WPI-CS-TR-98-8 August 1998

Data Warehouse Maintenance Under Concurrent Schema and Data Updates

by

Xin Zhang
Elke A. Rundensteiner

Computer Science Technical Report Series

WORCESTER POLYTECHNIC INSTITUTE

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

Data Warehouse Maintenance Under Concurrent Schema and Data Updates*

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 built by gathering information from several information sources and integrating it into one repository customized to users' needs. Recently proposed view maintenance algorithms tackle the problem of (concurrent) data updates happening at different autonomous ISs, whereas the EVE system addresses the maintenance of a data warehouse after schema changes of ISs. The concurrency of schema changes and data updates performed by different ISs still remains an unexplored problem however. This paper now provides a solution to this problem that guarantees the concurrent view definition evolution and view extent maintenance of a DW defined over distributed ISs. To solve that problem, we introduce a framework called SDCC (Schema change and Data update Concurrency Control) system. SDCC integrates existing algorithms designed to address view maintenance subproblems, such as view extent maintenance after IS data updates, view definition evolution after IS schema changes, and view extent adaptation after view definition changes, into one system by providing protocols that enable them to correctly co-exist and collaborate. SDCC tracks any potential faulty updates of the DW caused by conflicting concurrent IS changes using a global message labeling scheme. An algorithm that is able to compensate for such conflicting updates by a local correction strategy, called Local Compensation (LC), is incorporated into SDCC. The correctness of LC is proven. Lastly, the overhead of the SDCC solution beyond the costs of the known view maintenance algorithms it incorporates is shown to be neglectable.

1 Introduction

1.1 Background — View Maintenance for Data Warehousing

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 [Wid95, GM95, MD96] has importance for many applications in large-scale environments composed of numerous heterogeneous and distributed ISs, such as the WWW. Queries can be answered and analysis can be performed quickly and efficiently at the warehouse since the integrated information is materialized and hence is directly available at the warehouse, with differences already resolved.

Such large-scale environments are often plagued with dynamic continuously changing ISs, which not only modify their data contents, but also their schemas, their interfaces, as well as their query capabilities. Practically all past and current research in view maintenance has focused on the propagation of data updates from ISs to the

*This work was supported in part by the NSF NYI grant #IRI 94-57609. Dr. Rundensteiner would also like to thank IBM for the IBM Partnership Award as well as for IBM Fellowships for her students.

warehouse [AAS97], [ZWGM97], [ZGMW96], [ZGMHW95], [BLT86], [GJM96], [Wid95], [CTL+96]. Two exceptions are Gupta et al. [GMR97] and Mohania et al. [MD96] who both propose algorithms for how to keep a materialized view extent up-to-date when the view definition itself is explicitly changed by the user (for example, when a user drops one attribute from the SELECT clause of a view definition.) To the best of our knowledge, the EVE project [NLR98, RLN97, LNR97a, NR98a, NR98b, Nic98] is the first attempt of addressing the problem of how to preserve the view definition itself under capability (schema-level) changes of ISs instead of having the view simply become undefined.

In such modern distributed environments, ISs are typically owned by different information providers and hence are independent and autonomous. This implies they will update their data and schemas independently and without possibly any concern for how this may affect the DW defined upon them. They will not be aware of nor willing to wait until the DW manager has successfully processed all previous changes from other ISs and updated the warehouse appropriately. Rather, as assumed in recent data-update driven view maintenance approaches such as ECA [ZGMHW95], Strobe [ZGMW96], and SWEEP [AAS97], they may do the IS updates in any order and at any time. In that case, any new data update at an IS may affect the process of view maintenance for the previous data updates. Current view maintenance algorithms only handle the concurrency of data updates at ISs, while this paper is the first to address the concurrency problem between data updates and schema changes in such environments.

1.2 Example of Problem: Data Warehouse Maintenance over Evolving ISs

We now illustrate with an example how current technology [AAS97, GMR97, ZGMW96] would fail to handle this problem of concurrent data updates and schema changes. Assume we have two information sources IS1 and IS2 with relations R and S, respectively. Figure 1 shows the initial extent of R and S, with DW defined by the following SQL query:

$$\begin{array}{ll} \text{CREATE VIEW} & V \text{ AS} \\ \text{SELECT} & IS1.R.A,\ IS2.S.B \\ \text{FROM} & IS1.R,\ IS2.S \\ \text{WHERE} & IS1.R.B = IS2.S.B \end{array} \tag{1}$$

We assume there is one schema change SC at R of IS1 to drop the attribute IS1.R.B from IS1.R. Before IS1 drop the attribute IS1.R.B it will let middle layer handle the SC. This action could be activated by a trigger. Then assume that in response our view rewriter, which could be a human view administrator as in [GMR97] or an automatic view synchronizer tool as in [LNR97b], would create the following view rewriting:

$$\begin{array}{ll} \text{CREATE VIEW} & V' \text{ AS} \\ \text{SELECT} & IS1.R.A,\ IS2.S.B \\ \text{FROM} & IS1.R,\ IS2.S \end{array} \tag{2}$$

The idea here is that V' preserves as much as possible of the originally specified view V. Comparing the two view definitions before (V) and after (V') the schema change, we note that the IS schema change effectively triggered the

