A Compensation-based Approach for Materialized View Maintenance in Distributed Environments *

Songting Chen, Xin Zhang and Elke A. Rundensteiner
Department of Computer Science
Worcester Polytechnic Institute
Worcester, MA 01609-2280
{chenst, xinz, rundenst}@cs.wpi.edu

Abstract

Data integration over multiple heterogeneous data sources has become increasingly important for modern applications. The integrated data is usually stored in materialized views (MV) to allow better access, performance and high availability. MV must be maintained after the data sources change. In a loosely-coupled environment, such as the Data Grid, the data sources are autonomous. Hence the source updates can be concurrent and cause erroneous maintenance results. State-of-the-art maintenance strategies apply compensating queries to correct such errors, making the restricting assumption that all source schemata remain static over time. However, in such dynamic environments, the data sources may change not only their data but also their schema, query capabilities or semantics. Consequently, either the maintenance queries or compensating queries would fail.

In this paper, we propose a framework called DyDa that overcomes these limitations and handles both source data updates and schema changes. First, we identify three types of maintenance anomalies, caused by either data updates and/or rename and/or drop schema changes. We propose a compensation algorithm to solve the first two types of anomalies. We identify that the third type of anomaly is caused by the violation of dependencies between the maintenance processes. We propose a detection and correction algorithm to remove such anomalies based on the formalisms of dependencies. A new view adaptation algorithm is designed to incrementally adapt some complex updates introduced by the correction algorithm.

Put together, these algorithms are the first complete solution to the concurrency problems for MV maintenance in loosely-coupled environments. We have implemented the DyDa system. The experimental results show that our new concurrency handling strategy imposes a minimal overhead on normal data update processing while allowing for the extended functionality to maintain the materialized views even under concurrent schema changes.

Keywords: View Maintenance, View Synchronization, View Adaptation, Complex Updates.

1 Introduction

1.1 Materialized Views in Dynamic Environments

With the information explosion on the World Wide Web, the transformation and integration of data from multiple heterogeneous data sources is ubiquitous to many modern information systems and e-business applications. One basic requirement is to integrate data with rich structures, such as relational, XML, spreadsheets, etc. There

*This work is supported in part by several grants from NSF, namely, NSF NYI grant IIS-9796264, NSF CISE Instrumentation grant IRIS 97-29878, and NSF grant IIS-9988776.

is growing interest in the database community to focus on schema integration to achieve the interoperability between such heterogeneous sources. One common technique is to use schema mappings [MBR01, MHH00] to specify how the data of one schema is transformed to another. A view query is one way to specify such a mapping. Schema mapping are used extensively in variety of applications, such as data integration, physical database design like XML to relational mapping, or the semantic Web [LGMT03].

In dynamic environments like the WWW, the data sources may change their schema, semantics as well as their query capabilities. In correspondence, the mapping or view definition must be maintained to keep consistent [LNR02, VMP03]. Moreover, in a loosely-coupled environment, such as the Data Grid [JR03], the data sources are typically owned by different providers and function independently from each other. Hence they may commit update transactions without any concern about how those changes may affect the mapping or views defined upon them. While in combination, the autonomous source schema restruction poses new challenges for data integration.

Materialized view (MV), proven to be an excellent technique in decision support applications, would continue to be useful in this scenario to preserve the integrated data to ensure better access, performance and high availability. MV must be maintained when the sources change. This has been extensively studied in the past few years [AASY97, SBCL00, ZGMHW95]. However, it is not sufficiently explored in this new dynamic environment. As we will illustrate via examples in Section 2.2, when maintaining a source update, we may need to query the data sources for more information by issuing *maintenance queries* [ZGMHW95]. However, the *maintenance queries* may either return erroneous query results due to concurrent data updates or may even fail completely due to concurrent schema changes. Such failure of maintenance remains unsolved.

While recent work [AASY97, SBCL00, ZGMHW95] proposed *compensation-based* solutions to remove the effect of concurrent data updates from query results, we demonstrate that these existing solutions would fail under source schema changes. The reason is that if the source schema has been concurrently changed, neither *maintenance* nor *compensation queries* would get any query response from data sources due to the discrepancy of the source schema with the schema required by the queries. Interleaving of concurrent source data and schema changes even complicates the maintenance further.

1.2 State-of-the-Art Materialized View Maintenance Techniques

We distinguish between three MV maintenance tasks, namely, View Maintenance (VM), View Synchronization (VS) and View Adaptation (VA) as explained below. VM [ZGMHW95, AASY97, SBCL00] maintains the view extent under source data updates. In contrast to VM, VS [LNR02] aims at rewriting the view definition when the schema of a source has been changed, or in a more general sense, evolving the schema mappings after schema changes [VMP03]. Thereafter, View Adaptation (VA) [NR99, GMRR01] incrementally adapts the view extent

to again match the newly changed view definition. We will review these algorithms in more detail in Section 2.1.

However, as indicated in Section 1.1, the data sources are autonomous and may undergo changes at any time. Thus during the MV maintenance, other source updates may occur and conflict with the current maintenance processes, causing the concurrency problems as will be illustrated by an example in Section 2.2. While such concurrency problems have received increased attention in recent years, all existing work [ZGMHW95, AASY97, SBCL00, ZRD01] is restricted to handle pure data updates only. They make the restricting assumption that the schemata of all sources remain stable over time. As motivated extensively in Section 1.1, this is a rather limiting and unrealistic assumption for many real world environments and applications. SDCC [ZR02] is the first work to study the problem arising from the concurrency of both source data updates and schema changes. The SDCC approach integrates existing VM, VS and VA algorithms into one system with a protocol that all data sources must abide by to enable correct collaboration. This protocol however has the limitation of requiring sources to fully cooperate by first announcing an intended schema change, then waiting for permission from the maintenance modules to execute this schema change at their local database. In essence, the concurrency problem is avoided at the cost of the autonomy of the sources. For many environments such full cooperation and willingness to delay any update of a data source is too restrictive and often impractical. Hence, in such environments, the SDCC solution is not applicable.

1.3 Contributions of this Work

In this paper, we propose a general approach for dynamic MV management, called the DyDa framework to solve the concurrency problems. DyDa maintains the MV defined over distributed data sources without posing any restrictions on any source update transactions. In other words, the restrictive assumption of requiring cooperative data sources [ZR02] is dropped. In summary, the contributions of this work are:

(1) We identify all possible maintenance concurrency problems, namely, caused by source data updates, source rename schema changes and source drop schema changes. We introduce DyDa framework to solve all these anomalies while the data sources have the full autonomy to commit any types of transactions.

(2) We introduce a compensation algorithm to remove the effects of concurrent data updates and rename schema changes to restore the correct maintenance query results.

(3) We formalize the anomalies caused by concurrent drop schema changes as the violations of dependencies between the maintenance processes. We then develop algorithms to detect and correct any violated dependencies. A novel view adaptation algorithm is proposed to process the resulting mixed data updates and schema changes.

(4) We have implemented our techniques in our DyDa system which has been demonstrated at SIGMOD' 2001 [CZC+01]. The experimental results show that our new concurrency handling strategy imposes a minimal

overhead on data update processing while allowing for the extended functionality to maintain the MV even under concurrent schema changes.

(5) The conference paper of this work [CCZR04] discusses the general theory for solving anomalies caused by schema changes. In this paper, we present an overall solution framework and a compenhensive performance study for all types of anomalies. We also extend it by a different treatement for rename and drop schema change, which further improves the performance. A more detailed VA algorithm for complex update maintenance is presented. The proofs of the correctness of our approach are included here.

In the next section, we present the background materials necessary for the remainder of the paper. Section 3 describes our proposed architecture of the DyDa framework and explains the overall concurrency control strategies. Section 4 introduces a compensation algorithm to solve the concurrency caused by data updates and rename schema changes. Section 5 formalizes the dependencies between the maintenance processes and their relationship to the concurrency caused by drop schema change. An algorithm is proposed to detect and correct the violated dependencies. Section 6 introduces a new view adaptation algorithm to adapt multiple distributed schema changes and data updates required by the dependency correction algorithm. Section 7 discusses the experimental results. Section 8 reviews related work, while Section 9 concludes the paper.

2 Background Materials

2.1 View Maintenance Techniques Revisited

We first briefly review and generalize three basic maintenance techniques, namely, View Maintenance (VM), View Synchronization (VS) and View Adaptation (VA) for a single source update via an example below.

