Detection and Correction of Conflicting Source Updates for View Maintenance *

Songting Chen, Jun Chen, Xin Zhang and Elke A. Rundensteiner
Worcester Polytechnic Institute, 100 Institute Road, Worcester, MA 01609
{chenst, junchen, 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 for better access, performance and high availability. Such views must be maintained after the data sources change. In a loosely-coupled environment, such as the Data Grid, the source updates are autonomous and may cause erroneous maintenance results. State-of-the-art maintenance strategies apply compensating queries to correct such errors. However, they assume that the source schema remain static. This is an unrealistic assumption for such dynamic environments, where the data sources may change not only their data but also their schema, query capabilities or semantics. Consequently, either the maintenance or compensating queries may fail. In this paper, first, we analyze the maintenance errors and classify them into different classes of dependencies. We then propose Dyno, a two-pronged strategy composed of dependency detection and correction algorithms to handle these new classes of concurrency. Our techniques are not tied to specific maintenance algorithms nor to a particular data model. To our knowledge, this is the first complete solution to the view maintenance concurrency problems for both data and schema changes. Our experimental results illustrate that Dyno imposes an almost negligible overhead on existing maintenance algorithms for data updates while now allowing for this extended functionality.

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 is thus a 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 [13, 14] to specify how the data of one schema is transformed to another. A view query is one way to specify such a mapping. Schema mappings are used extensively in a variety of applications, such as data integration, physical database design like XML to relational mapping, or the semantic Web [10].

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 be kept consistent [9, 17]. Moreover, in a loosely-coupled environment, such as the Data Grid [8], 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 mappings or views defined upon them. Such autonomous source schema restructuring poses new challenges for data integration.

Materialized views (MV), proven to be an excellent technique in decision support applications, continue to be useful in this scenario to preserve the integrated data to ensure better access, performance and high availability. Materialized views must be maintained when the sources change. This has been extensively studied in the past few years [1, 15, 20]. However, it is not sufficiently explored in this new dynamic environment. As we will illustrate via examples in Section 1.2, when maintaining a source update, we may need to query the data sources for more information by issuing *maintenance queries* [20]. However, in these new autonomous and dynamic environments, such queries may either return erroneous 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 [1, 20] has proposed *compensation-based* solutions to remove the effect of concurrent data up-

* This work was supported by several grants from NSF, namely, NSF NYI grant #IRI 97–96264, NSF CISE Instrumentation grant #IRIS 97–29878, and NSF grant #IIS 9988776.

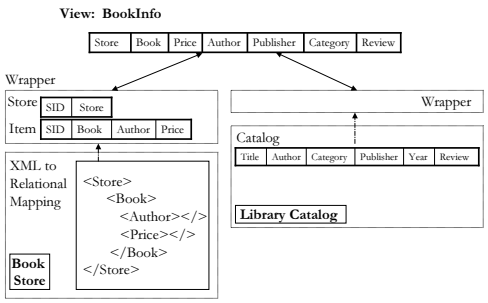

Figure 1. View and Source Description

dates from query results, we demonstrate that these existing solutions 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 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. Examples of Maintenance Anomalies

Example 1 *Assume we want to integrate book data from the Retailer and Library category to provide the user the sales as well as the detailed book information (Figure 1). The 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 wrapper. The integrated view BookInfo from both data sources is defined by Query (1).*

Assume a new book is inserted into the Library catalog. This new book is extracted by the wrapper as "ΔC = ('Data Integration Guide', 'Adams', 'Engineering', 'Princeton' ...)". To determine its delta effect on the view, an incremental maintenance query (Query (2)) [20] will be generated by decomposing the view query (1) into individual source queries. Two different anomalies can be distinguished:

```
CREATE VIEW   BookInfo AS
SELECT        Store, Book, I.Author, Price, Publisher, Category, Review
FROM          Store S, Item I, Catalog C
WHERE         S.SID = I.SID
              AND I.Book = C.Title
```
(1)

```
SELECT   Store, Book, Author, Price
FROM     Store S, Item I
WHERE    Book = 'Data Integration Guide'
         AND S.SID = I.SID
```
(2)

(a) **Duplication Anomaly***: Assume that before answering query (2), the Item table committed a data update ΔI = insert (10, 'Data Integration Guide', 'Adams', 35.99). 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 ΔI, the same tuple would be inserted into the view again. A duplication anomaly occurs due to concurrent data updates [20].*

(b) **Broken Query Anomaly***: Now assuming the designer may tune the XML-to-Relational mapping which results in one single table StoreItems in Figure 2. Query (2) then faces a schema conflict and cannot succeed since both Store or Item relations are no longer available.*

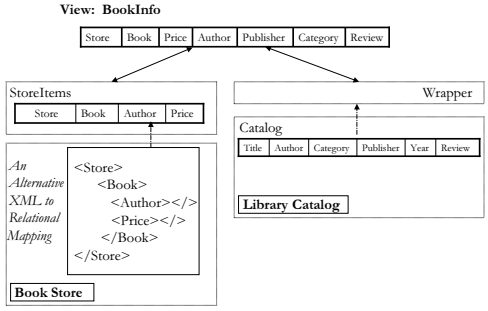

Figure 2. Change of Store Schema

This illustrates that the known view maintenance solutions are not sufficient when the sources undergo schema changes that invalidate the view definition. The reason is that the maintenance queries are constructed by outdated schema knowledge and would not return any query results.

1.3. Our Contributions

In this paper, we propose a solution, called Dyno, capable of dealing with the new types of maintenance conflicts under both source data and schema changes. Dyno enables data sources to autonomously commit all types of updates. This provides great flexibility for data integration in loosely-coupled environments, such as the Data Grid [8]. To our knowledge, Dyno is the first complete solution to the view maintenance anomaly problems. In summary, the contributions of this work are:

(1) We identify that the maintenance anomaly problem is caused by the violation of dependencies between source updates. We formally characterize and classify these anomalies as dependencies.