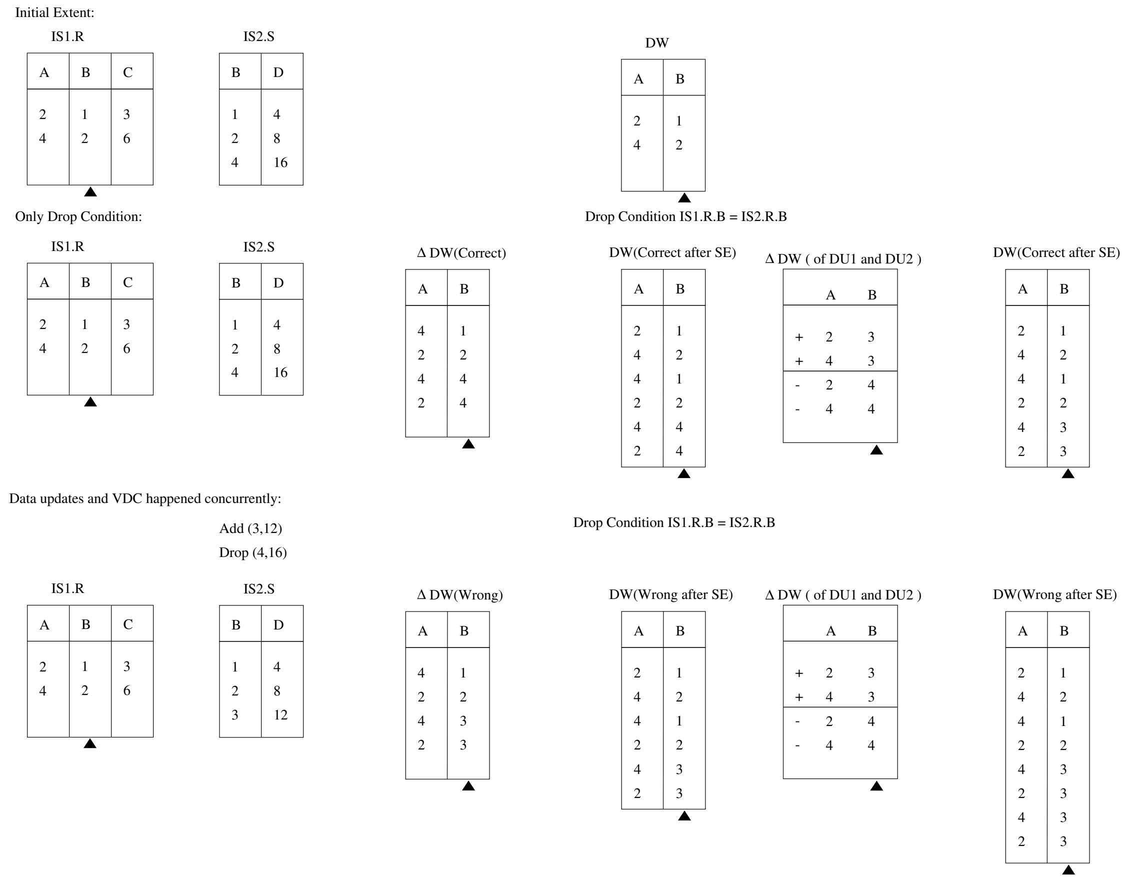

Figure 1: Example of Two Different Orderings of Schema Changes and Data Updates.

following View Definition Change (VDC) to be applied to the view V: "drop-condition $IS1.R.B = IS2.S.B$" from V. Using Gupta et al.'s approach [GMR97], in order to get the new extent of the evolved view definition after this VDC, we need to send down the following adaptation query Q3 to IS1.R to get the additional tuples for V'[1]:

$$\begin{array}{ll} \text{SELECT} & IS1.R.A,\ IS2.S.B \\ \text{FROM} & IS1.R,\ IS2.S \\ \text{WHERE} & IS1.R.B \neq IS2.S.B \end{array} \tag{3}$$

We assume here (as done in Gupta et al. [GMR97]), that the incremental maintenance query 3 can be processed before the actual drop of IS1.R.B is committed to IS1. Now assume two data updates at IS2 that occur concurrently with this SC change at IS1: DU1 to drop tuple $< 4, 16 >$ from S, and DU2 to insert tuple $< 3, 12 >$ into S. First, we assume the quiescence between the SC and two DUs is long enough for the middle layer handling the SC. The query result of Q3, denoted by QR3, shown as table 'ΔDW(Correct)' in Figure 1, correctly reflects the state of the IS space. After the two data updates have been handled next by the middle layer (say, the SWEEP View Maintenance Algorithm [AAS97]), it would drop tuples $\{< 2, 4 >, < 4, 4 >\}$ from DW, and insert $\{< 2, 3 >, < 4, 3 >\}$ to the DW.

[1] This adaptation query will add more tuples to the existing view extent by querying the IS to compensate for the dropping of a condition. For brevity reasons, we hide the details of how to process the query Q3 in our distributed system, instead see Section 6 for more details on this.

As the result, the final DW will be consistent with the ISs after handling those two data updates.

Then, we assume the quiescence between the SC and two DUs is too short to complete Q3 first. As result, DU1 and DU2 happened at IS2 before Q3 is being fully handled by IS2. The query result QR3, shown as table 'ΔDW(Wrong)' in Figure 1, is affected by DU1 and DU2 and contains some wrong tuples. This now results in the wrong extent of DW. Next the two data updates are handled by the middle layer. We still need to drop tuples $\{< 2, 4 >, < 4, 4 >\}$ from DW, and insert $\{< 2, 3 >, < 4, 3 >\}$ to the DW. As result, the DW denoted as 'DW (wrong after SE)' will have duplicate tuples $\{< 2, 3 >, < 4, 3 >\}$, and thus is not consistent with the ISs.

1.3 Our Solution Approach

Our approach towards addressing this problem is to attempt to utilize existing solution procedures for view maintenance such as View Synchronization (VS), View Adaptation (VA) and View Maintenance (VM) as much as possible, and to adapt them as needed to address this overall concurrent maintenance problem. In particular for the purpose of this paper, we select the newest solutions proposed in the literature for each of the tasks at hand, namely, SWEEP [AAS97] for data updates of the ISs (VM) , VA [GMR97] for extent updates after view definitions changes (VA), and EVE [LNR97b] for schema changes of the ISs (VS). Our integration solution approach is however generic, and consequently other algorithms could be easily plugged in for these three subtasks of VS, VA or VM.