Example 1 *Assume we want to integrate data from the book Retailer and Library category to provide the user the sales as well as the detailed book information (Figure 1). The book Retailer data, being in the XML format, is mapped into the relational tables Store and Item as a relational wrapper view. The Library catalog of the detailed book information can be accessed by a general-purpose wrapper, which is used to execute a query and extract source changes to notify the view manager. Now the integrated view BookInfo from both data sources can be defined by the SQL query in Equation (1).*

```
CREATE VIEW  BookInfo AS
SELECT       Store, Book, I.Author, Price, Pub-
             lisher, Category, Review
FROM         Store S, Item I, Catalog C
WHERE        S.SID = I.SID
             AND I.Book = C.Title
```

(1)

```
SELECT   Book, Author, Price, Review
FROM     Catalog C
WHERE    Book = 'Data Integration Guide'
```

(2)

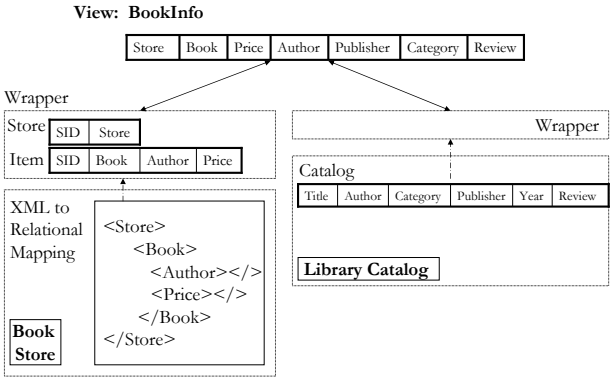

Figure 1: Description of View and Data Sources

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 DU to calculate the delta change on the view extent. In Example 1, assume there is a new book to be inserted into the Item table. This new book is extracted by the wrapper as "ΔI = insert (10, 'Data Integration Guide', 'Adams', 35.99)". To determine its delta effect on the view, an *incremental maintenance query* (Query 2) [ZGMHW95] will be generated by decomposing the view query (1) into individual source queries. In other words, we compute $\Delta V = \Delta I \bowtie Store \bowtie Catalog$. We generalize this maintenance process M as "$M(DU) = r(VD)r(DS_1)r(DS_2)...r(DS_n)w(MV)c(MV)$", where VD is the view definition, DS_i is the data source with index i, $r(DS_i)$ is the query sent to DS_i, $w(MV)$ and $c(MV)$ are write and commit of the MV, respectively.

2.1.2 View Synchronization (VS)

View Synchronization (VS) [NLR98, LNR02], on the other hand, aims at evolving the view definition when the schema of the base relation has been changed. Note that, a general mapping adaptation technique could also serve similar purpose [VMP03]. Two primitive types of source schema changes (SCs) that may affect the view defined upon them are considered: the *RenameSC* that renames the source attributes or relations and the *DropSC* that deletes attributes or relations. Note that add relations or attributes do not directly change the views, but they will be put into some knowledge bases [NLR98, LNR02].

It is straightforward to handle *RenameSC* by just modifying the corresponding view definition. To handle *DropSC*, the basic idea is to find some alternative sources to replace the dropped data. For example, assume the

Review attribute is no longer considered important and kept in the *Catalog* relation. The view synchronization process will locate some alternative table *Comments* in *ReaderDigest* data source for replacement based on the containment information. The new integration and rewritten query are shown in Figure 2 and Query (3).

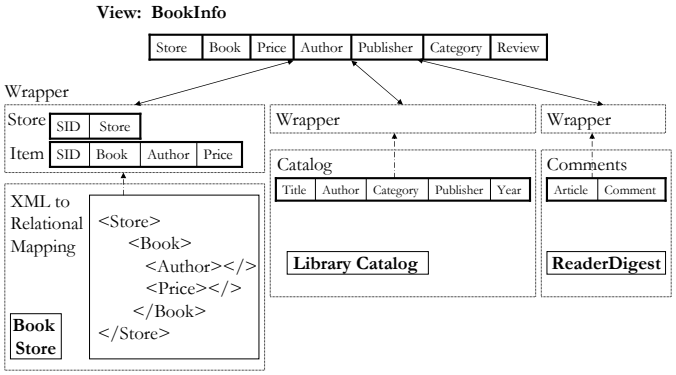

Figure 2: View Synchronization for Drop Review Attribute

$$
\begin{array}{ll}
\text{CREATE VIEW} & \text{BookInfo AS} \\
\text{SELECT} & \text{Store, Book, I.Author, Price, Publisher, Category, Comment as Review} \\
\text{FROM} & \text{Store S, Item I, Catalog C, Comments M} \\
\text{WHERE} & \text{S.SID = I.SID AND C.Title = M.Article AND I.Book = C.Title}
\end{array} \tag{3}
$$

We generalize this algorithm using the notations in Table 1.

Notation	**Meaning**
V	Old view definition, defined as $R_1 \bowtie R_2 \bowtie ... \bowtie R_n$
V^{new}	New view definition, defined as $R_1^{new} \bowtie R_2^{new} \bowtie ... \bowtie R_n^{new}$
R_i	Relation with index i.
R_i'	Relation R_i after a number of updates.
R_i^{new}	The relation R_i is replaced by R_i^{new} after view synchronization.
$\sigma(R_i)$	State of relation R_i describing both its schema and data.

Table 1: Notations

Assume a relation R_i is dropped or some of its attributes are dropped, VS will find a replacement R_i^{new} based on the containment information for the dropped relation R_i [LNR02]. The view V is rewritten as $V^{new} = R_1 ... \bowtie R_{i-1} \bowtie R_i^{new} \bowtie R_{i+1} ... \bowtie R_n$. Note that R_i^{new} can be a join over several tables. In the above example,

VS locates an alternative source table *Comments* that can be used to replace the *Review* attribute in the view definition. A join is necessary between *Catalog'* and *Comments* to avoid a cartesian product. The resulting join $Catalog' \bowtie Comments$ can be viewed as a replacement of the original $Catalog$ relation, denoted as $Catalog^{new}$. We generalize this view synchronization process as $M = r(VD)w(VD)c(MV)$. Note that here the VD is an in-memory data structure. The $w(VD)$ is just to modify that in-memory view definition in order to generate the maintenance query. The actual physical update of view definition (e.g., updating system catalog) is done in the $c(MV)$. One important feature of the view synchronization process is that the rewriting of the view query is not restricted to be equivalent to the original one in terms of their extent [LNR02, VMP03]. This is reasonable for information integration over a large scale and dynamic data sources. However, such non-equivalent rewriting makes the next step, view adaptation necessary.

2.1.3 View Adaptation (VA)

View Adaptation (VA) [NR99, GMRR01] incrementally adapts the view extent after the rewriting of the view definition. Since *RenameSC* will not affect the view extent, there is no need to do any view adaptation work. For *DropSC*, since the rewritten view definition may not be equivalent to the original one, adaptation of the view extent to be consistent with the new view definition is mandatory. The basic idea in [NR99] is to first compute the delta between the old relation and the new replaced relation.

Assume the original view V is defined as $V = R_1 \bowtie ... \bowtie R_{i-1} \bowtie R_i \bowtie R_{i+1}... \bowtie R_n$ and the new view $V^{new} = R_1 \bowtie ... \bowtie R_{i-1} \bowtie R_i^{new} \bowtie R_{i+1}... \bowtie R_n$. To incrementally compute V^{new} from V, we computer $\Delta R_i = R_i^{new} - R_i$. The view delta change is then evaluated as $\Delta V = \Delta R_i \bowtie R_1... \bowtie R_{i-1} \bowtie R_{i+1}... \bowtie R_n$.

In combination with the view synchronization, the full maintenance process for a *DropSC* is generalized as "$M(DropSC) = r(VD)w(VD)r(DS_1)r(DS_2)...r(DS_n)w(MV)c(MV)$". The maintenance of a *RenameSC* is generalized as "$M(RenameSC) = r(VD)w(VD)c(MV)$". Note that these high-level generalizations of various maintenance processes in this section are independent of the particular VM, VS or VA algorithms or even the underlying data model, which makes our proposed concurrency control strategy in this paper generally applicable to these known work.

2.2 View Maintenance Anomalies

2.2.1 Illustrative Examples

Using a motivating example, we now illustrate different types of the view maintenance anomalies that may arise when the data sources are autonomous.

Example 2 *We use the data sources and view in Example 1. As mentioned in Section 2.1.1, given a data*

update, "ΔI = insert (10, 'Data Integration Guide', 'Adams', 35.99)", the View Maintenance process will issue an incremental view maintenance query Q defined in Query (2). Two different anomalies can be distinguished during the processing of this maintenance query:

(a) **Duplication Anomaly***: Assume that before the execution of query (2), the* Catalog *table committed a data update ΔC = ('Data Integration Guide', 'Adams', 'Engineering', 'Princeton' ...). This new tuple would be included in the query result of query (2). Thus one final tuple ('Amazon', 'Data Integration Guide', 35.99, 'Adams', 'Princeton', 'Engineering') will be inserted into the view. However, later when the view manager processes ΔC, the same tuple would be inserted into the view again. A duplication anomaly occurs due to concurrent data updates [ZGMHW95].*

(b) **Broken Query Anomaly***: Now assuming the Review attribute is no longer kept in the Catalog table before query (2) is processed. Query (2) then faces a schema conflict and cannot succeed since the required column Review is no longer available.*

Hence the anomaly is caused by the autonomy of sources that may conflict with the view maintenance process. A concurrent data update may result in an *incorrect query result* returned by a maintenance query while a concurrent schema change may result in a query failed to be processed by the respective data source.

2.2.2 Types of Maintenance Anomalies

We formally define the anomalies below.

Definition 1 *Assume one update $w(DS_i)$ at source DS_i and one update $w(DS_j)$ at source DS_j. Maintenance of neither update has finished yet. We say that the update $w(DS_j)$ conflicts with the maintenance $M(w(DS_i))$ iff the source update $w(DS_j)$ is committed at DS_j* **before** *the query $r(DS_j)$ of $M(w(DS_i))$ is answered. We call such a conflict* **view maintenance anomaly**.

Note that the anomaly would never occur for $M(RenameSC)$ because it does not send any maintenance queries (Section 2.1.3). While the anomaly could occur during either $M(DU_1)$ or $M(DropSC_1)$ due to another concurrent DU_2, $RenameSC_2$ or $DropSC_2$.

