WPI-CS-TR-01-17 July 2001

# Detection and Correction of Conflicting Concurrent Data Warehouse Updates

by

Songting Chen
Jun Chen
Xin Zhang
Elke A. Rundensteiner

## 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

July 23, 2001

**Abstract**

A Data Warehouse Management System (DWMS) maintains materialized views derived from one or more information sources (ISs) under source changes. Given the dynamic nature of modern distributed environments such as the WWW, both source data and schema changes are likely to occur autonomously and even concurrently. Strategies proposed in the recent literature to achieve DW consistency typically are based on issuing maintenance queries to ISs and then applying compensating queries to correct any errors in the DW refreshs. However, these solutions are limited to handle 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, we introduce strategies for the detection of these dependencies between updates based on cyclic dependency analysis. Two, we introduce a conflict resolution strategy based on reordering and merging affected updates. We have proven the correctness of our solution, i.e., that the DWMS refreshs the data warehouse correctly in situations not handled by previously DWMS solutions.

This proposed solution has been successfully implemented in our Dynamic Data Warehousing system, called DyDa. DyDa is the first system that can correctly maintain a DW under all classes of concurrency. The experimental results show that our new concurrency handling strategy imposes a minimal (smaller than 5%) overhead to allow for this extended functionality. Our advanced concurrency detection strategy even succeeds in reducing the expensive cost of maintenance aborts and hence restarts of schema change processing under a high system load, thus achieving overall improved maintenance performance.

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

# 1 Introduction

## 1.1 Introduction to Data Warehouse Environment

Data warehouses (DW) [GM95, MD96] are built by gathering information from information sources (ISs) and integrating it into one virtual repository customized to users' needs. Data warehousing is important for many applications such as E-commerce, decision support systems and web-site management that must function in large-scale environments composed of distributed sources. One important task of a Data Warehouse Management System (DWMS) is to maintain the DW upon changes of ISs.

The requirements of an IS system, like most other systems, are likely to change during its life-cycle. The extent of changes in a typical working relational database system is illustrated in [Sjo93], which documents the measurement of schema evolution during the development and initial use of a health management system used at several hospitals. There was an increase of 139% in the number of relations and 274% in the number of attributes in the system during the nineteen-month period of study. In [Mar93], significant changes (about 59% of attributes on the average) were reported for seven applications. These applications ranged from project tracking, real estate inventory and accounting and sales management, to government administration of the skill trades and apprenticeship programs. Furthermore, in distributed environments, ISs are typically owned by different information providers and hence function independently from one another. This implies they will update their data and schemas concurrently without any concern for how this may affect the materialized views defined upon them [Wid95, RKZ00]. They are generally not aware of nor willing to synchronize with the DWMS's refresh process, thus resulting in concurrency between these updates.

While such concurrency problems have received increased attention in recent years, practically all existing work [ZGMHW95, AASY97, SBCL00, ZRD01] focuses on View Maintenance (VM) to refresh the data warehouse extent under concurrent source data updates only. All prior work assumes a static environment, and would fail under source schema changes as we describe below.

## 1.2 The Maintenance Anomaly Problem

During the DW maintenance processing of a source update the DWMS may need to query the ISs for more information by issuing so called *maintenance queries* [ZGMHW95]. In a fully concurrent environment, the relationship between the DWMS and the ISs is loosely-coupled. That is all IS updates are committed without any concern of how the DWMS incorporates 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 IS space. When processing a source update, the DWMS assumes the ISs that it is sending queries to be in the state when that update was originally committed. This is however not necessarily true because the ISs could continue to change both their data and schema autonomously. Thus the *maintenance queries* that the DWMS generates for that IS may either return erroneous query results [ZGMHW95] (if the related data has meanwhile been changed by an IS data update) or even fail completely (if the schema of the IS referred in the query has meanwhile been changed by an IS schema change). We refer to this as the DW maintenance anomaly problem.

While recent work in the literature [ZGMHW95, AASY97, SBCL00, ZRD01] proposed *compensation-based* solutions that correct the erroneous query results caused by concurrent data updates, we demonstrate in this paper 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* can get any query response from ISs due to the discrepancy of source schema with the schemata types required by these queries. A detailed example of this problem is illustrated in Section 2.

### 1.3 Our Contributions

In this paper, we now propose the first solution capable of dealing all types of concurrency conflicts under both source data and schema changes. Our system which we call DyDa (Dynamic Data warehouse maintenance) [CZC+01] supports for all ISs to operate both independently and autonomously without any synchronous cooperation. In summary, our contributions now are:

(1) We identify the problem that the maintenance processing of a batch of updates in the order they are received by the DWMS may lead to either incorrect results or even fail completely. We demonstrate that this maintenance anomaly problem arises due to the violation of dependencies between source updates. We categorize and formalize these dependencies, namely, concurrent dependencies and semantic dependencies.

(2) We propose three dependency detection strategies and a dependency correction algorithm. We design a solution called *Dyno* that combines these two techniques to handle all types of concurrency in a fully dynamic environment. We prove our solution correct, and also analyze it completely.

(3) We have implemented the *Dyno* solution in our DyDa [CZC+01] system as a proof of concept. This is the first system with a complete solution to all concurrency problems.

(4) Our experimental results confirm that our *Dyno* solution imposes a minimal overhead to allow for this functionality of extended concurrency handling. Our advanced concurrency detection strategy even succeeds in reducing the expensive cost of maintenance aborts and hence restarts of schema change processing under a high system load, thus achieving overall improved maintenance performance.

### 1.4 Outline of Paper

In the next section we give a motivating example of the maintenance anomaly problem. Section 3 introduces a brief overview of the DyDa architecture. We assume this framework as basis to study the problem and to formulate our solution. Section 4 develops a formal characterization of the concepts of dependency. Section 5 discusses the correctness criteria for dependency violation correction. Section 6 proposes the solution techniques and integrates them into a complete *Dyno* solution strategy. 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). VM [ZGMHW95, AASY97, SBCL00] aims to maintain 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 information source has been changed. Thereafter, View Adaptation (VA) [GMR95, NR99] incrementally adapts the view extent to again match the newly changed the view definition.

Thus 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, DWMS uses one of the many VM algorithms 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). This relationship is illustrated in Figure 1.