In order to put the three view maintenance algorithms together, we need some mechanism to order and coordinate the execution of the algorithms. For this purpose, we develop an overall control strategy to handle both messages as well as data exchanges between the modules supporting functions necessary for view maintenance. Our high-level control system is called SDCC (Schema change and Data update Concurrency Control system). This system includes two key modules for the coordination of VS, VA, and VM. One is the Update Message Queue (UMQ), that orders the data and schema update messages from the ISs and associates appropriate labels with all messages so to support the detection of faulty query results due to concurrency conflicts. The other one is the Central Control (CC) unit that incorporates protocols for the coordination of the execution of VS, VA and VM supported by the UMQ. Once faulty query results are detected by SDCC, we have designed a fault-correction algorithm. This algorithm corrects the faulty query result using local compensation queries only, hence called LC. We prove that the SDCC system successfully solves the problem of concurrent data updates and schema changes, while imposing neglectable overhead on the performance of view maintenance.

1.4 Contributions

Our work makes the following contributions:

1. We are the first to formally characterize the problem of concurrent schema and data updates, which leads to the development of a strategy for the *detection of faulty data warehouse updates* in the context of the view maintenance process.

2. We develop a framework, called SDCC, as a general *solution approach* for this problem. SDCC succeeds to exploit existing techniques for view maintenance, view adaptation, and view synchronization by incorporating protocols for the integration of these existing technologies.

3. Once conflicting updates are detected by SDCC, SDCC employs an algorithm for the correction of this problem. The algorithm, called *local compensation* (LC), solves the problem using local correction queries only. LC exploits the query template concept to abstract the adaptation process, thus allowing us to easily plug in different view adaptation algorithms.

4. SDCC and in particular the LC algorithm are *proven to be correct.*

5. We conduct an initial evaluation of the SDCC system by analyzing the *overhead of SDCC* in terms of the number of messages and data-sizes transferred on the network as compared to the known view maintenance algorithms it incorporates. We show this overhead to be neglect-able.

1.5 Outline of Paper

In the next section, we briefly review related research. We give the basic definitions in Section 3. In Section 4, we define the maintenance concurrency problem. Section 5 describes the framework of the SDCC system. In Section 6, we present the LC algorithm and gives out a proof of SDCC and particular the LC algorithm. Section 7 discusses the overhead of the SDCC system. Section 8 identified four levels of concurrency handling for data warehouse maintenance. In Section 9, we conclude and discuss future directions of our work.

2 Related Work

Data warehousing has been recognized as a good technique for simplifying information access in distributed environments. Many efforts have focused on the issue of how to maintain a data warehouse in dynamic environments ([ZGMHW95], [ZGMW96], [AAS97]). They all are concerned with data updates at the ISs, and how to maintain the view extent at the data warehouse in the context of such updates. In the last few years, new algorithms have been developed in particular to handle concurrent data updates between independent ISs. 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. However, their work is also limited to improving performance of warehouse maintenance for data updates only. In this paper, we are instead considering higher-level control issues over both schema changes and data updates of ISs.

Recently, the EVE project [LNR97b, LKNR98, NR98a, NR98b] has studied the problem of how to maintain a data warehouse not only under data but also under schema changes. We have developed the EVE (Evolvable View Environment) system [LNR97a] in which we add the much needed flexibility to the view evolution process by extending the SQL view definition language, now called Evolvable-SQL, to include view evolution preferences, indicating for example which components of the view are dispensable, essential, or replaceable by alternate components. 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. Concurrency of schema or data updates have not yet been considered in EVE, however, they are now the focus of this current paper.

Gupta, Mumick and Rao introduced a solution for updating the view extent of a data warehouse when the user changes the definition of the view [GMR97]. They assume, however, that the underlying ISs are static (neither data nor schema updates occur) and rather a user explicitly requested a modification directly of the view specification. They propose a set of adapting SQL queries to update the view extent of a materialized view incrementally after the different types of view definition changes. We heavily borrow from their work for addressing a sub-problem of our overall task, namely, we translate schema changes of an IS into a sequence of view definition changes and thus are able to apply their view extent adaption queries to solve our data maintenance after IS change problem. 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.

3 Background

3.1 Notations

Our system will handle two types of data updates: insertions and deletions. For convenience, we adopt an approach similar to [BLT86] and use *signs* on tuples: + to denote an inserted or existing tuple, and - to denote a deleted tuple. The propagation rules for tuple signs are listed in Table 1 [2].

t	$\delta_{cond}(t)$	$\Pi_{proj}(t)$
+	+	+
-	-	-
t_1	t_2	$t_1 \times t_2$
+	+	+
+	-	-
-	-	+
-	+	-

Table 1: Sign Propagation Rules.

[2]In the title of the table, t means tuple, $\delta_{cond}(t)$ means tuple after selection based on condition *cond*, $\Pi_{proj}(t)$ means tuple after projection. $t_1 \times t_2$ means cardition product of t_1 and t_2. This table is used to deside the sign of the tuples after operations.

Table 2 defines the main notations that will be use in the remainder of this paper. A sequential number, unique for each update, will be generated by our SDCC system whenever the IS update message reaches the system as further explained in Section 6. This number is denoted by "n" in Table 2.

In addition, like [ZGMHW95], we define two binary operators, called + and -, that operate on relations with signed tuples. For a relation r, let $pos(r)$ denote the tuples in r with a plus sign and let $neg(r)$ denote the tuples with a minus sign. Then $r_1 + r_2 = (pos(r_1) \cup pos(r_2)) - (neg(r_1) \cup neg(r_2))$, and $r_1 - r_2 = r_1 + (-r_2)$.

