WPI-CS-TR-99-20 July 1999

DyDa: Dynamic Data Warehouses Maintenance in a Fully Concurrent Environments

by

Xin Zhang
Elke A. Rundensteiner

Computer Science
Technical Report
Series

WORCESTER POLYTECHNIC INSTITUTE

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

Contents

DyDa: Dynamic Data Warehouse Maintenance in a Fully Concurrent Environment

Xin Zhang and Elke A. Rundensteiner

Department of Computer Science
Worcester Polytechnic Institute
Worcester, MA 01609-2280
xinz | rundenst@cs.wpi.edu

Abstract

Data warehouses (DW) are an emerging technology to support high-level decision making by gathering information from several distributed information sources (ISs) into one repository. In dynamic environments such as the web, DWs must be maintained in order to stay up-to-date. Recently proposed view maintenance algorithms tackle this problem of DW management under *concurrent data updates* (DU) at different ISs, whereas the EVE system is the first to handle *non-concurrent schema changes* (SC) of ISs. However, the concurrency of schema changes by different ISs as well as the concurrency of both interleaved schema changes (SC) and data updates (DU) still remain unexplored problems.

In this paper, we propose a solution framework called DyDa that successfully addresses both problems. The DyDa framework detects concurrent SCs by the broken query scheme and conflicting concurrent DUs by a local timestamp scheme. A fundamental idea of the DyDa framework is the development of a two-layered architecture that separates the concerns for concurrent DU and concurrent SC handling without imposing any restrictions on the fully concurrent execution of the ISs. At the lower (query engine) level of the framework, it employs a local correction algorithm to handle concurrent DUs, and a local name mapping strategy to handle concurrent rename-SCs that rename either attributes or relations at the IS space. At the higher (DW management) level, it addresses the problem of concurrent (drop-SC) operations that drop attributes or relations from the IS space. For the later, the view synchronization (VS) algorithm is modified to keep track of view evolution information as needed for handling sequences of concurrent SCs. We also design a new view adaption (VA) algorithm, called Map-VA, that incrementally adapts the view extent for a modified view definition even under interleaved SCs and DUs. Put together, these algorithms provide a comprehensive solution to DW management under concurrent SCs and DUs. This solution is currently being implemented within the EVE data warehousing system.

1 Introduction of Data Warehousing

1.1 Background — DW Management

Data warehouses (DW) are built by gathering information from several ISs (Information Sources) and integrating it into one virtual repository customized to users' needs. Data warehousing [GM95, MD96] has importance for many applications in large-scale environments composed of heterogeneous and distributed ISs, such as travel services, E-commerce, decision support systems, web-site management and other web related applications.

In such modern 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 and without possibly any concern for how this may affect the DW defined upon them. They generally are not aware of nor willing to wait until the DW manager has successfully processed all previous changes from other ISs and updated the warehouse appropriately.

There are three types of tasks related to DW management in distributed environments. The most popular research area is the incremental maintenance of materialized views under distributed ISs. Such view maintenance (VM) algorithms [ZDR99, AAS97, ZGMW96, ZGMHW95] maintain the extent of the data warehouse whenever a data update (DU) occurs at the IS space. The second research area called view synchronization (VS) [NLR98, RLN97, LNR97] is concerned with evolving the view definition itself whenever there is a schema change (SC) of one of the ISs that results in a view definition to become undefined. The third research area, referred to as view adaptation (VA) [GMR97, MD96, NR99] is concerned with adapting the view extent incrementally after the view definition has been changed either directly by the user or indirectly by a view synchronization module.

Materialized view maintenance (VM) is the only area among those three that thus far has given attention to the problem of *concurrency of (data) updates* at ISs. Our recent work on Coop-SDCC [ZR99] is the first to begin to study the concurrency problem of both data updates and schema changes in such environments. The Coop-SDCC approach integrates existing algorithms designed to address the three DW management subproblems VM, VS and VA into one system by providing a protocol that all ISs must abide by and that as consequence enables them to correctly co-exist and collaborate. This solution has however the limitation of requiring information sources to cooperate by first announcing an intended SC, then waiting for the DW to finish servicing any outstanding requests, before being permitted to execute the schema update at the IS.

In this paper, we overcome the limitation of this previous solution. We now propose the DyDa framework that can handle fully concurrent SCs and DUs without putting any restriction on the timing of when an SC is allowed to take place nor requiring any cooperation from the ISs. Below we now illustrate what kind of problems occur when we release the restriction of cooperative ISs.

1.2 Motivating Example of the Concurrent SC Problem

During the process of managing a data warehouse which includes sending down various query requests to different ISs, new updates (DU or SC) could occur concurrently at some of the ISs that haven't yet been seen by the DW middleware. We call such updates *concurrent updates.* A concurrent *DU* will result in an *incorrect query result* returned by an IS [ZR99, QW97], whereas as we will illustrate below a concurrent SC results in a *broken query* that cannot be processed by the ISs, i.e., an error message.

Example 1 *Assume we have two information sources IS1 and IS2 with relations R and S, respectively. The view V of the data warehouse is defined by the SQL query in Equation (1). Assume a data update DU happens at R in IS1. This requires us to send the incremental view maintenance query Q [AAS97, ZGMW96] defined in Equation (2) down to IS2 to perform a remote join.*

$$\begin{array}{ll} \textsf{CREATE VIEW} & V\ \textsf{AS} \\ \textsf{SELECT} & IS1.R.A,\ IS1.R.B, \\ & IS1.R.C,\ IS2.S.D \\ \textsf{FROM} & IS1.R,\ IS2.S \\ \textsf{WHERE} & IS1.R.A\ =\ IS2.S.A \\ & AND\ IS1.R.B\ <=\ 10 \end{array} \tag{1}$$

$$\begin{array}{ll} \textsf{SELECT} & DU.A,\ DU.B,\ DU.C,\ IS2.S.D \\ \textsf{FROM} & DU,\ IS2.S \\ \textsf{WHERE} & DU.A\ =\ IS2.S.A \\ & AND\ DU.B\ <=\ 10 \end{array} \tag{2}$$

If during the transfer time of the query Q to IS2, IS2 has a schema change, e.g., IS2.S.D is dropped, then the query Q can no longer be processed by IS2. We call a query, such as Q, a broken query, because the query result cannot be computed by IS2. The data warehouse can no longer be maintained correctly because the incremental view maintenance process is based on obtaining the results of the maintenance queries. A similar broken query problem also holds for view adaptation queries [GMR97, NR99] as they are also send down to the IS space (or for that matter for any query send down to the IS space) and thus face the issue of the IS schema changing unexpectedly.

1.3 Our Solution Approach — The DyDa Framework

In this paper, we present a general approach called the **Dy**namic **Da**ta warehouse management (DyDa) framework that is the first to solve the above problem. DyDa maintains views in a data warehouse defined over a space consisting of dynamic ISs with fully concurrent SCs and DUs without posing any restrictions on the ISs. In other words, the restrictive assumption of the previous and only other attempt of solving this problem [ZR99], namely the assumption of cooperative ISs that delay the schema evolution of their database until receiving permission from the data warehouse, is dropped.

When an unexpected schema change happens in the IS space, then the view adaptation queries sent by the VA algorithm and the incremental view maintenance queries send by the VM algorithm down to the ISs may be broken. The DyDa framework incorporates a strategy to detect the cause of such broken queries. Depending on the identified cause of the problem (i.e., the type of concurrent SC), the DyDa framework incorporates several strategies to handle it and maintain the data warehouse successfully.

1.4 Contributions of this Work

- We characterize the problem of maintenance under concurrent SCs, which we call the broken-query problem.

- We devise a strategy for the detection of concurrent SCs based on the broken query concept.

- We analyze the VM, VS and VA algorithms in the literature to determine if and how they are affected by concurrent SCs.

- We identify the problem of an incorrect view extent being generated by the VA algorithm for an SC that aborts the on-going VM process of previous DUs.
- We introduce the overall solution framework called DyDa that adapts a two-layered architecture that separates the concerns for concurrent DU and concurrent SC handling without imposing any restrictions on the fully concurrent execution of the ISs.
- We design a VA algorithm called Map-VA that correctly (and incrementally) adapts a view extent even under multiple SCs and interleaved DUs and SCs.
- We revise the VS algorithm [RLN97] to keep version-information and thus to provide the information to the Map-VA algorithm to enable it to perform concurrent view adaptation.
- We prove the correctness of our solution approach, in particular, the Map-VA algorithm.
- We extend the taxonomy of consistency levels of the DW from the literature to cover concurrent environments and IS schema changes, and then classify the consistency level of the DW reached by the DyDa framework.

1.5 Outline of Paper

In the next section, we define basic concepts needed for the remainder of the paper. We present the formal definitions of a maintenance-concurrent update and broken query in Section 3. Section 4 describes the architecture of the DyDa solution framework in detail. Section 5 explains the two levels of concurrency control used in the DyDa framework. Section 6 discussed the properties of concurrent updates. Section 7 proposes a new VA algorithm designed for the DyDa framework and explains why and how it solves the concurrent SC problem. Section 8 reviews related work. In Section 9, we conclude and discuss future directions of our work.

2 Background Material

2.1 Notations

Table 1 defines the main notations that will be used in the remainder of this paper. A sequential number, unique for each update, will be generated by the DyDa system whenever the IS update message reaches the system. This number is denoted by n in Table 1.

A **schema change (SC)** denotes a primitive change that occurs at the schema of an IS. We distinguish between three types of schema changes: The SCs that rename attributes or relations at ISs, called **RenameSC**; the SCs that delete attributes or relations from ISs, called **DropSC**; and the SCs that add attributes or relations to ISs, called **AddSC**.

Notation	Meaning
IS[i]	Information source with subscript i.
X(n)[i]	X is an update (SC or DU) from IS[i] at sequence number n. Sequence number of update is unique for all updates of all ISs.
Q(n)	Query used to handle update X(n)[i].
Q(n)[i]	Sub-query of Q(n) sent to IS[i].
QR(n)[i]	Query result of Q(n)[i].
QR(n)	Query result of Q(n) after re-assembly of all QR(n)[i] for all i.

Table 1: Notations and Their Meanings.

2.2 Assumption

In this paper, we keep the network FIFO assumption for the DW system built on top of distributed ISs from [ZGMHW95, ZGMW96, AAS97, ZDR99].

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

Lemma 1 *An SC cannot be concurrent with any DU that happened at the same IS.*

The proof is straightforward. Because they both come from the same IS, they have to have some ordering among themselves by Assumption 1.

Lemma 2 *The order of receiving DUs and SCs by the DW is guaranteed to reflect the partial orders in which the DUs and SCs from the same IS actually happened.*

Lemma 2 is correct due to Assumption 1. Note that ordering among updates at different ISs is however not guaranteed to be reflected in their arrival order at the DW.

