WPI-CS-TR-01-17 Jul 2001

Detection and Correction of Conflicting Concurrent Data Warehouse Updates

by

Songting Chen
Jun Chen
Xin Zhang
Elke A. Rundensteiner
Revised on Jan. 2002

Computer Science
Technical Report
Series

WORCESTER POLYTECHNIC INSTITUTE

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

Detection and Correction of Conflicting Concurrent Data Warehouse Updates *

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

Abstract

A Data Warehouse Management System (DWMS) maintains materialized views derived from one or more data sources under source changes. Given the dynamic nature of modern distributed environments, both source data and schema changes are likely to occur autonomously and even concurrently. Data warehouse maintenance strategies proposed in the recent literature typically issue maintenance queries to the data sources and then apply compensating queries to correct any errors in the delta refreshs. However, these existing solutions are limited to handling pure data updates only, making the restricting assumptions that (1) the schemata of all sources remain stable over time, and (2) maintenance queries are never broken by source schema changes.

In this paper, we introduce a formal framework that successfully lifts these restrictions. In particular, we characterize two classes of dependencies between concurrent update messages not currently handled in the literature. We then propose a two-pronged solution strategy tackling these dependencies: one, a dependency detection strategy based on dependency safety analysis, and two, a conflict resolution strategy based on reordering and merging affected updates. The DWMS now refreshs the data warehouse correctly in situations not handled by previous DWMS solutions. This proposed solution has been successfully implemented in our Dynamic Data Warehousing system, called DyDa, the first system that can correctly maintain a data warehouse view under all classes of concurrency. The experimental results show that our new concurrency handling strategy can be plugged into any data warehouse system, imposing an almost negligible overhead on existing data updates maintenance solutions to allow for this extended functionality.

Keywords: Data Warehouse, Maintenance Query, Concurrent Updates, Broken Query, Dependency Detection, Dependency Correction.

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

1 Introduction

1.1 Introduction to Data Warehouse Environment

Data warehouses (DW) [GM95, MD96] are built by gathering information from data sources and integrating it into one virtual repository customized to users' needs. One important task of a Data Warehouse Management System (DWMS) is to maintain the DW upon changes of the data sources, since for most data sources, frequent data updates are common, for example in stock price marketing or telephone call recording. In addition, the requirements of a data source are likely to change during its life-cycle, which may force schema changes for the data source. A schema change could occur for numerous other reasons, including design errors, schema redesign during the early stages of database deployment, the addition of new functionalities and even new developments in the modeled application domain, such as new tax laws or Y2K problems. Even in fairly standard business applications, rapid schema changes have been observed. In [Mar93], significant changes (about 59% of attributes on the average) were reported for seven different applications over relational databases. A similar report can also be found in [Sjo93]. These applications ranged from project tracking, accounting and sales management, to government administration. In the field of information integration, data sources also have been found to be extremely volatile to the extent that some of them may be temporarily or even permanently unavailable [IFF+99].

Data sources are typically owned by different providers and hence function independently from one another. In a fully concurrent environment, the relationship between the DWMS and the data sources is loosely-coupled [ZGMHW95]. That is, source updates are committed without any concern of how the DWMS will incorporate them. This causes new problems for DW maintenance. When processing a source update, the DWMS may need to query the data sources for more information by issuing so called *maintenance queries* [ZGMHW95]. Given a concurrent source update, the *maintenance queries* may either return erroneous query results or even fail completely. These problems, which we call the *DW maintenance anomaly problems*, are illustrated via a motivating example in Section 1.2.

While recent work [ZGMHW95, AASY97, SBCL00] has proposed *compensation-based* solutions to remove the *concurrent data updates*' effect from the query results, we demonstrate that these existing solutions will fail under source schema changes. The reason of this new anomaly problem is that neither *maintenance queries* nor *compensation queries* may be able to get any query response from data sources due to the discrepancy of the schema of the data source with the schemata required by these queries. With the interleaving of concurrent source data and schema changes from distributed data sources, maintenance becomes even more complicated. This is the problem we now address in this work.

1.2 The Maintenance Anomaly Problem

In a fully concurrent environment, the relationship between the DWMS and the data sources is loosely-coupled. That is all source updates are committed without any concern of how the DWMS may or may not incorporate them. Thus new problems arise for DW maintenance. Intuitively, the problem is how to refresh the DW while the DWMS no longer knows the current state of the underlying sources. When processing a source update, the DWMS may need to query the data sources for more information by issuing so called *maintenance queries* [ZGMHW95]. They then assume that the data sources are in the state that the update was just committed in. This is however not necessarily true because the data sources could continue to change both their data and schema autonomously. Thus the *maintenance queries* that the DWMS generates for that data source may either return erroneous query results [ZGMHW95] (if the related data has meanwhile been changed by a source data update) or even fail completely (if the schema of the data source referred in the query has meanwhile been modified by a source schema change). These problems, which we call the *DW maintenance anomaly problems*, are illustrated via a motivating example in Section 2.

While recent work in the literature [ZGMHW95, AASY97, SBCL00, ZRD01] has proposed *compensation-based* solutions to remove the *concurrent data updates*' effect from the query results, we demonstrate in this paper that these existing solutions will fail under source schema changes (see the example in Section 2). The reason of this new anomaly problem is that neither maintenance queries nor compensation queries may be able to get any query response from data sources due to the discrepancy of schema of the data source with the schemata types required by these queries. With the interleaving of concurrent source data updates and schema changes over distributed data sources, things become even more complicated. This is exactly the problem we now successfully tackle in this work.

1.3 Our Contributions

In this paper, we now propose a solution, called Dyno, capable of dealing with all types of concurrency conflicts under both source data and schema changes. Dyno enables all data sources to operate autonomously for all update types. To the best of our knowledge, this is the first complete solution to all identified DW concurrency problems. In summary, the contributions 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 dependencies.

(2) We propose a suite of methods for the detection of different classes of dependency problems during DW maintenance and develop a dependency correction algorithm to eliminate anomalies once identified.

(3) We design a new view adaptation algorithm for processing an executable plan generated by the *Dyno* algorithm that now may contain batches of mixed data updates and schema changes, which cannot be handled by previous VM, VS or VA algorithms.

(4) We have implemented the *Dyno* solution in our DyDa data warehousing system demonstrated at *SIGMOD'*2001 [CZC+01]. Our experimental results confirm that *Dyno* imposes an almost negligible overhead on data update processing while now offering comprehensive support for concurrency handling.

1.4 Outline of Paper

In the next section we give a motivating example of the maintenance anomaly problem. Section 3 introduces a general data warehouse architecture. Section 4 provides a formal characterization of the concepts of dependency. Section 5 discusses the correctness criterion for dependency violation correction. Section 6 proposes a complete solution to the problem in the form of the *Dyno* algorithm. It also proves the termination and correctness of this approach. Section 7 discusses the experimental results. Section 8 reviews related work, while Section 9 concludes the paper.