Based on the types of $w(DS_i)$ and $w(DS_j)$, there are three types of maintenance anomalies (six cases) as listed in Table 2. Among them, the anomalies I are caused by concurrent DU, while the anomalies II are caused by concurrent $RenameSC$. The anomalies of type III are due to concurrent $DropSC$. The two cases in Example 2 are anomalies of type I and III, respectively.

Notice that for the anomalies I in Table 2, we still could get some query results returned, however the results may be incorrect. For the anomalies of II and III in Table 2, we may not even be able to get any query result

Type	Cases
I	A DU_2 concurrent to the maintenance of $M(DU_1)$
	A DU concurrent to the maintenance of $M(DropSC)$
II	A $RenameSC$ concurrent to the maintenance of $M(DU)$
	A $RenameSC$ concurrent to the maintenance of $M(DropSC)$
III	A $DropSC$ concurrent to the maintenance of $M(DU)$
	A $DropSC_2$ concurrent to the maintenance of $M(DropSC_1)$

Table 2: Three Types of Maintenance Anomalies.

back due to the schema inconsistency between the maintenance query and the underlying sources. In this paper, we will introduce a comprehensive solution framework to solve all these anomalies.

3 The DyDa Framework

Figure 3 depicts the architecture of our DyDa view management system. The framework is divided into three spaces: view space, view manager space and remote source space. The view space houses the extent of the MV. It receives view deltas from the view manager space to refresh the MV. The remote source space is composed of data sources and their corresponding wrappers. We assume that all data source transactions are local to their respective sources and every data update and schema change at a data source is reported to the view manager once committed at the data source (or the changes can be detected and extracted by the wrapper).

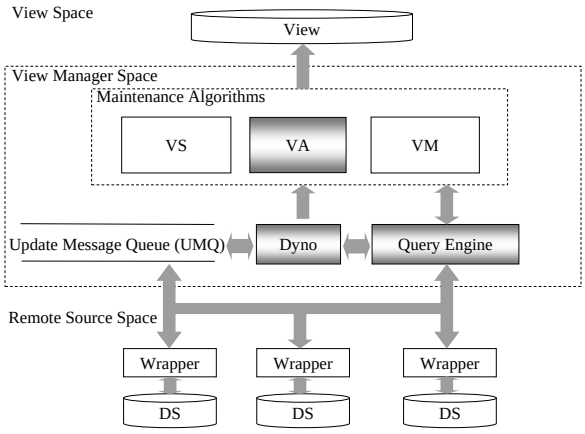

Figure 3: Architecture of DyDa Framework

The view manager aims to maintain the MV under source updates. It consists of the general view management algorithms, such as VS, VA and VM as introduced in Section 2.1. These three algorithms allow the system to

handle individual data update and schema change. According to the type of source update, the view manager may use either VS, VA, or VM algorithms to update the MV correspondingly. In this layer, there are also *Update Message Queue (UMQ)* and the *Query Engine*. The UMQ collects and manages the committed updates from the data sources, which are either data updates or schema changes. After maintaining the head update in the queue, UMQ will remove it from the queue and start processing the next. The Query Engine is responsible for query processing, namely, sending down maintenance queries to individual data sources and then collecting and assembling the query results.

Furthermore, in Figure 3 we indicate all modules which contain extended functionalities in order to handle the anomaly problems as described in Section 2.2.2 by shading them. We propose different strategies to handle the three types of anomalies in Table 2. In particular, we distinguish between two categories of our concurrency control strategies, namely, intra-maintenance process and inter-maintenance process.

Intra-maintenance process strategy solves the concurrency within one maintenance process. More precisely, it solves the anomalies of type I and II by this technique. There has been extensive study in the literature [ZGMHW95, AASY97, ZGMW96, SBCL00] to solve anomaly I, namely, the concurrency caused by data updates. The basic idea is to use *compensation* to remove any erroneous tuples from the query result. We extend this method to compensate the maintenance query when there also exists conflicting *RenameSC*. *Query Engine* employs this algorithm when processing the maintenance query to solve the anomalies of type I and II within one maintenance process (Section 4). The essence of this technique is that it solves the anomalies in a fine level without affecting the whole maintenance process.

Inter-maintenance process strategy globally schedules the maintenance processes, which is particular important to solve anomaly of type III, i.e., concurrent *DropSC*. While the state-of-the-art view maintenance algorithms assume the order of maintenance of updates simply based on their arrival order, however, such order is no longer appropriate and may cause anomaly III. We formally identify that such anomalies are due to the violation of dependencies between the maintenance processes. To solve this, we introduce a dynamic maintenance scheduler (*Dyno*) to correct such anomalies by rescheduling the maintenance order (Section 5). An advanced VA algorithm (Section 6) is designed to process the mixed batches of updates scheduled by *Dyno*.

In the rest of this paper, we will demonstrate how these two different level strategies maintain the MV consistent with the data sources even all three anomalies occur in an interleaved fashion.

4 A Compensation Strategy for Anomalies I and II

In this section, we will introduce a compensation-based algorithm to solve the anomalies of type I and II when processing a maintenance query $r(DS_i)$. First, we need to determine the updates $< w(DS_i) >$ that are concurrent to this query. Similar to the prior work [AASY97, ZGMHW95], we rely this on a FIFO assumption.

Assumption 1 *The network communication between an individual data source and the view manager is FIFO.*

Assumption 1 guarantees that when we receive the query result of $r(DS_i)$, all updates $w(DS_i)$ committed prior than the answering of query $r(DS_i)$ have already arrived the view manager in UMQ. If there exists DU among $< w(DS_i) >$, then anomaly I occurs. If there exists $RenameSC$, then anomaly II occurs. If there exists $DropSC$, then anomaly III occurs.

When the concurrent updates $< w(DS_i) >$ contain only DUs, a number of algorithms in the literature [AASY97, ZGMHW95, ZGMW96] propose to use compensation queries to remove the erroneous tuples from the query results. Take the SWEEP algorithm in [AASY97] for example, suppose the returned maintenance query result $r(DS_i)$ is $\Delta R_j \bowtie R_i'$ instead of $\Delta R_j \bowtie R_i$. This error query result can corrected by $\Delta R_j \bowtie R_i'$ - $\Delta R_j \bowtie \Delta R_i$, where $\Delta R_i =< w(DS_i) >$.

However, if there exists $RenameSC$ in $< w(DS_i) >$, this basic compensation idea fails because the maintenance query $r(DS_i)$ faces a schema conflict and fails to return any query results. But since the underlying data is still available and the only difference is the names, we can rewrite the query $r(DS_i)$ by specifying the new names of DS_i. The high-level abstraction of this algorithm is shown in Figure 4.

```
GLOBAL DATA
UMQ: Queue;   /* Update Message Queue */
DropSCFlag = FALSE: Boolean; /* DropSC flag, set true by UMQ_Manager */
MT: NameMappingTable /* Name in View Definition .vs. Current Name */;

Boolean FUNCTION Extended_Compensation(Query Q, QueryResult QR)
CDU: ConcurrentDUSet;
CRen: ConcurrentRenameSCSet;
BEGIN
1. CollectConcurrentUpdates(UMQ, QR, CDU, CRen);
2. if CRen is empty then
                return compensation_algorithm(Q, QR, CDU);
3. else
                UpdateMappingTable(MT, CRen);
                return Query_Engine(Q, QR);
END

Boolean FUNCTION Query_Engine (Query Q, QueryResult QR)
Query Q';
BEGIN
1. Q'=RewriteQuery(Q, MT);
2. ExecuteQuery(Q', QR);
3. If DropSCFlag = true then return false;
4. else
        return Extended_Compensation(Q', QR);
END
```

Figure 4: Compensation Algorithm for Anomalies I and II

We employ a name mapping table which describes the metadata name in the view definition and its current name. These two names are initially equivalent. As mentioned early, the $Query\_Engine$ is used to execute the maintenance query which is decomposed from the view definition by either VM or VA. This maintenance

query will first be rewritten using the current name of data sources. Then we apply our proposed compensation algorithm to correct the maintenance query results if no anomaly III, i.e., *DropSC* occurs [1].

Our extended compensation algorithm first collects the concurrent source updates in line 1 by the FIFO assumption. If there is no concurrent *RenameSC*, we can apply any of the existing compensating algorithm [AASY97, ZGMHW95, ZGMW96] for data updates in line 2. If the concurrent *RenameSC* does occur, we need to update the name mapping table to change the current metadata name and re-execute this query.

Theorem 1 *Assume the data source state $\sigma(DS_i)$ evolves to the state $\sigma(DS_i')$ by updates $\{w(DS_i)\}$. The maintenance query result $r(\sigma(DS_i'))$ can be compensated to $r(\sigma(DS_i))$ by our compensation algorithm iff $w(DS_i)$ is either DU or RenameSC.*

Proof: First, we define $\sigma_D(DS_i)$ as the data state and $\sigma_S(DS_i)$ as the schema state of DS_i. By definition, we have $\sigma(DS_i) = (\sigma_D(DS_i), \sigma_S(DS_i))$. Since $\sigma(DS_i) \overset{\{w(DS_i)\}}{\longrightarrow} \sigma(DS_i')$, we have $(\sigma_D(DS_i), \sigma_S(DS_i)) \overset{\{w(DS_i)\}}{\longrightarrow} (\sigma_D(DS_i'), \sigma_S(DS_i'))$. Next, the query $r(\sigma(DS_i))$ can also be precisely defined as $r(\sigma_D(DS_i), \sigma_S(DS_i)) = P(\sigma_D(DS_i))$. That is the query r is written based on the schema state of $\sigma_S(DS_i)$ and executed predicates P on data state $\sigma_D(DS_i)$.