Theorem 1 *The order of receiving DUs and SCs by the DW cannot be guaranteed to reflect the order in which the DUs and SCs actually happened in the IS space when the DUs and SCs come from different ISs.*

Intuitive Proof: The ISs are distributed and connected to the DW via a network, and the delay of the network communication between different ISs may be different, so we cannot determine which one is the first of two DUs from different ISs.

2.3 Consistency Levels of the Data Warehouse State

Zhuge et al. [ZGMHW95, ZGMW96] define different notions of consistency of a view extent depending on how the updates are incorporated into the view at the data warehouse. Since these definitions are somewhat under specified, it becomes necessary to further refine them, especially for the definition of IS space states and the order of IS space states in the context of distributed information sources. [ZDR99] proposed a new set of definitions of consistency levels for DWs built upon distributed information sources. We here define additional consistency levels for DWs management under SCs and DUs from distributed ISs.

Definition 1 *A* **legal IS space state** *from the DW point of view is defined recursively as follows:*

a. The initial state of an IS space, ISS_0, is a legal IS space state.

b. For a sequence of actual DUs at one IS_i for some i, denoted by $DU_{i,1}$, $DU_{i,2}$, ..., $DU_{i,n}$, an IS space state generated by applying any subsequence of $DU_{i,1}$, $DU_{i,2}$, ..., $DU_{i,k}$ with $1 \leq k \leq n$ to this IS_i is a legal IS space state.

c. For any pair of data updates DU_i and DU_j from different ISs IS_i and IS_j with $i \neq j$, both the IS space state generated by applying DU_i to a legal IS space state is called legal, and the IS space state generated by applying DU_j to a legal IS space state is called legal.

For example, if there are two data updates DU_1 and DU_2 from different ISs of the IS space, then there could be three legal IS states, which are we only have committed DU_1, we only have committed DU_2 or we have already committed both.

Definition 2 *We consider a DW state to be legal if the DW state can be generated from a legal IS space state.*

Definition 3 *A DW state is called quiet if there is no un-handled DU in the data warehouse.*

Definition 4 *A quiet IS space state is a legal IS space state that corresponds to a quiet DW state by Definition 3.*

Definition 5 *A* **state order tree** *for a given information space* ISP *and a set D of data updates DU_i with i = 1, ..., k, applied to ISs in the space* ISP *is defined to be a rooted acyclic directed tree where each node N represents a legal IS space state ISS_i by Definition 1 and each directed edge E from a node $\mathbf{ISS}_j$ to a node $\mathbf{ISS}_k$ labeled with the data update DU_l indicates that IS space state $\mathbf{ISS}_k$ can be derived from the IS space state $\mathbf{ISS}_j$ by applying the data update DU_l. The root of the tree is the initial state of the IS space* ISP *(or the quiet IS space state from which all updates in D started), denoted by ISS_0. The leaves of the tree are quiet IS space states reachable from ISS_0 of* ISP *by applying all updates in D in some order as long as they generate legal IS space states.*

The state order tree effectively represents the different legal application orders of a set D of data updates to an IS space I.

Definition 6 *Every directed path from the root to a leaf node of a state order tree as defined in Definition 5 is said to correspond to a* **legal order** *of* **IS space states**.

Definition 7 *Five consistency levels of the DW in distributed environments from the DW point of view can now be defined as follows:*

- *Convergence: In any quiet state of the DW, the DW state is legal by Definition 2.*
- *Weak Consistency: All states of the DW are legal by Definition 2.*

- *Consistency: Weak consistency and the DW states correspond to* **one** *legal order of the IS space states as defined by the state order tree given in Definition 6.*
- *Strong Consistency: Consistency and convergence.*
- *Complete Consistency: Strong Consistency and all the states in a legal order of the state order tree as defined by Definition 6 have corresponding legal DW states.*

So far the DyDa framework can reach the Strong Consistency level.

3 Problem Definition

3.1 Definition of the Maintenance-Concurrent Problem

While the concept of maintenance concurrency has previously been defined for data updates only [ZDR99], we now extend it to incorporate chema changes.

Definition 8 *The query result QR(n) of a data update DU(n)[i] is the result of the view maintenance query Q(n) generated by the VM algorithm in order to maintain the extent of the view.*

Definition 9 *The query result QR(n) of a schema change SC(n)[i] is the result of the view adaptation query Q(n) generated by the VA algorithm in order to adapt the extent of the view.*

Definition 10 *Let $X(n)[j]$ and $Y(m)[i]$ denote either DUs or SCs on IS[j] and IS[i] respectively. We say that the update X(n)[j] is* **maintenance-concurrent** *(in short concurrent) with the update Y(m)[i], denoted X(n)[j] $\vdash$ Y(m)[i], iff: i) $m < n$, and ii) X(n)[j] is received at the DW* **before** *the answer QR(m)[j] of update Y(m)[i] is received at DW. We say that the update X(n)[j] is* **maintenance-concurrent**, *if X(n)[j] is* **maintenance-concurrent** *with at least one update Y(m)[i].*

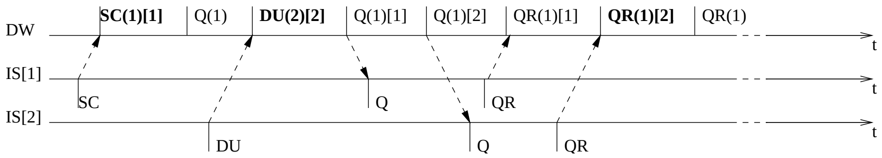

Figure 1: Time Line for a Maintenance Concurrent Data Update.

Figure 1 illustrates the concept of a **maintenance-concurrent** update defined in Definition 10 with a time line. We note that in Figure 1, the messages only get time stamps assigned at the DW layer. That means that the **maintenance-concurrent** update is defined with respect to the data warehouse layer instead of the IS layer. Assume we have one data warehouse DW and two information sources IS[1] and IS[2]. First, there is a schema change SC at IS[1]. Then, there is a data update DU at IS[2]. From the figure, we can see that the SC is received by the DW before the DU, but DU occurs at IS[2] before the adaptation query Q(1)[2] of SC arrives at IS[2], that is, DU occurs before the query result QR(1)[2] arrives at the DW. So, here the DU is maintenance concurrent with SC by Definition 10.

There are four types of **maintenance-concurrent** updates listed in Table 2 in the order of the easiest to the hardest in terms of handling them.

Type	Meaning	Denoted By[1]
I	A **maintenance-concurrent** DU happened when handling a DU	$DU_h - DU_{mac}$
II	A **maintenance-concurrent** DU happened when handling a SC	$SC_h - DU_{mac}$
III	A **maintenance-concurrent** SC happened when handling a DU	$DU_h - SC_{mac}$
IV	A **maintenance-concurrent** SC happened when handling a SC	$SC_h - SC_{mac}$

Table 2: Four Types of Maintenance-Concurrent Updates.

3.2 Definition of Broken Query Problem

Definition 11 *A query is called a* **broken query** *if it cannot be processed because the schema of the IS expected by the query is not consistent with the actual schema of the IS encountered during query processing.*

We distinguish between three types of broken queries based on the type of SC (see Section 2.1) causing the problem as well as the characteristics of the IS space available for the system to deal with this problem.

- **Type 1: Broken Queries caused by a RenameSC.** In this case the data is still there but only the accessing interface of this IS relation has changed. We can use a name mapping strategy to get to the desired data.

- **Type 2: Broken Query caused by a DropSC with Replacement.** While the data is really gone, we are able to find the required information from an alternate source that holds duplicate data.

- **Type 3: Broken Query caused by DropSC without Replacement.** The data has really been dropped, and the system is not able to identify an alternate source for the data.

AddSC will not result in broken queries as they do not interfere with the interpretation of any existing query.

Assumption 2 *A broken query as defined in Definition 11 will be returned by the IS space as an empty query result with a "broken query" error message.*

[1] DU_h (or SC_h) denotes the DU (or SC) that is currently being handling by the DyDa system. DU_{mac} (or SC_{mac}) denotes the DU (or SC) that is a **maintenance-concurrent** DU (or SC).

In other words, we assume the ISs are not smart enough to analyze a broken query and return a partial query result to the data warehouse. In the following sections, we will show that the proposed solution framework DyDa can handle all three types of broken queries, while keeping the data warehouse up-to-date.

Theorem 2 *If a* **maintenance-concurrent** *SC breaks a maintenance query sent by the VM algorithm with the term broken query defined in Definition 11, then the* **maintenance-concurrent** *SC also makes the view definition maintained by VM undefined.*

Proof: From [AAS97, ZGMW96, ZGMHW95] we can see that the view maintenance query is a part of the view definition query. As illustrated in Example 1, the maintenance query 2 utilizes components of the view definition query 1 only besides the DU which is not generated from the IS but instead send down from the DW to the IS. So, if the view maintenance query is broken, then the original view definition is broken too. In Section 4, we use this property to decide the overall state transitions of the DyDa system.

4 The DyDa Framework

4.1 Overall Architecture

The DyDa framework is divided into three spaces: DW space, middle space, and IS space (see Figure 2).[2] 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 information sources and their corresponding wrappers. Wrappers are responsible for translating queries, returning query results, and sending notifications of DUs and SCs of the information sources to the middle space.

Symbol	Meaning
V	View definition affected by either Schema Change (SC) or Data Update (DU).
V'	Evolved view definition of affected view V.
DW	Data warehouse.
ΔV	Incremental view extent of data warehouse, i.e., set of tuples to be inserted into or removed from the extent of view V.
CQR	Query result that is returned by QE.
VS-VA	All information VA module requires from VS module for view adaptation. It includes: V, V', Meta-knowledge, Synchronization-mapping.
VAQ	View Adaptation Query.
VAQR	View Adaptation Query Result.

Table 3: Meaning of Symbols Used in Figure 2.

[2]Figure 2 depicts the modules of our proposed framework and the data flow between them. Table 3 lists the meaning of each symbol that appears in the framework.