2 The Maintenance Anomaly Problem

We distinguish between three DW maintenance tasks, namely, View Maintenance (VM), View Synchronization (VS) and View Adaptation (VA) as explained below. VM [ZGMHW95, AASY97, SBCL00] maintains the DW view extent under source data updates. In contrast to VM, VS [LNR01, NLR98] aims at rewriting the DW view definition when the schema of a source has been changed. To handle the delete of any schema information of a data source, VS tries to locate an alternative source for replacement to keep the view available. Thereafter, View Adaptation (VA) [NR99] incrementally adapts the view extent to again match the newly changed view definition.

For a single (non-concurrent) *data update* (DU) or a single *schema change* (SC), the processing steps of the DWMS have been well defined in the literature. For a single DU, the DWMS uses one of the many VM algorithms [ZGMHW95, AASY97, SBCL00] proposed in the literature to refresh the data warehouse. For a single SC, DWMS first engages the VS to rewrite the affected view definition(s) and then the VA to incrementally repopulate the extent of the modified view(s) [LNR01].

If there is no concurrency, namely, the data warehouse maintenance completes before the next source update occurs, then the VM incorporates each source *data update* (DU) while VS and VA together incorporate the source *schema change* (SC) into the data warehouse. However, as we have mentioned, the data sources are autonomous and may undergo changes at any time. Thus during the data warehouse maintenance of one update message, other source updates may occur causing the

maintenance anomaly problem as illustrated by the example below.

Example 1 *Assume we have four data sources with one relation each as shown in Figure 1.*

DS 1: Customer(Name, Address, Phone): Customer Info.
DS 2: Tour(TourID, TourName, Age, Type, NoDays): Tour. Info.
DS 3: Participant(Participant, TourID, StartDate, Loc): Participant. Info.
DS 4: FlightRes(Name, Age, FlightNo, Dest): Reservation. Info.

Figure 1: Description of Data Sources.

The view Asia-Customer of the data warehouse is defined by the SQL query in Equation (1). Assume the data update "insert ΔF = ('Ben','MA',123456) into the Customer relation". In order to determine the delta effect on the DW extent, this now requires us to send the incremental view **maintenance query** *Q [ZGMHW95] defined in Equation (2) down to the FlightRes relation.*

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

$$\begin{array}{lll} \text{SELECT} & 'Ben' \text{ as } Name,\ F.Age,\ F.FlightNo,\ F.Dest & \\ \text{FROM} & FlightRes\ F & \\ \text{WHERE} & F.Name = 'Ben' & (2) \\ & AND\ F.Dest = 'Asia' & \end{array}$$

Let us now distinguish between two different scenarios that may arise:

- Duplication Anomaly*: If during the transfer time of the query Q to the relation FlightRes in the DS4, FlightRes has already committed a new data update ΔF = insert ('Ben', 18,'AA3456','Asia'). This new tuple would also be included in the join result of Q and a tuple ('Ben', 18, 'AA3456','Asia') would be inserted into the view. However, later when the DW starts processing ΔF = insert('Ben', 18,'AA3456','Asia'), the same tuple would be inserted into the view again. A* ***duplication anomaly*** *appears, as has also been observed by [ZGMHW95].*
- Broken Query Anomaly*: If during the transfer time of the query Q to DS4, the FlightRes relation in DS4 has a schema change, e.g., the attribute FlightRes.Age is dropped, then the query Q faces a schema conflict. In this case, the selected attribute Age is not available. Thus the query Q cannot be processed by DS4 due to the inconsistency between the schemata specified in the query ("F.Age") and the schema of the underlying source (no such attribute). We then say that the query Q is* **broken**.

An intuition of the **broken query** problem is that the old view definition is no longer consistent with the underlying sources. To solve the problem, we have to rewrite the view definition immediately so as to make future maintenance work applicable.

The timeline of either of those two scenarios is shown in Figure 2. We can see that when the DWMS wants to incorporate a source data update, it may send maintenance queries down to a data source. If

at the time before answering the maintenance queries, another SC has already been committed at this data source, these queries will not succeed due to the changed source schema.

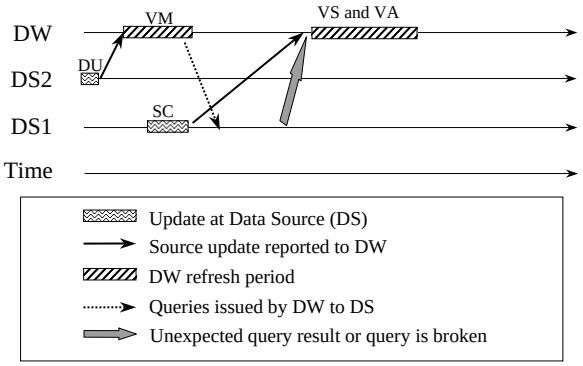

Figure 2: Interleaved Processing in DWMS

From the example above, we know that the concurrency is caused by the autonomy of sources that commit updates that may conflict with the DW maintenance process. A concurrent DU may result in an **incorrect query result** returned by a maintenance query while a concurrent SC may result in a **broken query** that cannot be processed by the respective data source, i.e., an error message is returned.

3 The Data Warehouse Framework

To set the context of our solution, we first introduce the general data warehouse management framework as depicted in Figure 3.

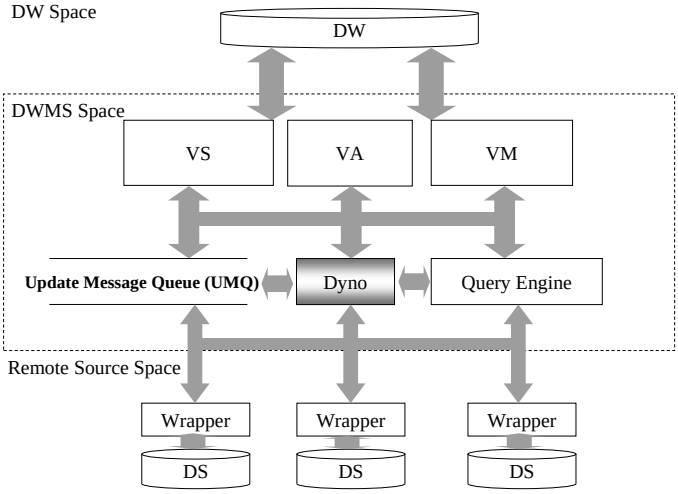

Figure 3: The DataWarehouse Architecture

As typically assumed in the literature [ZGMHW95, AASY97], the framework is divided into three spaces. The DW space houses the extent of the data warehouse. It receives view delta data from the middle space to update the DW. The remote source space is composed of remote data sources and their corresponding wrappers. We assume that every update at a data source is reported to the DWMS once committed.

The DWMS space maintains the DW under source updates. It consists of the general DW management algorithms, such as VS [LNR01], VA [GMR95, NR99] and VM [ZGMHW95, AASY97, SBCL00] for maintenance of different type of source updates, namely, data updates and schema changes. The Update Message Queue (UMQ) collects the updates from the data sources. The Query Engine is responsible for query processing, that is, decomposing the view queries to individual source queries, sending down these queries to data sources and then collecting and assembling query results.

Most DWMS solutions maintain the updates simply based on their arrival order which might introduce concurrency problems as described in Section 2. In this paper, we instead propose a new strategy, called Dyno (DYNamic reOrdering) to detect concurrent updates and correct them by adjusting their processing order. Dyno is a general strategy, in particular, a dynamic scheduler for concurrency handling in DW maintenance. Hence as shown in Figure 3, we propose to plug Dyno into this framework to coordinate with various other components to handle the maintenance concurrency problems.