1). We first consider the case when all $w(DS_i)$ are data updates $\{DU_i\}$, i.e., $\sigma_S(DS_i) = \sigma_S(DS_i')$. Hence $r(\sigma_D(DS_i'), \sigma_S(DS_i)) = r(\sigma_D(DS_i'), \sigma_S(DS_i')) = P(\sigma_D(DS_i'))$. This is a known problem and a number of existing solutions [AASY97, SBCL00, ZGMHW95], generalized as function *Comp* that can generate the query result $P(\sigma_D(DS_i))$ by $Comp(P(\sigma_D(DS_i')), \{DU_i\})$.

2). Next, we consider the case when in addition to $\{DU_i\}$, $\{w(DS_i)\}$ also contains *RenameSC*, i.e., $\{w(DS_i)\} = \{DU_i\} \cup \{RenameSC_i\}$. We have $\sigma_D(DS_i) \overset{\{DU_i\}}{\longrightarrow} \sigma_D(DS_i'')$ and $\sigma_S(DS_i) \overset{\{Rename_i\}}{\longrightarrow} \sigma_S(DS_i'')$. The original query $r(\sigma_D(DS_i''), \sigma_S(DS_i))$ is not valid, because the data state and schema state is not consistent. The rewritten query $r(\sigma_D(DS_i''), \sigma_S(DS_i''))$ returns the result $P(\sigma_D(DS_i''))$.

An important property of *RenameSC* is that it only modifies the schema name not the actual data, i.e., $\sigma_D(DS_i') = \sigma_D(DS_i'')$. Hence we back to first case by applying $P(\sigma_D(DS_i)) = Comp(P(\sigma_D(DS_i'')), \{DU_i\}) = Comp(P(\sigma_D(DS_i')), \{DU_i\})$. Note that this technique does not work for *DropSC* since the data state is also changed by *DropSC*. ∎

5 Detection and Correction of Anomaly III

Section 4 guarantees that if there is no concurrent *DropSC*, the compensation solution can successfully generate the correct maintenance query result. However, if there exists any concurrent *DropSC*, i.e., anomalies of type

[1] The *DropSCFlag* is an important optimization as we will see later.

III, the solution no longer works. In this section, we describe that the reasons for this are the violations of dependencies between maintenance processes. We first formalize the dependencies and point out their relationship to the anomalies III then propose algorithms to solve them.

5.1 Dependencies among Maintenance Processes

5.1.1 Concurrent Dependency

There are two cases for anomalies III, namely, the maintenance process $M(DU_1)$ or $M(DropSC_2)$ conflicts with another $DropSC_3$. More formally, assume the DU_1 or $DropSC_2$ occurs at source DS_i and the $DropSC_3$ occurs at DS_j, respectively. Their maintenance processes are generalized as "$M(DU_1) = r_1(VD)r_1(DS_1)r_1(DS_2)...r_1(DS_n)$ $w_1(MV)c_1(MV)$", "$M(DropSC_2) = r_2(VD)w_2(VD)r_2(DS_1)r_2(DS_2)...r_2(DS_n)w_2(MV)c_2(MV)$" and "$M(DropSC_3)$ $= r_3(VD)$ $w_3(VD)r_3(DS_1)$ $r_3(DS_2)...r_3(DS_n)w_3(MV)c_3(MV)$" in Section 2.1. By Definition 1, there is conflict between $r_1(DS_j)/DropSC_3$ or $r_2(DS_j)/DropSC_3$.

Notice that there is also a read-write conflict on the view definition between these maintenance processes, i.e., $r_1(VD)/w_3(VD)$ and $r_2(VD)/w_3(VD)$. Interestingly, this conflict on the view definition is the reason for the conflict between $r_1(DS_j)/DropSC_3$ or $r_2(DS_j)/DropSC_3$. The rationale is that the query $r_1(DS_j)$, $r_2(DS_j)$ has been constructed based on the read of the view definition $r_1(VD)$, $r_2(VD)$. For instance, in Example 1.b, the maintenance query (2) is constructed based on the view definition query (1) over $Catalog$. If this view definition read $r_1(VD)$, $r_2(VD)$ conflicts with $w_3(VD)$, the constructed query $r_1(DS_j)$, $r_2(DS_j)$ may no longer reflect the actual schema of DS_j.

Definition 2 *Let $w(DS_i)$ and $w(DS_j)$ denote two updates committed on data sources DS_i and DS_j, where $w(DS_i)$ is either a DU or DropSC. The view manager has not finished maintenance for either of them. We say that maintenance process $M(w(DS_i))$ is* ***concurrent dependent*** *(**CD**) on maintenance process $M(w(DS_j))$, denoted by $M(w(DS_i)) \xleftarrow{cd} M(w(DS_j))$ iff $M(w(DS_i))$ contains read view definition and $w(DS_j)$ is a DropSC that $M(w(DS_j))$ contains write view definition.*

Concurrency dependency defines the relationship between maintenance processes over critical resource *view definition*. There is a close relationship between concurrent dependency and anomaly of type III as will be explained in Section 5.2. Note that there are several differences between the *concurrent dependencies* and *wait-for* dependencies in traditional transactions [BHG87]. First, the conflict is on the view definition not on the actual tuples. Second, even if the maintenance of a sequence of updates is processed in a serial fashion, dependencies between them may still occur. The rationale is that the source updates are committed autonomously and thus may conflict with any ongoing maintenance processes. Third, the dependency direction is always from a write to a read of the view definition since the concurrent schema change may invalidate the old view definition and

consequently any ongoing maintenance processes. The concurrency of the second case of Example 2 in Section 2.2.1 is of type "$DU \xleftarrow{cd} DropSC$". An example of "$DropSC_1 \xleftarrow{cd} DropSC_2$" will be given in Section 5.3.

5.1.2 Semantic Dependency

The MV is maintained consistent if it reflects some valid state of each data source, $\sigma(DS_i)$. Assume the state of DS_i evolves as $\sigma(DS_i) \xrightarrow{\Delta^1} \sigma(DS_i^1) \xrightarrow{\Delta^2} \sigma(DS_i^2)$. It is important for MV to maintain Δ^1 and Δ^2 in that order. If the MV maintains Δ^2 first, then MV reflects the data source state $\sigma(DS_i')$ as $\sigma(DS_i) \xrightarrow{\Delta^2} \sigma(DS_i')$, which is neither $\sigma(DS_i^1)$ nor $\sigma(DS_i^2)$. In this case, *Strong consistency* [ZGMHW95] that MV reflects the valid state of data sources in the same order cannot be achieved. Furthermore, the MV consistency may not even *converge*, i.e., the final state may be invalid too. For example, assume that there are two updates from the same relation, either an insert tuple A followed by a delete A, or a rename B to C and then rename C to D. If we reverse their maintenance order, we cannot correctly maintain the MV.

Thus it is necessary to preserve the processing order of updates from shared resources such as the same tuple, the same attribute or the same relation in the examples above. For simplicity, we employ the idea of the same relation here. We now formally define this as a **semantic dependency** (**SD**).

Definition 3 *Assume two updates $w_1(DS_i)$ and $w_2(DS_i)$ from data source DS_i, then $M(w_2(DS_i))$ is **semantic dependent (SD)** on $M(w_1(DS_j))$, denoted by: $M(w_2(DS_i)) \xleftarrow{sd} M(w_1(DS_i))$ iff $w_1(DS_i)$ is committed before $w_2(DS_i)$.*

5.2 Dependency Properties

The two types of dependencies, *concurrent dependency* and *semantic dependency* share an important property, namely, both represent constraints on the maintenance order between updates. Hence we now abstract them as one common concept.

Definition 4 *For two updates m_1 and m_2, we define $M(m_2)$ is **dependent on** $M(m_1)$, denoted by $M(m_2) \leftarrow M(m_1)$ if either $M(m_2)$ is* concurrent dependent *on $M(m_1)$ by Definition 2 or $M(m_2)$ is* semantic dependent *on $M(m_1)$ by Definition 3.*

Definition 5 *Given two updates m_1 and m_2 in the UMQ. If m_1 precedes m_2 in the Update Message Queue (UMQ), then we denote this by "$pos(m_1, UMQ) \prec pos(m_2, UMQ)$". We define the **dependency relationship** between $M(m_1)$ and $M(m_2)$ to be:*

1. ***independent** iff there is no dependency between $M(m_1)$ and $M(m_2)$ by Definition 4.*
2. ***safe dependent** iff pos(m1, UMQ) $\prec$ pos(m2, UMQ) and all dependency orders between $M(m_1)$ and $M(m_2)$ by Definition 4 are $M(m_2) \leftarrow M(m_1)$.*

3. ***unsafe dependent*** *iff pos(m1, UMQ)* $\prec$ *pos(m2, UMQ) and there is at least one dependency* $M(m_1) \leftarrow M(m_2)$.

Consider the second case of Example 1. The concurrent dependency is $DU \xleftarrow{cd} DropSC$. However, since the $pos(DU, UMQ) \prec pos(DropSC, UMQ)$, this dependency is *unsafe* by Definition 5. It is obvious that if $M(m_2)$ is dependent on $M(m_1)$, then the maintenance $M(m_1)$ *must* be processed *before* $M(m_2)$. For a *semantic dependency*, the required order is obvious as discussed in Section 5.1.2. For a *concurrent dependency*, as shown in Section 5.1.1, the write view definition operation has to be done *first* to solve the read-write conflict on view definition. The concurrent schema change invalidates the view definition, hence rewriting it becomes critical.