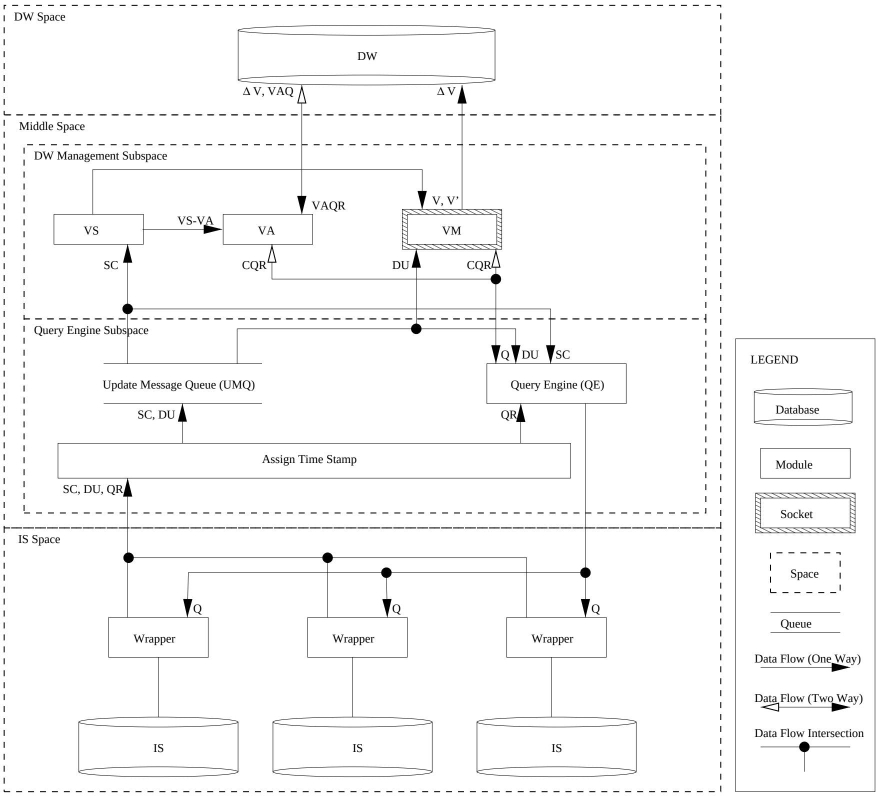

Figure 2: Architecture of DyDa Framework

The middle space is the integrator of the DyDa framework. It can be divided into two subspaces. The higher-level subspace is called the **DW management subspace**. All the DW management algorithms, like VS, VA and VM, are located in this subspace. The lower-level subspace is called the **query engine subspace**. This subspace is composed of the Query Engine (QE) and its supporting modules, namely, the Update Message Queue (UMQ) module and Assign Time Stamp module. The two subspaces effectively correspond to two different levels of concurrency control as will be presented in Section 5. The key idea is that **maintenance-concurrent** DUs will be handled locally by the QE module at the lower-level of the middle space, so that DW management algorithms at the upper-level, such as VS, VM and VA, are shielded from and will never be aware of any **maintenance-concurrent** DUs. The correctness of and justification for this two-layered concurrency control approach will be discussed in depth in Section 5.

4.2 State Transition of Higher Level of the DyDa Framework

The DyDa framework focuses on how to handle interleaved SCs and DUs. This requires the collaboration between the VS, VA and VM modules in terms of control flow and data exchanges as outlined below.

Figure 3 shows the state transition diagram of the DyDa framework at the higher-level of the middle space, hence focusing on **maintenance-concurrent** SCs only. There are four types of

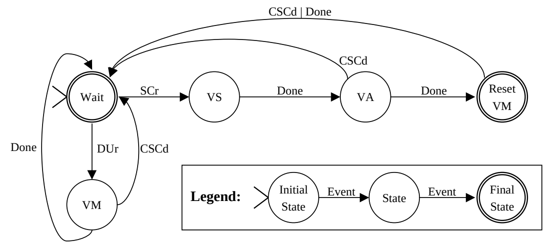

Figure 3: State Diagram of DW Management Subspace for Concurrent SCs.

Event	Meaning
SCr	the event of receiving a notification of an SC.
CSCd	the event of detecting a concurrent SC.
DUr	the event of receiving one DU.
Done	state finished normally with no concurrent SC.

Figure 4: Events of Transition Diagram of DW Management Subspace.

events shown in the diagram as depicted in Figure 4. As we can see from Figure 3, the *DW management* subspace starts in the *Wait* state waiting for an incoming SC or DU. Once an SC is received, the VS module will generate synchronized view definitions for all view definitions affected by this SC. After the view synchronization, the state will change to the *VA* state. In the *VA* state, the VA module will adapt the view extent to account for the modification of its view definition. If a concurrent SC is detected during this extent adaptation process, the VA process will be aborted, and the state changes back to the *Wait* state to handle the next update. Otherwise, it goes to the *Reset-VM* state to reset the VM module in order to be able to maintain the new view definitions for future data updates. If a concurrent SC is detected while in the *Reset-VM* state, the state will change back to the *Wait* state. Once a DU is received while in the *Wait* state, the VM module will handle it. If a concurrent SC is detected while in the *VM* state, then the state changes to the *Wait* state.

State	Meaning of State
Wait	*DW management* subspace is idling to wait for SC or DU to happen.
VS	VS module is handling the evolution of views affected by the detected SC.
VA	VA module is adapting the extent of the data warehouse in order to keep the extent consistent with the evolved view definition.
Reset VM	VM module is creating a new instance of VM maintenance process in order to maintain the evolved view.
VM	VM module is maintaining the extent of the data warehouse for a DU.

Table 4: States of Transition Diagram of DW Management Subspace DyDa Framework (Figure 3).

There are two final states in the state diagram (Figure 3), namely, the *Wait* state and *Reset-VM* state. The data warehouse extent is updated in either of these two states. Especially, the *Reset-VM* state will update DW for SCs. Note here that there are three back arcs from *VM* state to *Wait* state, from *VA* state to *Wait* state, and from *VM Reset* state to *Wait* state. They can lead the system to loop once for every **maintenance-concurrent** SC. We require a quiescence time period of the IS space in terms of SCs in order to propagate such updates to the DW extent (e.g., reach the *ResetVM* state), similar to how Strobe [ZGMW96] requires this for concurrent DUs. Note that information sources are relatively stable in terms of their schema, while data updates are likely to be more frequent. Based on this observation the potential infinite wait for SCs is not likely to be any issue in practice for the DyDa system.

4.3 Effect of Maintenance-Concurrent Updates on Existing DW Management Algorithms

Here we briefly analyze how **maintenance-concurrent** DUs and SCs affect the VS, VA and VM modules. In the DyDa framework, we decide to let the query engine (QE) module as explained in Section 4 fix the problem of any **maintenance-concurrent** DU before the query results reach the DW management modules. So, the **maintenance-concurrent DUs** have no effect on the three modules VA, VM and VS.

However, the three modules have a different degree of awareness of **maintenance-concurrent SCs**. There is no concept of **maintenance-concurrent** SCs for the VS module, because the VS module never sends any query down to the IS space. While the VM module will send queries down to the IS space for view maintenance, it assumes the view definition that it is maintaining will not be changed. So if a maintenance query is broken by a **maintenance-concurrent** SC, the VM module has to be reset by the DyDa system (see Section 4) so to work with the newly updated view definition that has been generated by the VS module to take care of that **maintenance-concurrent** SC. The VA module also sends down queries to the IS space to adapt the view extent. If a view adaptation query is broken by a **maintenance-concurrent** SC, then the VA module needs to reinitialize the adaptation process.

5 Two Levels of Concurrency Control of DyDa Framework

Based on the DyDa framework we defined in Section 4, we are going to discuss the properties of concurrent updates. From Lemma 1 (Section 2.2), the real handling order of DUs and SCs from different ISs is not critical for the correctness of the final state of the DW as long as we keep the partial orders. The handling is not required to be same as the receiving order as long as the order of the updates from the same IS is preserved. The DyDa system will handle DUs and SCs one by one, and the order of handling DUs and SCs is same as the order of receiving DUs and SCs at the DW. If two DUs and SCs are received at exactly the same time, then the DyDa system will generate a random order for those updates.

5.1 Low Level Concurrency Control at QE Subspace

Different from the detection of concurrent DUs, which is based on the local timestamp assigned to DUs upon their arrival at the data warehouse [ZDR99], the detection of concurrent SCs is based on identifying when a submitted query is broken and hence returned unanswered (see Section 3.2 and Assumption 2).

In the DyDa framework, in order to separate out the handling of concurrent data updates and schema changes, there are two levels of concurrency control corresponding to the two sub-spaces of the *middle* space. At the lower level of concurrency control (the *query engine subspace*), the concurrent DUs as well as the concurrent SCs of the type **RenameSC** will be handled by the QE. The *DW management subspace* supports the higher level of concurrency control. The management algorithms (e.g., VM, VS and VA) at that subspace cooperate with each other in order to handle schema changes of the type **DropSC**. **AddSC** SCs do not render view definitions undefined and hence do not affect VM, VS nor VA. Thus they do not need to be handled.

5.1.1 Using Local Correction to Handle Maintenance-Concurrent DUs

All queries from the data warehouse modules down to the IS space will first go through the QE module (Figure 2). This includes incremental view maintenance queries, view adaptation queries, or view recomputation queries. Given that all three query types are "extent-related queries", the query engine will use the local correction (LC) algorithm[3] described in [ZDR99] to successfully fix all the side effects of concurrent DUs on these queries before passing the corresponding query results up to the next layer of the system. This results in the concurrent DUs to be transparent to all the modules in *DW management* subspace. And by using the local correction algorithm, there is no possibility to be faced with an infinite wait due to the recursive correction of queries, as is a problem in the Strobe solution [ZGMW96]. Details of the local correction scheme we adopt for DyDa and its proof of correctness are beyond the scope (and space limitations) of this paper, but can be found in [ZDR99].

5.1.2 Using Name Mapping to Handle Maintenance-Concurrent RenameSCs

In addition to the above service, more features have been added to the query engine of DyDa in order to handle concurrent SCs of the type **RenameSC**. In particular, the solution strategy for handling the **RenameSC** is based on a temporary name mapping table inside the query engine. For this, there are two cases we need to consider:

Case 1: The QE detects a RenameSC message received at the UMQ. From then onwards, the QE will first locally in any query send to that information source rename the attributes or relations that have been changed as indicated by the RenameSC. This renaming process is encapsulated inside the QE module. The upper level modules like VS, VA and VM modules will not be aware of it, i.e., the query results they receive still use the original schema and its names.

Case 2: It could also happened that the QE has already send off the queries to the underlying information sources before the RenameSC is detected by the QE. In this case, the QE would receive a broken query message back from the respective information source. Then, QE will search through the UMQ for any RenameSC operation that affects the query. It then modifies the query and tries to process it again. The fact that the RenameSC would be in the UMQ is guaranteed by the FIFO behavior of the network (Assumption 1). When there is no related RenameSC that will affect the query found in the UMQ, then there must be an alternate SC from the same IS where the broken query was send to, i.e., in our case it would be a DropSC. The QE module cannot handle

[3]The LC algorithm so far can handle SELECT-FROM-WHERE queries.

a broken query caused by a DropSC, and hence it will report an error message (i.e., $CSCd$) to the maintenance algorithms in the *DW management* subspace.

Lemma 3 *Whenever there is a concurrent SC that results in a broken query, then the SC causing this problem can be found in the UMQ.*

Proof: If an SC will break a query result, then the SC and the broken query result must come from the same IS, and the SC will have happened earlier than the processing of the query at the IS. So by the Assumption 1 (FIFO network), DW must have received the SC before receiving the broken query. Hence the SC can be found in the UMQ when the broken query is received.