(2) We propose several dependency detection strategies to find the view maintenance anomalies at different stages and experimentally evaluate their trade-offs.

(3) We propose dependency correction algorithm to adjust the violated dependencies and consequently eliminate the view maintenance anomalies.

(4) We design an overall framework called *Dyno* (DYNamic reOrdering) which integrates both dependency detection and correction algorithms to ensure correct view maintenance. While we use the relational model in our implementation, our techniques are general and independent of any data model.

(5) We have implemented the *Dyno* solution in our DyDa [3] prototype system. Our experimental results confirm that *Dyno* imposes an almost negligible overhead on data update processing while now offering comprehensive support for concurrency handling.

The paper is organized as follows. Section 2 introduces our maintenance framework. Section 3 provides a formalism of dependencies. Section 4 presents the complete Dyno solution. We sketch a view adaptation algorithm for support of Dyno in Section 5. The experimental study is discussed in Section 6. Section 7 reviews related work, while Section 8 concludes the paper.

2. View Management Framework

To set the context for our solution, we first introduce the proposed view management framework in a loosely-coupled environment (Figure 3). The remote source space is composed of a large number of heterogeneous and autonomous data sources. These sources join, leave, or change their capabilities dynamically. A data source is "integrated" into our framework via a wrapper that serves as a bridge between the source space and the view manager. The wrapper is assumed to be intelligent so that it extracts not only raw data, but also metadata information, such as changes at the schema level or relationships with other sources. The view manager maintains the materialized view under these source updates.

The view manager must accomplish three view maintenance tasks, namely, View Maintenance (VM), View Synchronization (VS) and View Adaptation (VA). A well studied process, VM [1, 15, 20] maintains the view extent under source data updates. VS [9] aims at evolving the view definition when the source schema has been changed. Note that a general-purpose mapping evolution module [17] could also implement this function. Thereafter, View Adaptation (VA) [7] incrementally adapts the view extent so that it again matches the newly changed view definition. Our view manager incorporates these general view management algorithms. The Update Message Queue (UMQ) buffers the updates from the data sources. The Query Engine executes the maintenance queries generated by VM, VS or VA and assembles the query results into one final result to be integrated into the view.

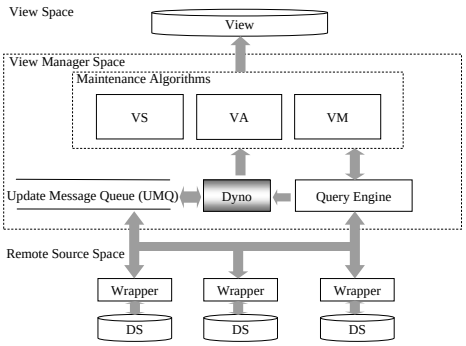

Figure 3. The View Management Framework

Most maintenance systems usually process the source updates simply based on the order they arrive at the view manager [1, 15, 20, 9]. This however might introduce maintenance anomaly problems as illustrated in Section 1.2. In this paper, we propose a new strategy, called Dyno, to detect concurrent updates and correct them by *re-scheduling* their maintenance order. Dyno is a general concurrency control strategy independent of the choices of the particular VM/VS/VA algorithms or the underlying data models. In this paper, we will work with views defined using SQL to keep the description simple.

3. Classes of Dependency Relationships

3.1. Types of Maintenance Anomaly

Definition 1 *We first define the view maintenance processes based on the types of source updates.*

(1) Given a source data update (DU), a view maintenance (VM) algorithm first reads the view definition, decomposes the view query into individual source queries, then probes each source and finally refreshes the view. Without loss of generality, we define this maintenance process 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 source query, w(MV) and c(MV) are write and commit of the materialized view MV, respectively.*

(2) Given a source schema change (SC), a view synchronization (VS) or mapping evolution algorithm rewrites the view definition to become well defined over the changed data sources. Then view adaptation (VA) adapts the view extent to again become consistent

with the new view definition. We denote this maintenance process as "$M(SC) = r(VD)w(VD)r(DS_1)$ $r(DS_2)...\ r(DS_n)w(MV)c(MV)$*"* [1].

Definition 2 *Assume one update* $w(DS_i)$ *at source* DS_i*, one update* $w(DS_j)$ *at source* DS_j *and a view* V *defined upon both sources. The maintenance of* V *for 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. *In particular, if w(*DS_j*) is a schema change, then we say the query r(*DS_j*) is* broken *and call this a* broken query anomaly.

Note that a *broken query anomaly* may not always cause the query fail. E.g., a schema change modifies only the attributes not included in the query. Based on the type of source update and maintenance process, there are four types of anomalies:

(1) A data update DU_1 conflicts with $M(DU_2)$;
(2) A data update DU_1 conflicts with $M(SC_1)$;
(3) A schema change SC_1 conflicts with $M(DU_1)$;
(4) A schema change SC_1 conflicts with $M(SC_2)$.

The two anomalies in Examples 1.a and 1.b are of type (1) and (3), respectively. The concurrent data updates, namely, anomalies of type (1) and (2), modify the sources' *extent* and thus may cause erroneous maintenance query results. Concurrent schema changes, namely, anomalies of type (3) and (4), modify the sources' *schema* which may no longer match the schema specified in the maintenance query. The latter two types are broken query anomalies.

Unlike traditional read-write conflicts in transaction theory [2], the write (source update) in a loosely-coupled environment is autonomous and out-of-the-control of the view manager. Hence a locking mechanism is not an appropriate solution in this environment. Prior efforts [1, 15, 20] propose special-purpose compensation algorithms to remove the erroneous tuples from the query result. However they only address the anomaly of type (1). We notice that the anomaly of type (2) is similar to type (1), i.e., they are both caused by concurrent data updates. Hence any compensation algorithms over distributed sources [1, 19, 11], can be applied to solve these two anomalies by correcting any errors from the maintenance query result. We omit the review of these known algorithms. Instead, we focus on addressing the as of now unsolved broken query anomalies, i.e., type (3) and (4).