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.
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] of all i.
VDC(n)(m)	Primitive view definition change caused by SC(n) with subscript m. More than one VDCs could be caused by one SC, i.e., m $\geq$ n.
Q(n)(m)	Query used to adapt view after VDC(n)(m).
Q(n)(m)[i]	Sub-query of Q(n)(m) sent to IS[i].
QR(n)(m)[i]	Query result of Q(n)(m)[i].
QR(n)(m)	Query result of Q(n)(m) after re-assembly of all QR(n)(m)[i] for all i.
ΔQ(n)(m)(p)[i]	Local compensating query generated by our system addressing the concurrency of QR(n)(m)[i] and concurrent X(p)[i].
ΔQR(n)(m)(p)[i]	Query result of ΔQ(n)(m)(p)[i].

Table 2: Notations and Their Meanings.

3.2 Definitions of Basic Terms

Definition 1 *The SQL views at the DW (data warehouse) are assumed to be Select-From-Where queries with a conjunction of primitive clauses in the WHERE clause. A DW query hence is defined by:*

$$\begin{array}{ll} \text{CREATE VIEW} & V \text{ AS} \\ \text{SELECT} & R_1.\bar{D}_1,\ ...,\ R_n.\bar{D}_n \\ \text{FROM} & R_1,\ ...,\ R_n \\ \text{WHERE} & C_1\ AND\ C_2\ ...\ AND\ C_m \end{array} \quad (4)$$

where $C_i, i = 1..m$ are primitive conditions, and $\bar{D}_j, j = 1..n$ is a subset of attributes of the relation R_j, respectively.

Definition 2 *There are two types of* **schema changes** *in our framework:*

- *A* **schema change (SC)** *denotes a primitive change that occurs at the schema of one of the information sources. In our current system, SC could be: add-attribute, add-relation, change-attribute-name, change-relation-name, drop-attribute, and drop-relation.*

- *A* **view definition change (VDC)** *denotes a primitive change of one of the view definitions in the warehouse. It could be add-attribute, add-relation, add-condition, delete-attribute, delete-relation, and delete-condition.*

The VDCs can be either explicitly requested by a user [GMR97] for a given view, or they could be automatically generated by a view evolution system like EVE [LNR97b] in response to IS schema changes that indirectly affect a view definition. The relationships between one schema change (SC) of an IS and the corresponding set of view definition changes (VDCs) of a dependent view that may be triggered by this SC are given in Table 3. For each VDC, there is a corresponding adaptation query that adapts the extent of the view after the VDC[3]. Based on Table 3, we can see that a set of adaptation queries may be required in order to fix the views affected by a single SC.

Schema Change	View Definition Change (VDC)[4]
add-relation	None
change-rel-name	None
drop-relation	delete-relation(delete-attribute | delete-condition)[add-relation(add-attribute | add-condition)]
add-attribute	add-attribute
change-att-name	None
drop-attribute	(delete-attribute | delete-condition) [add-attribute | add-condition]

Table 3: Relationship between SC and VDC (View Definition Changes)

Definition 3 *A data update (DU) denotes (a table of) changes of the extent of one of the information sources. It could be tuples to be added to that information source, or tuples to be dropped from that information source, denoted by "+" or "-" respectively (see Section 3.1 above). We say that the set of data updates at information source j at time m, denoted by DU(m)[j], is an insertion (deletion) set, if DU(m)[j] contains only tuples that have been added to (deleted from) IS[j].*

4 Definition of the Maintenance Concurrent Problem

A **maintenance-concurrent** data update is a data update that unexpectedly affects the query result used by the middle layer to handle other data updates. Here is the formal definition.

Definition 4 *Let $X(n)[j]$ and $Y(m)[i]$ denote either data updtes or schema changes on IS[j] and IS[i] respectively, m and n is the time stamp assigned to the updates. We say that the update X(n)[j] is* **maintenance-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.*

[3] Please see Section A and Table 5 for a full description of how Gupta et. al.'s [GMR97] view adaptation solution can be applied to our problem of data maintenance after IS change.

[4] In Table 3 we use BNF notations: "[ ]" means optional, "( )" means group, and '|' means "or". The order in the expression shows the order the VDCs would be executed on the DW.

Definition 5 *We say that the update X(n)[j] is* **maintenance-concurrent** *X(n)[j], if X(n)[j] is* **maintenance-concurrent** *with at least one update Y(m)[i] for some m in the system.*

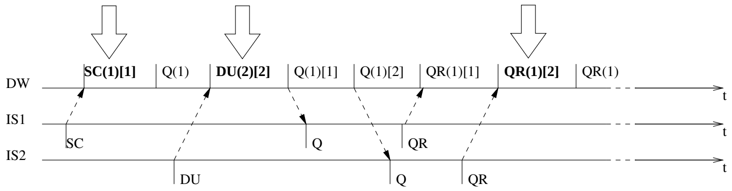

Figure 2: Time Line for a Maintenance Concurrent Data Update.

Figure 2 illustrates the concept of a **maintenance-concurrent** data update defined in Definition 4 with a time line illustration. Assume we have one data warehouse DW and two information sources IS1 and IS2. First, there a schema change SC occurs at IS1. Then, there is a DU at IS2. From the figure, we can see that the SC is received by the DW before the DU, but DU occurs at IS2 before the adaptation query of SC arrives at IS2 (The SC, DU and QR are pointed by arrows). So, here the DU is maintenance concurrent with SC by Definition 4.

Any remote query send down to the information source space has the possibility of causing the **maintenance-concurrent** problem, including the adaption queries for view extents that require remote queries (see Appendix A).

5 The SDCC Framework

5.1 Assumptions

In our work, we make the following simplifying based on the following assumptions:

Assumption 1 *Information sources are semi-cooperative. For data updates, ISs will first commit the update, and then notify the middle layer. For schema changes, ISs will first notify the middle layer about the intended schema change and proceed with the actual change only after it received acknowledgement from the data warehouse.*

Assumption 2 *Each IS only has one relation.*

Assumption 3 *Network communication between IS and DW is FIFO.*

5.2 The Overall Architecture of SDCC system

The Schema change and Data updates Concurrency Control (SDCC) framework we have designed to address this problem of DW maintenance under concurrent data and schema change is depicted in Figure 3[5]. We assume the existence of wrappers that convert heterogeneous ISs to a common data model, say in our case the relational model.