■

If more than one SC related to the query is found in the UMQ, and at least one of them is not RenameSC (i.e., at least one DropSC exists), then the QE cannot handle the broken query anymore. It will report an error message (i.e., $CSCd$) to the *data warehouse maintenance* subspace.

If the modified query based on the RenameSC breaks again due to another RenameSC in the UMQ, the QE module will modify the query and resubmit it until either the query succeeds, or a DropSC is encountered in the UMQ. In the highly unlikely case that the ISs were to continuously issue new RenameSCs, then the QE might get stuck in an longer processing time.

In summary, DyDa solves the type I and II **maintenance-concurrent** problems (see Table 2) at the QE level by the local correction strategy and part of type III and IV **maintenance-concurrent** problems (see Table 2) by a local naming mapping strategy.

5.2 High Level Concurrency Control at DW Management Subspace

Because the QE cannot handle the concurrent DropSC problem, the VS, VM and VA modules in the *DW management subspace* need to handle it.

5.2.1 The VS Module and Maintenance-Concurrent SCs

From Definition 10, we know that VS will never have any trouble with concurrent SCs, because it will not send any queries down to IS space. However, the VS module needs to provide information to the VA module to help VA to adapt the view extent under such concurrent SCs (Figure 2).

From the state diagram (Figure 3) in Section 4.2, we can see that for every SC, the VS module will be called to generate a new version of the view definition to account for this SC. We now propose to extend the VS module to keep track of the versions of all rewritings of the affected view definitions. This would allow VS to provide the VA module with the history of how the view definition evolved in order to adapt it correctly. Information related to each version that is kept includes: the new view definition, the evolution mapping, the schema change that triggered the rewriting, and the meta knowledge used for synchronization. The most recent version will be kept at the head of the version list, and the oldest version will be kept at the tail. Every view definition in the system will keep such a list of versions.

From the point of view of VS, all SCs happen sequentially. If two SCs come to the *middle* space at the same time, they will be assigned a random handling order.[4] In a distributed environment as we are assuming, there is no convenient way to determine when the two SCs happened at the IS space relative to each other (unless they both come from the same IS), plus they may indeed have truly occured at the same time. The more important reason is that the issue of which of the SCs happened first for two SCs from two autonomous ISs is not related to the correctness of the VS, though it may affect the quality of the views generated by VS [LKNR99].

The VS module will evolve a view always based on its most recent version. Once the VA module successfully adapts the view extent, the "versions" of the views can be cleaned up. The detailed algorithm of how to use the "versions" is described in Figure 5.

Algorithm of VS with Versions

```
INPUT:   View definition with version extension;
         SC.
OUTPUT:  Updated view definition.

System is in the VS state in Figure 3
 (VS module is going to be invoked)
Get the SC to be handled by the VS module.
FOR every view V in the system
  Get the most recent version of that view V
  IF V is affected by the SC
    DO View Synchronization on it and make the new view
      definition into the most recent version of that view V.
  END IF
  IF the VS module no longer can evolve V
    we drop this view V.
  END IF
END FOR
```

Figure 5: Algorithm of VS with Versions.

After we apply the *extended VS* module with versions kept for each view, the *VA* module will know the history of how each view definition evolved and thus is capable of generating the corresponding view adaptation queries for it. If the *extended VS* module drops a view as being no longer salvageable, i.e., empty extent, then the *VA* module doesn't need to adapt it.

5.2.2 The VM Module and Maintenance-Concurrent SCs

If VM encounters a concurrent SC, it recognizes that by the fact that it will receive a broken query passed up from the QE module (Section 3.2). Because the VM algorithms in the literature haven't considered thus far how to handle broken queries, we put the responsibility of this handling on the

[4] If two SC changes come to the middle space of DyDa framework at same time, we will pick a preferred order of handling the SCs based on a quality-cost model described in [LKNR99]. The decision of the handling order of a set of SCs is out of the scope of this thesis.

VA module proposed in Section 7. Thus in our system existing VM algorithms from the literature are kept intact. Instead the VM algorithm simply stops (abnormally) and put the DU that it is currently handling into the concurrent relationship set of the SC. That DU will then be handled later by the VA algorithm when it is adapting the extent of the view as discussed in detail in Section 7.

Due to pushing the **maintenance-concurrent** SC handling responsibility outside of VM, all the view maintenance (VM) algorithms, e.g., PSWEEP [ZDR99], SWEEP [AAS97], Strobe [ZGMW96], ECA [ZGMHW95] and [BLT86], could be used as VM module of the DyDa framework.

In the next section, we are going to discuss the concurrency relationship between different DUs and SCs.

6 Detection of Concurrent Relationship Between Updates

We have observed in Section 4 that some of the updates from the ISs are concurrent to each other, while some of them are not. Figure 3 (Section 4) shows that the DyDa system will handle the updates in the order they are received at the DW. During the processing of some updates, the process could be aborted by some concurrent SCs. Then the handling of the updates will be postponed until the VA process of handling the concurrent SCs can also take care of them. In order to do that, we need to remember the association between different updates.

Definition 12 *A* **Concurrent Update Set** *is a set of updates (DU or SC) that all have been aborted by an SC, and need to be handled at the same time when handling that SC.*

The concurrent update set is useful for the View Adaptation algorithm, because we need to know who is concurrent with whom (means which updates have to be handled by the VA together with a given SC due to the SC having broken the handling of those updates).

Definition 13 *A* **Concurrent Relationship Sequence** *is a sequence of updates (DUs or SCs) together with their respective concurrent updates set for each SC in the receive order.*

Example 2 *Assume we have following concurrent relationship sequence:*

DU1, SC1(DU1), SC2(SC1(DU1)), DU2, SC3(SC2(SC1(DU1)), DU2)

This means that SC1 aborts the process of DU1 handling, and SC2 aborts the process of the SC1 handling, so the VA for SC1 has to handle SC1, SC2 and DU1. SC3 aborts the handling of SC2 and DU2. So the VA process of SC3 has to handle SC3, SC2, SC1, DU2 and DU1. Here, we notice that the VA algorithm must have the capability of handling multiple SCs (Section 7), as well as handling the previously aborted DUs.

The problem of multiple SCs and aborted interleaved DUs will be handled in Section 7.2 and 7.5. The *concurrent relationship sequence* can be easily generated during the maintenance of the updates. Whenever there is an aborted process maintaining an update U_i, U_i will be put into the concurrent update set of the SC that caused the abortion.[5] The *concurrent relationship sequence* should be kept in the UMQ. Here is an example of how to generate the concurrent relationship sequence during the processing.

Example 3 *For example if we have the following updates received and the maintenance process:*

Updates	*DU1,*	*SC1,*	*SC2,*	*DU2,*	*SC3*
What Happened	*abort due to SC2*	*abort due to SC3*	*done*	*abort due to SC3*	*done*

The receive order is: DU1, SC1, SC2, DU2 and SC3. During the process of each update, the VM process of DU1 is aborted because of the concurrent SC2. So DU1 is put into the concurrent update set of SC2. The VA process of SC1 is aborted because of concurrent SC3, so SC1 is put into the concurrent update set of SC3. The VA process of SC2 finished normally. The VM process of DU2 is aborted because of concurrent SC3, so the DU2 is also put into the concurrent update set of SC3. So, the final concurrent relationship sequence is:
DU1, SC1, SC2(DU1), DU2, SC3(SC1, DU2)

7 View Adaptation Algorithm For Handling Maintenance-Concurrent SCs

VA is the least developed technology in the literature compared to VM and VS. We are aware of only two works, namely, [GMR97, NR99]. Both deal with non-concurrent (single) SCs. We now propose a new VA algorithm that maintains the extent of the data warehouse after view synchronization in an interleaved **maintenance-concurrent** SC and DU environment. The VA algorithm we propose is an extended version of the SYNCMAA [NR99] algorithm that is now capable of handling **maintenance-concurrent** DropSCs. We call it Map-VA. Since the Map-VA algorithm cannot adapt all possible view definitions, a view recomputation strategy, called Mac-Recompute, is proposed as an alternate strategy for any views that cannot be handled by Map-VA. By combining Mac-Recompute and Map-VA algorithms into one VA module, the DyDa framework can handle all kinds of view definitions under concurrent DropSCs.

7.1 View Recomputation in a Concurrent DU and SC Environment

Practically, any kind of view recomputation strategy [ZGMHW95] could be used as the Mac-Recompute algorithm as long as it can bring the extent of the data warehouse into a state consistent with the underlying information sources. However, the DyDa framework needs to know to which state the Mac-Recompute strategy brings the DW to, i.e., which DUs and SCs have been handled

[5] If there is more than one SC that causes the abortion of the VM process of that DU, then the DU will be associated with the first SC that breaks the view maintenance query.

by the Mac-Recompute and which ones not, so that the DyDa framework can continue future maintenance.

The Mac-Recompute algorithm will recompute the extent based on the new view definition generated by the VS module. If during the processing the Mac-Recompute receives a broken query message from the QE, the system state will be go back to the wait state as illustrated in Figure 3. All the DUs that happen during the query processing of Mac-Recompute will be treated as **maintenance-concurrent** DUs (Definition 10). The Mac-Recompute algorithm will handle all the DUs that the VM process aborts and all the SCs that the VA process aborts due to this SC. This then leads the VS module to generate a new view definition that then is again recomputed by the Mac-Recompute algorithm.

After the Mac-Recompute algorithm adapts the view extent, it will modify the UMQ, e.g., remove DUs and SCs out of the UMQ that related to this view and were received earlier than the successful recomputation query result because they have already been handled by this recomputation session.

7.2 The Map-VA Algorithm

The Map-VA algorithm is an extended version of the SYNCMAA [NR99] algorithm in the sense of decoupling itself from any particular view synchronization (VS) algorithm [NR98]. Namely, the Map-VA algorithm requires the old view definition, the new view definition, the old view extent, evolution mapping provided by the modified VS module.[6] The main task of the Map-VA algorithm is to generate the SQL queries to be applied to the view extent and then to calculate the new view extent. Because the generation of the SQL queries requires the *evolution mapping*, we call the algorithm Map-VA. The *evolution mapping* is a structure composed of a list of pairs of attributes, relations or conditions that encodes how the old view definition gets translated into the new view definition.