Theorem 2 *An anomaly III occurs during the maintenance* $M(w(DS_i))$ *only if there is at least one* unsafe concurrent dependency $M(w(DS_i)) \xleftarrow{cd} M(w(DS_j))$.

An anomaly of type III implies an *unsafe* dependency, but not vice versa. The proof is straightforward. If an anomaly III occurs during the maintenance of $w(DS_i)$, by Definition 1, then there is a $DropSC$ denoted as $w(DS_j)$ that conflicts with $M(w(DS_i))$. By Definition 2, there is *concurrent dependency* $M(w(DS_i)) \xleftarrow{cd} M(w(DS_j))$. Since $M(w(DS_i))$ is scheduled before $M(w(DS_j))$, this *concurrent dependency* is unsafe. ∎

Since the two types of dependencies both represent constraints between maintenannce processes, we now put them together in a common structure.

Definition 6 *A Dependency Graph is a directed graph G=(V,E) with the set of nodes V denoting all updates* m_i *in the UMQ and with the set of directed edges E denoting the dependencies* $e(m_i, m_j)$ *between two updates* m_i *and* m_j *iff a* concurrent dependency *or* a semantic dependency *exists between* $M(m_i)$ *and* $M(m_j)$.

The complexity of identifying *concurrent dependencies* between maintenance processes is $O(mn)$, where m is the number of $DropSC$ and n is the number of updates. The reason is that each *concurrent dependency* involves at least one $DropSC$. In the worst case, one $DropSC$ would have one *concurrent dependency* to all other updates. Second, the complexity of building *semantic dependencies* between updates is $O(n)$, where n is the number of updates. To achieve this, we can create one bucket for each data source and scan the list of updates once. Thus the time complexity of building a *dependency graph* is $O(mn) + O(n)$, i.e., $O(mn)$.

5.3 Cyclic Dependencies

A set of dependencies may comprise a cycle as illustrated by the example below. This is similar to the deadlock in serializability theory [BHG87]. Given the source relations from Example 1, let us refer to the drop of *Review* attribute as SC_1. Now assume the *Bookstore* tunes the mapping of the XML documents as shown in Figure 5, which we refer as SC_2.

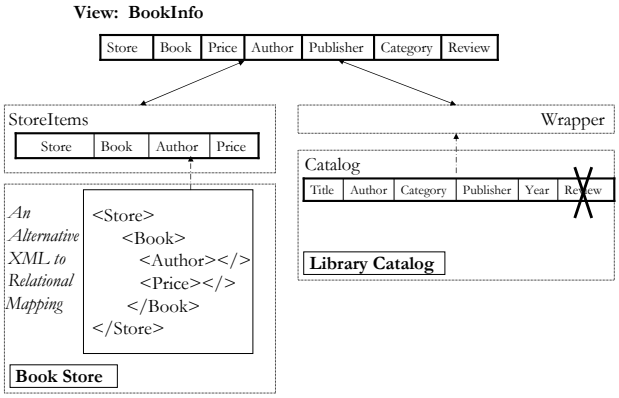

Figure 5: Concurrent Tuning of XML Mapping

Assume these schema changes SC_1, SC_2 have already been committed at their data sources. If we process SC_1 first, the view definition may be rewritten into Query (3) (Section 2.1.2). However, this new view definition is no longer consistent with the sources since the table *Store* or *Item* is no longer available due to SC_2. Similarly, if we process SC_2 first, the view definition may be rewritten into Query (4). Again, the view definition is not valid since the attribute *Review* is no longer available due to SC_1. Hence the maintenance query constructed based on either of these two view definitions would fail. We call such situation a *cyclic* dependency.

$$\begin{array}{lll} \text{CREATE VIEW} & BookInfo' \text{ AS} & \\ \text{SELECT} & Store,\ Book,\ S.Author,\ Price,\ Publisher,\ Category,\ Review & \\ \text{FROM} & StoreItems\ S,\ Catalog\ C & \\ \text{WHERE} & S.Book\ =\ C.Title & \end{array} \tag{4}$$

By Definition 2, there are concurrent dependencies $M(SC_1) \xleftarrow{cd} M(SC_2)$ and $M(SC_2) \xleftarrow{cd} M(SC_1)$. This comprises a cycle. Intuitively, the reason is that all these updates in a "cycle" modify the source schema. If the view manager rewrites the view definition based on a subset of them, the view definition is still not consistent with the underlying sources. To handle such cyclic dependencies, aborting some of the source updates as often used to resolve the deadlock in traditional databases is not feasible since the source updates have already been autonomously committed and cannot be aborted. A viable idea is to consider all these updates *at the same time* as we will elaborate later.

5.4 Dyno: Detection and Correction of *Unsafe* Dependencies

After we detect an *unsafe* dependency between the maintenance processes, we need to reschedule the maintenance processes to turn the *unsafe* dependencies into *safe* ones (or equivalently speaking, we need to reorder the updates

in the UMQ). We achieve this by sorting the *dependency graph* constructed during the detection phase.

Theorem 3 *Given a fixed number of updates, if the dependency graph is acyclic, we can obtain a maintenance order with all dependencies safe.*

Theorem 3 holds since given an acyclic dependency graph, we can simply apply a *topological sort* algorithm [Tar72] to obtain a partial order of nodes. The complexity is $O(n+e)$, where n is the number of nodes (updates) and e is the number of edges (dependencies). This way we obtain an order of updates that has all dependencies in their safe direction.

However, if the dependency graph is cyclic as shown in Section 5.3, the *topological sort* algorithm cannot generate a partial order [Tar72]. For this, we first identify all cycles in the dependency graph (similar to identifying *strong connected components* in [Tar72], with complexity also $O(n + e)$). Traditional transaction processing [BHG87] breaks the cycle (or deadlock) by removing one of the nodes in the cycle, in other words, aborting one of the transactions. However, this strategy is not appropriate here because the source update is autonomous and hence not abortable. Instead of removing one node, we propose to *merge* these nodes (updates) into a *merged* one to be processed at a time. The intuition of the merge operation is that since we cannot process these updates separately, we instead process them in one *atomic* batch. This however requires a new view adaptation algorithm capable of processing such combined batches of updates which we will describe in Section 6. After removing all cycles in the dependency graph, we can apply *topological sort* again to the now acyclic dependency graph to obtain a maintenance order with all dependencies safe.

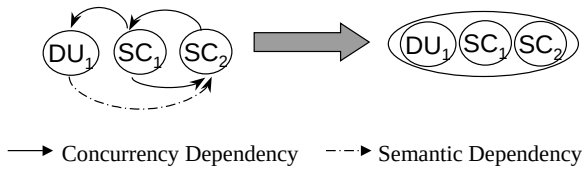

Figure 6: Examples of Unsafe Dependency Correction

Figure 6 depicts this dependency correction algorithm for our running example. Assume in the view (1) of Example 1, three updates occur, namely, one data update DU_1 (ΔI in Example 2) then two schema changes (SC_1 and SC_2 in Section 5.3), in that order. Since DU_1 and SC_2 are from the same source, there is a semantic dependency between them. Several *concurrent dependencies* are *unsafe* initially, such as $DU_1 \longleftarrow SC_1$ and $SC_1 \longleftarrow SC_2$. Figure 6 illustrates the merge step of these three nodes into one big node to make the dependency graph acyclic. The final schedule is to maintain these updates altogether in one batch.

We now introduce a dynamic maintenance scheduler *Dyno* utilizing the above basic idea for solving the anomaly of type III. Figure 7 details the *Dyno* algorithm. Dyno checks the *DropSCFlag* in line 1 before the maintenance processing. If there is no *new DropSC* after the *last* correction, we can avoid the detection and correction steps because the newly arriving *DU*s or *RenameSC*s will not introduce any additional *unsafe*

```
PROCESS Dyno()
X: UpdateMessage;
Success: Boolean;
BEGIN
    LOOP (FOREVER)
1:      IF Test_If_True_Set_False(DropSCFlag) THEN
            // If no new DropSC change, we can avoid detection and correction
2:          UMQ.Build_Dependency_Graph();
3:          UMQ.Topological_Sort_with Cycle_Merge();
4:      ENDIF
5:      X = UMQ.GetHead();
6:      Success = View_Maintenance(X); // Start maintenance of update X;
7:      IF Success = TRUE THEN // If no anomaly III occurs during maintenance;
8:        UMQ.RemoveHead();
        // If anomaly III occurs, it will be corrected in the next loop;
    ENDLOOP
END
```

Figure 7: Dyno: DYNmaic reOrdering Algorithm

dependencies. If a *DropSC* did occur, Dyno will construct the *dependency graph* (line 2) and correct all *unsafe* dependencies (line 3) in the UMQ. After that, Dyno starts to maintain the update (line 6). If no anomaly of type III occurs during maintenance, then the head update will be removed and *Dyno* proceeds to process the next update. Recall that the anomalies I and II will be solved within one maintenance process. If the anomaly III did occur, Dyno kicks off the correction algorithms in line 1-3 to reschedule the maintenance order. It is straightforward to prove that since Dyno provides a maintenance order having all the dependencies $safe$, no anomaly of type III would occur.

Theorem 4 *Given a number of updates, ΔX_1, ..., ΔX_m, the anomaly of type III caused by ΔX_i will not occur under the maintenance order scheduled by Dyno.*

6 A Novel VA Algorithm for Processing Complex Updates

In Section 5.4, the correction algorithm generates a maintenance order for a number of updates. However, if there is any cycle in the dependency graph, some updates will be *merged*. Such *combined* updates now could contain both schema and data updates over different data sources. To our knowledge, state-of-the-art view maintenance algorithms [AASY97, SBCL00, LNR02] cannot maintain such mixed updates in one maintenance process. Below we introduce a batch extension of previous view maintenance algorithms for processing such merged updates.