[5]For simplicity, in the remainder of the discussion, we assume only one data warehouse created over multiple ISs.

In our data warehousing middle space, there are three major components that maintain the views and their extent, and that must thus be coordinated:

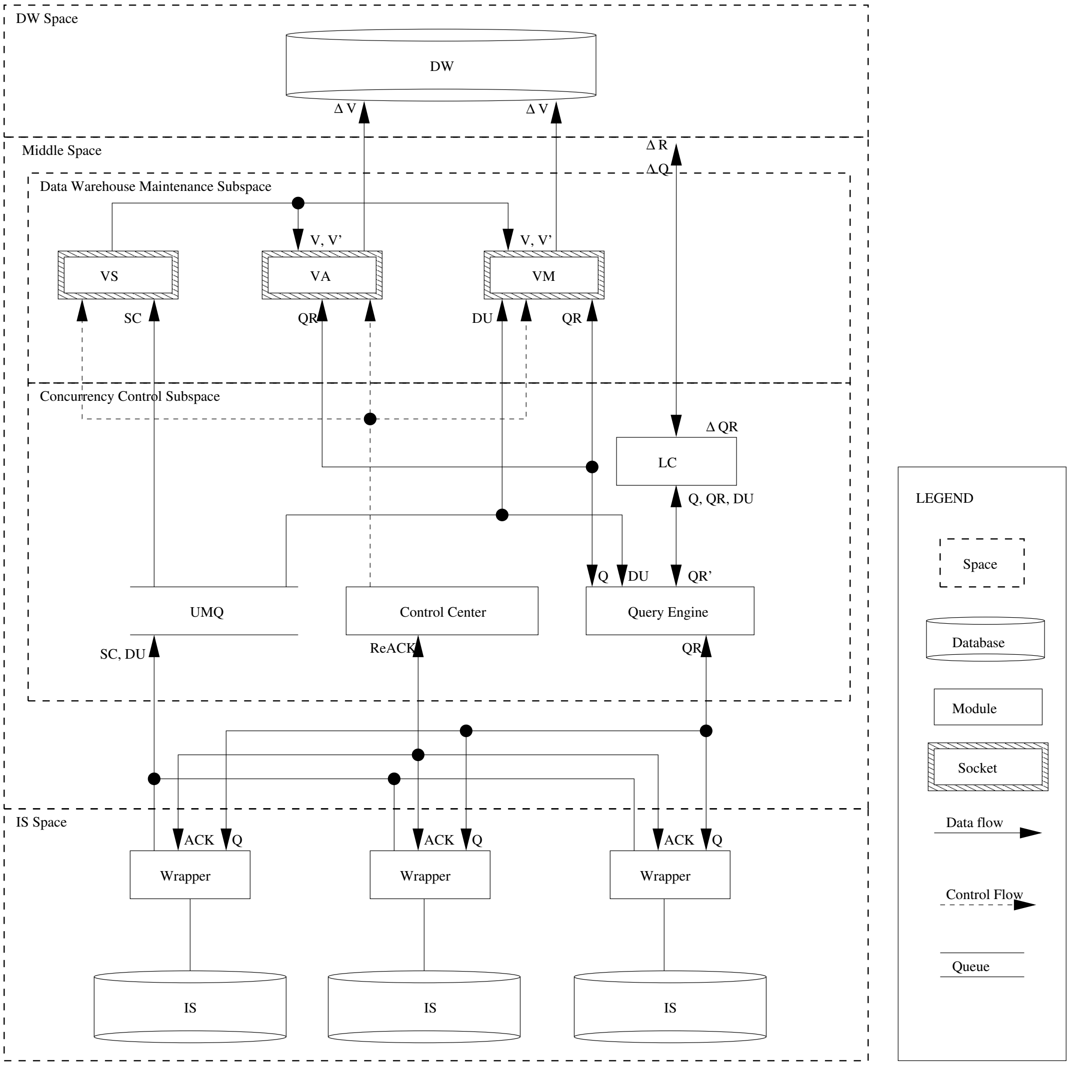

Figure 3: Framework of Local Compensation/SDCC System

- the View Maintainer (VM) that maintains the extent of a view under IS data updates.
- the View Synchronizer (VS) that rewrites the view definition in response to IS schema changes.
- the View Adaptor (VA) that adapts the extent of a view after its definition has been changed.

SDCC is general in the sense that it is not restricted to the use of a particular view maintenance algorithm. In order to fit existing incremental view maintenance algorithms into the SDCC framework, we only must extend them to be able modify the view definition they work with, i.e., to add the capability to update their data structures depending on the View Definition Change (VDC). Clearly, existing algorithms such as SWEEP [AAS97] are not equipped with this capability, however, this extension is relatively straightforward[6]. Figure 3 shows the modules of our framework and the data flow between them[7]. Table 4 lists the meaning of each symbol that appears in the framework figure.

Symbol	Meaning
V	a 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. It shows which tuples will be inserted into or removed from extent of view V.
ΔQ	Local Compensation (LC) query.
ΔR	a temporary relation created at local database of DW.
DU	a data update
SC	a schema change.
Q	query.
UMQ	update message queue.
ACK	acknowledgement sent by DW in order to let IS proceed with SC.
ReACK	acknowledgement sent by IS in order to let DW know that IS has finished SC.