Example 4 *We assume the old view definition as described in Equation 3, and the new view definition as defined by Equation 4.*

$$
\begin{array}{ll}
\text{CREATE VIEW} & V \text{ AS} \\
\text{SELECT} & IS1.R.A,\ IS1.R.B,\ IS1.R.C,\ IS2.S.D \\
\text{FROM} & IS1.R,\ IS2.S \\
\text{WHERE} & IS1.R.A = IS2.S.A \\
 & AND\ IS1.R.B <= 10
\end{array}
\tag{3}
$$

$$
\begin{array}{ll}
\text{CREATE VIEW} & V' \text{ AS} \\
\text{SELECT} & IS3.T.A,\ IS3.T.B,\ IS2.S.D \\
\text{FROM} & IS3.T,\ IS2.S \\
\text{WHERE} & IS3.T.A = IS2.S.A \\
 & AND\ IS3.T.B <= 10
\end{array}
\tag{4}
$$

Then the mapping generated by the view synchronization algorithm can be described by:

[6] The details of how to generate the evolution mapping from the schema change and meta-knowledge used for VS are straight-forward. Please reference [NR98] for how the schema change and meta-knowledge are defined.

$$
\begin{array}{llll}
IS1.R.A & & \rightarrow & IS3.T.A \\
IS1.R.B & & \rightarrow & IS3.T.B \\
IS1.R.C & & \rightarrow & null\ (IS1.R.C\ is\ dropped) \\
IS1.R & & \rightarrow & IS3.T \\
IS1.R.A & = IS2.S.A & \rightarrow & IS3.T.A\ =\ IS2.S.A \\
IS1.R.B & <= 10 & \rightarrow & IS3.T.B\ <= 10
\end{array}
$$

The view adaptation algorithm [NR99] has four steps. The VA algorithm first needs to calculate which column of the old view extent should be kept by the view definition rewriting. We denote that part of the old extent of V as V^0. Then, it will calculate the difference in terms of extent of the replacement and the original relation (or attribute) ΔR in the new view definition. Third, it will calculate the effect of ΔR on the extent of the new view definition. This effect, denoted by ΔV, is the difference between V^0 and the extent of the new view definition. At last, it will calculate the new view extent out of V^0 and ΔV. V' = V^0 $(\cup|-)$ ΔV. Within this four steps, there are six view adaptation queries generated. The functions of the six view adaptation queries used in the Map-VA algorithm are defined in Table 5.

Query V^0	calculate potential extent of new view definition by projection on old view extent.
Query R_V	determine data from deleted relation R in old view extent.
Query S_{NV}	calculate data from substitute relation S in new view definition by querying the IS of S.
Query ΔR	calculate difference between the query results of R_V and S_{NV}.
Query ΔV	calculate difference between new view extent and V^0 by querying the IS space.
Query V	calculate new view extent from query result V^0 and query result ΔV.

Table 5: Six Queries of View Adaptation

There are two cases of using the Map-VA algorithm. First, we can use Map-VA with VS that can provide the extra information of how the view extent changed. Second, we can use Map-VA in a more general way that we only know the old view definition, new view definition and the mappings. If the view extent parameter is equivalent in the first case, then we only need to calculate V_0, and V_0 is V'. If the view extent parameter is superset, subset, or don't care or there is no view extent, the Map-VA will go through all the six queries. Note the query 4 and 5 will be expensive to calculate, so if the query result is null for either of them, we can stop and directly use V_0 as V'.

Here is an example of how Map-VA uses the *evolution mapping* to help the view adaptation.

Example 5 *Assume we have three relations R, S and T in three information sources IS1, IS2 and IS3 respectively. We define view V by query 3 in Example 4. The old view extent is materialized in a table called "OldView". The new view definition is defined by query 4 in Example 4. The evolution mapping is shown in Example 4. Then we can generate the following query to create* V^0.

$$
\begin{array}{ll}
\textsf{CREATE TABLE} & V^0\ \textsf{AS} \\
\textsf{SELECT} & IS1.R.A\ \textsf{AS}\ IS3.T.A,\ IS1.R.B\ \textsf{AS}\ IS3.T.B,\ IS2.S.D \\
\textsf{FROM} & OldView
\end{array}
\tag{5}
$$

Next, in step 2 we calculate ΔR. This process is divided into three steps. First, we need to get the data from the deleted relation R (or the relation of the deleted attribute) in the old view extent. We call it R_V by query 6.

$$\begin{array}{ll} \textsf{CREATE TABLE} & R_V \ \textsf{AS} \\ \textsf{SELECT} & IS1.R.A \ \textsf{AS} \ IS3.T.A, \ IS1.R.B \textsf{AS} \ IS3.T.B \\ \textsf{FROM} & OldView \end{array} \tag{6}$$

Then, we calculate the corresponding data from the substituted relation S (or the relation of the substituted attribute) in the new view definition. We call it S_{NV}.[7]

$$\begin{array}{ll} \textsf{CREATE TABLE} & S_{NV} \ \textsf{AS} \\ \textsf{SELECT} & IS3.T.A, IS3.T.B \\ \textsf{FROM} & IS3.T \\ \textsf{WHERE} & IS3.T.B \ <= \ 10 \end{array} \tag{7}$$

Finally, we calculate ΔR out of R_V and S_{NV}.

$$\begin{array}{ll} \textsf{CREATE TABLE} & \Delta R^- \ \textsf{AS} \\ \textsf{SELECT * FROM} & R_V \\ \textsf{EXCEPT} & \\ \textsf{SELECT * FROM} & S_{NV} \end{array} \tag{8}$$

$$\begin{array}{ll} \textsf{CREATE TABLE} & \Delta R^+ \ \textsf{AS} \\ \textsf{SELECT * FROM} & S_{NV} \\ \textsf{EXCEPT} & \\ \textsf{SELECT * FROM} & R_V \end{array} \tag{9}$$

In step 3, based on ΔR, we calculate ΔV by sending down ΔR to the information sources.

$$\begin{array}{ll} \textsf{CREATE TABLE} & \Delta V^- \ \textsf{AS} \\ \textsf{SELECT} & \Delta R^-.A, \ \Delta R^-.B, \ IS2.S.D \\ \textsf{FROM} & \Delta R^-, IS2.S \\ \textsf{WHERE} & \Delta R^-.A \ = \ IS2.S.A \end{array} \tag{10}$$

$$\begin{array}{ll} \textsf{CREATE TABLE} & \Delta V^+ \ \textsf{AS} \\ \textsf{SELECT} & \Delta R^+.A, \ \Delta R^+.B, \ IS2.S.D \\ \textsf{FROM} & \Delta R^+, IS2.S \\ \textsf{WHERE} & \Delta R^+.A \ = \ IS2.S.A \end{array} \tag{11}$$

Finally, in step 4 we calculate the new extent of the view definition by $V' = V^0 \cup (-) \ \Delta V$, and update the data warehouse.

$$\begin{array}{ll} \textsf{CREATE TABLE} & NewView \ \textsf{AS} \\ \textsf{SELECT * FROM} & V^0 \\ \textsf{UNION} & \\ \textsf{SELECT * FROM} & \Delta V^+ \\ \textsf{EXCEPT} & \\ \textsf{SELECT * FROM} & \Delta V^- \end{array} \tag{12}$$

The intermedia result of the six view adaptation queries described in Table 5 is described in Figure 6. As we can see, we first calculate the V_0 to figure out how many tuples from the old view extent could be in the new view extent. Then, we calculate the R_V and S_{NV} in order to calculate ΔR to figure out what's the difference between the original relation and the new replacement. Then, pass the ΔR around and calculate what's the effect ΔV on the view extent of the difference. Last, we update the extent V_0 with ΔV to calculate the new view extent V'.

[7]NV stands for new view.

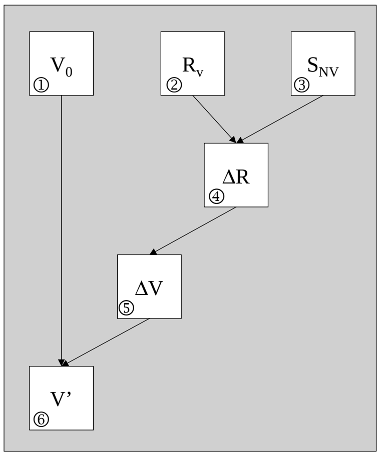

Figure 6: Query Flow of Map-VA Algorithm.

7.2.1 The Map-VA Algorithm and Maintenance-Concurrent SC

A **maintenance-concurrent** SC will disturb the view adaptation query that is send down to the information sources. The query engine would report a "broken query" error message to the VA module. The Map-VA algorithm has been developed to work with the VS module. The essence of algorithm, namely the six queries generated by the Map-VA algorithm are given in Figure 5, while details of algorithm can be found in Appendix A.

The queries V^0, R_V, ΔR, and V can all be completed at the data warehouse site, and hence will not be affected by **maintenance-concurrent** SCs. Whereas the S_{NV} and ΔV queries will be sent to the information sources for processing and hence have the possibility of being affected by the **maintenance-concurrent** SCs.

Lemma 4 *If a* **maintenance-concurrent** *DropSC breaks the view adaptation queries* S_{NV} *or* ΔV*, then the* **maintenance-concurrent** *DropSC affects the new view definition that is maintained by the view adaptation query.*

Intuitive Proof: As can be seen by Example 5, query 7 is a subquery of the new view definition, and queries 10 and 11 uses components (attributes in select clause, relations in from clause, and conditions in where clause) from the new view definition. Queries 10 and 11 use $\Delta R^{+(-)}$ that has been send from DW to IS. Therefore, if those queries are broken, then the components from the new view definition are also removed by the DropSC. In other words, DropSC also affects the new view definition.

■

As the state transition flow shows in Figure 3, the VA process will be aborted and go to the wait state in order to handle that DropSC later.

7.2.2 The Map-VA Algorithms and Multiple SCs

As described in Figure 3, whenever a **maintenance-concurrent** SC is detected, VA will abort the adaptation transaction. However, when later the VA tries to handle the **maintenance-concurrent** SC, the previous aborted SCs have to be handled as well. Therefore, the Map-VA algorithm has to handle the new view definition generated by VS module for multiple SCs.

The *Map-VA* algorithm requires the new view definition, the old view definition, and the evolution mapping provided by the VS process. For every version of the view definitions stored in the history table of the specific view, there is one evolution mapping for it. In order to handle the multiple schema changes, we need to merge the evolution mapping of all view definitions in the version history. Table 6 shows all possible transformations of the *evolution mappings* due to **maintenance-concurrent** SCs. The merging of the *evolution mappings* will be handled by the Map-VA algorithm before it is trying to do the view adaptation. The list of *evolution mappings* will be combined one by one from the oldest to the newest. If the combination is not shown in Table 6, then that means that the combination of the two operations has no effect on each other. Hence the new evolution mapping will keep both of them.