6.1 Preprocessing Step of the Source Updates

After *merging* cyclic dependent updates as described in Section 5.4, we have a *complex* update containing updates from multiple sources. We first partition these updates based on the data source DS_i that they originate from. After that, we further partition the updates from the same data source DS_i into two subgroups, namely, the data updates group defined as $< DU_i >$, and the schema changes group defined as $< SC_i >$. Without loss of generality, we define such a merged update as: $U = \{(< SC_1 >, < DU_1 >), (< SC_2 >, < DU_2 >), ..., (< SC_n >, < DU_n >)\}$, where n is the number of data sources.

For such a merged update, first, the schema changes in $< SC_i >$ can sometimes be combined, e.g., if rename A to B and then B and C occur in the same data source, we could simply rename A to C. Second, the data updates may be inconsistent with their schema due to some schema changes in between. For example, assume two inserts into the same relation with a drop attribute in between, the latter tuple will have fewer columns. Thus our first step is to preprocess these updates in U from the same source to adjust these differences and to enable us to maintain them in one batch.

Unifying Schema Changes: Given a group of schema changes $< SC_i >$ from one data source DS_i, we rewrite $< SC_i >$ to an equivalent group. This would optimize the maintenance, because those removed SC_i need not be processed.

Table 3 shows all possible combinations T(SC_1, SC_2) between two SCs (SC_1, SC_2) with SC_1 the row entry and SC_2 the column entry. Here R, S, T represent relations, R.a, R.b, R.c represent attributes. If the entry of the combination in Table 3 is empty, then it means that the combination of the two operations has no effect on each other. Hence the combined result will keep both of them.

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

Table 3: Combination Rules between Two SCs

Finally we define $< SC_i' >$ as the schema changes set after combining the schema changes in $< SC_i >$ pairwisely using the rules above.

Unifying Data Updates: We then try to combine the data updates in $< DU_i >$, some of which might be of different schemata. To achieve this, 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 the new relation R_i'. These common attributes are actually the original R_i's attributes minus the dropped ones. The purpose of this projection is to make the

$< DU'_i >$ schemata consistent with each other while still correct for maintenance. We justify this below.

Lemma 1 *All DU'_i must have the same schemata.*

Proof: We prove this by contradiction. Suppose that one tuple A contains one more attribute than another tuple B. This extra attribute must be either an added attribute of A or a dropped attribute of B. 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 attribute will not appear in $attr(R'_i) \cap attr(R_i)$. ∎

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: +(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 $attr(R_2) \bowtie attr(R'_2) = \{A\}$. We get $< DU'_2 >= \Pi_A < DU_2 >=< +(3), +(4), -(5) >$, which are schemata consistent. Now let's examine the new view definition V^{new}, a possible rewriting might be $V^{new}(A,B,C) = R_1(A,B) \bowtie \Pi_{A,C}(\Pi_A R'_2(A,D) \bowtie R_3(A,C))$. Since only attribute A of R'_2 is involved in the view definition V^{new}, we confirm that $< DU'_2 >$ is sufficient for the view maintenance.*

Using the process described above, we can convert U into $U' = \{(< SC'_1 >, < DU'_1 >), (< SC'_2 >, < DU'_2 >), ..., (< SC'_n >, < DU'_n >)\}$. We characterize the relationship between $\{SC'_i\}$ and $\{DU'_i\}$ as follows:

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

In the next section, we will show that we can also safely have $< DU'_i >= \emptyset$ when $< SC'_i >$ contains a *"Drop Attribute"*.

6.2 Incremental View Adaptation Step

Now we are ready to incorporate these modified updates U' into the MV. Recall that the view manager process involves two steps to incorporate schema changes, namely, view rewriting (VS) and then view adaptation (VA). Below we describe how to maintain U' by these two steps.

1. View Rewriting by View Synchronization: We first apply view synchronization (VS) to all $< SC'_i >$ in U', i=1..n. Note that we rewrite the view definition for each schema change in $< SC'_i >$ by the VS techniques described in Section 2.1.2. Here we denote the old view definition as $V = R_1 \bowtie R_2 \bowtie ... \bowtie R_n$ and the new view definition as $V^{new} = R_1^{new} \bowtie R_2^{new} \bowtie ... \bowtie R_n^{new}$.

In particular, we have the following possible rewritings for each relation R_i. If the updates contain *drop relation*, then by Section 6.1 we know that $< SC'_i >$ has only that *drop relation* after merging the schema

changes. Thus the rewriting is just to find an alternative table for replacement. If the updates contain *drop attributes*, alternative tables and additional joins may be needed as described in Section 2.1.2. If the updates do not contain any $DropSC$, then there might be only name changes on the view definition by $RenameSC$. In summary, we have each new source relation as:

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

The meanings of R_i^{new}, R_i' and R_i are described in Table 1. Here we assume a valid view rewriting exists. Otherwise the MV would become invalid and cannot be maintained. Take the cyclic dependency example in Section 5.3, the correct view definition taking both SC_1 and SC_2 into consideration should be:

CREATE VIEW	BookInfo AS	
SELECT	Store, Book, S.Author, Price, Publisher, Category, Comment as Review	
FROM	StoreItem S, Catalog C, Comments M	(5)
WHERE	S.Book = C.Title AND C.Title = M.Article	

2. Incremental View Adaptation: In order to incrementally adapt the view extent, we need to determine the delta change. Here the old view extent is $V = R_1 \bowtie R_2 \bowtie ... \bowtie R_n$ and the new view extent is $V^{new} = R_1^{new} \bowtie R_2^{new} \bowtie ... \bowtie R_n^{new} = (R_1 + \Delta R_1) \bowtie (R_2 + \Delta R_2) \bowtie ... \bowtie (R_n + \Delta R_n)$. Comparing the old and the new view extent, the delta change is:

$$\begin{aligned} \Delta V &= \Delta R_1 \bowtie R_2 \bowtie ... \bowtie R_i \bowtie ... \bowtie R_n \\ &+ R_1^{new} \bowtie \Delta R_2 \bowtie R_3 \bowtie ..... \bowtie R_i \bowtie ... \bowtie R_n + ... \\ &+ R_1^{new} \bowtie ... \bowtie R_{i-1}^{new} \bowtie \Delta R_i \bowtie R_{i+1} \bowtie ... \bowtie R_n + ... \\ &+ R_1^{new} \bowtie ... \bowtie R_i^{new} \bowtie ... \bowtie R_{n-1}^{new} \bowtie \Delta R_n. \end{aligned} \tag{6}$$

The ΔR_i is defined as

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

If there is no $DropSC$, then the delta changes are only the data updates. If there is a *"drop relation"*, since

only pre-state R_i is interested, we can safely discard $< DU'_i >$. Moreover, since some of R_i data is already dropped, we instead compute the above set-difference by replacing R_i with $\Pi_{R_i} V$ [NR99]. However, if there is *"drop attribute"*, then $< DU'_i >$ might not be empty. We prove that we can also need not explicitly consider $< DU'_i >$ since its delta effect is implicitly evaluated in $\Pi_{attr(R_i)}(R'_i \bowtie R_i^1 \bowtie R_i^2 ... \bowtie R_i^m)$. The reason is simply because $\Pi_{(attr(R_i) \cap attr(R'_i))}(R'_i) = \Pi_{(attr(R_i) \cap attr(R'_i))} R_i + \Pi_{(attr(R_i) \cap attr(R'_i)} < DU'_i >$. Thus whenever there is *DropSC* in $< SC_i >$, we can safely discard any $< DU_i >$, which further simplifies our preprocessing step in Section 6.1.

Theorem 5 *Assume the complex updates U include ΔR_1, ..., ΔR_n, which evolves each table state as: $\sigma(R_i) \xrightarrow{\Delta R_i} \sigma(R'_i)$. The MV state that reflects each individual table state is maintained from $(\sigma(R_1), ..., \sigma(R_n))$ to $(\sigma(R'_1), ..., \sigma(R'_n))$ by our adaptation algorithm.*

Proof: Assume the updates to be maintained in a batch are ΔR_1, ΔR_2, ..., ΔR_n. Each ΔR_i includes two update sets, namely, $\{DU_i\}$ and $\{SC_i\}$. Correspondingly, the state of the relation $(\sigma_D(R_i), \sigma_S(R_i))$ evolves to $(\sigma_D(R'_i), \sigma_S(R'_i))$ by $\{DU_i\}$ and $\{SC_i\}$. The original view schema state reflects each of the original table schema state. More formally, $\sigma_S(V) = (\sigma_S(R_1), ..., \sigma_S(R_n))$. The original view data state state reflects each of the original table data state, formally as $\sigma_D(V) = (\sigma_D(R_1), ..., \sigma_D(R_n))$

The unifying step for $\{SC_i\}$ in Section 6.1 is trivial, i.e., the resulting $\{SC'_i\}$ is equivalent to $\{SC_i\}$. The view definition V is then rewritten to V^{new} by $V \xrightarrow{VS_i(\{SC'_i\})} V^{new}$, i.e., employ VS algorithm for each SC'_i. Since $\sigma_S(R_i) \xrightarrow{\{SC'_i\}} \sigma_S(R'_i)$, we have $\sigma_S(V^{new}) = (\sigma_S(R'_1), ..., \sigma_S(R'_n))$. In other words, the new view definition reflects the new schema state of the sources.