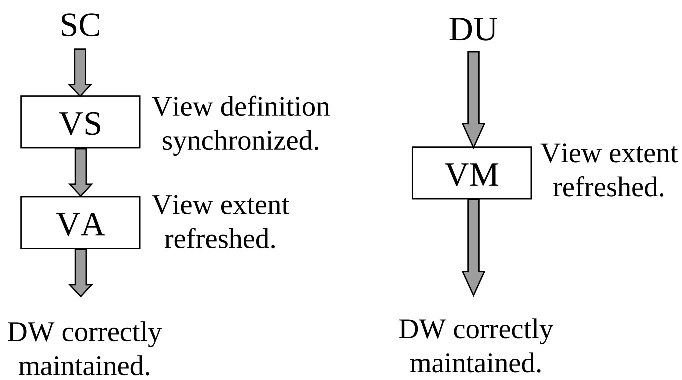


Figure 1: Sequential Processing of a Single DU/SC

If there is no concurrency, then the VM could incorporate the source *data update* (DU) successfully while VS and VA is responsible for incorporating the source *schema change* (SC). In a fully concurrent data warehouse environment, it is unrealistic to require all ISs to always cooperate with the DWMS to assure such a sequential processing order. Thus during the DW maintenance of one update message, other IS updates may occur causing the DW maintenance problem.

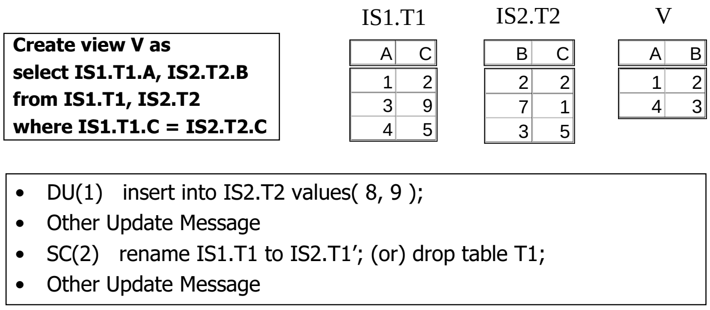


Figure 2: Example of Maintenance Anomaly Problem

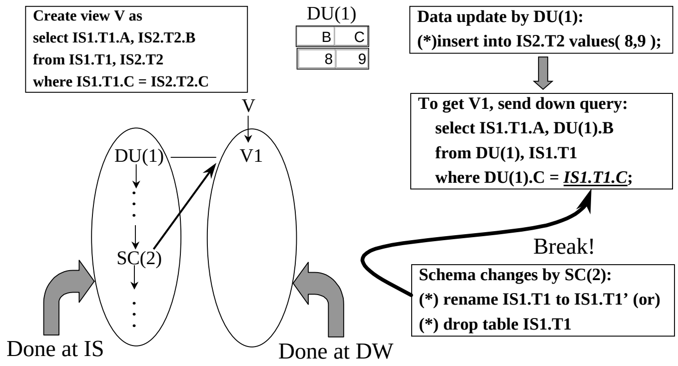


Figure 3: Broken Query Problem

Figure 2 details an example of an *aborted* maintenance process. We assume the view V defined on IS1.T1 and IS2.T2. Assume the DWMS receives several update messages about changes committed at IS1 and at IS2 in the order shown in Figure 2. Figure 3 illustrates what happens when the DWMS tries to process DU(1). To process DU(1) and calculate the delta change to refresh V into V1, the DWMS needs to issue a *maintenance query* [AASY97] to the IS1.T1 to fetch a delta view extent. But we notice that IS1.T1 has already been renamed to IS1.T1' (or even dropped completely) before the query arrives at IS1. Thus the query will be *broken* and the processing of DU(1) has to be *aborted*. Note that the *compensation* strategy proposed to tackle DU concurrency in [ZGMHW95, AASY97, SBCL00] doesn't help in this case, because here we cannot fetch any result at all due to the incompatibility of the schema.

The timeline of such a scenario is shown in Figure 4. We can see that when DWMS wants to refresh the DW content due to DU(1) at IS2, it may send maintenance queries down to IS1. At that time another SC is already committed at IS1. But the DWMS does not yet know about it. Thus this query that sent down to IS1 does not succeed due to the changed schema in IS1.

From the above example we can conclude that in a fully concurrent environment sequential processing of updates in UMQ is not always feasible, i.e., the maintenance process may be aborted. The reason for this is that the DWMS cannot always keep synchronized with the current state of all ISs. In the example shown above, the process of handling DU(1) is dependent on the handling of SC(2). In other words, DU(1) cannot be processed successfully until after SC(2) has been processed, even though DU(1) was received by the DWMS before SC(2).

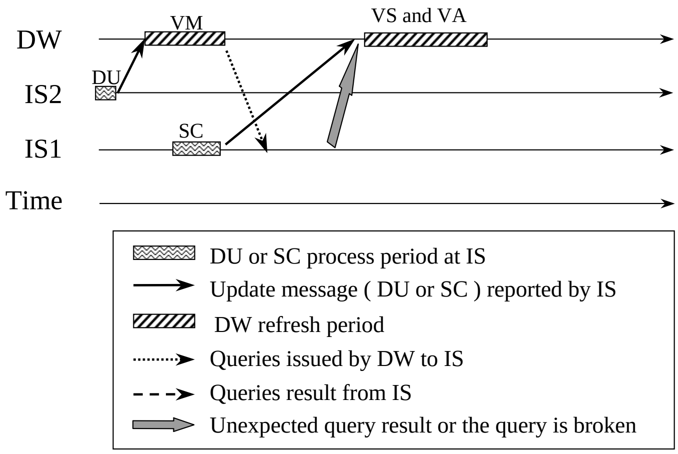


Figure 4: Interleaved Processing in DWMS

# 3 The DyDa Framework

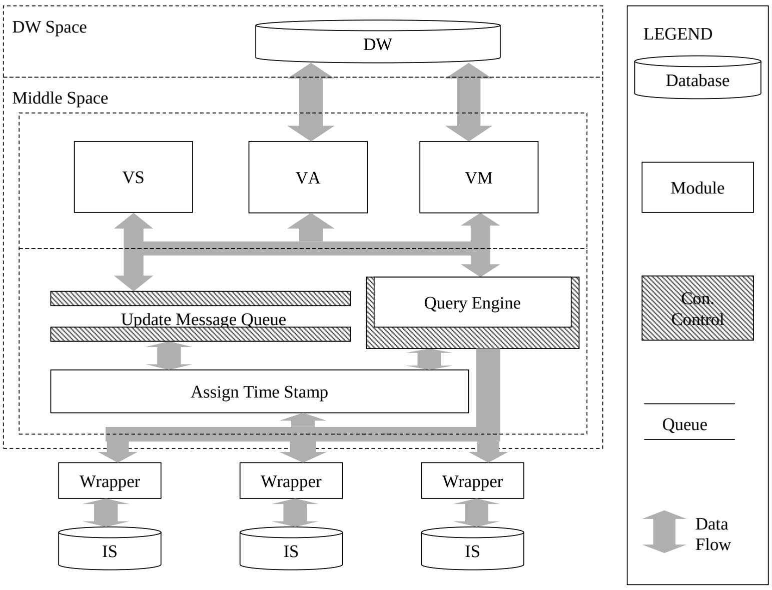


Figure 5: DyDa Architecture

The solution we propose in this paper has been developed as part of the dynamic data warehousing project funded by NSF, called DyDa [CZC+01] system that aims at resolving the concurrency problems. To have a better understanding of the context of our solution, we first introduce our DyDa framework.