1 Note that the rewritten view is not restricted to be equivalent to the original one [9, 17], making the adaptation step necessary. Second, 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 rewrite of view definition, e.g., updating system catalog, is done in the w(MV).

3.2. Concurrent Dependency

The anomalies of type (3) and (4) are caused by source schema changes. Compensation algorithms [20, 1, 15] do not work since the maintenance query may have no query result returned at all, as illustrated by Example 1.b. We observe that although such schema conflicts occur during query processing at the sources, the cause are the read-write conflicts of the *view definition*.

Assume a data update $w_{DU}(DS_i)$ at source DS_i and a schema change $w_{SC}(DS_j)$ at DS_j, respectively. Their maintenance processes are "$M(w_{DU}(DS_i)) = r_1(VD)r_1(DS_1)r_1(DS_2)...r_1(DS_n)w_1(MV)c_1(MV)$" and "$M(w_{SC}(DS_j)) = r_2(VD)w_2(VD)r_2(DS_1)r_2(DS_2)$ $...r_2(DS_n)w_2(MV)c_2(MV)$" by Definition 1. By Definition 2, the read $r_1(DS_j)$ of the maintenance process $M(w_{DU}(DS_i))$ may conflict with the schema change $w_{SC}(DS_j)$.

Notice that there is also a read-write conflict on the view definition between these two maintenance processes, i.e., $r_1(VD)/w_2(VD)$. Interestingly, this conflict on the view definition is the reason for the conflict between $r_1(DS_j)$ and $w_{SC}(DS_j)$. The rationale is that the query $r_1(DS_j)$ has been constructed based on the read of the view definition $r_1(VD)$. For instance, in Example 1.b, the maintenance query (2) is constructed based on the view definition query (1) over $Store$ and $Item$. If this view definition read $r_1(VD)$ conflicts with $w_2(VD)$, the constructed query $r_1(DS_j)$ may no longer reflect the actual schema of DS_j. Based on the observations above, we define a *concurrent dependency* between maintenance processes.

Definition 3 *Let* $w(DS_i)$ *and* $w(DS_j)$ *denote two updates committed on data sources* DS_i *and* DS_j*. 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 the 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 VD while* $M(w(DS_j))$ *contains write VD.*

Concurrency dependency defines the relationship between maintenance processes over the critical resource, view definition. Unlike broken query anomaly, it is independent of how and when the maintenance queries are processed. Their relationship is explained in Section 3.4. There are also several differences between the *concurrent dependencies* and *wait-for* dependencies in traditional transactions [2]. 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.

3.3. Semantic Dependency

A materialized view is consistent if it reflects some valid state of each data source. A valid state of DS_i, denoted as $\sigma(DS_i)$, describes both its data content and metadata. Assume the state of DS_i evolves as $\sigma(DS_i) \xrightarrow{\Delta^1} \sigma^1(DS_i) \xrightarrow{\Delta^2} \sigma^2(DS_i)$. It is important for MV to be refreshed by Δ^1 before Δ^2 in that order. If we maintain 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 $\sigma^1(DS_i)$ nor $\sigma^2(DS_i)$. In this case, the *Strong consistency* [20] that MV reflects the valid state of data sources in the same order cannot be achieved. Furthermore, the MV consistency is not even *converging*, i.e., the final state may be invalid too. For instance, in Example 1, assume there is a data update Δ^1 = insert into Item (10, 'Data Integration Guide', 'Adams', 35.99) followed by a delete Δ^2 = delete from Item (10, 'Data Integration Guide', 'Adams', 35.99). If we reverse the maintenance order, we cannot find the tuple to delete. We say that the maintenance $M(\Delta^2)$ is *semantic dependent* on $M(\Delta^1)$. More formally,

Definition 4 *Let ΔX and ΔY denote two updates at source DS_i, then maintenance $M(\Delta X)$ is semantic dependent **(SD)** on $M(\Delta Y)$, denoted by $M(\Delta X) \xleftarrow{sd} M(\Delta Y)$, iff ΔY is committed before ΔX.*

Note that if there are other types of constraints between maintenance processes, they can be treated similarly.

3.4. Dependency Properties

The two types of dependencies share an important property, namely, both represent constraints on the processing order between maintenance.

Definition 5 *For two updates ΔX and ΔY, we say the maintenance M(ΔX) is dependent on M(ΔY), denoted by M(ΔX)←M(ΔY), if M(ΔX) is concurrent dependent on M(ΔY) by Definition 3 or M(ΔX) is semantic dependent on M(ΔY) by Definition 4.*

Notice that transitivity holds, i.e., if M(ΔX)←M(ΔY) and M(ΔY)←M(ΔZ), then M(ΔX)←M(ΔZ). Furthermore if M(ΔX) is dependent on M(ΔY), then the maintenance M(ΔY) must be processed *before* M(ΔX). For a semantic dependency, such required order is obvious as discussed in Section 3.3. For a concurrent dependency, as shown in Section 3.2, the write view definition operation has to be done *first* to solve the read-write conflict on the view definition. The concurrent schema change invalidates the view definition, hence rewriting it becomes critical.

Definition 6 *Given two updates ΔX and ΔY in the Update Message Queue (UMQ). We denote "pos(ΔX, UMQ) $\prec$ pos(ΔY, UMQ)" if ΔX precedes ΔY in the UMQ. This presents their sequential maintenance order. We define the **dependency relationship** between maintenance processes M(ΔX) and M(ΔY) to be:*

1. ***independent** iff there is no dependency between ΔX and ΔY by Definition 5.*
2. ***safe dependent** iff pos(ΔX, UMQ) $\prec$ pos(ΔY, UMQ) and all dependencies between M(ΔX) and M(ΔY) by Definition 5 are M(ΔY)←M(ΔX).*
3. ***unsafe dependent** iff pos(ΔX, UMQ) $\prec$ pos(ΔY, UMQ) and there is at least one dependency M(ΔX)←M(ΔY).*

For example, the concurrent dependency of the broken query anomaly in Example 1.b is M(DU)$\xleftarrow{cd}$M(SC). Since pos(DU, UMQ) $\prec$ pos(SC, UMQ), this dependency is *unsafe* by Definition 6.