Next, we consider the data state of the view. Assume the rewritten view as $V^{new} = R_1^{new} \bowtie R_2^{new} \bowtie ... \bowtie R_n^{new}$. To incrementally maintain the V^{new} from $V = R_1 \bowtie R_2 \bowtie ... \bowtie R_n$, it is known to compute ΔR_i^{new} as the set difference between R_i^{new} and R_i then apply Equation 6 to correctly compute ΔV.

We now prove that $\sigma_D(V^{new}) = (\sigma_D(R'_1), ..., \sigma_D(R'_n))$.

1) If ΔR_i does not contain *DropSC*, then $\Delta R_i^{new} = \{DU_i\}$ (*RenameSC* will not change data.); 2) If R_i is dropped, then $\sigma_D(R'_i) = \emptyset$. Since R_i is replaced by R_i^{new}, the V^{new} no longer reflects any state of R_i; 3) If some of the R_i's attributes are dropped, the R_i^{new} is in the form of $(R'_i \bowtie R_i^1 \bowtie R_i^2 ... \bowtie R_i^m) = (R_i + \{DU_i\}) \bowtie R_i^1 \bowtie R_i^2 ... \bowtie R_i^m$. In the first the third case, the $\{DU_i\}$ are both considered in ΔR_i^{new}, while the second case is trivial. Hence $\sigma_D(V^{new})$ reflects the new data state of each $\sigma_D(R'_i)$. ∎

Theorem 6 *The proposed techniques achieve strong consistency for MV maintenance.*

Proof: The proof of the overall correctness is basically a combination of the theorems we have already developed. We start with the static case assuming there are n source updates ΔU_1, ..., ΔU_n. The *Dyno* technique in Section 5 will schedule these maintenance processes that is free of anomaly III (Theorem 4). Assume the resulting order of

updates for maintenance is $\Delta U'_1$, ..., $\Delta U'_m$. The correctness of each maintenance $M(\Delta U'_i)$ without any anomalies is guaranteed by Theorem 5, which will generate a number of maintenance queries. Moreover, even if there are anomalies I and II when processing these maintenance queries, they can be compensated by Theorem 1. Hence, MV reflects the correct states after each $M(\Delta U'_i)$. Finally, given the *semantic dependency* constraints posed by *Dyno*, there is a partial order between $\Delta U'_i$. Thus the MV achieves *strong consistency* but not *complete consistency* because some of the intermediate states may be missing due to the merge step.

Now we consider dynamic case, i.e., new updates occur during the maintenance. If the new updates are *DU*s or *RenameSC*s, then they can be compensated by Theorem 1. If the new updates are *DropSC*s, then the maintenance is aborted and *Dyno* reschedules the whole processes. We then back to the static case. ∎

7 Experimental Evaluation

7.1 Experiment Testbed

We have implemented the above techniques in our DyDa [CZC+01] system, using Java as development language and Oracle8i as view servers and data source servers. In our experimental setting, there are six sources evenly distributed over three different source servers with one relation each. Each relation has four attributes and contains 100, 000 tuples. There is one materialized one-to-one join view defined upon these six source relations containing all twenty four attributes, residing on a fourth server. All experiments are conducted on four Pentium III PCs with 256MB memory each, running Windows NT and Oracle8i.

7.2 Individual Update Processing

We first study the individual update processing of view maintenance. We distinguish between two classes of updates, i.e., data update and schema change. We further distinguish between *RenameSC* and *DropSC*, because we expect that they have significant differences in maintenance costs.

Figure 8 depicts the average view maintenance cost under our basic experimental setting, measured in seconds (depicted on the y-axis) for different types of updates. We find that the cost for *DU* maintenance is the least while the cost of *DropSC* maintenance is significant. This is because the latter invokes both VS and VA modules. This observation provides us some intuition that it is more costly to abort and re-process a *DropSC*.

In the next three experiments, we will study the system performance under various kinds of anomalies described in Section 2.2.2, all of which are supported in our DyDa system.

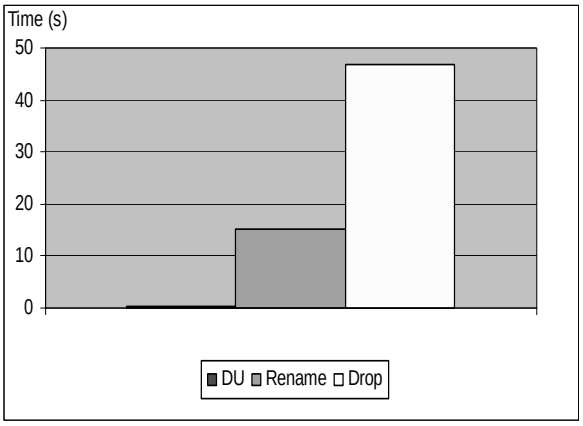

Figure 8: Comparison of Individual Update Processing Types

7.3 Study of Compensation for Anomaly I

Note that our DyDa system extends the ordinary view manager functionality to also deal with concurrent schema changes. We first study the overhead that such extended functionality may bring to the normal system's data update processing. We examine our compensation algorithm in Figure 4. Since we employ a *DropSCFlag* to indicate any concurrent *DropSC*, when there are only data updates, we can avoid the construction of dependency graph and correction step. Thus the extra cost to the existing data update maintenance algorithms is only O(1). In this experiment, we compare DyDa to SWEEP [AASY97] as the VM algorithm under a number of concurrent data updates from distributed sources to measure the extra overhead that the DyDa framework may be imposing.

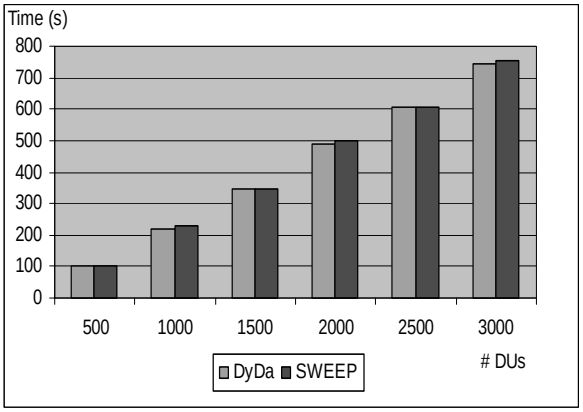

Figure 9: SWEEP vs. DyDa on Data Update Processing

Figure 9 depicts the total view maintenance cost measured in seconds (depicted on the y-axis) under different numbers of source data updates (depicted on the x-axis). From the result, we find that the extra cost is almost neglectable (less than 3%) for a number of data updates in our environmental settings. We thus conclude that

the DyDa system imposes little extra cost on data update processing while offering added support for concurrent schema change processing.

7.4 Abort Cost of Anomalies II and III

Recall that a maintenance query may fail due to the existence of some concurrent schema changes, i.e., anomalies II and III. Once a concurrent *DropSC* causes the query failure, the view manager has to abort all previous maintenance work and redo it again imposing some extra cost on the view maintenance process. While in comparison, if a concurrent *RenameSC* occurs, we simply rewrite the query using new names and try the new query again without aborting any prior effort as described in Figure 4. The extra abort cost of anomaly II is thus less than that of the anomaly III.

In this experiment, we study the cost of all four cases of anomalies II and III. To observe the exact abort cost, we employ controlled cases here, i.e., one data update processing aborted by one schema change and one schema change processing conflicts with another one. Two different environment settings are compared. First, we measure the maintenance cost of all updates by spacing them far enough so that each source update occurs after the completion of the previous view maintenance step. This way they will not interfere with each other, which can be considered to be the minimum cost as no concurrency handling cost would arise. Second, we allow the anomalies to occur by spacing the updates close enough and measure the cost.

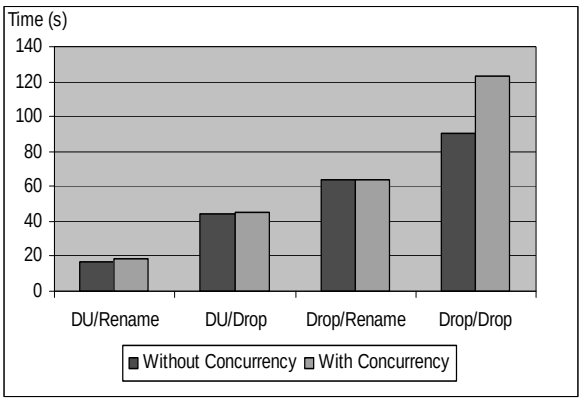

Figure 10: Abort Cost for Anomaly II and III

Figure 10 depicts the total view maintenance cost in terms of seconds (depicted on the y-axis) for the different types of anomalies II and III. We find that the extra cost of aborting $M(DropSC)$ by another *DropSC* is more significant than any others. The reason is the complete abort of $M(DropSC)$, which is the most expensive maintenance process as observed in Section 7.1.

We also find that the abort cost of anomalies type II is small because the loss is just one maintenance query. All other maintenance efforts completed to that point can be kept since we can simply rewrite the query and

try again. Finally, since data update maintenance is the least costly process, even if it's completely aborted and redone again, the cost is still insignificant.

7.5 Mixed Update Processing

We now study how DyDa performs in an environment composed of a random mixture of both data updates and schema changes.