Figure 5 depicts the architecture of the DyDa system. The framework is divided into three spaces: DW space, Middle space, and IS space. The DW space houses the extent of the data warehouse. It receives queries from the middle space bundled together with the data to update the data warehouse.

The IS space is composed of ISs and their corresponding wrappers. We assume that all IS transactions are local and every data update and schema change at an IS is reported to the DWMS once committed at the IS (or the delta changes can be detected and extracted by, for example the wrapper). Note that the DW and IS space setting is similar to that in any of the other related projects, such as [ZGMHW95, AASY97, ZRD01].

The middle space is the integrator of the DyDa framework, i.e., the DWMS system that aims to maintain the DW under source updates. It can be divided into two subspaces. The higher-level subspace is called the DW Management Subspace consists of the general DW management algorithms, such as VS, VA and VM. With the combination of these three modules, the system is able to handle non-concurrent data updates and schema changes. According to the type of each source update, DyDa may use either VS, VA, or VM algorithms to update the DW correspondingly.

The lower layer consists of the Time Stamp Manager, the Update Message Queue (UMQ) and the Query Engine. The Time Stamp Manager assigns timestamps to source updates as soon as they arrive at the middleware to make it possible to trace sequences of these updates globally. The Update Message Queue (UMQ) collects and manages the committed updates from the ISs, which are either data updates (DUs) or schema changes (SCs). The Query Engine in the middleware is responsible for query processing, that is, decomposing the view queries into individual IS queries, sending down these queries to ISs and then collecting and assembling query results.

The concurrency handling of the DW maintenance anomaly problem resides in the lower layer. There are various kinds of concurrency as illustrated in Section 4. Basically, the Query Engine employs a local compensation [AASY97] algorithm to handle concurrent DUs. We further extend the functionalities of UMQ to find an executable plan of updates when concurrent SC occurs, termed *Dyno*, which is the focus of this paper.

We make the following assumptions in our DyDa framework, as also made by other DW work [ZGMHW95, ZGMW96, AASY97].

**Assumption 1** *The network communication between an individual IS and the DWMS is FIFO.*

**Assumption 2** *All transactions of an IS are local to this IS (i.e., not distributed). In our current work, every IS transaction contains only one update either a data update or a schema change. The study of batching updates, a relatively straightforward extension, is left for future work.*

## 4 Classes of Dependency Relationships

The motivating problem in Section 2 clearly illustrates that there are dependencies between source updates that make view maintenance impossible. In this section we first analyze these dependencies. Then we formally 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 DWMS: *concurrency dependency* and *semantic dependency*. The example we discussed earlier illustrates concurrency dependencies. As we have seen in Section 2, concurrency dependencies are caused by the asynchronicity between the update processes at the ISs and the DW refresh processes at the DWMS side. We first define some notations.

**Definition 1** *With "i" a unique number for each IS and "n" a globally unique timestamp value assigned to each message (either an source update, a maintenance query, or query result), we define the following notations:*

*(1) Let DU(n)[i] and SC(n)[i] denote a data update or a schema change committed on IS[i] with the global timestamp "n" assigned by DWMS when this update arrives at DWMS.*

*(2) Let Q denote the set of maintenance queries generated by the VM (or VA) algorithm if the update is a DU (or SC). In particular, we use DU(m)[i].Q[k] to denote one maintenance query issued to IS[k] when processing the DU(m)[i], and use SC(m)[i].Q[k] to denote one maintenance query issued to IS[k] when processing the SC(m)[i].*

*(3) Let QR denote the set of query results returned by ISs. In particular, we use DU(m)[i].QR(n)[k] to denote the result of the query DU(m)[i].Q[k], and use SC(m)[i].QR(n)[k] to denote the result of the query SC(m)[i].Q[k]. "n" denotes a global timestamp assigned by DWMS when the query result arrives at DWMS.*

**Definition 2** *Given two update messages m1 and m2 in UMQ. If m1 precedes m2 in UMQ, then we denote this by "pos(m1, UMQ) $\prec$ pos(m2, UMQ)".*

Intuitively, the reason for DW maintenance anomaly problem is that the DW *maintenance query* is affected by a concurrent IS update. We formalize such concurrency below.

**Definition 3** *Let X(n)[j] and Y(m)[i] denote DUs and/or SCs committed on IS[j] and IS[i] respectively. We say that Y(m)[i] is* ***concurrent dependent (CD)*** *on X(n)[j], denoted by:*

1. *Y(m)[i] $\xleftarrow{cd}$ X(n)[j], if pos(Y(m)[i], UMQ) $\prec$ pos(X(n)[j], UMQ)*
2. *X(n)[j] $\xrightarrow{cd}$ Y(m)[i], if pos(X(n)[j], UMQ) $\prec$ pos(Y(m)[i], UMQ)*

*iff:*

1. *X(n)[j] and Y(m)[i].Q[j] both refer to a common relation on IS[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 DWMS* ***before*** *the maintenance query result Y(m)[i].QR(k)[j].*

For example, assume the time when the DWMS receives an update $DU_i$ from some $IS_k$ is $t_1$. To process $DU_i$, the DWMS issues some maintenance query DU(i)[k].Q[m] and sends it to some $IS_m$. The time when the DWMS receives the query result is $t_2$. Assume there is another update $SC_j$ from $IS_m$ that arrives at the DWMS between $t_1$ and $t_2$. According to Assumption 1, $SC_j$ is committed **before** the query *DU(i)[k].Q[m]* arrives at $IS_m$. Thus $DU_i$ is **concurrent dependent (CD)** on $SC_j$ since the maintenance query of $DU_i$ is influenced by $SC_j$ and thus may fail.

**Definition 4** *There are four kinds of **CDs**:*
*(1) DU(1)* $\xleftarrow{cd}$ *DU(2) or DU(2)* $\xrightarrow{cd}$ *DU(1): The process of DU(1) is **CD** on the process of DU(2);*
*(2) SC* $\xleftarrow{cd}$ *DU or DU* $\xrightarrow{cd}$ *SC: The process of SC is **CD** on the process of DU;*
*(3) DU* $\xleftarrow{cd}$ *SC or SC* $\xrightarrow{cd}$ *DU: The process of DU is **CD** on the process of SC;*
*(4) SC(1)* $\xleftarrow{cd}$ *SC(2) or SC(2)* $\xrightarrow{cd}$ *SC(1): The process of SC(1) is **CD** on the process of SC(2);*

Clearly, the example shown in Section 2 is a **CD** of type "DU$\xleftarrow{cd}$SC".