Theorem 1 *A maintenance query $r(DS_i)$ of a maintenance process M(ΔX) is broken only if at least one* unsafe *dependency M(ΔX)←M(ΔY) exists, where ΔY is from DS_i.*

A broken query implies an unsafe dependency, but not vice versa. The proof of Theorem 1 can be found in [4]. Hence if we find a processing order that makes all dependencies safe, then no broken query anomaly would occur.

3.5. 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 [2]. Given the source setting from Example 1, let us refer to the XML document change in Example 1.b as SC_1. Now assume the $Review$ attribute of the library $Catalog$ table is no longer considered important and dropped. We refer this change as SC_2.

Assume these schema changes SC_1 and SC_2 have already been committed at their data sources. If we process SC_1 first, the view definition is rewritten into Query (3). However, this new view definition is no longer consistent with the sources since the attribute *Review* is no longer available due to SC_2. Similarly, if we process SC_2 first, the view definition may be rewritten into Query (4) by locating some alternative data source *ReaderDigest* for replacement [9]. Again, the view definition is not valid since the tables *Store* and *Item* are no longer available due to SC_1. Hence the maintenance query constructed based on either of these two view definitions would fail.

```
CREATE VIEW   BookInfo' AS
SELECT        Store, Book, S.Author, Price, Publisher,
              Category, Review                          (3)
FROM          StoreItems S, Catalog C
WHERE         S.Book = C.Title

CREATE VIEW   BookInfo' AS
SELECT        Store, Book, S.Author, Price, Publisher,
              Category, R.Comments as Review
FROM          Store S,Item I, Catalog C, ReaderDigest R  (4)
WHERE         S.SID = I.SID
              AND I.Book = C.Title
              AND C.Title = R.Article
```

Each of these two maintenance processes involves a read and a write of the view definition. Hence there are dependencies M(SC_1)←M(SC_2) and M(SC_2)←M(SC_1) by Definition 3. This comprises a cycle. Intuitively, such cyclic dependencies may result in a deadlock in the sense that we have maintenance processes waiting for each other. To handle such cyclic dependencies, aborting some of the maintenance processes 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. We have to develop some alternative solution to address this maintenance deadlock problem as will described in Section 4.

Definition 7 *Given a set of updates, any order of maintenance is called a* ***legal order*** *if all* dependencies *are* safe *by Definition 6.*

Having all *dependencies* safe, by Theorem 1, means no broken query anomalies would occur. Definition 7 thus establishes the correctness criterion for solving the *broken query anomalies*.

4. Dyno: A Dynamic Scheduler

4.1. Detection of Unsafe Dependencies

4.1.1. Dependency Graph. Our dependency detection algorithm has two steps. One, we construct a *dependency graph*, where each node represents a maintenance process for each update in the UMQ and the directed edges are either *concurrent dependencies* or *semantic dependencies* between these processes. Two, we check if there are any *unsafe dependencies* in the graph as defined by Definition 6.

Given a sequence of updates, we determine the *concurrent dependency* between the maintenance of two updates M(ΔX) and M(ΔY) as follows. If ΔY is a schema change and modifies any metadata, such as attribute or relation, that is included in the view query, then we draw a *concurrent dependency* edge, namely M(ΔX) $\xleftarrow{cd}$ M(ΔY) by Definition 3. The reason is that M(ΔX) will read the view definition which has been invalidated by the update ΔY. It is also straightforward to identify *semantic dependencies*. That is, each pair of maintenance processes of two adjacent updates to the same relation is assigned a *semantic dependency* edge.

We now examine the complexity of building such a *dependency graph*. First, the complexity of identifying *concurrent dependencies* between maintenance processes is O(mn), where m is the number of schema changes and n is the number of updates. The reason is that each *concurrent dependency* involves at least one schema change. In the worst case, one schema change would have one *concurrent dependency* with 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 complexity of building a *dependency graph* is $O(mn)$ + $O(n)$, i.e., $O(mn)$.

If there are only data updates, then there is no concurrent dependency. Since all the semantic dependencies are safe initially, we can optimize the detection by setting a *schema change flag* indicating whether any schema changes have been logged into the UMQ. If only data updates have occurred thus far, we can avoid building the *dependency graph* altogether - thus reducing the complexity to O(1).

4.1.2. Timing of Detection Execution. The detection algorithm in Section 4.1.1 can have different modes in terms of when it is executed, namely, *pre-exec detection* and *in-exec detection* modes. The *pre-exec static detection mode* detects the unsafe dependency *before* the maintenance begins in order to avoid any potential conflicts. For instance, for the broken query anomaly case in Example 1.b, before the view manager processes the insert, it discovers that the drop *Store* operation has already been buffered into the UMQ which forms an *unsafe* concurrent dependency with the insert. Thus we need not attempt to maintain the insert now by sending down a maintenance query (2) to probe the tables *Store* and *Item*. This query will surely be broken. Our goal is to avoid such computation waste, which we call *abort cost*, that would have to be discarded later. On the other hand, the *in-exec dynamic detection mode* detects any *unsafe* dependencies *during* the maintenance using the *broken query scheme*. That is, whenever a query fails, the Query Engine (in Figure 3) is responsible to detect such failure. Then there must exist an *unsafe* dependency by Theorem 1.

4.1.3. Pessimistic vs. Optimistic Strategies. We designed two complete detection strategies in a dynamic context based on the detection modes discussed in Section 4.1.2, namely, optimistic and pessimistic strategy. Both produce correct final maintenance results. The choice of strategy is largely based on the frequency and types of concurrency among these updates.