4 Classes of Dependency Relationships

The motivating problem in Section 2 illustrates that the known view maintenance solutions are not applicable when the view definition is not consistent with the underlying sources. We discover that the reason for this are the dependencies between source updates. In this section, we first formalize these dependencies and then define the view maintenance anomaly problem based on the concept of dependency.

4.1 Concurrency Dependency

There are two kinds of dependencies between source updates received by a DWMS: *concurrency dependencies* and *semantic dependencies*. Example 1 illustrates that concurrency dependencies are caused by the asynchronicity between the update processes at the data sources and the DW refresh processes at the DWMS. Below, we first describe an assumption and define some notations.

Assumption 1 *Network communication between an individual data source and the DWMS is FIFO [ZGMHW95, AASY97].*

Assumption 1 guarantees that whenever an erroneous query result is returned to the DWMS from a data source, the update at that source that caused the problem has already prior arrived in the

DWMS. This way we can find the conflicting concurrent updates.

Definition 1 *We then define the following notations:*

(1) Given two update messages m1 and m2 in the UMQ. If m1 precedes m2 in the UMQ, then we denote this by "pos(m1, UMQ) $\prec$ pos(m2, UMQ)".

(2) DU(n)[i] and SC(n)[i] denote a data update or a schema change, respectively, committed on data source with index 'i' and with a global unique "n" indicating the time this update message arrives at the DWMS.

(3) Q denotes the maintenance query generated by the VM (or VA) algorithm if the update is a DU (or an SC). We use DU(m)[i].Q[k] to denote one maintenance query issued to a data source with index 'k' when processing the DU(m)[i], and use SC(m)[i].Q[k] to denote one maintenance query issued to data source 'k' when processing the SC(m)[i].

(4) QR denotes the query results returned by data sources. In particular, we use DU(m)[i].QR(n)[k] or SC(m)[i].QR(n)[k] to denote the result of the query DU(m)[i].Q[k] or SC(m)[i].Q[k]. "n" denotes the time when this query result message arrives at the DWMS.

Intuitively, the reason for the DW maintenance anomaly problem is that the DW maintenance query is affected by a concurrent source update. We now formalize this notion of being affected by a concurrent update.

Definition 2 *Let X(n)[j] and Y(m)[i] denote DUs and/or SCs committed on data source 'j' and data source (i) respectively. We say that Y(m)[i] is* ***concurrent dependent*** *(**CD**) on X(n)[j], denoted by:* $Y(m)[i] \xleftarrow{cd} X(n)[j]$
iff:

1. *X(n)[j] and Y(m)[i].Q[j] both refer to a common relation on data source (j), and*
2. *there is at least one query result Y(m)[i].QR(k)[j] such that $n < k$. The later means that X(n)[j] is received by the DWMS* ***before*** *the maintenance query result Y(m)[i].QR(k)[j].*

For example, in the *broken query anomaly* case of Example 1, the maintenance query of the data update is broken and the result arrives at the DWMS after the drop operation. Hence the data update is *concurrent dependent* on the drop operation.

There are four kinds of **Concurrent Dependencies**:
(1) DU(1) $\xleftarrow{cd}$ DU(2): The process of DU(1) is *concurrent dependent* on the process of DU(2);
(2) SC $\xleftarrow{cd}$ DU: The process of SC is *concurrent dependent* on the process of DU;
(3) DU $\xleftarrow{cd}$ SC: The process of DU is *concurrent dependent* on the process of SC;
(4) SC(1) $\xleftarrow{cd}$ SC(2): The process of SC(1) is *concurrent dependent* on the process of SC(2).

The two cases of Example 1 are of *concurrency dependency* types "DU$\xleftarrow{cd}$DU" and "DU$\xleftarrow{cd}$SC", respectively.

Note that concurrent DUs (namely the *concurrency dependency* cases (1) and (2) modify the sources' *content* and thus may invalidate the results returned by the maintenance queries. Concurrent SCs (*concurrency dependency* cases (3) and (4) modify the underlying sources' *schema* which may break the maintenance queries.

We propose to apply the compensation algorithm [AASY97] to correct any errors from the maintenance query result in the Query Engine module, hence solving the dependency types (1) and (2). We omit the discussions of this algorithm here. For details please refer to [AASY97]. In the remainder of this paper, we focus on solving the latter two *concurrency dependency* problems triggered by the schema changes, namely, DU$\xleftarrow{cd}$SC and SC$\xleftarrow{cd}$SC. Hence from now onwards, when we refer to *concurrent dependencies*, we mean the dependencies of type (3) and (4).

4.2 Semantic Dependency

A **semantic dependency** concerns the semantic requirement of the processing order of the updates from the same resources. Let's take for example the view defined in Equation (1). Assume there are two SCs: SC(1) and SC(2). SC(1) renames the relation *FlightRes* to *FlightReservation* and then SC(2) renames the relation *FlightReservation* to *FR*. Obviously, if we reverse the processing order, we cannot proceed. The processing of SC(2) is *dependent* on the processing of SC(1). It's apparent that we must process the update messages in the order they are received, and not in any other order.

We assume we preserve the processing order of updates from shared resources such as the same relation, attribute or tuples. We adopt the same relation here for simplicity. We now formally define this type of **semantic dependency (SD)**.

Definition 3 *Let X(n)[i] and Y(m)[i] denote either DUs or SCs on the same data sourse (i), then X(n)[i] is* ***semantic dependent (SD)*** *on Y(m)[i], denoted by:*
X(n)[i] $\xleftarrow{sd}$ *Y(m)[i]*
iff:

1. *m < n, and*
2. *X(n)[i] and Y(m)[i] both refer to a shared resource on data source (i), such as the same relation.*

4.3 Dependency Properties

The two types of dependencies share an important property: both represent constraints on the processing order between updates. Hence we now abstract them in a common manner.

Definition 4 *For two updates m1, m2 in UMQ, we say m1 is* ***dependent on*** *m2, denoted by m1←m2, if either m1 is* concurrent dependent *on m2 by Definition 2, or m1 is* semantic dependent *on m2 by Definition 3.*

Lemma 1 *For two updates m1 and m2, if m1 is dependent on m2 by Definition 4, then m2 must be processed by the DWMS* ***before*** *m1.*

For a semantic dependency, the required order given above is obvious as discussed in Section 4.2. For a concurrent dependency, since the source schema is changed, the old view definition is out of date and has to be rewritten to be consistent with the new schema of the underlying sources immediately. For example, in the second case of Example 1, the drop attribute has to be processed first to rewrite the view definition (the rewritten view is presented in Section 4.4), otherwise the processing of the insert utilizing the old view definition would not succeed.