Note that concurrent DUs, (namely the Dependency cases 1 and 2 in Def 4) modify the ISs' *content* which may invalidate the results returned by the maintenance queries. Concurrent SCs, (namely the Dependency cases 3 and 4 in Def 4) modify the underlying ISs' *schema* which may break the maintenance queries. As mentioned in Section 3, the concurrency caused by DUs is handled in the Query Engine by applying a compensation-based algorithm [AASY97]. For case 2, the DWMS will send a maintenance query generated by VA [GMR95, NR99] that may also get erroneous results due to concurrent DUs. They can be compensated similarly. Instead in this paper we focus on solving the latter two **CD**s, namely, DU$\xleftarrow{cd}$SC and SC$\xleftarrow{cd}$SC, i.e., concurrency triggered by SCs.

The complexity of building such **CD** graph for updates in UMQ is O($n^2$), where n is the number of updates. Because in the worst case, each pair of updates would have a **CD**.

### 4.2 Semantic Dependency

A **semantic dependency** concerns the semantic requirement of the processing order of the updates from the same resources. Figure 6 shows an example of a semantic dependency. Assume a view V defined on IS1.T1 and IS2.T2. Assume two SCs: SC(1) and SC(2). SC(1) renames IS2.T2 to IS2.TT and then SC(2) drops IS2.TT.C. On the left side of Figure 6 we can see that the sequential processing order of SC(1) and SC(2) is correct. But if we reverse the processing order, we cannot proceed because SC(2) cannot be processed before SC(1). That is, the IS2.TT is undefined in DWMS if we have not yet processed SC(1). The process of SC(2) is *dependent* on the process of SC(1). It's apparent that we must refresh the DW in the order they are received, and not in any other order.

Thus it is necessary to preserve the processing order of updates from shared resources such as the same relation in this case. We now formally define that this type of **semantic dependency** (**SD**):

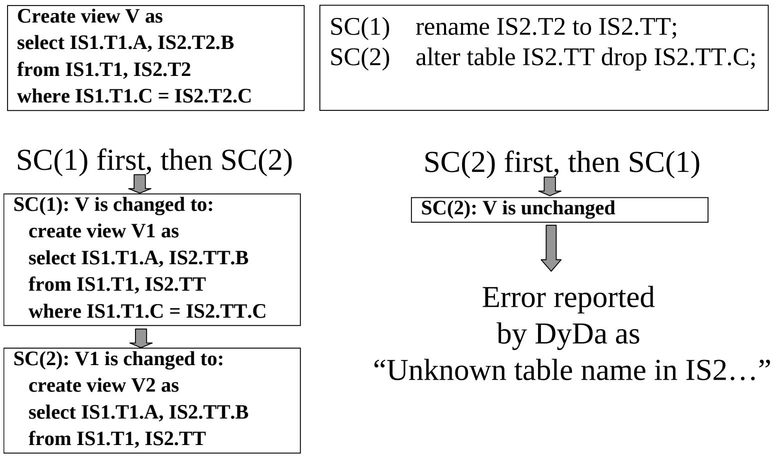


Figure 6: Example of Semantic Dependency

**Definition 5** *Let X(n)[i] and Y(m)[i] denote either DUs or SCs on the same information sourse IS[i], then X(n)[i] is* ***semantic dependent (SD)*** *on Y(m)[i], denoted by:*
*Y(m)[i]* $\xrightarrow{sd}$ *X(n)[i], if pos(Y(m)[i], UMQ)* $\prec$ *pos(X(n)[i], UMQ)*
*iff:*

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

It is clear that the complexity of building such a **SD** graph for updates in UMQ is O(n), where n is the number of updates.

### 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 6** *For two updates m1, m2 in UMQ, we define m1 is* ***dependent on*** *m2, denoted by m1*$\leftarrow$*m2 if pos(m1, UMQ)* $\prec$ *pos(m2, UMQ) or m2*$\rightarrow$*m1 if pos(m2, UMQ)* $\prec$ *pos(m1, UMQ), if either m1 is* ***CD*** *on m2 by Def 4, or m1 is* ***SD*** *on m2 by Def 5. We define the* ***dependency order*** *between m1 and m2 to be the direction of the dependency arrow. Otherwise we say the dependency order between m1 and m2 is null.*

**Lemma 1** *For two updates m1, m2 in UMQ, if m1 is dependent on m2 by Def 5, then m2 must be processed* ***before*** *m1.*

For example, in Figure 2, the SC(2) has been processed before DU(1), or the processing of DU(1) could never succeed.

**Definition 7** *For two update messages m1 and m2 in UMQ, we define the* ***dependency relationship*** *between m1 and m2 to be:*

1. ***independent*** *iff there is no dependency between m1 and m2 by Def 6.*
2. ***safe dependent*** *iff pos(m1, UMQ)* $\prec$ *pos(m2, UMQ) and all dependency orders between m1 and m2 by Def 6 are from m1 to m2 (m1*$\rightarrow$*m2).*
3. ***unsafe dependent*** *iff pos(m1, UMQ)* $\prec$ *pos(m2, UMQ) and and there is at least one dependency in UMQ in the order from m2 to m1 (m1*$\leftarrow$*m2).*

The CD of the example in Figure 2 is DU(1)$\xleftarrow{cd}$SC(2), however, since the pos(DU(1), UMQ) $\prec$ pos(SC(2), UMQ), this CD is **unsafe** and the *broken query* occurs.

### 4.4 Cyclic Dependencies

Given a set of dependencies in UMQ, they may comprise a circle. Figure 7 depicts a cyclic dependency example. Below we now illustrate that neither of the two processing orders, i.e., SC(1) by SC(2) or SC(2) by SC(1), can succeed.

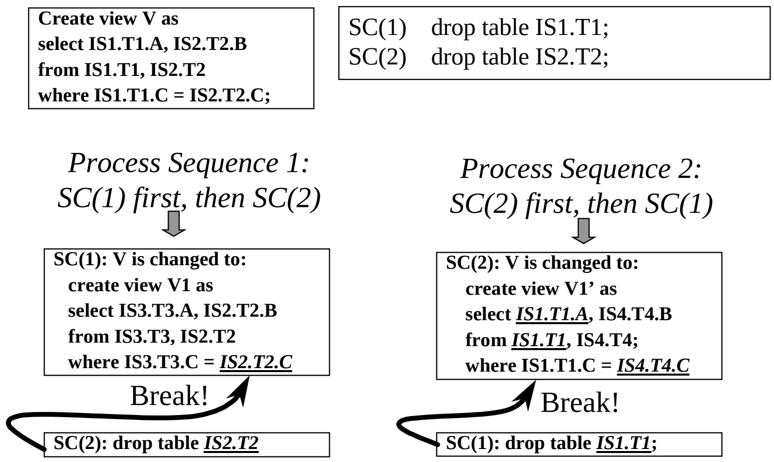


Figure 7: Example of Circle Dependency

If we process SC(1) first, we cannot succeed. When DWMS processes SC(1), let us assume by VS [NLR98], the DWMS finds a replacement for this relation, say, IS3.T3, and rewrites the view definition V. After that, it issues a VA [GMR95, NR99] query (as shown in the view definition of Figure 7) down to both IS2 and IS3 to adapt the new view extent. But at that time we notice that IS2.T2 has been dropped by SC(2). The query fails because it has IS.T2 referred.