We employ a mixture of updates with three *DropSC*, three *RenameSC* and one hundred data updates over all six sources. We apply a worst case study here, i.e., there is no schema changes could be combined and no data updates could be discarded as described in Section 6.2.1. If so, the performance should be better than we will find here. In this experiment, we vary the time interval between the *DropSC*, *RenameSC* and data updates.

Figure 11 depicts the abort cost and the overall maintenance cost (which includes the abort cost) when only varying the time interval between *DropSC* from 0s to 45s. 0s means that all updates flood into the view manager before any maintenance kicks in. From Figure 11, we see that this case has the best performance. This is because the system is able to correct all *unsafe* dependencies at once for all updates, thus no anomalies type III would occur during maintenance processing. When the time interval between *DropSC* increases, the new *DropSC* could break the ongoing maintenance work, hence the cost increases. The cost reaches a high peak when the new *DropSC* always occurs near the end of the current $M(DropSC)$, resulting in the maximum abort cost. After the interval is larger than the maintenance time, there is no conflicts between the *DropSC*. Hence the cost significantly decreases.

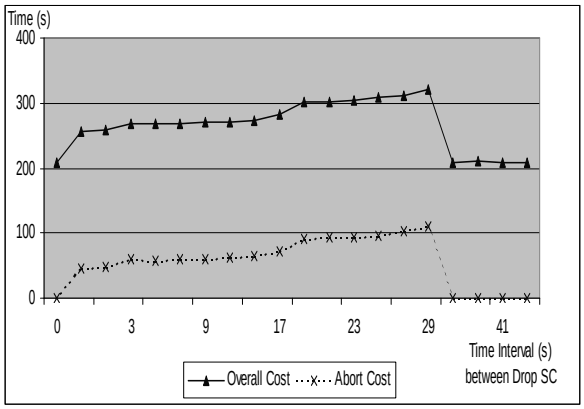

Figure 11: Varying Time Interval between DropSCs

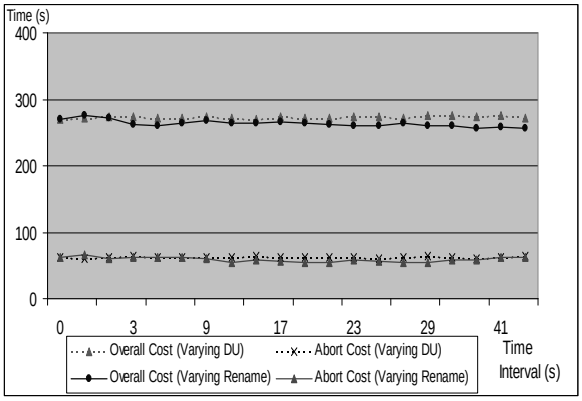

Figure 12: Varying Time Interval between DUs and RenameSCs

Next, we fix the time interval between the *DropSC* to 10s, but vary the time interval between data updates and *RenameSC*, respectively. Figure 12 depicts the abort cost and the overall maintenance cost for both cases. From the figure, we see that the system performance as well as the abort cost remain stable because the

concurrency between *DropSC* is not affected. Although the rate of the other two types of concurrency may vary, it seems not to affect the system performance much as observed in Section 7.3 and 7.4.

8 Related Work

Schema mapping [MBR01, MHH00] specifies how to map the data from one schema to another to achieve interoperability of heterogeneous data sources. A variety of modern applications requires schema mapping as foundations, such as data integration for heterogeneous sources, XML to relational mapping or semantic Web [LGMT03]. With the popular usage of WWW, the application environment becomes increasingly complex and dynamic. The data sources may change their schema, semantics as well as their query capabilities. In correspondence, the mapping or view definition must be maintained to keep consistent. In EVE [LNR02] system, the view definition evolves after the source schema changes. In [VMP03] the authors propose to incrementally adapts the schema mapping to the new source or target schema or constraints.

In a loosely-coupled environment, such as the Data Grid [JR03], the data sources may wish to contribute some of their sources but also want to keep autonomous such that they may commit update transactions without any concern of how those changes may affect the mapping or views defined upon them [ZGMHW95].

Materialized views, as proven to be a big success in the decision support applications, would continue to be effective in this scenario. Materialized views must be maintained when the source changes, which has been extensively studied in the past few years [AASY97, SBCL00, ZGMHW95, CGL$^+$96, GL95, LMSS95]. However, most of these work assume a static schema, which is no longer a valid assumption in the dynamic environment. While in [AASY97, SBCL00, ZGMHW95], the authors proposed *compensation-based* solutions to remove the effect of concurrent data updates from query results, these solutions would fail under source schema changes. [ZR02] assumes a fixed synchronization protocol between the view manager and data sources to resolve the concurrency problem. This restricts the autonomy of sources in that the sources have to wait before applying any schema change. Our proposed solution successfully drops this restricting assumption. [CCR02] employs a multiversion concurrency control algorithm to avoid anomalies assuming there are enough system resources to materialize auxiliary data.

9 Conclusions

To our knowledge, our work is the first to address the view maintenance anomaly problem under both concurrent source data updates and schema changes without placing any restrictions on the sources. We first identify three types of potential maintenance anomalies. We prove our compensation solution is able to handle both concurrent data and rename scheme changes. In the case of concurrent drop schema changes, we propose to remove such

anomaly by re-scheduling the maintenance order and use our new view adaptation algorithm to maintain the resulting complex update. With our proposed approach, the view manager is able to handle any type of anomalies. The data sources in our environment achieve complete autonomy in that they can commit either data updates or schema changes without coordinating with the view manager.

We have implemented our DyDa system. The experimental results show that our new concurrency handling strategy imposes a minimal overhead on normal data update processing while still allowing for the extended functionality to maintain the view even under concurrent schema change processing. We also extensively study the concurrency handling cost for different type of anomalies and the overall system performance in a mixed updates environment.

Acknowledgements We would like to thank Bin Liu, Jun Chen, Andreas Koeller in the Database Systems Research Group at WPI for discussions and helping implement the DyDa system. We are also grateful to Lucian Popa, Richard Sidle, Jun Rao and Latha Colby at IBM Almaden Research Center for the valuable suggestions.

References

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

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

[CCR02] J. Chen, S. Chen, and E. A. Rundensteiner. A Transactional Model to Data Warehouse Maintenance. In *International Conference on on Conceptual Modeling (ER)*, pages 247–262, 2002.

[CCZR04] S. Chen, J. Chen, X. Zhang, and E. A. Rundensteiner. Detection and Correction of Conflicting Source Updates for View Maintenance. In *Proceedings of IEEE International Conference on Data Engineering*, 2004.

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

[CZC+01] J. Chen, X. Zhang, S. Chen, K. Andreas, and E. A. Rundensteiner. DyDa: Data Warehouse Maintenance under Fully Concurrent Environments. In *Proceedings of SIGMOD, system demo*, page 619, 2001.

[GL95] T. Griffin and L. Libkin. Incremental Maintenance of Views with Duplicates. In *Proceedings of SIGMOD*, pages 328–339, 1995.

[GMRR01] A. Gupta, I. S. Mumick, J. Rao, and K. A. Ross. Adapting Materialized Views after Redefinitions: Techniques and a Performance Study. *Information Systems*, 2001.

[JR03] A. Jagatheensan and A. Rajasekar. Data Grid Management Systems. In *SIGMOD*, page 683, 2003.

[LGMT03] A. Y. Levy, Z. G.Ives, P. Mork, and I. Tatarinov. Piazza: Data management infrastructure for semantic web applications. In *WWW*, pages 556–567, 2003.

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

[LNR02] A. M. Lee, A. Nica, and E. A. Rundensteiner. The EVE Approach: View Synchronization in Dynamic Distributed Environments. *In IEEE TKDE*, 2002.

[MBR01] J. Madhaven, P. A. Bernstein, and E. Rahm. Generic Schema Matching with Cupid. In *International Conference on Very Large Data Bases*, pages 49–58, 2001.

[MHH00] R. J. Miller, L. M. Haas, and M. A. Hernández. Schema Mapping as Query Discovery. In *International Conference on Very Large Data Bases*, pages 77–88, 2000.

[NLR98] A. Nica, A. J. Lee, and E. A. Rundensteiner. The CVS Algorithm for View Synchronization in Evolvable Large-Scale Information Systems. In *Proceedings of EDBT*, pages 359–373, 1998.

[NR99] A. Nica and E. A. Rundensteiner. View Maintenance after View Synchronization. In *Proceedings of IDEAS*, pages 213–215, 1999.

[SBCL00] K. Salem, K. S. Beyer, R. Cochrane, and B. G. Lindsay. How To Roll a Join: Asynchronous Incremental View Maintenance. In *Proceedings of SIGMOD*, pages 129–140, 2000.

[Tar72] R. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM J. Computing*, 1(2), June 1972.

[VMP03] Y. Velegrakis, R. J. Miller, and L. Popa. Mapping Adaptation under Evolving Schemas. In *VLDB*, 2003.

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

[ZGMW96] Y. Zhuge, Héctor García-Molina, and J. L. Wiener. The Strobe Algorithms for Multi-Source Warehouse Consistency. In *PDIS*, pages 146–157, December 1996.

[ZR02] X. Zhang and E. A. Rundensteiner. Integrating the Maintenance and Synchronization of Data Warehouses Using a Cooperative Framework. *Information Systems*, 27(4):219–243, 2002.

[ZRD01] X. Zhang, E. A. Rundensteiner, and L. Ding. PVM: Parallel View Maintenance Under Concurrent Data Updates of Distributed Sources. In *Data Warehousing and Knowledge Discovery, Proceedings*, pages 104–113, 2001.