Table 4: Notation used for Framework of SDCC System in Figure 3

To order and synchronize the execution of VM, VA and VS, there are two major components in the concurrency control subspace of the SDCC framework: Update Message Queue (UMQ) and Control Center (Figure 3). UMQ collects all messages related to the SCs and DUs sent to our system from the underlying information sources and establishes an order among them by assigning unique numbers to them based on the order they arrived at the middle space. The Control Center handles the cooperation protocol between the data warehouse and the information sources to make sure that the data warehouse will be updated correctly, as further explained below.

The Query Engine (QE), also situated in the concurrency control subspace of the SDCC framework (Figure 3), is the common module used by VS, VA and VM. It will connect the modules with the underlying ISs. The query engine makes the system flexible, because it allows any VA, VM and VS algorithms to be easily plugged into our SDCC system. QE not only processes queries send down from the SDCC system to the distributed information space and receives the query result from the ISs, it also analyzes the UMQ to find out if there is any **maintenance-concurrent** data update at that IS. If there is, it will invoke LC to correct the query result before returning the query result to the upper level, such as the VA module. The QE will detect the **maintenance-concurrent** DU using the following strategy: All the query results received by the Query Engine (QE) will have the following attributes as part of the message: QR(n)(..)[i][8], where "n" denotes the update that generated this query, i.e., the update with the sequential number "n", and "[i]" denotes the fact that the query result comes from the IS[i]. So QE checks if there is any $DU(m)[i]$ with $m > n$ in the UMQ. If there is any, then that update is a **maintenance-concurrent**

[6] One trivial though not most efficient solution would be to simply reconstruct the module of SWEEP for the new view definition.

[7] To show the main idea clearly, the Meta Knowledge Base (It, introduced in the EVE system, is used to store the meta knowledge, such as schema of information source.) and View Knowledge Base (It, introduced in the EVE system, is used to store the view definitions in the data warehouse.) are not showed in the figure, neither are the subscribers of ΔQ, ΔQR, Q, QR, DU, and SC shown.

[8] (..) means none or more (k). So, it could be QR(n)[i], or QR(n)(m)[i].

DU by Definition 4.

Each data warehouse maintenance module connects to SDCC system by a socket that is responsible for assigning a number to the query generated by the module and send down to the QE for processing. This number is always set to be identical to the sequential number of the update that is currently being processing. From the sequential number, we can easily know which update is handled by this query. QE will assign the same sequential number to the query result as that associated with of its corresponding query. In this way, we can keep track of every message transferred inside the middle space, allowing us to easily detect **maintenance-concurrent** DUs. The detailed algorithm of QE is given below.

```
PROCEDURE: QE
Input: Q(n)[i] as Query
Output: QR(n)[i] as Query Result
Algorithm:
    send Q(n)[i] to IS[i];
    receive QR(n)[i] from IS[i];
    IF (DU(m)[i] exists in Update Message Queue AND m > n)
        /* concurrent DU happened */
        QR(n)[i] = LC(Q(n)[i], DU(m)[i], QR(n)[])
    END IF
    RETURN QR(n)[i];
END PROCEDURE
```

The LC module, in the concurrency control subspace of the SDCC framework (Figure 3), is designed to generate a compensation query to fix the faults in the affected query result caused by concurrent data updates once detected by QE. The detailed algorithm for LC is given in Section 6.

The control center controls all the executions of data warehouse maintenance algorithms. It coordinates the middle space and the IS space by the following IS cooperation protocol:

IS cooperation protocol:

- `IS will send out DU notification after it finished a DU.`
- `IS will send out SC notification when it plans to do schema update. It will wait for the ACK from the DW before executing the schema update.`

CC cooperation protocol:

- `Middle space will send out ACK to IS to notify it once the maintenance of the data warehouse for a SC has been completed.`
- `IS will send back Re-ACK to middle space to notify it off the schema update completion at the IS.`

The cooperation protocol is created based on the assumption of semi-cooperating ISs that is justified in Appendix A.4. From this protocol, we can see that the IS cannot execute the schema change unless the middle space permits it, while the IS can send out the notification of intended schema changes freely. This protocol will not restrict the execution of data updates at the IS space.