If we process SC(2) first, we still cannot succeed. This time, assume that the DWMS finds another replacement IS4.T4 and rewrites the view definition V triggered by SC(2). It issues a VA query both

IS1 and IS4 to compute the new view extent. At that time, it finds that the IS1.T1 has already been dropped. So the query *breaks* and the processing of SC(2) *aborts*.

In the example shown in Figure 7, we notice that the two updates *are dependent on each other*, i.e., the dependency orders as defined in Def 6 between them comprise a circle. We give a formal definition of this dependency circle concept now.

**Definition 8** *For n update messages $m[i_1]$,$m[i_2]$,...,$m[i_n]$ in UMQ with $i_j < i_{j+1}$, we say the dependencies among these update messages comprise a* ***dependency circle*** *if they satisfy the following:*

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

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

### 4.5 Definition of DW Maintenance Anomaly Problem

We now are able to characterize the DW maintenance anomaly problem.

**Theorem 1** *The DW maintenance anomaly problem corresponds to the existence of* ***unsafe*** *concurrency dependencies between updates.*

Thus to resolve the anomaly is to find a processing sequence to make all dependencies **safe**.

## 5 Correctness Criteria for Update Message Processing

We now introduce a series of correctness criteria that a solution to the problem illustrated above should meet. For simplicity, we first give a correctness definition assuming a static state of UMQ. In an actual environment, the UMQ is of course dynamic, with updates incoming as well as others being removed at any time. Later we show the correctness definition of a static state of UMQ is sufficient for a solution, as long as the solution takes incoming update messages into account.

From the DW's point of view, we call the order of the updates incoming from the ISs the **receiving order**. By Assumption 1, we know that for one specific IS, its updates arrive at UMQ in a strict sequential order, namely, in the order in which they were actually committed at the IS. At time t, the order of messages in UMQ of a DWMS is called **a static snapshot of the UMQ order at time t**.

**Definition 9** *We say that a DW refresh process* ***succeeded*** *if it finishes the maintenance and updates the DW's database. Similarly we say that a DW refresh process* ***failed*** *if it is aborted by a concurrent SC by Def 4.*

**Definition 10** *Given a static snapshot of UMQ, any order of successfully processing these updates is called a* ***successful order*** *of this snapshot of UMQ. If we can reorganize the updates in a snapshot of UMQ to eliminate all unsafe dependencies, any such resulting reorganization order is called a* ***legal order*** *of this snapshot of UMQ.*

Def 10 establishes the correctness criteria for a solution strategy of the anomaly problem defined in Section 4.5. That is, the solution must always be able to find a **legal order** of update messages to make all dependencies **safe**.

**Theorem 2** ***Legal orders*** *of a snapshot of UMQ exist iff there are no cyclic dependencies.*

We can construct a *dependency graph* (DG) of UMQ that includes both **CD**s and **SD**s (as described in Section 6.1.1). If the graph contains a cycle, obviously we can not find a processing (acyclic) order for these cyclic-dependent updates. If it is instead a DAG (Directed Acyclic Graph), we can easily obtain an order that does not violate the order imposed by all dependencies by traversing the graph, which represents one legal processing order.

# 6 A Complete Solution Strategy: Dyno

We now introduce our solution strategy called *Dyno*. A complete solution must incorporate the following three functionalities: maintenance of source update, dependency detection and error correction operations. The *Dyno* solution entails three components: dependency detection methods for the two types of dependencies; dependency correction operations that rearrange or merge updates in UMQ in order to fix unsafe dependencies; and lastly an overall control strategy to integrate the former two components into a complete one.

### 6.1 Detection of Dependencies

#### 6.1.1 Dependencies Detection Method

We can construct a *dependency graph* (DG) which includes both **CD**s and **SD**s in the UMQ. Given a snapshot of UMQ, we can discover the **CD** between two updates, m1 and m2 using the method described below: If the *maintenance query* generated for maintenance of m1 refers to the same relation as m2, then there is a **CD**, namely m1 $\xleftarrow{cd}$ m2. The reason is that m2 may affect this *maintenance query*. Since the *maintenance query* is decomposed from DW view definition, we can infer if two updates may have **CD** by just referring to the view definition.

It is straightforward to construct **SD**s between updates, i.e., each two adjacent updates from the same relation has a **SD**. Note that we put both **CD** and **SD** in the same graph because they are actually both constraints on processing order information.

After the construction of DG, we can easily check if a dependency (or an edge in DG) is **safe**. by using Def 7.

#### 6.1.2 Time to Apply Detection

We further propose three different detection strategies with the same functionalities as described above, but only differ in terms of the time they are applied: *pre-exec static detection* method, *in-exec dynamic detection* method and *post-exec static detection* method.

A **pre-exec static detection method** detects the dependency **before** the head update in UMQ is processed. Its principle is to analyze the content of all other updates in UMQ and the maintenance queries to be generated by the VM or VS/VA modules respectively to discover potential dependencies between them. Given a snapshot of the UMQ, this method can detect all the concurrency and semantic dependencies between update messages in this snapshot. For example, in Figure 3, before DWMS processes *DU(1)*, we can discover *SC(2)* that is already in the UMQ which forms an unsafe concurrent dependency with *DU(1)*. Thus we need not bother to send down a maintenance query to IS1 which will surely break. Note that the **pre-exec static detection method** by itself if not sufficient because some SC that occurs after the pre-exec detection would still break the maintenance query.

An **in-exec dynamic detection method** detects the unsafe dependency **during** processing of the head update in UMQ. Its principle is to detect if any *maintenance query* failed due to the incompitable schema of ISs. However, in some cases, this detection method cannot catch all unsafe dependency as illustrated below.

Assume an IS relation is dropped and recreated or two relations switch their names. The *maintenance query* may succeed but operated on different data. The **in-exec dynamic detection method** can not detect such special "broken" query problem.

A **post-exec static detection method** is thus introduced to compensate for the previous two methods. Its task is just to make sure the IS relation which the maintenance query succeeds in operating upon is the original one. Note that we do not need to examine update messages which have arrived after the returned query results.

### 6.1.3 An Interpreted Detection Strategy in *Dyno*

Based on these three detection strategies, we distinguish between two kinds of complete detection strategies to our problem: optimistic and pessimistic. Both kinds lead to correct final results, and the choice of which kind of strategy to utilize is largely based on the expected behavior of the data warehouse environment in terms of concurrency as we experimentally illustrated in a later section.