1. **Pessimistic detection strategy**: A pessimistic solution aims to anticipate and ideally prevent any bro-

ken query anomaly and its overhead during maintenance. For this, we utilize the detection algorithm from Section 4.1.1 in a *pre-exec detection* mode to detect all unsafe dependencies that may occur *before* maintenance starts, hence named *pessimistic*. The pre-exec detection by itself is *not* sufficient, because a schema change that occurs after the pre-exec detection phase but while the maintenance process is ongoing could still break that maintenance process. Hence, we need to also employ the *in-exec detection* mode as supplementary method to assure correctness.

2. **Optimistic detection strategy**: An optimistic solution aims to avoid any regular detection overhead during maintenance. To achieve this, we just employ the detection algorithm in the *in-exec detection* mode. Compared to previous pessimistic detection, it saves the *pre-exec detection* cost. However, it cannot prevent any broken query anomaly that could have been avoided by pessimistic detection and hence may endure more *abort costs*. Our experimental study comparing these two strategies is described in Section 6.4.

4.2. Static 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*. This strategy assumes a fixed number of updates, hence named static correction. We will extend this to the dynamic context in Section 4.3.

Theorem 2 *Given a set of updates, if the dependency graph is acyclic, a legal order of maintenance exists.*

Theorem 2 holds since given an acyclic dependency graph, we can apply *topological sort* algorithm [16] to obtain a partial order of nodes. The time complexity is O(n+e), where n is the number of nodes (updates) and e is the number of edges (dependencies). Thus 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 3.5, the *topological sort* algorithm cannot generate a partial order. For this, we first identify all cycles in the dependency graph (similar to identifying *strong connected components* [16]) with time complexity also O(n+e). Traditional transaction processing [2] breaks the cycle (or deadlock) by removing one node in the cycle, in other words, aborting one transaction. However, this strategy is not appropriate here because the source updates are autonomous and already committed at the sources. Instead of removing one node, we propose to *merge* these nodes (updates) into a *merged* one to be processed at one time. The intuition of merge is that since we cannot process these updates separately, we instead process them in one atomic batch. This requires a view adaptation algorithm capable of processing such combined batches of updates (see Section 5). After removing all cycles, we can apply topological sort to the now acyclic dependency graph to obtain a legal order.

An alternative simplistic solution may be to merge all the updates whenever there is a broken query anomaly. We argue that this is not a good solution. First, more intermediate MV states would be missing due to such a blind merge. Second, since it is more costly to process a big update than a small one, correspondingly, the *abort cost* of such a large maintenance will also increase and in fact it would be more likely to occur due to the extended duration of the maintenance process. In comparison, our solution maintains the updates in the smallest possible granularity and refreshes the view as quickly as possible.

Assume in the view (1) of Example 1, three updates occur, namely, one data update DU_1 (ΔC in Example 1.(a), then two schema changes (SC_1 and SC_2 in Section 3.5), in that order. The initial dependency graph is shown on the left part of Figure 4. 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 4 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 (see Section 5). Figure 5 shows a more complex example of dependency correction.

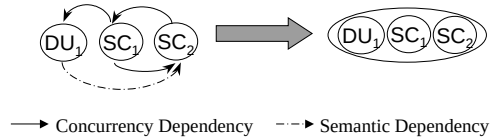

Figure 4. Unsafe Dependency Correction for View (1)

4.3. Dyno Solution: Pulling It All Together

The correction algorithm in Section 4.2 assumes a static *dependency graph*. We now extend this algorithm into a dynamic context by taking newly arriving updates into consideration. Our proposed solution Dyno combines the *pessimistic detection strategy* in Section 4.1.3 and the *static dependency correction* algorithm in Section 4.2. We will explain this decision experimentally in Section 6.

Figure 6 details the *Dyno* algorithm. Dyno checks the schema change flag (see Section 4.1.1) in line 1 before the

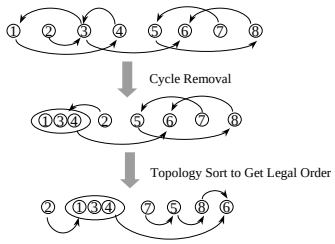

Figure 5. Complex Example of Dependency Correction

maintenance processing. If no *new* schema change has occurred after the *last* correction, we can avoid the detection and correction steps because the newly arriving data updates will not introduce any additional *unsafe* dependencies [2]. If a schema change did occur, Dyno will construct the *dependency graph* (line 4) and correct all *unsafe* dependencies (line 5) in the UMQ. After that, Dyno starts to maintain the update (line 9). If no broken query occurs during maintenance, then the head update will be removed and Dyno proceeds to process the next update. If a broken query did occur, Dyno kicks off the correction algorithms in lines 1-5 to obtain a new *legal order*. Thereafter, Dyno processes the head update in the UMQ in line 9.

Figure 7 presents some maintenance-related background processes. The *UMQ_Manager()* enqueues any newly arriving update and sets the schema change flag. *Query_Engine()* will be called by the *View_Maintenance()* during the maintenance. Once a maintenance query is *broken*, it will be detected by the *in-exec detection* mechanism and *View_Maintenance()* will be aborted.

4.4. Termination and Correctness of Dyno

Correctness of Dyno. We now argue that Dyno can always obtain a legal order. A new schema change ΔX that conflicts with the current maintenance M(ΔY) may occur in three different periods. First, it may happen before the current maintenance process and thus can be detected by the pre-exec detection. The conflicts will then be corrected by obtaining a new legal order. Or it may occur during the ongoing maintenance and cause a broken query. *Dyno* would detect it by the in-exec detection strategy and again correct it. In either case, the view manager will have a legal order of updates and no broken query anomaly would occur by Section 4.2. Finally, if ΔX occurs after the maintenance commits, then ΔX does not conflict with the maintenance process M(ΔY). See [4] for the proof that our proposed solution achieves *strong consistency* [20].