	$S \rightarrow T$	drop S	$b \rightarrow c$	drop b	drop $Cond_c$
$R \rightarrow S$	$R \rightarrow T$	drop R	-	-	-
$a \rightarrow b$	-	-	$a \rightarrow c$	drop a	-
add $Cond_c$	-	-	-	-	Cancel "add $Cond_c$" and "drop $Cond_c$"

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

Table 6: Transformation Rules of the Evolution Mapping

Example 6 *Assume we have a view defined as follows:*

$$\begin{array}{ll} \text{CREATE VIEW} & V \text{ AS} \\ \text{SELECT} & IS1.R.A,\ IS1.R.C,\ IS2.T.B,\ IS2.T.D \\ \text{FROM} & IS1.R,\ IS2.T \\ \text{WHERE} & IS1.R.C\ =\ IS2.T.D \end{array} \tag{13}$$

Assume that SC "drop IS1.R" happened, and the VS module found the substitution "IS2.S". So the new view definition (version 1) is defined by Equation 14. Then another SC "drop IS2.S" happened, and the VS module found the substitute "IS3.P". So the new view definition (version 2) is defined by Equation 15.

$$\begin{array}{ll} \text{CREATE VIEW} & V' \text{ AS} \\ \text{SELECT} & IS2.S.A,\ IS2.T.B, \\ & IS2.S.C,\ IS2.T.D \\ \text{FROM} & IS2.S,\ IS2.T \\ \text{WHERE} & IS2.S.C\ =\ IS2.T.D \end{array} \tag{14}$$

$$\begin{array}{ll} \text{CREATE VIEW} & V'' \text{ AS} \\ \text{SELECT} & IS2.T.B,\ IS3.P.C, \\ & IS2.T.D \\ \text{FROM} & IS3.P,\ IS2.T \\ \text{WHERE} & IS3.P.C\ =\ IS2.T.D \end{array} \tag{15}$$

The evolution mapping from the original view definition to the version 1 view definition (in Equation 14) is given in the first column of Table 7. The evolution mapping from the view definition *version 1 to version 2 is described in the second column of Table 7. The third column of Table 7 shows the new merged evolution mapping for these two SCs.*

Version 1	Version 2	Merged Evolution Mapping
$IS1.R.A \to IS2.S.A$	$IS2.S.A$ dropped	$IS1.R.A$ dropped
$IS1.R.C \to IS2.S.C$	$IS2.S.C \to IS2.P.C$	$IS1.R.A \to IS2.P.C$
$IS1.R \to IS2.S$	$IS2.S \to IS2.P$	$IS1.R \to IS2.P$

Table 7: Evolution Mapping Merging Example

After the VA module has calculated the merged *evolution mapping*, it can start to do the adaptation. We use the original view definition as the old view definition, the most recent view definition as the new view definition, the merged *evolution mapping* as the *evolution mapping*, and *merged containment constraint* as the containment constraint. Then, we can apply the *Map-VA* to do the view adaption based on those inputs (Section 7.2).

The *Map-VA* thus effectively handles multiple SCs. Hence the DyDa can handle concurrent SCs.

As long as the view definitions in the DW hold the assumption of the SYNCMAA algorithm [NR99][8], we can always use the above method to handle multiple SCs. For the view definitions for which the assumption does not hold, we use *Mac-Recompute* instead.

7.3 Example of Problem of Previous Aborted DUs

When the VM module handles the DU by sending the queries down to the ISs, the unexpected SC could make those queries no longer be processed by the ISs. In that case, the VM process will be aborted and the DU can no longer be handled by the VM until the concurrent SC handled. Hence, we postpone the DU handle at the VA module. During the VA process of that concurrent SC, the VA module will send down the view adaptation queries that will incrementally adapt the view extent. While, the DU that was previously aborted could also affect the original view extent in the old view extent, which was not going to be updated by the view adaptation extent. In order to result in a correct view extent, we have to do additional handles accompanied with the VA process to handle the abnormal in the original view extent by the aborted DU.

Figure 7 depict the problem of the aborted DU in a more straight way. The figure show the composition of the new view extent after one SC and one DU. The new view extent is composed of three parts. They are original extent, SC effect, and DU effect, where SC effect and DU effect has overlap. When the DU process is aborted, then the whole DU effect is not updated to the DW. And, the VA query will only calculate the SC effect including the overlap part. So, the DU effect part without the overlap was not calculated. That part need a special handling. The following examples will show what exactly the part that is miss calculated.

[8] *inclusion assumption*: In any rewriting of the view V, all join attributes of R replaced in the WHERE clause of V' are among the attributes replaced in the SELECT clause.

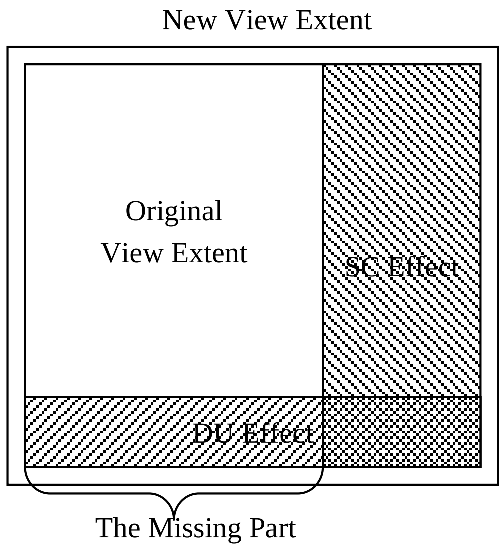

Figure 7: Composition of New View Extent after SC and DU.

Definition 14 *If a view V is defined upon relation R, we use following notations to show the related operations on relation R:*

$VE(V, R)$	*Means a part of view extent of view V that is from relation R.*
$V(R)$	*Means apply all the related projection and selection operations in view V on the extent of relation R.*
$V(..., ...)$	*Means apply all the related projection, selection, and join operations in view V on the enclosed relations.*

Example 7 *Assume we have three relations R, S, and T from different information sources. We define one view definition V as: $V = V(R, S)$. In the following example, S is deleted and the view definition is updated to use T instead. So the evolved view definition is $V' = V'(R, T)$. If we denote the fragment of the extent of R that participates in view extent of V as $VE(V, R)$ and S as $VE(V, S)$, then $V = V(VE(V, R), VE(V, S))$. Using the same notation, $V' = V'(VE(V', R), VE(V', T))$.*

First, let us explain the idea of the six steps for view adaptation by using the new notations.

- Calculate V_0 from the original view extent that still will possibly remain in the new view extent, denoted by $V'(VE(V, R), VE(V, S))$
- Calculate S in the original view extent under new view definition V denoted by $V'(VE(V, S))$.
- Calculate ΔS as difference between $V'(T)$ and $V'(VE(V, S))$, that is: $\Delta S = V'(T) - V'(VE(V, S))$.
- Calculate ΔV from ΔS, that is: $\Delta V = V'(R, \Delta S)$
- Add ΔV to the original extent of V, we get:

$$
\begin{aligned}
V' &= \underline{V_0} + \underline{\Delta V} \\
&= V'(VE(V,R),VE(V,S)) + \underline{V'(R,V'(T) - V'(VE(V,S)))} \\
&= \underline{V'(VE(V,R),VE(V,S)) + V'(R,V'(T))} - V'(R,V'(VE(V,S))) \\
&= \underline{V'(R,V'(T))} + V'(VE(V,R),VE(V,S)) - \underline{V'(R,V'(VE(V,S)))} \\
&= V'(R,T) + \underline{V'(V'(VE(V,R)),V'(VE(V,S))) - V'(V'(R),V'(VE(V,S)))} \\
&= V'(R,T)
\end{aligned}
$$

From [NR99], we know that V' has less conditions or attributes than V. Hence $V'(VE(V,R))$ is the same as $V'(V'(R))$. Therefore $V'(VE(V,R),VE(V,S))$ is canceled with $V'(V'(R),V'(VE(V,S)))$.

Second, let's see an example of what will happen, if this VA process has to handle interleaved DUs that have been aborted before.

Example 8 *We assume all definitions as in Example 8. In addition, we assume R has $DU1$, S has $DU2$ and T has $DU3$ and the VM processes of all three DUs are aborted because of that schema change. Let's go through the previous steps.*

$V_0 = V'(VE(V,R),VE(V,S))$.

$\Delta S = V'(T + DU3) - V'(VE(V,S))$.

$\Delta V = V'(V'(R + DU1),\Delta S)$.

$$
\begin{aligned}
V' &= V_0 + \Delta V \\
&= V'(VE(V,R),VE(V,S)) + \underline{V'(V'(R'),(V'(T') - V'(VE(V,S)))} \\
&= \underline{V'(VE(V,R),VE(V,S)) + V'(V'(R'),V'(T')} - V'(V'(R'),V'(VE(V,S))) \\
&= V'(V'(R'),V'(T')) + V'(VE(V,R),VE(V,S)) - \underline{V'(V'(R+DU1),V'(VE(V,S)))} \\
&= V'(V'(R'),V'(T')) + \underline{V'(VE(V,R),VE(V,S)) - V'(V'(R),V'(VE(V,S)))} \\
&\quad - V'(V'(DU1),V'(VE(V,S))) \\
&= V'(R',T') - \underline{V'(V'(DU1),V'(VE(V,S)))}
\end{aligned}
$$

However, we know the V' should be $V'(R',T')$. So $V'(V'(DU1),V'(VE(V,S)))$ is missing from this view adaptation query. Note that $V'(DU1)$ and $V'(VE(V,S))$ are both known at the middle space, so we can do local correction for it.

Theorem 3 *The DropSC cannot abort the VM process of the DU that comes from the same IS.*

Proof: This can be easily proven by the VM procedure of a DU. Whenever a DU arrives in the mediator, the VM procedure will only send queries to other relations to do the join with this DU. So in that case, the SC on the original IS will not affect the process of that DU.

■

Lemma 5 *The DU from the same IS as the DropSC will cause an abnormal result for the VA process if the DU is not successfully handled by the VM process before the VA process of this SC.*

We observe that the data updates of the relations affected by DropSC (in our example these are the data update $DU2$ of original relation S and the data update $DU3$ of the substitute relation T) are successfully handled by the view adaptation query. However, the data updates of the unaffected relations are not fully handled. In our example this is $DU1$ of S.

Lemma 6 *Previous DUs of the relations added by the DropSC that occured before DropSC will be automatically handled by the VA handling process described in Section 7 without requiring any compensation.*

Intuitively proof: we want to send an adaptation query down to calculate the updated part of the view extent. Because one relation is deleted, so we reuse the part of the view extent which the DU has not affected. ΔS is calculated by the current extent of T and the original extent of S in the view extent. So we get the current extent of T (substitute relation) and the current extent of R (other relations except dropped/substitute relation), hence the DU of T is automatically handled.

Lemma 7 *The DUs of the dropped relation have no effect on the view adaptation query result.*

Lemma 8 *That means several DUs happened on that relation S, and then one SC happens later, if we know there will be a dropSC that will drop the S, then we don't need any kind of view maintenance for the S, as long as we do the little correction $V'(V'(DU1), V'(VE(V, S)))$.*

Theorem 4 *DUs of the relations not related to the dropSC that occur in time before the SC will need extra correction on the view adaptation query result in order to incorporate the DUs into the view extent.*

That is because we remove too much, we need to compensate by putting it back.