1. **Optimistic detection strategy**: An optimistic solution aims to minimize the performance overhead during normal processing and then has to endure some cost to recover if a problem actually happens. It employs both *in-exec detection* and *post-exec detection* strategies. Whenever a maintenance query fails, the DWMS aborts the on-going processing of current update. After some necessary recovery, the process resumes to handle the next update message.
2. **Pessimistic detection strategy**: A pessimistic solution aims to minimize or even prevent any aborts of the maintenance process at the cost of an added performance overhead during normal processing. A pessimistic strategy thus attempts to anticipate, detect and ideally prevent any unsafe dependency, thus avoiding aborts and their overheads. Thus it utilizes a *pre-exec detection* method to detect dependencies *before* processing as much as possible, hence the name pessimistic. But as indicated in Section 6.1.1, it still needs to employ the *in-exec* and *post-exec detection* as supplementary detection methods to assure complete detection.

Our *Dyno* solution employs the "Pessimistic Detection Strategy" as shown in Figure 8.

1. **pre-exec static detection method:**

   Before the DW maintenance of head update in UMQ: Check if this update is involved in any unsafe dependency.
2. **in-exec dynamic detection method:**

   Activated only when the maintenance query failed due to the discrepancy of schema: Abort current processing.
3. **post-exec static detection method:**

   After the success of maintenance query: It aims to make sure the "succeeful" query does operate upon the desire relation.

Figure 8: Detection Strategy of *Dyno*

### 6.2 Static Correction of Unsafe Dependencies

After we have detected an unsafe dependency between two updates, we need to determine how to change the unsafe dependency into a safe one. Based on the dependency graph constructed during the detection, we propose a solution that employs two *dependency correction operations* to achieve this goal, namely, by either rearranging updates in UMQ, or by merging updates in the UMQ.

In particular, assume 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 Def 7. We propose operation *Op-Precede* to precede m2 before m1 thus correct this **unsafe** dependency. The intuition of this method is that after we reorder these two updates, their processing order would fit their dependency.

If there exists another dependency m1→m2, which comprises a cycle, we instead propose operation *Op-MergeForward* to merge m2 into m1, i.e., we eliminate the dependency by merging the two or more updates into one which will be processed by the DWMS in one refresh process [1].

The dependency correction algorithm **SDC** for a static snapshot of UMQ is shown in Figure 9:

```
Procedure StaticCorrection()
Begin
    Given a Snapshot of UMQ;
    while(there exists unsafe dependency in this snapshot)
    begin
        pick one unsafe dependency;
        if it forms a cycle with other dependencies then
            Op-MergeForward all related updates;
        else
            Op-Precede one message to another;
    end;
end;
```

Figure 9: SDC: Static Dependency Correction Algorithm

Figure 10 depicts the steps of SDC when applying to a snapshot of UMQ. First, SDC finds that the **CD** 1←4 is **unsafe**, the it precedes 4 before 1. Second, SDC discovers that **SD** 4←3 becomes **unsafe** now, it precedes 3 before 4. Finally, there is a cycle between 4 and 6, then SDC merges them. Now all the dependencies are now **safe**, thus the SDC stops.

**Proof of Termination of SDC:** Assume if the algorithm does not stop, there must be some corrected dependencies turning back to unsafe since there are finite unsafe dependencies. In this case, a cycle is found and related updates are merged. This results in the reduced number of updates. Thus in the worst case we get one *big* update that contains all original updates and even though the algorithm would still stop.

We can further conclude that the termination means there exists no **unsafe** dependency, which

[1] We have developed advanced algorithms for VM/VS/VA to process multiple updates in a time, which is out of scope of this paper. This is why we mention that the processing of batch updates is relatively straightforward and left for future work.

corresponds to a **legal order**. By Def 10, we know that our *static dependency correction algorithm* is **correct**.

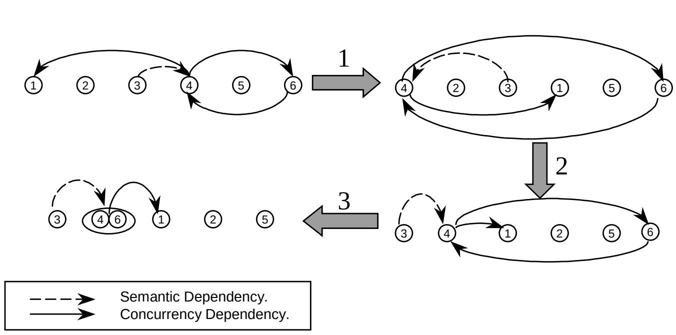


Figure 10: Example of Dependency Correction

### 6.3 *Dyno* Solution: Pulling It All Together

With the aid of dependency detection strategies in Section 6.1.3 and the SDC (static dependency correction) algorithm in Section 6.2, we now propose our *Dyno* solution for dynamic dependency detection and correction. The *Dyno* first uses *pre-exec detection* strategy **before** starting the processing of the head update in UMQ. *In-exec detection* is used **during** the DW maintenance. Upon detection of any unsafe concurrency dependency, the *Dyno* solution uses SDC to turn any unsafe dependencies to safe.

Figure 11 shows a detailed flow chart of the *Dyno* algorithm. Each step is identified by a number. Just before the process of the head update in UMQ, **Step 1** detects all dependencies and constructs a dependency graph given a snapshop of UMQ. **Step 2** checks if the head update in UMQ is involved in any unsafe dependencies. **Step 3** applies *static dependency correction algorithm* to find a legal processing order. In **Step 4** we start processing of that head update. During the maintenance, *in-exec dependency detection* (not shown in the flowchart) method is employed to detect any new unsafe concurrent dependency by a broken query scheme. If the current process is aborted, **Step 5** undoes any effect of this aborted process from step 4 and goes to step 1. **Step 6** uses a *post-exec static detection* method after successful processing. **Step 7:** If any unsafe concurrent dependency has been detected by *post-exec dependency detection*, then Step 7 undoes any effect caused by the completed process from step 4 and goes to step 1. **Step 8:** The head update is removed by Step 8 from UMQ and go to step 1.