2 Hence this flag not only helps in a data update only environment as in Section 4.1.1, but also helps in a mixed update context to reduce the number of executions of dependency detection.

```
GLOBAL DATA
UMQ: Queue;  /* Update Message Queue */
NewSchemaChangeFlag = FALSE: Boolean; /* Flag for new schema change, set true by UMQ_Manager */
BrokenQueryFlag = FALSE: Boolean; /* Broken query flag, set true by Query_Engine */

PROCESS Dyno()
VAR X: UpdateMessage;
BEGIN
      LOOP (FOREVER)
          // Pessimistic detection strategy: pre-exec detection
1:        IF Test_If_True_Set_False (NewSchemaChangeFlag) THEN
2:            // If no new SC, we can avoid detection because new DUs will not introduce unsafe dependencies
3:            // otherwise, we atomically test&set the flag and start detection and correction algorithms.
4:            UMQ.Build_Dependency_Graph(); // Build dependency graph as in Section 5.1.1
5:            UMQ.Topological_Sort_with Cycle_Merge();  // Static Correction as in Section 5.1.2
6:        ENDIF
7:        X = UMQ.GetHead();
8:        BrokenQueryFlag = FALSE;
9:        View_Maintenance(X);  // Start maintenance of update X and commit to the view if succeed.
10:           // It will call Query_Engine() to process query. If broken query occurs, the maintenance will abort.
11:       IF BrokenQueryFlag = FALSE THEN // If no broken query occurs during View_Maintenance();
12:           UMQ.RemoveHead();
              // If broken query occurs, it will be corrected in the next loop;
      ENDLOOP
END
```

Figure 6. Dyno: DYNmaic reOrdering Algorithm

```
PROCESS UMQ_Manager()
VAR X: UpdateMessage;
BEGIN
       LOOP (FOREVER)
1:     UMQ.Receive_Update(X); /* receive and enqueue the updates from sources*/
2:     IF X.type = SC  THEN Set_True(NewSchemaChangeFlag); /* if a new SC arrives */
       ENDLOOP
END

PROCESS Query_Engine(Query Q)
VAR Success: Boolean;
      QR: QueryResult;
BEGIN
1:    Success = ExecuteQuery(Q, &QR); /* Execute Query, in-exec detection of broken query */
2:    IF Success = TRUE THEN return (QR);
3:    ELSE
4:        BrokenQueryFlag = TRUE; /* Broken query occurs */
5:        return (NULL);
END
```

Figure 7. Background Processes

Termination of Dyno. The only possibility that may cause *Dyno* to loop infinitely without any view refreshs is that continuously new schema changes escape the *pre-exec detection* and always break the ongoing maintenance. However, such case is unlikely, because for a view never to be refreshed by any update, it would require (1) a frequent and continuing stream of source schema changes, and (2) the schema change to always arrive after the *pre-exec detection* phase. We experimentally study this in Section 6.4.

5. Processing of Merged Updates

Part of our correction strategy as outlined in Section 4.2 is that if there is any cycle in the dependency graph, some updates will be *merged* into a batch node. Such batch node

would contain both schema and data updates from different data sources. We have developed an algorithm capable of maintaining such view update batches [4]. While this algorithm and details are beyond the scope of this paper, we briefly sketch its intuition below.

The first step is to preprocess the updates. We first partition these updates into *n* groups for each DS_i. Then we further partition the updates from the same source DS_i into two subgroups, namely, the data update subgroup $\{DU_i\}$ and the schema change subgroup $\{SC_i\}$. We combine schema changes in $\{SC_i\}$. For example, the update sequence "rename A to B" then "rename B to C" can be combined to "rename A to C". The data updates in $\{DU_i\}$ may be schema inconsistent if there are schema changes between them. For example, given "insert (3,4)", "drop first attribute", "insert (5)", we have two data updates with different schemata. We solve that by projecting only the attributes in the rewritten view definition. Now we have "insert (4),(5)", which are homogeneous update tuples that can be merged.

After preprocessing, we rewrite the view definition based on the combined schema changes. A general view synchronization [9] algorithm can accomplish this task. As a result, the old view definition $V = R_1 \bowtie R_2 \bowtie ... \bowtie R_n$ is rewritten into some (possibly not equivalent) $V' = R_1^{new} \bowtie R_2^{new} \bowtie ... \bowtie R_n^{new}$, where R_i^{new} represents either the new state of relation R_i after one or more updates or a replaced relation if R_i has been dropped. For the cyclic dependency example in Section 3.5, the view rewriting will take both schema changes into consideration. The final view definition is given in Query (5).

```
CREATE VIEW     BookInfo' AS
SELECT          Store, Book, S.Author, Price, Publisher,
                Category, R.Comments as Review
FROM            StoreItems S, Catalog C, ReaderDigest R
WHERE           S.Book = C.Title
                AND C.Title = R.Article
```
(5)

The final step is to incrementally adapt the view extent to be consistent with the new view definition. Here we represent the new view as $V' = R_1^{new} \bowtie R_2^{new} \bowtie ... \bowtie R_n^{new} = (R_1 + \Delta R_1) \bowtie (R_2 + \Delta R_n) \bowtie ... \bowtie (R_n + \Delta R_n)$, where ΔR_i stands for data updates or the difference between the old and the replaced relation. With $R_i, R_i^{new}, \Delta R_i$, we are able to calculate ΔV using Equation 6 and adapt the view.

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

6. Experimental Evaluation

6.1. Experimental Testbed

We have implemented the *Dyno* algorithm and embedded it into the DyDa [3] system. DyDa integrates VM, VS and VA algorithms in order to maintain the materialized views under both data and schema changes. In DyDa, we apply SWEEP [1] algorithm to compensate for concurrent data updates to solve the anomalies (1) and (2). With the integration of Dyno, the DyDa system solves the anomalies (3) and (4) and is now capable of handling any concurrent data or schema changes. The prototype system is implemented in JAVA, using JDBC to connect to view and source servers. In our experimental setting, there are six relations evenly distributed over three different source servers with two relations each. Each relation has four attributes and contains 100,000 tuples. The materialized view resides on a fourth view server. The view is defined as a one-to-one join among six relations and includes all twenty four attributes. All experiments are conducted on four Pentium III PCs with 256MB memory each, running Windows NT and Oracle8i.