Definition 5 *For two update messages m1 and m2, we define the* ***directed dependency relationship*** *between m1 and m2 to be:*

1. ***independent*** *iff there is no dependency between m1 and m2 by Definition 4.*
2. ***safe dependent*** *iff pos(m1, UMQ) ≺ pos(m2, UMQ) and all dependencies between m1 and m2 by Definition 4 are m2←m1.*
3. ***unsafe dependent*** *iff pos(m1, UMQ) ≺ pos(m2, UMQ) and there is at least one dependency m1←m2 in the UMQ.*

The *concurrent dependency* of the second case in Example 1 is DU$\stackrel{cd}{\leftarrow}$SC. However, since pos(DU, UMQ) ≺ pos(SC, UMQ), this dependency is **unsafe** by Definition 5.

We now are ready to characterize the DW *broken query* problem.

Theorem 1 *The broken query problem corresponds to the existence of some* ***unsafe*** *concurrency dependency between updates.*

This is because, the *broken query* problem occurs when a source schema change affects the maintenance process of a previous source update. Thus there must be a *concurrent dependency* between these two updates that is *unsafe* by Definition 5.

By Theorem 1, to resolve the DW anomaly problem is thus to find a processing sequence of updates to make all dependencies **safe** by Definition 5.

4.4 Cyclic Dependencies

A set of dependencies may comprise a cycle as now illustrated by an example. Given the data source relations from Example 1, assume an SC_1 (drop relation Customer) and SC_2 (drop attribute

FlightRes.Age). Based on the view synchronization (VS) algorithm in [LNR01], the data warehouse may rewrite its view definition for each schema change as shown in Equations (3) and (4), respectively. The basic idea of this rewriting is that if a relation is dropped, we try to locate an alternative relation for replacement. If an attribute is dropped, we try to locate an attribute from another relation for replacement along with an appropriate join. In our example, this is join condition *"F.Name = T.TourName"* shown in Equation (4).

```
CREATE VIEW   Asia − Customer' AS
SELECT        P.Participant, F.Age,
              F.FlightNo, F.Dest
FROM          Partipant P, FlightRes F
WHERE         P.Participant = F.Name
              AND F.Dest = 'Asia'                    (3)
```

```
CREATE VIEW   Asia − Customer' AS
SELECT        C.Name, T.Age,
              F.FlightNo, F.Dest
FROM          Customer C, FlightRes F, Tour T        (4)
WHERE         C.Name = F.Name AND
              F.Dest = 'Asia' AND
              F.Name = T.TourName
```

Assume both SC_1 and SC_2 have been committed. If we process SC_1 first, the view is rewritten into Equation (3) by replacing the *Customer* by the relation *Participant*. However, we are unable to adapt the view extent, because the *F*.Age attribute is no longer available. Similarly, if we process SC_2 first, the view is rewritten to Equation (4) by including the *Tour* relation and a corresponding join [1]. Again, we are unable to proceed because the *Customer* has been dropped.

We thus notice that these two updates *are dependent on each other*, i.e., the dependency orders as defined in Definition 4 between them comprise a cycle. We give a formal definition now.

Definition 6 *For n update messages m[i_1], m[i_2], ..., m[i_n], we say the dependencies among these update messages comprise a* ***dependency cycle*** *if they satisfy the following:*

1. *for* $1 \leq k < n$: $m[i_k] \leftarrow m[i_{k+1}]$ *if* $pos(m[i_k], UMQ) \prec pos(m[i_{k+1}], UMQ)$.
2. *and* $m[i_n] \leftarrow m[i_1]$.

Intuitively, such a "cycle" of dependency edges in a dependency graph may result in a deadlock in the sense that we have maintenance processes waiting for each other. Dependencies forming such a cycle may be all concurrency dependencies, or may be a mixture of semantic and concurrent dependencies. They can never be all just semantic dependencies. This is because the semantic dependency directly relates to the sequence in which updates were committed by a data source and such a commit sequence does never comprise a cycle.

5 Correctness Criterion for Update Message Processing

We now introduce a correctness criterion for DW maintenance processing. We start with a correctness definition assuming a fixed set of updates. In the next section, we will show how to adjust this to also

[1] The rewriting strategies, i.e., the view synchronization (VS) solutions are described in [LNR01, NLR98].

take the newly incoming updates into consideration.

From the DWMS's point of view, we call the arrival order of source updates the *receiving order*. First, for one specific data source, by the Assumption 1, its updates arrive at the DWMS in a strict sequential order, namely, in the order in which they were committed at that data source. Second, the order among updates from different sources is not critical [2]. However, as shown in Example 1, the DW maintenance based on the *receiving order* may cause broken query anomaly. We hence define an order of updates that can avoid such anomaly.

Definition 7 *Given a sequence of updates, any order of processing these updates is called a* ***legal order*** *if it keeps all* dependencies safe.

Having all *concurrent dependencies safe*, by Theorem 1, means no *broken queries* will occur. To have all *semantic dependencies safe*, we have a processing order consistent with the order of updates at each data source. The DW is thus able to maintain updates in a *legal order* correctly assuming the DW maintenance modules VM,VS and VA process each update are correct. The reason is that the maintenance process of one update will not be affected or aborted by other updates, thus as long as the maintenance algorithms are correct, the refreshing of the DW is correct.

Defintion 7 thus establishes the correctness criterion for a solution strategy of the anomaly problems defined in Section 2.

6 Dyno: A Dynamic Scheduler

In this section, we first introduce dependency detection and correction algorithms. Then we propose a DYNamic reOrdering strategy, called *Dyno* that integrates the former two components to search for a *legal order* for updates at run-time.

6.1 Detection of Unsafe Dependencies

6.1.1 Dependencies Detection Method

Our dependency detection method is composed of two steps. First, we construct a **dependency graph**, where the nodes are updates and the directed edges are either *concurrent dependencies* or *semantic dependencies*. Second, we check if there are any *unsafe dependencies* as given by Definition 5.

Given a sequence of updates, we can determine the *concurrent dependency* between two updates m1 and m2 using the following method. If the *maintenance query* generated for maintenance of m1 refers

[2]Note that here, for simplicity of description, we assume there is no user required update processing order, such as foreign key constraints. However, they can be viewed as a third category of dependency and can be treated similarly.

to the same relation as m2, then there is a *concurrent dependency*, namely m1 $\xleftarrow{cd}$ m2, by Definition 2. The reason is that m2 may affect this *maintenance query*. Since the *maintenance query* is constructed solely based on the knowledge from the DW view definition, we can infer if two updates may have a *concurrent dependency* relationship simply by analyzing the view definition. No other information is needed.

It is straightforward to identify *semantic dependencies* between updates, namely, each pair of two adjacent updates from the same relation is assigned a *semantic dependency* edge.

We now examine the time complexity of building such a *dependency graph*. First, the complexity of identifying *concurrent dependencies* between updates is O(n^2), where n is the number of updates. That is, because in the worst case, each pair of two updates would have one *concurrent dependency* in between (maybe two as described in Section 4.4).

Second, the complexity of building *semantic dependencies* between updates is O(n), where n is the number of updates. To achieve this, since the updates arrive at the DW from the same source have already been ordered by Assumption 1, we can create one bucket for each data source and scan them once.

Thus the time complexity of building a *dependency graph* is up to $O(n^2) + O(n)$, i.e., $O(n^2)$. Note that, if there are only data updates, the *maintenance query* would never break. Thus there is no *unsafe* dependency [3]. In this case, we optimize by setting a *schema change flag* indicating if any schema change occurs. If there is no schema change, we avoid building the *dependency graph* altogether thus reducing the time complexity to O(1).