The right side of Figure 11 shows the detailed steps to get an "executable" head update, i.e., an update not involved in any unsafe dependency. It will first construct the dependency graph and check if the head update is involved in any unsafe dependencies. If so, it applies SDC to correct the dependencies thus making the first head update executable. Note that all of these are done in a static

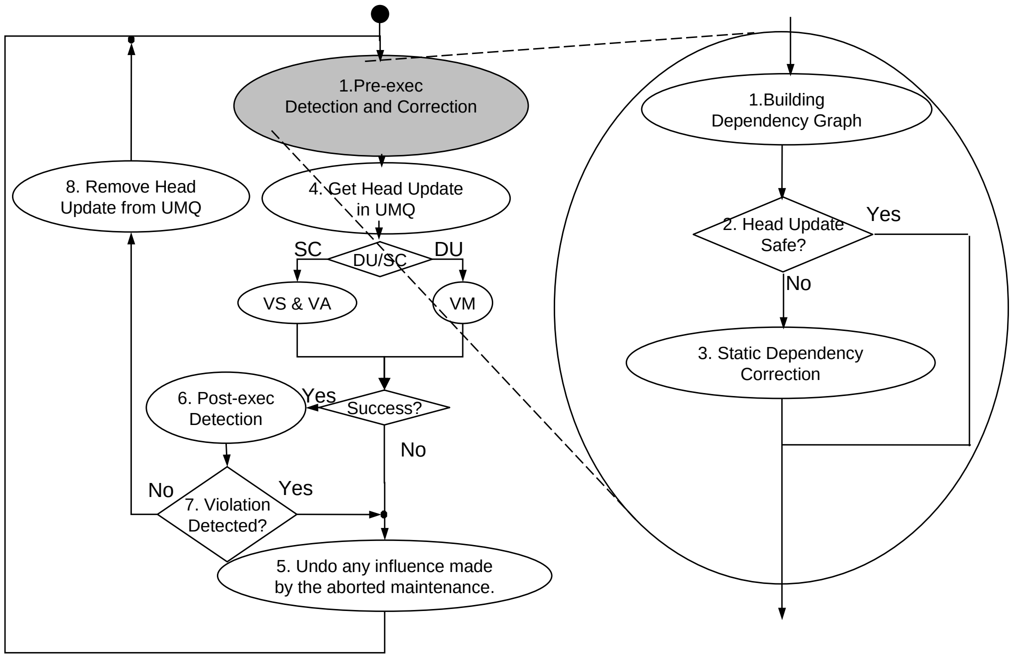


Figure 11: The *Dyno* Flowchart

snapshop of the UMQ.

### 6.4 Termination and Correctness of the Dyno Solution

We have already proven the termination and correctness of our *static dependency correction strategy* (see Figure 9) given a static snapshot. We now briefly prove the correctness of *Dyno* in a dynamic context with further consideration of the new incoming updates.

We first look at the termination. The only possibility that may cause the *Dyno* infinite is that (1) there are unlimited schema changes at ISs, and (2) the incoming updates always aborts the DW maintenance process causing the DW never to be refreshed. Such a degenerate condition is highly unlikely and we believe it is in general not addressable.

Similarly, the *static dependency correction algorithm* fails only when some new concurrent updates arise. In this case, our *in-exec detection* and *post-exec detection* would be sufficient to detect any of new unsafe dependencies resulting in recorrection of them. Given limited updates, the processing order that the *static dependency correction algorithm* generates would finally succeed. Thus we can conclude that *Dyno* is correct.

### 6.5 Consistency Level Achieved by *Dyno*

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

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

Clearly, *Dyno* achieves "Strong Consistency". This is because, first, the reordering will keep all semantic dependencies thus the DWMS would process the IS updates in the same order as they commit at IS. Second, however, since *Dyno* would merge updates thus not every update corresponds to a distinct DW state. Lastly, the correctness of *Dyno* guarantees that each state of DW is correct as long as VM/VS/VA are correct. In conclusion, *Dyno* reaches "Strong Consistency".

# 7 Experimental Evaluation

## 7.1 Experiment Testbed

We have implemented the *Dyno* algorithm in our data warehousing system DyDa [CZC+01]. The integration of this module makes DyDa capable of maintaining the DW even under concurrent data updates and schema changes. In the DyDa system, we apply the SWEEP [AASY97] algorithm to compensate for the concurrent DUs, thus solving the first two dependencies problems (see Section 4.1). DyDa is implemented using java, jdbc to connect to Oracle8i as DW servers and IS servers. In our experimental setting, there are four information sources with one relation each. Each relation has two attributes and contains 10, 000 tuples. There is one materialized join view defined upon these four IS relations. We have conducted our experiments on a Pentium III PC with 256M memory, running Windows NT and Oracle8i.

## 7.2 Study of DU Processing

Note that our *Dyno* algorithm extends the system's functionality to now also deal with concurrent SCs. We first study the overhead such extended functionality may bring to the normal system's DU processing. Clearly, any extra cost would come from the detection process of *Dyno*, i.e., it would examine the UMQ before/after the processing of current DUs, but an abort would not occur without SCs.

The static *pre-exec detection strategy* involves the construction dependency graph of both **CD**s and **SD**s, which has a time complexity of $O(n^2)$. However, since if there are only DUs in UMQ, no concurrent dependencies would exist thus the maintenance query never breaks. Thus we can optimize the dependency graph construction by adding a pre-scan of the UMQ. Whenever there is no SC, we can safely avoid the construction the whole dependency graph thus reducing the time complexity to O(n). The *post-exec detection strategy* also has the time complexity of O(n). That is so because it will trace the relation that the maintenance query operated upon by scanning the UMQ once. The *in-exec detection strategy* would actually not be activitated since there is no SC thus no abort would ever occur.

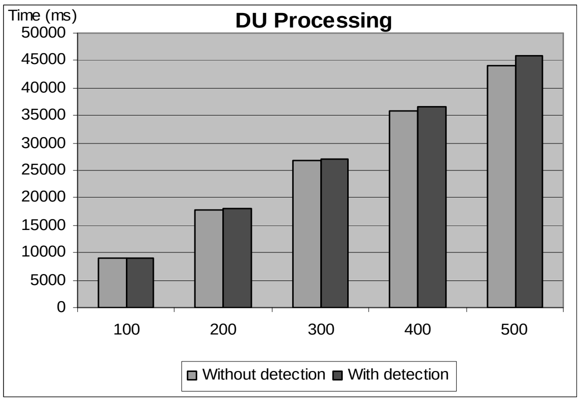


Figure 12: Comparison of DU Processing with/without Detection Enabled

Figure 12 depicts the total DW maintenance cost measured in milliseconds (depicted on the y-axis) with or without detection enabled under different numbers of IS DUs (depicted on the x-axis). From the result, we find that the detection cost (both pre-exec and post-exec detection) is small for any number of DUs. It is actually less than 5% of the total cost in the cases observed. We thus conclude that the *Dyno* algorithm imposes little extra cost on DU processing.

### 7.3 Cost of Broken Query

Recall that a maintenance query would break due to the existence of some concurrent SCs. There are two kinds of broken query problems, namely, a DU processing is aborted by an SC or an SC processing is aborted by another SC. Once such a broken query occurs, the DWMS has to drop all previous maintenance work and redo it again imposing an extra cost on DW maintenance which we refer to as the cost of the broken query, i.e., the maintenance abort cost.