5.3 Execution Strategy of the SDCC System

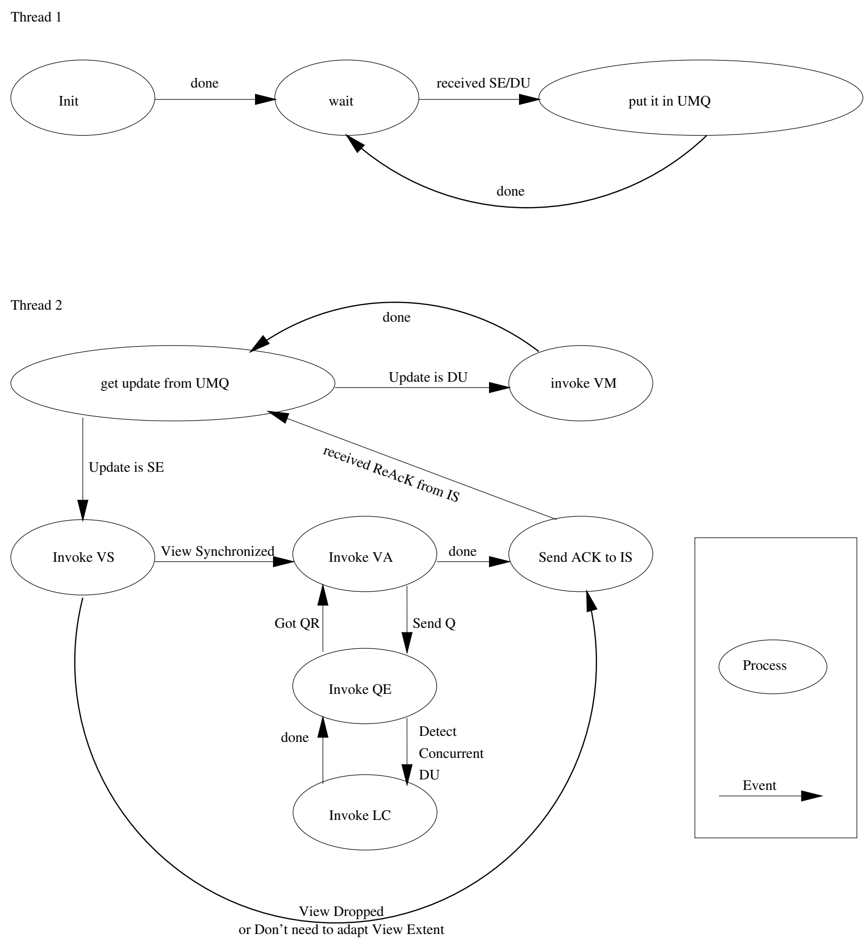

Figure 4: Event-Process Diagram of SDCC System

There are two major threads running in parallel in the system. The first thread depicted in Figure 4 will initialize the system and continue to collect all the data updates and schema changes reported by any IS and store them in the UMQ. The second thread in Figure 4 is getting the next update from the UMQ for processing by the SDCC system. If the update is a data update, SDCC will invoke the VM process to handle the view extent update. If the update is a schema change, SDCC will first invoke VS[9] . If the synchronization is successful, VS will then invoke VA to update the extent of the current modified view definition. In the case of **maintenance-concurrent** DUs happening at one IS during the process, QE will call LC for the compensation process. At last, SDCC will send ACK down to IS to initiate the IS cooperation protocol. After receiving Re-ACK from the IS, the overall process of this second

[9]For details on view synchronization, please reference EVE papers [LNR97b, NR98a, NR98b, Nic98] to synchronize the view definition

thread repeats and the SDCC will get the next update from the UMQ. If the synchronization failed or the extent of the affected view does not need to be adapted (like in the case of a change-rel-name request for example), SDCC will directly send ACK.

The algorithm used for VS could be any one of the view synchronization algorithms of the EVE system [LNR97b]. The algorithm used for VM could be Strobe [ZGMW96], SWEEP [AAS97] or any other of the recently developed maintenance algorithms that operate correctly in a concurrent data-update environment. The VA algorithm, for example, could be Gupta et. al [GMR97]'s view adaptation algorithm[10]. The maintenance algorithms employed by the SDCC system are not required to handle the concurrency problem. As long as all the ISs communication goes through the QE, then the protocols introduced in this paper will assure correct overall synchronization and functioning of all processes. In this paper, we use a subset of Gupta's algorithm for VA to illustrate the LC algorithm for the compensation of this VA as further detailed in Section 6.

6 LC Algorithm: Compensating for Maintenance Concurrent Updates

There could be four kinds of concurrency problems between SCs and DUs. They are DUs $\vdash$ SC, SCs $\vdash$ SC, DUs $\vdash$ DU, and SCs $\vdash$ DU ("$\vdash$" means **maintenance-concurrent** with as defined in Definition 4). Due to the assumption 1, the IS will not change until the SC is handled by the DW. For this reason, we can isolate the SCs $\vdash$ SC and SCs $\vdash$ DU problems from our current treatment as they will not cause any problem within the context of the SDCC framework. Given that the DUs $\vdash$ DU problem has already been addressed in the literature in the form of VM algorithms [AAS97, ZGMW96], we instead focus in the remainder of this paper on solving the DUs $\vdash$ SC problem. Hence, from here onwards, we only consider the case that SC is followed by possibly several conflicting DUs. This is both a reasonable as well as important problem to address. First, SCs tend to happen less frequently than DU. Second, the SC process, involving both view synchronization and view adaptation, may take a longer time to execute. Thus, it is likely that during the process of schema evolution of one IS there may be the concurrent data updates at other ISs. Below, we present the Local Compensation (LC) algorithm of the SDCC system, for which we asume the VA module is using the algorithm described in Gupta et. al.'s paper [GMR97] for view extent adaption. See Appendix A for a detailed treatment of the VA algorithm.

6.1 Query Template

The LC algorithm is generic, in the sense that it works with the query generated by the VA but does not care how it will be generated[11]. The LC algorithm in generally applicable for a diversity of VA algorithms due to the query template that is used to generate the LC query. A query template is a parameterized SQL query where relation

[10]The DSRG lab at WPI is working on a second VA algorithm [Nic98] to properly match the characteristics of the concurrent environments, but it has not yet been completed.

[11] The query will be broken down for each IS. We assume one IS only has one relation. It is possible to extend the LC algorithm to handle more than one relation in one IS.

names are variables that can be instantiated to specific relations. For the LC, in order to compensate for a faulty query result, we need to know the query that caused the fault and then re-apply this query to the data update. For this purpose, we employ the concept of a query template to help us generate a compensating query as illustrated by the example below.

Example 6.1: In this example, assume we have the query:

$$\begin{array}{ll} \text{SELECT} & R.A \\ \text{FROM} & R \\ \text{WHERE} & R.B > 5 \end{array} \tag{5}$$

We can abstract a query template from the query given in Equation (5) by replacing the relation name "R" with the variable "\$R", resulting in the template in Equation (6):

$$\begin{array}{ll} \text{SELECT} & \$R.A \\ \text{FROM} & \$R \\ \text{WHERE} & \$R.B > 5 \end{array} \tag{6}$$

Then assuming the relation "S" has the same schema as "R" has, we can apply this query template to "S" as well as to "R", we then get the following:

$$\begin{array}{ll} \text{SELECT} & S.A \\ \text{FROM} & S \\ \text{WHERE} & S.B > 5 \end{array} \tag{7}$$

6.2 General Description of Local Compensation Strategy

Local compensation is only concerned with the particular query[12]that has been sent to the IS and is affected by **maintenance-concurrent** data updates of that IS. LC tries to erase the faults caused by the **maintenance-concurrent** data update from the query result that is to be returned back to the data warehouse middle layer. In Section 5.2, we describe the concurrent data update detection procedure that determines if a query result (QR) is potentially faulty. And now we must determine *how* that query was affected by data updates in order to design LC to correct it locally.