6.2. Study of Data Update Processing

We first study *Dyno*'s overhead for data update processing. Clearly, any extra cost would be caused by the overhead of detection. Since broken queries will not occur without the presence of schema changes, we can avoid the construction of a *dependency graph* during the *pre-exec detection* (Section 4.1.1). This reduces the time complexity to O(1). The *in-exec detection strategy* are never launched since aborts would never be caused by data updates.

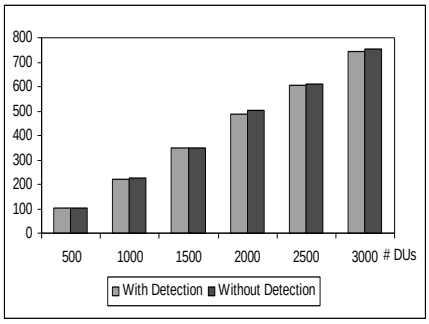

Figure 8. DU Processing and Detection

Figure 8 depicts the total view maintenance cost measured in seconds (depicted on y-axis) with or without detection enabled for different numbers of source data updates (depicted on x-axis). The overhead of detection is almost unobservable in all cases. Since the detection cost is trivial,

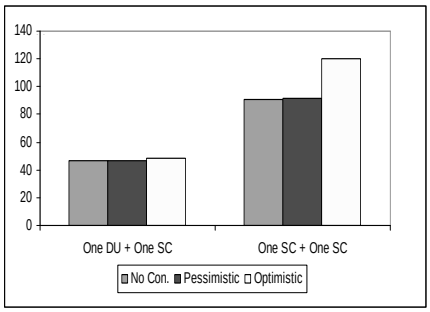

Figure 9. Cost of Broken Query

i.e., O(1), we predict that such cost would remain small under a larger number of data updates. We conclude that *Dyno* imposes little extra cost on data update processing.

6.3. Cost of Broken Query

The broken query anomaly is caused by the existence of concurrent schema changes. Once a query is *broken*, the view manager has to drop previous maintenance work [3] and redo it. This imposes wasted effort on view maintenance, which we call *abort cost*.

In this experiment, we study the *abort cost* under different types of anomalies, namely, type (3) and (4) as defined in Section 3.1. Two workloads have been chosen. The first is one data update followed by one conflicting drop attribute schema change. The second is one drop attribute schema change followed by a conflicting rename relation schema change. In both cases, the second schema change may cause the maintenance query of the first update to fail. Here three different environmental settings are compared. First, we measure the maintenance cost [4] of all updates by spacing them far apart, so they won't interfere with each other since each next update occurs after the completion of the previous maintenance. This represents the minimum cost as no concurrency handling is needed (grey bar in Figure 9). Second, we apply the *pessimistic strategy* to discover any potential concurrency conflicts before processing. This tries to avoid the occurrence of any broken query (black bar in Figure 9). Third, we apply the *optimistic strategy* (depicted by a white bar in Figure 9). Thus only after the broken query occurs and is detected, do we resolve the conflicts and restart the maintenance. Clearly, more aborts may occur.

In Figure 9 the cost of aborting schema change processing is significant compared to that of data update processing. That is, the white bar of "one SC + one SC" (where the abort of the schema change maintenance occurs) is much higher than the other two. The reason is that the schema change processing is time consuming compared to data update processing. It is thus costly to redo the schema change maintenance process. Secondly, we find that the *pessimistic strategy* does indeed help to reduce the expensive abort. Based on the observations in Section 6.2 and 6.3, we have chosen to employ the pessimistic strategy in Dyno.

6.4. Mixed Update Processing

We have observed that the most expensive extra cost is the abort of schema change processing. Using the focused experimental scenario in Figure 9, we were able to determine that the pessimistic strategy does help to reduce this expensive abort cost. However, a broken query may still occur even when employing the *pessimistic strategy* when the newly incoming update breaks the ongoing maintenance work. We now study under what conditions the broken query would occur and to what degree the pessimistic strategy helps to avoid this in mixed update environments.

6.4.1. Effects of Schema Changes on Abort. We study the effects of schema changes on the *abort cost*, in particular, the time interval between the schema changes and the number of schema changes. First, we employ a mixture of 200 data updates and one drop attribute operation and nine rename relation operations, both randomly generated over all six relations. The schema changes may abort the ongoing maintenance processes. In this experiment, we vary the time interval between two adjacent schema changes.

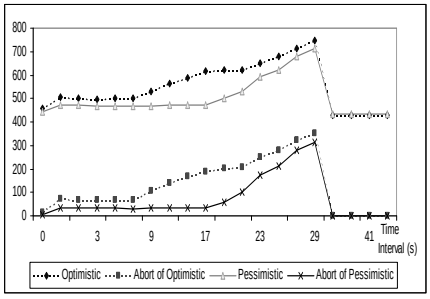

Figure 10. Time Interval of Schema Changes

Figure 10 depicts the maintenance costs for the optimistic and pessimistic strategies and their respective abort costs, when varying the delay between the schema changes from 0s to 45s (as depicted on X-axis). 0s means that all schema changes flood into the view manager before any maintenance kicks in. From Figure 10, we see that this case has the best performance for both strategies. This is because the system is able to correct all *unsafe* dependencies at once for

3 This abort is just to discard any temporary query results.

4 The maintenance cost includes the *abort cost* throughout our experiments.

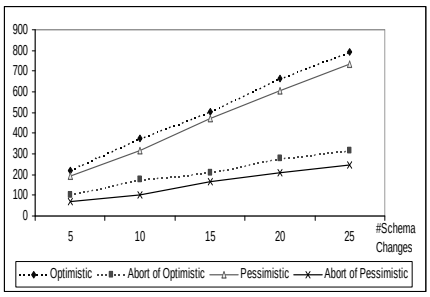

Figure 11. Increasing Number of Schema Changes

all updates. Thus no broken query would occur during the maintenance processing. When the time interval between schema changes increases, new updates could break the ongoing maintenance work. Thus the cost of both strategies increases. When the interval reaches a particular range, the cost reaches a high peak when the new schema change always occurs near the end of the current maintenance process. After the interval is larger than the maintenance time, the cost charted is only pure maintenance and no more due to aborts.