After construction of the *dependency graph*, we can easily check if a dependency edge in this graph is *safe* simply by checking the update positions in the UMQ based on Definition 5.

6.1.2 Pre-exec vs In-exec Detection

Different detection methods in terms of the time they are applied can be designed: *pre-exec detection* and *in-exec detection* method.

The *pre-exec static detection method* detects the dependency **before** the update message in the UMQ is processed in order to discover any potential conflicts with other updates before doing the maintenance.

Take the second case of Example 1. Before DWMS processes the insert, it discovers that the drop attribute operation already in the UMQ forms an *unsafe* concurrent dependency with the insert. Thus we need not bother to maintain the insert now and send down a maintenance query to DS_4 which surely would break. The main goal of this strategy is to avoid such waste of computation that later would have to be discarded.

[3] Semantic dependencies only would never be unsafe.

The *pre-exec detection method* by itself is not sufficient because a schema change that occurs after the *pre-exec detection* phase could still break the maintenance query.

The *in-exec dynamic detection method* is thus introduced for compensation purposes. It can detect any *unsafe* dependencies **during** the maintenance by a *broken query* scheme. That is, whenever a *broken query* occurs, an error message is reported from the data source. We thus know that there is an *unsafe* dependency in the UMQ. To optimize this, the change of the *schema change flag* (in Section 6.1.1) caused by a newly incoming conflicting schema change can terminate the current ongoing maintenance process before it encounters the *broken query*. This way we can save some further wasted effort.

6.1.3 Optimistic vs. Pessimistic Detection Strategies

The above two basic detection methods lead to two complete detection strategies: optimistic and pessimistic. Both lead to correct final results, and the choice of which strategy to use is largely based on the expected behavior of the data warehouse environment in terms of frequency of and types of concurrency among updates.

1. **Optimistic detection strategy**: An optimistic solution aims to minimize or completely avoid any the performance overhead during normal processing. Instead it may endure some extra overhead cost to recover if a problem actually happens. For this, we simply employ the *in-exec detection* method. By not employing *pre-exec detection*, thus we cannot prevent *broken queries* and thus some additional abort costs may arise.
2. **Pessimistic detection strategy**: A pessimistic solution aims to minimize or even prevent any aborts of the maintenance process at the cost of a continuous performance overhead during normal processing. In other words, a pessimistic strategy attempts to anticipate, detect and ideally prevent any *unsafe* dependency at run-time, thus avoiding the broken queries and their overheads. For this, we utilize a *pre-exec detection* method to detect as much as possible all *unsafe* dependencies *before* processing, hence the name *pessimistic*. But as indicated in Section 6.1.2, we still need to employ the *in-exec detection* as supplementary detection method to assure correctness.

An experimental comparison of these two strategies is described in Section 7.4.

6.2 Static Correction of Unsafe Dependencies

After we have detected an *unsafe* dependency between two updates using one of the strategies from Section 6.1.3, we need to determine how to change the *unsafe* dependency into a *safe* one. Based on the *dependency graph* constructed during the detection phase, we propose a solution that employs two *dependency correction operations* to achieve this goal.

In particular, assume we have detected that there is an *unsafe* dependency order between two update messages m1 and m2, i.e., m1 is before m2 in UMQ and m1 is dependent on m2 (m1←m2) by Definition 5. We propose to **reorder** them by placing m2 just before m1, which in this case would turn this dependency *safe*. The intuition of this method is that after we reorder these two updates, their processing order would obey the order imposed by their dependency constraints.

If there were also another dependency m2←m1 that comprises a cycle, *reordering* obviously can't make both of these two cyclic dependencies *safe*. We instead propose to **merge** m2 and m1 into one *combined* update {m1,m2} and place at m1's place. To generalize, we eliminate the dependency by merging the two or more updates into one *combined* update which will be processed by the DWMS in one refresh process. The intuition of this operation is that since only the combination of these updates reflects the real schema of the data sources, the DW must take all of them into consideration at the same time to be able to rewrite the view definition consistently with all data sources.

The DEpendency COrrection algorithm **Deco** applying both the reorder and merge strategies for a sequence of updates is shown in Figure 4.

```
Deco Algorithm
Input: A sequence of updates U
Output: A new sequence of U free of unsafe dependencies
begin
1: G ← Build Dependency Graph of U;
2: while(exists dependency (m1← m2) unsafe in G)
3: if exists (m2 ← m1) then  // merge step
4:    create m' = m1 + m2;
5:    replace m1 by m';
6:    remove m2 from G;
7: else insert m2 directly before m1 in UMQ; // reorder step
end
```

Figure 4: Deco: DEpendency COrrection Algorithm

Figure 5 depicts two dependency correction examples handled by Deco. Example 5.a illustrates the second case in Example 1, where a *concurrent dependency* between the insert (DU) and drop attribute (SC) exists. Deco reorders these two updates and the *concurrent dependency* becomes safe. Example 5.b illustrates the cyclic dependency described in Section 4.4, where two drop operations (SC1 and SC2) are dependent on each other. Deco merges these two updates into one combined update, thus the cyclic dependency disappears. This now requires that the VA algorithm [NR99] is capable of processing such combined batches of updates as described in Section 6.5.

Termination of Deco: We prove this by contradiction. Given a fixed set of updates, assume the algorithm does not stop. Then there must be some *corrected dependencies* in U turning back to *unsafe* since there are finite number of *unsafe* dependencies. In this case, a cycle is found and related updates are merged. This results in a reduced number of updates. If Deco doesn't terminate, the number

of updates would finally reduce to one, i.e., one *big* update that contains all original updates. The algorithm would still stop at that point.

Correctness of Deco: We can further conclude that the Deco algorithm terminates if and only if no more *unsafe* dependency exists. By Definition 7, we have a *legal order* of updates and thus the maintenance of these updates will be correct.

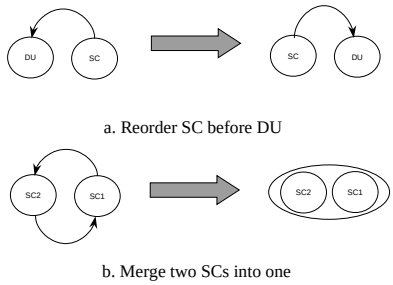

Figure 5: Examples of Unsafe Dependency Correction

6.3 *Dyno* Solution: Pulling It All Together

Our complete solution Dyno combines the *pessimistic dependency detection strategy* in Section 6.1.3 and the *Deco* dependency correction algorithm from Section 6.2. *Dyno* first uses the *pre-exec detection* strategy **before** starting the processing of the first update in the UMQ. *In-exec detection* is used **during** DW maintenance. Upon detection of any *unsafe* dependency, the *Dyno* solution uses *Deco* to turn any *unsafe* dependency into a *safe* one.

```
Dyno Algorithm
Input: Source updates dynamically arrive
Goal: DW maintaining these updates
begin
   while(1) begin
1:     if ( CheckSCFlag ) Deco(UMQ);
2:     h ← head update of UMQ;
3:     start DW Maintenance on h;
4:     if ( no broken query occurs )
5:         remove m from UMQ;
   end
end
```

Figure 6: *Dyno*: DYNamic reOrdering Algorithm

Figure 6 details the *Dyno* algorithm. Just before processing of the head update in the UMQ, **(1)** checks the schema change flag (Section 5.1.1). If there is no schema change, we can avoid the detection and correction step. Or Dyno will apply Deco to detect and correct any *unsafe* dependencies in the UMQ at that time. **(2) and (3)** get the head update in the UMQ and start the maintenance. During

the maintenance, **(4)**, the *in-exec detection* method is employed to detect any new *unsafe* dependency. If any broken query occurs, it will loop back to **(1)** to correct the anomaly problem. The head update will be removed from the UMQ in **(5)** if no broken query occurs. Then we continue the maintenance from **(1)**.

