schedulers which only maintain source updates one by one. Also each maintenance process only corresponds to a single source update. This limits its performance in a distributed environment where the maintenance of source update endures the overhead of network delay and IO costs for each maintenance query.

In this thesis work, we propose two different optimization strategies to improve the data warehouse maintenance performance for a given set of source updates in such a dynamic environment containing both data updates and schema changes. For the parallel maintenance, based on the DWMS_Transaction model [5], we formalize the constraints that arise in maintaining data and schema changes concurrently. We propose several parallel maintenance schedulers. We also prove the correctness of our parallel scheduling solution. For the batch processing, we propose a technique to preprocess and generate delta changes for each source. We also propose algorithm to adapt and maintain the data warehouse extent using these delta changes. Further optimization of the algorithm is achieved by using shared queries for the maintenance tasks.

Furthermore, we have designed and implemented both optimization strategies and incorporated them into the existing DyDa/TxnWrap [5, 31] system. We have conducted extensive experiments both on parallel and batch processing on a given set of source updates to investigate the performance achievable under various environment settings. Our findings include that for the parallel processing, there is 40 $\sim$ 50% performance improvement compared to sequential processing in environments that using single-CPU machines and the network delay is neglectable compared with the processing time. While for batch processing, there is likely to to be a 400 $\sim$ 500% improvement in environments where network delay is low.

6.2 Future Work

The following are possible tasks that can be done in the future:

1. Integrate these two optimization strategies. Batch processing works well when all the source updates come from the same source, while parallel processing prefers that the source updates are evenly distributed among all the underlying sources. Thus in a certain level, we can switch between different optimization stratgies. Another point is that we could apply batch processing at the first level, and parallel processing on these grouped source updates at a higher level.

2. Using different strategies to implement the batch maintenance queries to mea-

sure system behavior, for example, using a temporary table in the sources to return the result instead of issuing maintenance queries. That is, efficiently maintain the data warehouse in a environment that the sources are more co-operative.

3. To extend the solutions to handle the situation that multiple relations exist per source. Though we can simply think each relation as one source and still use the same proposed solutions, the problems are how to reduce the maintenance processes as well as the cost for those updates from the same source.

4. To incorporate the solutions in environments that multiple views are defined. The issues of the hierarchies and constraints exist in these views have to be considered in maintenances.

5. How the maintenance issues and their corresponding optimization strategies be adapted to non-relational databases?

Chapter 7

Appendix

7.1 Proof of Parallel TxnWrap Maintenances

Theorem 2 *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 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 Section 3.1, TxnID is the vector timestamps which records 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 [3], 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). Then 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 [3]. 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 3 *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 [3], 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 [3], then there are only two cases to consider:

   - $w_{\tau_j}[x_{\tau_j}] \ldots r_{\tau_k}[x_{\tau_i}]$ and $x_{\tau_j} < x_{\tau_i}$. 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. Contradiction: Since all cases lead to contraditions, the assumption is not correct. Thus, there is no version order edge $T_{\tau_i} \leftarrow T_{\tau_j}$ with TxnID $\tau_i < \tau_j$. So the MVSG is acyclic.

Bibliography

[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] 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.

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

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

[5] J. Chen and E. A. Rundensteiner. Txnwrap: A transactional approach to data warehouse maintenance. Technical Report WPI-CS-TR-00-26, Worcester Polytechnic Institute, November 2000.

[6] 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.

[7] 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.

[8] H. García-Molina, W. L., J. L. Wiener, and Y. Zhuge. Distributed and Parallel Computing Issues in Data Warehousing . In *Symposium on Principles of Distributed Computing*, page 7, 1998. Abstract.

[9] T. Griffin and L. Libkin. Incremental Maintenance of Views with Duplicates. In *Proceedings of SIGMOD*, pages 328–339, 1995.

[10] 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.

[11] 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.

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

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

[14] N. P. Huyn. *Maintaining Data Warehouse Under Limited Source Access*. PhD thesis, Stanford University, August 1997.

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

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

[17] A. J. Lee, A. Koeller, A. Nica, and E. A. Rundensteiner. Data Warehouse Evolution: Trade-offs between Quality and Cost of Query Rewritings. In *Proceedings of IEEE International Conference on Data Engineering*, Special Poster Session, page 255, March, Sydney, Australia 1999.

[18] A. M. Lee, A. Nica, and E. A. Rundensteiner. The EVE Approach: View Synchronization in Dynamic Distributed Environments. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2001.

[19] B. Liu, S. Chen, and E. A. Rundensteiner. Parallel Data Warehouse Maintenance. Technical Report WPI-CS-TR-02-08, Worcester Polytechnic Institute, Dept. of Computer Science, 2002.

[20] 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.

[21] 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.

[22] I. Mumick, D. Quass, and B. Mumick. Maintenance of Data Cubes and Summary Tables in a Warehouse. In *Proceedings of SIGMOD*, May 1997.

[23] A. Nica, A. J. Lee, and E. A. Rundensteiner. The CVS Algorithm for View Synchronization in Evolvable Large-Scale Information Systems. In *Proceedings of International Conference on Extending Database Technology (EDBT'98)*, pages 359–373, Valencia, Spain, March 1998.

[24] 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 (DW-DOT'98)*, Vienna, Austria, August 1998.

[25] 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.

[26] 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.

[27] D. Quass and J. Widom. On-line warehouse view maintenance. In *Proceedings of SIGMOD*, pages 393–404, 1997.

[28] 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.

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

[30] J. Yang and J. Widom. Temporal view self-maintenance in a warehousing environment. Technical report, Stanford University, Computer Science Department, February 2000.

[31] 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.

[32] 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.

[33] 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.

[34] Y. Zhuge, H. 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.