Differences between these two strategies can be observed. When a broken query happens, the optimistic strategy is able to correct the unsafe dependencies of all updates in the UMQ so far. The corrected plan for these updates is static in the sense that it fails to respond to any newly conflicting updates and has to endure the abort. In contrast, the pessimistic strategy with pre-exec detection has the potential to avoid this break and consistently performs better.

We then study the effects of the number of schema changes on performance. Setting the time interval between two schema changes to 25 seconds, Figure 11 depicts the maintenance and abort costs (depicted on the y-axis) with optimistic and pessimistic strategies, respectively. We vary the number of schema changes from 5 to 25, randomly generated over six relations. In particular, one drop operation is followed by several rename operations. Each rename operation may break the ongoing maintenance of the drop operation. As the number of schema changes increases, the abort cost increases as well for both strategies since more schema changes would introduce more conflicts between them. As expected, the system with pessimistic strategies still performs better due to its ability to avoid some aborts.

6.4.2. Effects of Data Updates on Abort. We now study how data updates affect the system performance, in particular, the broken query problems. We have a fixed number of schema changes, i.e., one drop attribute followed by four rename schema changes, and we fixed the time interval between them to be 25sec. We now vary the number of data updates. We measure the maintenance and abort costs for both optimistic and pessimistic strategies in Figure 12.

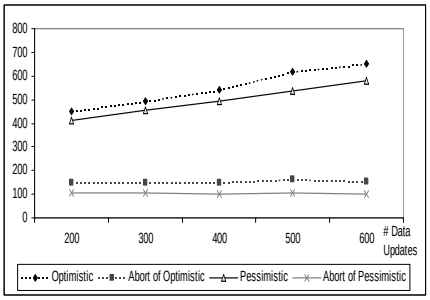

Figure 12. Increasing Number of Data Updates

From Figure 12, we can see that the abort cost remains consistent. Thus we conclude that the *abort cost* is not significantly affected by the data updates. This confirms that the major cause for the abort cost are schema changes as discussed in Figures 10 and 11. Finally, recall the potential infinite wait (or infinite number of aborts) of *Dyno* as stated in Section 4.4 is highly unlikely to occur in practice. First, there must be a continuous stream of new schema changes. However, in reality, schema changes are likely to be less frequent than data updates. Second, Figure 10 shows that the chance of abort is small when the schema changes are either close together (occurred altogether) or far apart from one another (not affecting each other at all). The abort cost reaches a high peak only when the time interval between schema changes is similar to the maintenance time of one schema change, which is a particular narrow range. Hence the infinite wait is highly unlikely to arise.

7. Related Work

Schema mapping [13, 14] 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 [10]. 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 [9] system, the view definition evolves after the source schema changes. In [17] the authors propose to incrementally adapts the schema mapping to the new source or target schema or constraints.

Maintenance of materialized view has been extensively studied in the past few years [1, 15, 20, 5, 6, 12]. However, most of these works assume a static schema. This is no longer a valid assumption in the dynamic environment. While in [1, 15, 20], 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. [18] 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.

In this paper, we identify that the new concurrency problems are caused by the read-write conflicts on the view definition. Unlike traditional serializability theory [2] that has full control to schedule the read/write operations to resolve the conflicts, in our context, the write (source update) is autonomous and hence locking is not an appropriate technique. Since the write is unabortable, we propose to process the related updates altogether to resolve the deadlock using a novel view adaptation algorithm.

8. Conclusions

In this paper, we illustrate that the materialized view maintenance anomaly problems in a loosely-coupled environment correspond to the problem of unsafe dependencies between source updates. We categorize the different types of dependency relationships that cause the anomaly problem. Then we propose a suite of detection methods for unsafe dependencies. We also introduce a dependency correction solution to eliminate unsafe dependencies. Finally we propose *Dyno* that combines both detection and correction strategies into one integrated solution. We show the correctness of *Dyno*, namely, that it enables the integrator to handle concurrent data and schema changes in a dynamic context. Dyno is a general strategy for handling view maintenance concurrency problems independent from the specific view maintenance algorithms, and thus has the potential to be plugged into any view system.

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

References

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

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

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

[4] S. Chen, X. Zhang, and E. A. Rundensteiner. A Compensation-based Approach for Materialized View Maintenance in Distributed Environments. Technical report, 2004.

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

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

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

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

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

[10] 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.

[11] B. Liu, , S. Chen, and E. A. Rundensteiner. A Transactional Approach to Parallel Data Warehouse Maintenance. In *DaWaK*, pages 307–316, 2002.

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

[13] J. Madhaven, P. A. Bernstein, and E. Rahm. Generic Schema Matching with Cupid. In *VLDB*, pages 49–58, 2001.

[14] R. J. Miller, L. M. Haas, and M. A. Hernández. Schema Mapping as Query Discovery. In *VLDB*, pages 77–88, 2000.

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

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

[17] Y. Velegrakis, R. J. Miller, and L. Popa. Mapping Adaptation under Evolving Schemas. In *VLDB*, pages 584–595, 2003.

[18] X. Zhang and E. A. Rundensteiner. Integrating the Maintenance and Synchronization of Data Warehouses Using a Cooperative Framework. In *Information Systems*, volume 27, pages 219–243, 2001.

[19] X. Zhang and E. A. Rundensteiner. PSWEEP: Parallel View Maintenance Under Concurrent Data Updates of Distributed Sources. *VLDB Journal*, 2003, to appear.

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