6.4 Termination and Correctness of Dyno

Dyno is correct in a static environment given a fixed number of updates (see Figure 4). We now briefly argue the correctness of *Dyno* in a dynamic context by now also considering newly incoming updates.

Correctness of Dyno: Assume a new conflicting update m occurs. First, it may happen before the current maintenance process and thus can be detected by *pre-exec detection*. Then the conflicts will be corrected into a conflict-free *legal order* by Deco. Or it may occur during the ongoing maintenance process and break it, *Dyno* would detect it by *in-exec detection* and again correct it by Deco. In either case, the DWMS will have a *legal order* of updates and thus correctly maintain the DW. Finally, if m occurs after the maintenance process, then no concurrency would happen.

Termination of Dyno: The only possibility that may cause *Dyno* to loop infinitely without any DW refresh commit is that we have continuously new schema changes arriving that would always escape the *pre-exec detection*, i.e., during the maintenance process and would always break the ongoing maintenance work. However, we argue that such case is unlikely because it would require (1) a frequent and continuing stream of schema changes at data sources, and (2) the schema change always arriving after the *pre-exec detection* phase causing the DW never to refresh any update. We further experimentally evaluate this in Section 7.4.

6.5 Processing of Merged Updates

In Section 6.2, the Deco algorithm will generate an executable order for updates in the UMQ. If no *Merge* operation occurred, previous DW maintenance algorithms, like VM, VS or VA that work on one individual update at a time can continue to be applied. However, if the *Merge* operation does happen, there will be some complex merged update message containing both SCs and DUs over distributed data sources. Current DW maintenance algorithms cannot handle such mixed updates at a time. Due to the limited space, below we briefly describe the algorithm we have developed for processing such merged updates.

Given a set of merged updates, DW first rewrites the view definition based on the schema changes within the merged updates. A general view synchronization (VS) [LNR01] algorithm can accomplish this job. As a result, the old view definition $V = R_1 \bowtie R_2 \bowtie ... \bowtie R_n$, is rewritten to $V' = R_1^{new} \bowtie R_2^{new} \bowtie ... \bowtie R_n^{new}$, 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.

In order to incrementally adapt the view extent, we need to figure out the delta change to get from V to V'. Here we present 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 is the difference between R_i^{new} and R_i. Here, new VA tries to restore the old state of the relation R_i from the view if R_i is dropped [NR99]. ΔR_i stands for data updates if no drop schema operation has occurred, or it would be the difference between the old relation and the replaced relation. With $R_i, R_i^{new}, \Delta R_i$, we are able to calculate ΔV as follows and thus incrementally adapt the view extent.

$$\begin{aligned} \Delta V &= \Delta R_1 \bowtie R_2 \bowtie ... \bowtie R_i \bowtie ... \bowtie R_n \\ &+ R_1^{new} \bowtie \Delta R_2 \bowtie R_3 \bowtie ..... \bowtie R_i \bowtie ... \bowtie R_n \\ &+ ... \\ &+ R_1^{new} \bowtie ... \bowtie R_{i-1}^{new} \bowtie \Delta R_i \bowtie R_{i+1} \bowtie ... \bowtie R_n \\ &+ ... \\ &+ R_1^{new} \bowtie ... \bowtie R_i^{new} \bowtie ... \bowtie R_{n-1}^{new} \bowtie \Delta R_n \end{aligned} \tag{5}$$

Let us now take the cyclic case in Section 4.4 as an example. First we rewrite the view as shown in Equation 6 based on both schema changes in the merged update instead of just one of them at a time.

```
CREATE VIEW   Asia − Customer' AS
SELECT        P.Name, T.Age,
              F.FlightNo, F.Dest
FROM          Participant P, FlightRes F, Tour T
WHERE         P.Name = F.Name AND
              F.Dest = 'Asia' AND
              F.Name = T.TourName
                                    (6)
```

Here, old view $V = Customer \bowtie FlightRes$ and new view $V' = Participant \bowtie (FlightRes' \bowtie Tour) = (Customer + \Delta Customer) \bowtie (FlightRes + \Delta FlightRes)$. We then calculate the view delta change by Equation 5 as follows. Since the relation Customer and part of the FlightRes have been dropped, we have to restore them from the view itself [4].

$$\begin{aligned} \Delta V &= \Delta Customer \bowtie FlightRes + Participant \bowtie \Delta FlightRes \\ &= \Delta Customer \bowtie \Pi_{FlightRes} V + Participant \bowtie \Delta FlightRes \\ &= (Participant - \Pi_{Customer} V) \bowtie \Pi_{FlightRes} V + Participant \bowtie (FlightRes' \bowtie Tour - \Pi_{FlightRes} V) \end{aligned}$$

[4]The assumptions that have to be held are described in [NR99].

While above just illustrates the intuitions, a more detailed solution can be found in [CZR02].

6.6 Consistency Level Achieved by *Dyno*

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

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

Clearly, *Dyno* achieves "Strong Consistency". This is so because, first, the reordering will keep all semantic dependencies thus the DWMS would process the source updates in the same order as they commit at the data source. The correctness of *Dyno* guarantees that each state of DW is correct. Second, however, since *Dyno* may merge updates thus not every update corresponds to a distinct DW state, hence cannot reach complete consistency. In conclusion, *Dyno* reaches "Strong Consistency".

7 Experimental Evaluation

7.1 Experimental Testbed

We have implemented the *Dyno* algorithm and embedded it into our data warehousing system DyDa [CZC+01]. The integration enables DyDa to maintain the DW under both concurrent data and schema changes. In DyDa, we apply the SWEEP [AASY97] algorithm to compensate for concurrent DUs, thus solving the first two dependency problems (see Section 4.1). DyDa is implemented using JAVA, using JDBC to connect to Oracle8i as DW server 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 data warehouse with one materialized join view defined upon these six source relations residing on a fourth DW server. All experiments are conducted on four Pentium III PCs with 256MB memory each, running Windows NT and Oracle8i.

7.2 Study of Data Update Processing

We first study the overhead that *Dyno* may bring to the system's data update processing. Clearly, any extra cost would be caused by the overhead of the detection process.

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* in Section 5.1.1 thus reducing the time complexity to O(1). The *in-exec detection strategy* would not ever be launched since aborts would never be caused by data updates only.