Lemma 9 *Correction for DUs as defined in Theorem 4 can be executed locally in the middle space.*

In particular based on the example shown in Example 9, the formula for correction is $V'(V'(DU1), V'(VE(V, S)))$. Based on the previous observation, we define the following problem of handling (aborted) DUs in the VA process.

7.4 Problem Definition of Aborted Interleaved DU

Definition 15 *When SCs and DUs happen interleaved, a later detected SC could abort the VM process handling previously received DUs. If the DU is from an IS different than that SC, this DU will possibly cause the state of the extent after the view adaptation of this SC to be incorrect. We refer to this problem as the* **aborted interleaved DU** *problem.*

Example 9 *Assume we have V defined as $R \times S$. The schemas and the extents of relations R, S and T are shown in Table 8. Assume we have three data updates: $DU1$ adds $< 3 >$ to relation R; $DU2$ adds $< d >$ to relation S, and $DU3$ adds $< e >$ to relation T.*

Originally the view extent of $V = R \times S$ is:

Relation	R	S	T
Attribute	A	B	C
Extent	$<1>$ $<2>$	$<a>$ $<b>$	$<a>$ $<c>$

Table 8: Initial Setup of Schemas and Extents

$$\begin{array}{c|cccc} A & 1 & 2 & 1 & 2 \\ B & a & a & b & b \end{array}$$

Assume the SC drops relation S and S is replaced by relation T. Then, we get the new view definition $V' = R \times T$. Assume the DUs happened before the SC, we recalculate the view extent after the three data updates, and then the extent of V'(expected) is $R \times T$:

$$\begin{array}{c|ccccccccc} A & 1 & 2 & 3 & 1 & 2 & 3 & 1 & 2 & 3 \\ C & a & a & a & c & c & c & e & e & e \end{array}$$

If we do the view adaptation, first V_0 is:

$$\begin{array}{c|cccc} A & 1 & 2 & 1 & 2 \\ B & a & a & b & b \end{array}$$

Then $VE(V, S)$ is:

$$\begin{array}{c|cc} B & a & b \end{array}$$

Hence ΔS^+ is $V'(T') - V'(VE(V, S))$:

$$\begin{array}{c|cc} C & c & e \end{array}$$

And ΔS^- is $V'(VE(V, S)) - V'(T')$:

$$\begin{array}{c|c} C & b \end{array}$$

Because extent of R' is:

$$\begin{array}{c|ccc} A & 1 & 2 & 3 \end{array}$$

$\Delta V^+ = R' \times \Delta S^+$ is:

$$\begin{array}{c|cccccc} A & 1 & 2 & 3 & 1 & 2 & 3 \\ C & c & c & c & e & e & e \end{array}$$

$\Delta V^- = R' \times \Delta S^-$ is:

$$\begin{array}{c|ccc} A & 1 & 2 & 3 \\ C & b & b & b \end{array}$$

Therefore, $V'(adapt) = V_0 + \Delta V^+ - \Delta V^-$ *is:*

$$\begin{array}{c|ccccccccc} A & 1 & 2 & 1 & 2 & 3 & \begin{pmatrix} 3 \\ b \end{pmatrix}^{*} & 1 & 2 & 3 \\ C & a & a & c & c & c & & e & e & e \end{array}$$

Comparing the extent of $V'(adapt)$ *and the expected extent of* $V'(expected)$*, we can see the difference is:*

$$\begin{array}{c|cc} A & \mathit{3} & \mathit{3} \\ C & a & b \end{array}$$

This extent can be calculated using the equation $V'(V'(DU1), V'(VE(V,S)))$ *as defined in Example 9, where* $V'(DU1)$ *is:*

$$\begin{array}{c|c} A & \mathit{3} \end{array}$$

and $V'(VE(V,S))$ *is:*

$$\begin{array}{c|cc} C & a & b \end{array}$$

7.5 Algorithm of Handling Aborted DU

By Example 8, we know that the incorrect part is defined by $V'(V'(DU1), V'(VE(V,S)))$. We notice that both $V'(DU1)$ and $V'(VE(V,S))$ are known at the mediator layer. Hence, we can correct this problem locally. The algorithm is shown in Figure 8.

8 Related Work

View Maintenance Algorithms. Self-maintenance [QGMW96, GJM97, SK98, GJM96, Huy96] is one of the means to maintain materialized views at the DW without accessing the base relations.

View maintenance methods concentrate instead on incrementally maintaining the extent of the DW when the materialized views are not self-maintainable. Zhuge et al. [ZGMHW95, ZWGM97] introduce 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 but again only for the concurrency problem between data updates while the schemata of all ISs are assumed to be static. Agrawal et al. [AAS97] propose the SWEEP-algorithm that can ensure consistency of the data warehouse in a larger number of cases compared to the Strobe family of algorithms. In a separate work, we have proposed the PSWEEP algorithm [ZDR99] that improves the performance of SWEEP by parallelizing the view maintenance processes of SWEEP. However, while this previous work focused on improving the performance of warehouse maintenance for data

*The $< 3,\ b >$ within a parenthesis means the tuple does not exist in the original view extent but removed by the adaptation query.

Algorithm for Fixing Aborted DU

```
INPUT:   Old View Extent;
         SC;
         S (dropped by SC);
         T (substitution);
         View Adaptation Query Result QR.
OUTPUT:  none.

FOR every DU_i in the concurrent update set of SC
  IF DU_i comes from same IS as the S
    skip this DU_i (DU_i doesn't need to be handle.)
  ELSE IF DU_i comes from same IS as the T
    skip this DU_i (DU_i has already been handled.)
  ELSE (DU_i comes from different IS of S and T)
    calculate Result_i = V'(DU_i) × V'(V(R_j))
    ( here i and j are the indices of the ISs.)
  END IF
END FOR
compensated the QR with Result_i.
```

Figure 8: Algorithm for Fixing Aborted DU

updates only, in this paper, we are instead considering higher-level control issues over both schema changes and data updates of ISs.

View Synchronization Algorithms. Recently, the EVE project [LKNR98, NR98] studied the problem of how to maintain a data warehouse not only under data but also under schema changes. To preserve view components of affected view definitions, the EVE system locates replacements for affected components from alternate ISs and then attempts to rewrite view definitions using these identified sources. This automation of the view rewriting caused by schema changes of ISs is called view synchronization. We can reuse the view synchronization algorithms [NR98] of the EVE system in our system proposed here to generate the new view definition for an SC. EVE therefore only handled non-concurrent SCs.

View Adaptation Algorithms. Gupta, Mumick and Rao introduced a solution for updating the view extent of a data warehouse when the user explicitly changes the definition of the view [GMR97]. Their techniques are however not designed to handle concurrent schema changes nor the interleaving of data and schema changes rather they assume a completely static IS environment.

In the context of the EVE system, Nica et. al. [NR99] proposed an algorithm of view adaptation for when the underlying base data is actually removed from the IS after a DropSC schema change. It creates a set of view adaptation queries that are executed in the data warehouse space to update the old view extent so to be consistent with the new view definition. In our DyDa system, we extent this work for concurrent SC environments.

Concurrency Control. Because of the basic nature of the loose relationship between the data warehouses and information sources (IS), concurrency control is more difficult in data warehousing environments. Such work is typically based on the assumption that individual ISs cannot be controlled by the data warehouse. So the traditional lock mechanism cannot be applied to the transactions of the data warehouse. Recently, researchers in this area study the concurrency control of updating and querying [KM99], as well as transactions including materialized views [KLM+97]. They mainly focus on the concurrency problem of how the users can use the data warehouse while the data warehouse is being updated, while we focus on the concurrency problem of how to make the DW updating transaction succeed while the ISs keep changing their data and schema.

9 Conclusions and Future Work

To our knowledge, our work is the first to address the data warehouse maintenance problem under fully concurrent data updates and schema changes of ISs. DyDa overcomes the limitation of the only previous approach of handling concurrent DUs and SCs by dropping its restrictive assumption [ZR99] and releasing the protocol between the *middle* space and the *IS* space.

In this paper, we first identified the broken query problem happened in the DW management under concurrent DUs and SCs. Then, we proposed DyDa solution framework that solves the problem in two layers. The query engine level handles the concurrent DUs by local correction and concurrent RenameSCs by local name mapping. The DW management level handles the concurrent DropSCs by newly proposed view adaptation algorithm Map-VA.

Besides the broken query problem discussed in this paper, in the future, we are going to work on the research issues related with the concurrent relationships between the SCs and DUs.

Acknowledgments. We would like to thank all the DSRG members located at FL319 in the Fuller Labs. In particular, we are grateful to A. Koeller and Y. Li for helping to implement the EVE system. We also thank A. Lee and A. Nica from the University of Michigan for interactions on the EVE project.

References

[AAS97] D. Agrawal, A. E. Abbadi, and A. Singh. Efficient View Maintenance at Data Warehouses. In *Proceedings of SIGMOD*, pages 417–427, 1997.

[BLT86] J. A. Blakeley, P.-E. Larson, and F. W. Tompa. Efficiently Updating Materialized Views. *Proceedings of SIGMOD*, pages 61–71, 1986.

[GJM96] A. Gupta, H. Jagadish, and I. Mumick. Data Integration using Self-Maintainable Views. In *Proceedings of International Conference on Extending Database Technology (EDBT)*, pages 140–144, 1996.

[GJM97] A. Gupta, H. V. Jagadish, and I. S. Mumick. Maintenance and Self Maintenance of Outer-Join Views. In *Next Generation Information Technologies and Systems*, 1997.

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

[GMR97] A. Gupta, I. S. Mumick, and J. Rao. Adapting Materialized Views after Redefinitions: Techniques and a Performance Study. Technical Report CUCS-010-97, Columbia University, 1997.

[Huy96] N. Huyn. Efficient View Self-Maintenance. In *Proceedings of the Workshop on Materialized Views: Techniques and Applications*, pages 17–25, June 1996.

[KLM+97] A. Kawaguchi, D. F. Lieuwen, I. S. Mumick, D. Quass, and K. A. Ross. Concurrency Control Theory for Deferred Materialized Views. In *ICDT*, pages 306–320, 1997.

[KM99] S. Kulkarni and M. Mohania. Concurrent Maintenance of Views Using Multiple Versions. 1999.

[LKNR98] A. J. Lee, A. Koeller, A. Nica, and E. A. Rundensteiner. Data Warehouse Evolution: Trade-offs between Quality and Cost of Query Rewritings. Technical Report WPI-CS-TR-98-2, Worcester Polytechnic Institute, Dept. of Computer Science, 1998.