In this experiment, we now study the cost of these two kinds of aborts. To observe the exact cost of a broken query, we employ simple case here, i.e., one DU processing aborted by an SC and an SC processing aborted by another one. Three different environment 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 [2]. This can be considered as the minimum cost since there is no concurrency handling cost. Second, we apply *pre-exec detection strategy* to discover the potential concurrency before processing thus trying to avoid the occurrence of any broken query. Third, we disable *pre-exec detection strategy* now only after the broken query occurs and is detected, do we correct the dependencies and restart the maintenance, thus more aborts may occur.

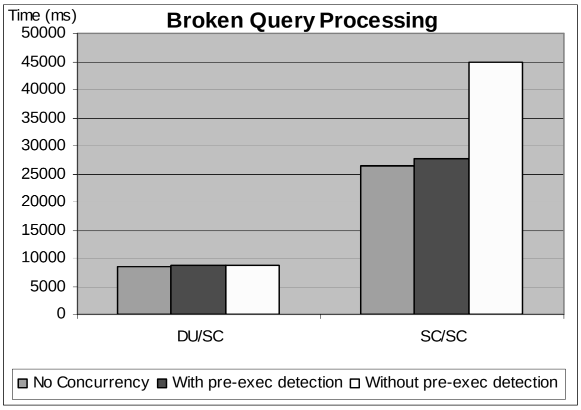


Figure 13: Cost of Broken Query

Figure 13 depicts the total DW maintenance cost in terms of milliseconds (depicted on the y-axis) under different kinds of concurrent updates. From the figure, we find that the cost of aborting SC processing is significant compared to that of DU processing, i.e., the white box of SC/SC (where the abort of SC occurs) is much higher than the other two, while in the case of DU/SC, the white box of DU/SC (where the abort of DU occurs) is similar to the other two. The reason is that the VS&VA modules involve rather complex operations for SC processing. Thus it is costly to redo the SC maintenance process. Secondly, we find that the *pre-exec detection strategy* does indeed help to avoid some of the broken queries to occur, i.e., the black and grey boxes are similar in both cases.

### 7.4 Mixed Update Processing

In the previous two experiments, we observe that the most expensive cost besides the normal DW maintenance processing is the abort of the SC processing, i.e., the cost of both dependency detection (Figure 12) and the abort of DU (Figure 13) is minimum in comparsion. Also we find that the *pre-exec detection strategy* does help to avoid the aborting of the SC processing (Figure 13). However, as mentioned before, the broken query may still happen even when employing *pre-exec detection strategy.* We now study how this strategy helps in a truly mixed update environment with both DUs and SCs.

We employ a mixture of updates with 5% of them being SCs and 95% being DUs, both randomly generated over all four ISs. In this experiment, we vary the time interval between updates as a

[2] Because the IS updates occur after the completion of DW maintenance.

parameter. When the time interval is short, the DWMS is under a high load and will receive more IS updates given a certain time thus resulting in more concurrency problems. In contrast, when the time interval is long, the DWMS is under a low load, i.e., less updates are received within a certain period and less concurrency would occur. Figure 14 depicts two lines for the maintenance cost with or without pre-exec detection, respectively, when varying the delay between these updates from 100ms to 350ms.

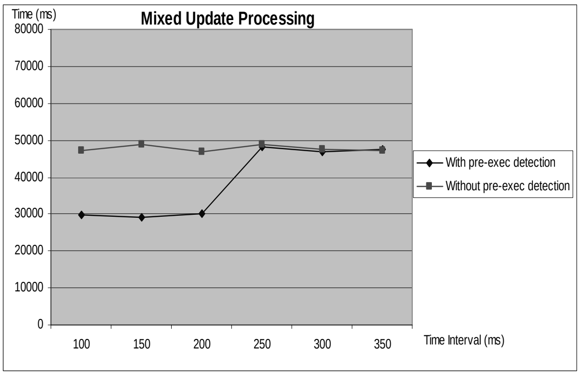


Figure 14: Mixed Update Processing

From Figure 14, we find that the system without pre-exec detection can never prevent the abort of SC processing, thus leading to high maintenance cost even for small time interval. In comparison, when the time interval between updates is short, the system with the *pre-exec detection* is able to discover the potential concurrent SCs thus avoiding the abort. The reason is that if an SC is propagated to the DWMS quickly, it's more likely that it can be caught during the *pre-exec detection* process. This leads to the observation that the *pre-exec detection* helps under a high system load. When the time interval increases, a concurrent SC may escape the pre-exec detection period thus the broken query problem is unavoidable.

# 8 Related Work

Maintaining materialized views 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. Such algorithms for maintaining a data warehouse under source data updates are called view maintenance algorithms [CGL+96, GL95, LMSS95]. 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 ISs is called View Synchronization.

Thereafter, View Adaptation (VA) [GMR95, NR99] incrementally adapts the view extent after the view definition has been redefined. View self-maintenance [QGMW96, Huy97] is one approach to maintain DW extent trying to limit the access the base relations. [MRSR01] proposes an optimization for multiple view maintenance by using some intermediate views with common subexpressions.

In approaches that need to send maintenance queries down to the IS space, concurrency problems can arise [ZGMHW95]. In their work, they introduced the ECA algorithm for incremental view maintenance under concurrent IS data updates restricted to a single IS. In Strobe [ZGMW96], they extend their approach to handle multiple ISs. 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 SWEEP. [SBCL00] proposes to only materialize delta changes of both ISs and views with timestamps, so the view is able to asynchronously refresh its extent. They also introduce an interesting propagation algorithm that could significantly reduce the number of compensation queries. However, none of them can handle the source schema change and the system would fail when such schema change occurs.

Our early work [ZR99] studies the problems of the DW refresh caused by the concurrency of IS data updates and schema changes. However it assumes that each IS reports a schema change and waits for permission from the DWMS before it commits. In other words, the ISs are assumed to be fully cooperative. Our work now drops this restricting assumption. [CR00] employs a multiversion concurrency control algorithm to handle the concurrency problem assuming there are enough system resources to materialize some extra data.

# 9 Conclusion

In this paper, we characterize the DW maintenance anomaly problem corresponds to the fact that the unsafe dependencies exist between update messages in a fully concurrent environment. We analyze and categorize the different types of dependency relationships between source updates. Then we propose different types of detection methods of these dependencies. Based on the dependency graph, we introduce the basic dependency correction solution in a static environment. Finally we propose and prove a complete solution strategy named *Dyno* which enables a DWMS to handle concurrent DUs and SCs in a dynamic context. We have implemented the *Dyno* solution in our DyDa system. The experimental results show that our new concurrency handling strategy imposes a minimal overhead to allow for this extended functionality. Our advanced concurrency detection strategy even succeeds in reducing the expensive cost of maintenance aborts and hence restarts of schema change processing under a high system load, thus achieving overall improved maintenance performance. Our future work includes batch updates processing and multiple view maintenance.

## 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.

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

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

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

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

[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.

[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.