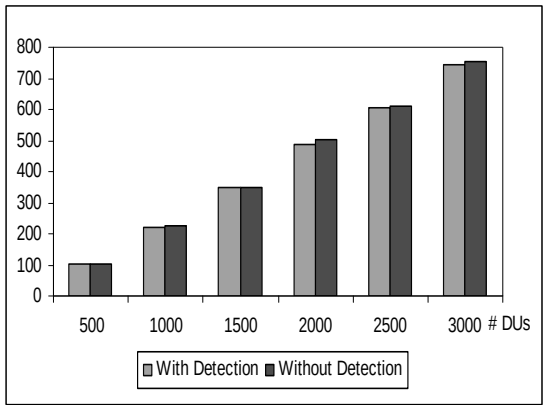

Figure 7: DU Processing with/Out Detection

Figure 7 depicts the total DW maintenance cost measured in seconds (depicted on the y-axis) with or without detection enabled for different numbers of source data updates (depicted on the x-axis). We find that the overhead of detection is almost unobservable in all cases. Since the detection cost is trivial, i.e., O(1), we can reasonably predict that such cost would be small even under a larger amount of data updates. We thus conclude that the *Dyno* algorithm imposes little extra cost on data update processing.

7.3 Cost of Broken Query

Recall that the broken maintenance query problem is caused by the existence of some concurrent schema changes. There are two kinds of broken query problems, namely, a data update maintenance processing is aborted by a schema change or a schema change maintenance processing is aborted by another schema change. Once such a broken query occurs, the DWMS has to drop all previous maintenance work and redo it. This imposes extra cost on DW maintenance, which we now will refer to as *maintenance abort cost*.

In this experiment, we study the cost of these two kinds of aborts. Two corresponding workloads have been chosen. The first is one data update followed by a conflicting schema change. The second is two conflicting schema changes. Then three different environmental settings are compared. First, we

measure the maintenance cost of all updates by spacing them far enough, so they won't interfere with each other [5]. This represents the minimum cost as no concurrency handling is needed (depicted by a grey bar in Figure 8). Second, we apply the *pessimistic strategy* to discover any potential concurrency conflicts before processing, thus trying to avoid the occurrence of any broken query (depicted by a black bar in Figure 8). Third, we apply the *optimistic strategy*. Thus only after the broken query occurs and is detected, do we correct the dependencies and restart the maintenance. In this case, more aborts may occur (depicted by a white bar in Figure 8).

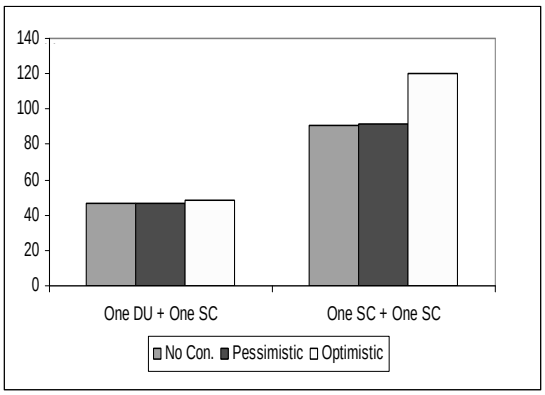

Figure 8: Cost of Broken Query

We find that the cost of aborting schema change processing is significant compared to that of data update processing, i.e., the white bar of SC/SC (where the abort of schema change occurs) is much higher than the other two. While for DU/SC, three costs are similar. The reason is that the schema change processing is rather complex and thus time consuming compared to data update processing. It is costly to redo the schema change maintenance process. Secondly, we find that the *pessimistic strategy* does indeed help to reduce the expensive abort requiring only a minimum overhead, i.e., the black and grey bars are similar in both cases.

7.4 Mixed Update Processing

Above, we observe that the most expensive extra cost besides the normal DW maintenance processing is the abort of the schema change processing. Also we find that the *pessimistic strategy* does help to reduce this expensive abort cost (Figure 8) via a simple setting. However, the broken query may still happen even if employing the *pessimistic strategy* when the newly coming update breaks the ongoing maintenance work. We now study under what conditions the broken query would occur and how the *pessimistic strategy* helps avoid this in mixed update environments.

[5] Because the source update occurs after the completion of DW maintenance of the previous update.

7.4.1 Time Interval between Schema Changes

In this experiment, we employ a mixture of 200 data and 10 schema changes, both randomly generated over all six relations. We vary the time interval between schema changes (as depicted on X-axis).

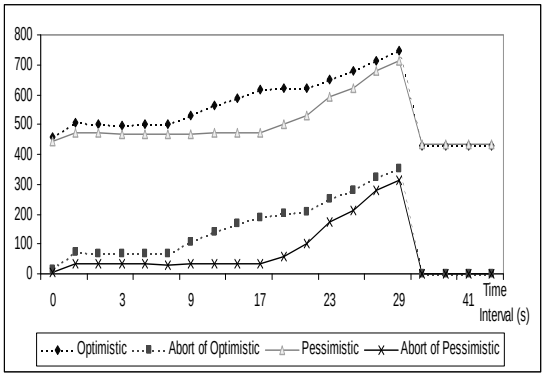

Figure 9: Time Interval of Schema Changes

Figure 9 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. 0s means that all schema changes flood into the DWMS before any maintenance. From Figure 9, we see that this case has the best performance for both strategies. This is obvious because the system is able to correct any *unsafe* dependencies once for all updates, thus no broken query would occur. 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 because the new schema change always occurs near the end of current maintenance. Even the *pre-exec detection* cannot avoid such broken queries in this case. After the interval is larger than the maintenance time, there will be no concurrency but only pure maintenance cost.

However, 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 until it breaks. In contrast, the *pessimistic strategy* with *pre-exec detection* has the potential to avoid this break and consistently performs better.

7.4.2 Number of Schema Changes

Given a time interval between schema change of 25s, Figure 10 depicts the maintenance costs (depicted on the y-axis) with *optimistic* and *pessimistic* strategies, respectively, when varying the number of schema changes from 5 to 25.

The broken query cost increases linearly for both strategies. Also, as expected, the system with *pessimistic strategies* still performs better due to its ability to avoid some broken queries.

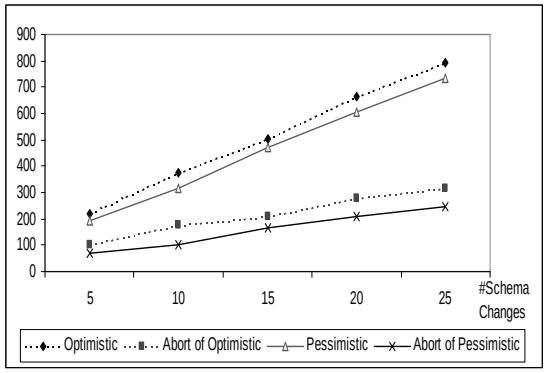

Figure 10: Increasing Number of Schema Changes

7.4.3 Effects of Data Updates

We now study how data updates affect the system performance, in particular, the broken query problems. We fix the number of schema changes and their time intervals, but vary the number of data updates.