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

[LNR97] A. J. Lee, A. Nica, and E. A. Rundensteiner. Keeping Virtual Information Resources Up and Running. In *Proceedings of IBM Centre for Advanced Studies Conference (CASCON'97), Best Paper Award*, pages 1–14, November 1997.

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

[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. The POC and SPOC Algorithms: View Rewriting using Containment Constraints in *EVE*. Technical Report WPI-CS-TR-98-3, Worcester Polytechnic Institute, Dept. of Computer Science, 1998.

[NR99] A. Nica and E. A. Rundensteiner. View Maintenance after View Synchronization. In *International Database Engineering and Applications Symposium (IDEAS'99)*, 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.

[QW97] D. Quass and J. Widom. On-Line Warehouse View Maintenance. In *Proceedings of SIGMOD*, pages 393–400, 1997.

[RLN97] E. A. Rundensteiner, A. J. Lee, and A. Nica. On Preserving Views in Evolving Environments. In *Proceedings of 4th Int. Workshop on Knowledge Representation Meets Databases (KRDB'97): Intelligent Access to Heterogeneous Information*, pages 13.1–13.11, Athens, Greece, August 1997.

[SK98] S. Samtani and V. Kumar. Maintaining Consistency in Partially Self-Maintainable Views at the Data Warehouse. In *Database and Expert Systems Applications (DEXA)*, pages 206–211, 1998.

[ZDR99] X. Zhang, L. Ding, and E. A. Rundensteiner. PSWEEP: Parallel View Maintenance Under Concurrent Data Updates of Distributed Sources. Technical Report WPI-CS-TR-99-14, Worcester Polytechnic Institute, Dept. of Computer Science, May 1999.

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

[ZGMW96] Y. Zhuge, H. Garcia-Molina, and J. L. Wiener. The Strobe Algorithms for Multi-Source Warehouse Consistency. In *International Conference on Parallel and Distributed Information Systems*, 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*, Montreal, Canada, August, 1999.

[ZWGM97] Y. Zhuge, J. L. Wiener, and H. Garcia-Molina. Multiple View Consistency for Data Warehousing. In *Proceedings of IEEE International Conference on Data Engineering*, pages 289–300, 1997.

A Algorithms of the Map-VA Strategy

In this appendix, we present the algorithms for generating the six view adaptation queries used by Map-VA in Chapter 7.2.

A.1 Query 1: Calculating V^0

Figure 9 describes how to generate the query V^0 from the old view definition and the evolution mapping. Query V^0 is used to calculate the potential tuples that will be in the new view extent.

Algorithm for Generating SQL query for V^0

```
 INPUT:   Old View Definition;
          Evolution Mapping.
 OUTPUT:  SQL that can create V^0.

   INIT SQL = "CREATE TABLE " + Name_V^0 + " AS " + "SELECT ";
   Get the schema of Old View Definition -- OVS.
  --- Create SELECT clause ---
   FOR every column name in OVS
     Set OCN = the specificed column name from OVS
     Set NCN = the new column name of OCN from Mapping
     IF NCN is null ( means OCN is dropped )
       skip this OCN
     ELSE (means OCN is either replaced or remains unchanged)
       IF NCN == OCN
         SQL = SQL + OCN + ", ";
       ElSE
         SQL = SQL + OCN + " AS " + NCN + ", ";
       END IF
     END IF
   END FOR
   erase last "," from SQL and do error checking,
    e.g.  empty SELECT clause.
  --- Create FROM clause ---
   SQL = SQL + "FROM " + Old View Name;
   Return SQL statement.
```

Figure 9: Algorithm for Generating Query V^0

A.2 Query 2: Calculating R_V

Figure 10 describes how to generate query R_V from the old view extent and the schema of the deleted relation R. Query R_V will get the attributes in the old view extent that come from relation R.

Example 10 *Assume we have the following schema of the old view extent:*
V: IS_R_A, IS_T_B
and the schema of deleted relation R is:
R: A, C

Algorithm for Getting original part of R in the old view extent

INPUT: Old View Exten OVE;
Schema of R from MKB.
OUTPUT: the extent of R in the original view.

01. Generate a SQL out of the view definition of the View and the schema of R.
02. Assume the view definition has following schema:
03. IS_R_A, and the schema of relation kept in MKB is local, like:
04. attributes A, B, C of R of IS.
05. IF the name of IS/R/A has the '_', it will be replaced to '__'.
06. First, we need to generate the global name of an attribute out of the schema of the R. For example:
07. A of R of IS --> IS_R_A
08. Set CN_1 = a set of column names from View Definition.
09. Set CN_2 = a set of column names from the global schema of R.
10. Compute intersection of two sets of names. $CN_I = CN_1 \cap CN_2$.
11. Generate SQL out of the intersection by using this template:

$$\begin{array}{ll} \text{CREATE TABLE} & R\_V \text{ AS} \\ \text{SELECT} & CN\_I \\ \text{FROM} & V \end{array} \tag{16}$$

Here 'V' is the table name of the old view extent.
12. send SQL in the local IS where stores the old view extent.

Figure 10: Algorithm for Generating Query R_V

From line 07 in Figure 10, we have the global schema of R is:
R: IS_R_A, IS_R_C
From their schemata, we can see that the data from relation R in the old view extent is attribute IS_R_A (line 08 to 10 in Figure 10). So the correct SQL query to represent R_V is:

$$\begin{array}{ll} \text{CREATE TABLE} & R\_V \text{ AS} \\ \text{SELECT} & IS\_R\_A \\ \text{FROM} & V \end{array} \tag{17}$$

A.3 Query 3: Calculating S_V

The query S_V extracts all view components related to S out of the new view definition and generates a SQL statement out of it. It then changes the global schema back to the local schema. Global schema is used to assign a unique id of one attribute of a relation of a IS. The global schema is used in the data warehouse to explicitly specify where to get information. For example, the global schema of an attribute named 'A' of relation named 'R' of information source named 'IS' is 'IS_R_A'. Local schema is used within one IS site.

We already have the view definition class modeled by an object representation of the view definition. The view definition is divided into small units called view components. View components could be the attribute in the SELECT clause, or the relation in the FROM clause, or the condition in the WHERE clause. So, we can get the specific view components by specifying the relation name. Then, the detailed steps are described in Figure 11.

Algorithm for Getting S in the New View Definition

```
INPUT:   schema of S;
         new view definition.
OUTPUT:  S_V

get the more recent version of rewriting from the original view.
get the SELECT and WHERE clause from view definition.
FOR every attribute component AC in the SELECT clause
  IF AC is related to relation S
    Put the attribute of AC in the SELECT clause of query S_V.
  END IF
END FOR
Put the relation S in the FROM clause of query S_V.
FOR every local condition component LCC in the WHERE clause
  IF LCC is related to relation S
    Put the condition of LCC in the WHERE clause of query S_V.
  END IF
END FOR
```

Figure 11: Algorithm for Generating Query S_V

Example 11 *Assume we have the old V defined as:*

$$\begin{array}{ll} \textsf{CREATE VIEW} & V \ \textsf{AS} \\ \textsf{SELECT} & IS\_R\_A,\ IS\_T\_B \\ \textsf{FROM} & IS\_R,\ IS\_T \\ \textsf{WHERE} & IS\_R\_C \ = \ 5 \end{array} \tag{18}$$

The SC is dropping the relation R, and the VS module uses relation S to replace relation R. Then we get V' defined as:

$$\begin{array}{ll} \textsf{CREATE VIEW} & V' \ \textsf{AS} \\ \textsf{SELECT} & IS\_S\_A,\ IS\_T\_B \\ \textsf{FROM} & IS\_S,\ IS\_T \\ \textsf{WHERE} & IS\_S\_C \ = \ 5 \end{array} \tag{19}$$

From the new view definition V', we can generate the query S_V:

$$\begin{array}{ll} \textsf{CREATE TABLE} & S\_V \ \textsf{AS} \\ \textsf{SELECT} & IS\_S\_A \\ \textsf{FROM} & IS\_S \\ \textsf{WHERE} & IS\_S\_C \ = \ 5 \end{array} \tag{20}$$

In order to process the query at the IS site of relation S, we map the query 20 to the local schema. Then we get:

$$\begin{array}{ll} \textsf{CREATE TABLE} & S\_V \ \textsf{AS} \\ \textsf{SELECT} & A \\ \textsf{FROM} & S \\ \textsf{WHERE} & C \ = \ 5 \end{array} \tag{21}$$

Then, we send that query to the IS site of relation S. The result will have only one attribute named A. In order to process the query result further at the Mediator *space, we change the local schema names back to global schema names: IS_S_A before passing the result back up.*

A.4 Query 4: Calculating ΔR

Figure 12 describes the algorithm for calculating the query ΔR.

Algorithms for Calculating Difference ΔR Between R_V and S_V

```
INPUT:   R_V;
         S_V.
OUTPUT:  ΔR.

Need to make R_V schema same as S_V based on mapping.
do projection on the S_V schema from R_V
Depending on the relationship between R and S we can make
    following queries:
IF R > S:
  CREATE TABLE ΔR AS
  SELECT IS_R_A as IS_S_A
  FROM R_V
  EXCEPT (SELECT IS_S_A
      FROM S_V)
END IF
IF R <= S:
  CREATE TABLE ΔR AS
  SELECT IS_S_A
  FROM S_V
  EXCEPT (SELECT IS_R_A as IS_S_A
      FROM R_V)
END IF
```

Figure 12: Algorithm for Generating Query ΔR

A.5 Query 5: Calculating ΔV

The query that is used to calcuate the ΔV from ΔR is generated by replacing the S with ΔR in the new view definition. The query break down and processing will be handled by the QE module.

Example 12 *Assume we have the view definition described in the query 19. Then, the query to calculate the ΔV is:*

$$\begin{array}{ll} \mathsf{CREATE\ TABLE} & \Delta V\ \mathsf{AS} \\ \mathsf{SELECT} & \Delta R\_A,\ IS\_T\_B \\ \mathsf{FROM} & \Delta R,\ IS\_T \\ \mathsf{WHERE} & \Delta R\_C\ =\ 5 \end{array} \tag{22}$$

A.6 Query 6: Calculating V

Figure 13 described the algorithm for generating the query that will calculate the extent of the new view definition.

Merge the V^0 and Delta_V

```
INPUT   V^0;
        Delta_V.
OUTPUT  new extent of V.

IF R > S
  SQL = "INSERT INTO V' AS " +
      "SELECT * " +
      "FROM ( SELECT * " +
          "FROM V^0 " +
          "EXCEPT (SELECT * " +
                "FROM Delta_V ))";
ELSE
  SQL = "INSERT INTO V' AS " +
      "SELECT * " +
      "FROM ( SELECT * " +
          "FROM V^0 " +
          "UNION (SELECT * " +
                "FROM Delta_V ))";
```

Figure 13: Algorithm for Generating Query that Updates Extent V