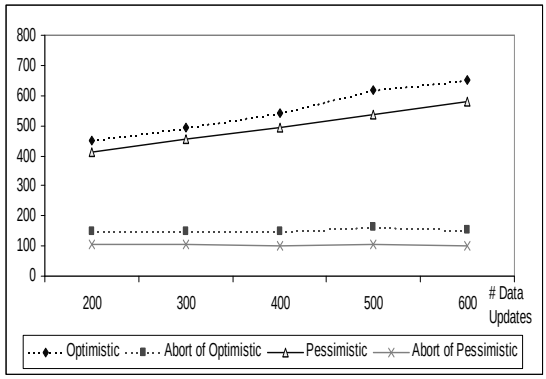

Figure 11: Increasing Number of Data Updates

From Figure 11, we can see that the broken query cost remains consistent thus is not related to the data updates. Hence the dominant effect on broken query cost are schema changes as discussed in Figures 9 and 10. Also we conclude that our *pessimistic strategy* does help improve the performance in all cases.

Finally, recall the potential infinite wait (or infinite number of aborts) of *Dyno* as stated in Section 6.4 is highly unlikely to occur in practice. Because first, there must be a continuous stream of new schema changes. However, schema changes are often less frequent than data updates in reality. Second, Figure 9 shows that the abort cost reaches a high peak only when the time interval is close to the maintenance cost. Otherwise it is small when the schema changes are either close or apart from one another. A high abort cost would only occur when the interval of the schema changes would be within a particular narrow range. Hence this is very unlikely to arise.

8 Related Work

Maintaining materialized view under source updates in a DW environment is one of the important issues of data warehousing [Wid95, RKZ00]. Initially, some research has studied incremental view maintenance assuming no concurrency [CGL+96, GL95, LMSS95]. Such algorithms for maintaining a data warehouse under source data updates are called view maintenance algorithms. More recently, the EVE project [LKNR99, NR98] studied the problem of how to maintain a data warehouse not only under data updates but also under schema changes. The essence of EVE is to automatically rewrite the view definitions when the base schema has been changed, and to try to locate the best replacements for affected view components. Such view rewriting caused by schema changes of sources is called View Synchronization. After the view definition has been redefined, View Adaptation (VA) [GMR95, NR99] incrementally adapts the view extent. View self-maintenance [QGMW96, Huy97] is one approach to maintain the DW extent trying to limit the access to the base relations. [MRSR01] proposes a strategy for multiple view maintenance by applying multiple query optimization techniques, i.e., by sharing some common expression computations.

In approaches that send maintenance queries down to the source space, concurrency problems can arise. In [ZGMHW95], they introduced the ECA algorithm for incremental view maintenance under concurrent data updates restricted to a single source. In Strobe [ZGMW96], they extend their approach to handle multiple sources. Agrawal et al. [AASY97] propose the SWEEP-algorithm that can ensure consistency of the data warehouse in a large number of cases compared to the Strobe family of algorithms. [ZRD01] improves upon the performance of SWEEP by parallelizing the maintenance process. [SBCL00] proposes to only materialize delta changes of both sources and views with timestamps, so the view is able to asynchronously refresh its extent. They also introduce a propagation algorithm that could significantly reduce the number of compensation queries. However, none of them can handle source schema changes and any of these proposed systems would fail under a mixture of concurrent data updates and schema changes. To our knowledge, we are the first to solve this challenging problem.

Our early work [ZR99, ZR01] studies the problems of the DW refresh caused by the concurrency of source data updates and schema changes. However it assumes that each source reports a schema change and then waits for permission from the DWMS before it commits the change to the source. In other words, the data sources are assumed to be fully cooperative. Our proposed solution now successfully drops this restricting assumption. [CCR00] employs a multiversion concurrency control algorithm to handle the concurrency problem assuming there are enough system resources to completely materialize versions of the source data and its updates. While in this current solution, no extra storage is needed and no versions are kept.

9 Conclusions

In this paper, we illustrate that the DW maintenance anomaly problem corresponds to the problem of unsafe dependencies between update messages reported to the DWMS. We analyze and categorize the different types of dependency relationships between source updates. 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 integrates both detection and correction strategies into one integrated solution. We prove the correctness of *Dyno*, namely, that it enables a DWMS to handle concurrent DUs and SCs in a dynamic context. Dyno is a general strategy for maintenance independent from the specific view maintenance algorithms, and thus has potential to be plugged into any DWMS system. We also develop a novel algorithm for the incremental adaptation of the data warehouse view extent under a mixed batch of updates. The experimental results show that our new concurrency handling strategy imposes an negligible overhead to allow for this extended functionality. Thus, adding Dyno to any DWMS system is a complete and win-win solution.

Acknowledgements The authors would like to thank students in the Database Systems Research Group at WPI for their valuable feedback. We are grateful to Bin Liu, Jehad Sabbah and Andreas Koeller for helping to implement the DyDa system.

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.

[CCR00] J. Chen, S. Chen, and E. A. Rundensteiner. TxnWrap: A Transactional Approach to Data Warehouse Maintenance. Technical Report WPI-CS-TR-00-26, Worcester Polytechnic Institute, December 2000.

[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 Demo Session*, page 619, Santa Barbara, CA, May 2001.

[CZR02] S. Chen, X. Zhang, and E. A. Rundensteiner. Dyda: A compensation based approach to dynamic data warehouse maintenance. Technical Report WPI-CS-TR-02-02, Worcester Polytechnic Institute, January 2002.

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

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

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

[Huy97] N. Huyn. Multiple-View Self-Maintenance in Data Warehousing Environment. In *International Conference on Very Large Data Bases*, pages 26–35, 1997.

[IFF+99] Z. G. Ives, D. Florescu, M. Friedman, A. Y. Levy, and D. S. Weld. An Adaptive Query Execution System for Data Integration. pages 299–310, 1999.

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

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

[LNR01] A. M. Lee, A. Nica, and E. A. Rundensteiner. The EVE Approach: View Synchronization in Dynamic Distributed Environments. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2001.

[Mar93] S. Marche. Measuring the Stability of Data Models. *European Journal of Information Systems*, 2(1):37–47, January 1993.

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

[MRSR01] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham. Materialized View Selection and Maintenance Using Multi-Query Optimization. In *Proceedings of SIGMOD'01*, pages 307–318, May 2001.

[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 International Conference on Extending Database Technology (EDBT'98)*, pages 359–373, Valencia, Spain, March 1998.

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

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

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

[RKZ00] E. A. Rundensteiner, A. Koeller, and X. Zhang. Maintaining Data Warehouses over Changing Information Sources. *Communications of the ACM*, pages 57–62, June 2000.

[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.

[Sjo93] D. Sjoberg. Quantifying Schema Evolution. *Information and Software Technology*, 35(1):35–54, January 1993.

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

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

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

[ZR99] X. Zhang and E. A. Rundensteiner. The SDCC Framework for Integrating Existing Algorithms for Diverse Data Warehouse Maintenance Tasks. In *International Database Engineering and Application Symposium*, pages 206–214, Montreal, Canada, August, 1999.

[ZR01] X. Zhang and E. A. Rundensteiner. Integrating the Maintenance and Synchronization of Data Warehouses Using a Cooperative Framework. In *Information Systems*, 2001.

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