The LC algorithm will generate a compensation query based on the template of the affected query. It will compensate for the effect of that update on that faulty query result returned to SDCC. In this way the query result will be corrected and made consistent with the original state of the IS space before the occurrence of the maintenance-concurrent data update. In order to erase the effect of a **maintenance-concurrent** data update on the affected query result, we perform the same query as the original query on the **maintenance-concurrent** data update to get the faulty tuples. To achieve this, in the first step, LC creates temporary tables for each conflicting data update. Then we apply the compensation query on the temporary tables and get the faulty tuples. In the last step, LC corrects the affected query result by either subtracting or adding the faulty tuples.

---

[12]We assume one IS only has one relation to simplify the sub-query for each IS in the treatment below. However, an extension to multiple relations is straightforward.

PROCEDURE: `LC`
Input: `Sub-query result QR(m)(n)[j]`
`Sub-query Q(m)(n)[j],` $1 \leq i, j \leq x$, `with x number of ISs`
`all` **maintenance-concurrent** `data updates DU(p)[j]-s,` $p > m$, `from Update Message Queue (UMQ).`
Output: `Adjusted QR'(m)(n)[i] with effect of all` **maintenance-concurrent** `DU(p)[j],` $p > m$, `erased.`
Algorithm:

1. `Create one temporary relational table ΔR_j for all` **maintenance-concurrent** `DU(p)[j].`
2. `Create one local compensating query ΔQ(m)(n)[j] from query template of Q(m)(n)[j] and the relational table ΔR_j created in step 1.`
3. `Get ΔQR(m)(n)[j] by executing LC query ΔQ(m)(n)[j] on the table ΔR_j.`
4. `Get QR'(m)(n)[j] by minus ΔQR(m)(n)[j] from the QR(m)(n)[j].`

END PROCEDURE

This LC procedure will be called by the SDCC system when an affected query result is detected. After collecting information about which query is affected, what the query result is and what the **maintenance-concurrent** data updates are, the LC procedure will correct the affected query result using this information. In step one, LC creates one temporary relational table for all the **maintenance-concurrent** data updates DU(p)[j] in the middle layer with a special sign field to show whether a tuple is deleted from or inserted to the IS[j]. 'p' denotes a higher sequence number than the sequence number of the affected query result QR(m)(n)[j], indicating this data update happened after the generated query. 'j' denotes the same subscript of the IS where this data update comes from as where the faulty query was being executed. 'n' denotes the subscript of a VDC of a set of VDCs caused by the schema change. Then, in step two, LC generates the local compensating query, denoted by ΔQ(m)(n)[j], from the original adaptation query Q(m)(n)[j] by using query template techniques described in Section 6.1. In step three, LC executes the compensation query on the temporary table and gets the faulty tuples denoted as ΔQR(m)(n)[j]. In the last step, LC separates out the faulty tuples from the affected query result, denoted by ΔQR(m)(n)[j], by subtracting the inserted tuples and adding the deleted tuples.

We observe from the description given above that the Local Compensation (LC) algorithm:

- sends no query down to information sources, and
- does all compensation locally within the data warehouse.

From this, we can conclude that:

- LC will not cause any further **maintenance-concurrent** problems to occur, and
- LC will efficiently correct any effects of **maintenance-concurrent** DUs due to performing only local rather than remote over the network requests.

6.3 Example of LC Algorithm

We now give an example to illustrate how the LC algorithm correctly solves the problem described in Section 1.2. Recall that we have two information sources IS1 and IS2 with relations R and S, respectively. The view definition V (Equation 1) is evolved to view definition V' (Equation 2) for the schema evolution SC of dropping the attribute IS1.R.B from relation IS1.R. The adaptation query Q3 is trying to adapt the extent of the view V to be consistent with the new definition V', while two **maintenance-concurrent** data updates affect the query result and finally result in a wrong extent of the view. In particular the tuples $\{ <4,3>, <2,3> \}$ in the query result of Q3 shown in Figure 1 are faulty, and thus need to be detected and compensated for.

In order to execute the query over distributed information sources, we break down the query Q3 for two information sources. Query (8) is sub-query of IS1, query (9) is sub-query of IS2, and query (10) is the assembly query.

$$\begin{array}{ll} \text{CREATE tempTABLE} & Q3-IS1 \\ \text{SELECT} & IS1.R.A,\ IS1.R.B \\ \text{FROM} & IS1.R \end{array} \quad (8) \qquad \begin{array}{ll} \text{CREATE tempTABLE} & Q3-IS2 \\ \text{SELECT} & IS2.S.B \\ \text{FROM} & IS2.S \end{array} \quad (9)$$

$$\begin{array}{ll} \text{SELECT} & Q3-IS1.A,\ Q3-IS2.B \\ \text{FROM} & Q3-IS1,\ Q3-IS2 \\ \text{WHERE} & Q3-IS1.B\ =\ Q3-IS2.B \end{array} \quad (10)$$

Only the query result, denoted by QR, of query Q3-IS2 is affected by the **maintenance-concurrent** data updates. The query result should return extent $\{<1>,\ <2>,\ <4>\}$ instead of extent $\{<1>,\ <2>,\ <3>\}$. We need to apply the LC algorithm to correct the faulty query result.

First, we create a local database "*R_LC*" for the two **maintenance-concurrent** data updates with signs. Tuple $<3,12>$ has '+' sign, and tuple $<4,16>$ has '-' sign. Then, we abstract the query template "Q3-Template" from the query "Q3-IS2":

$$\begin{array}{ll} \text{SELECT} & \$R.B \\ \text{FROM} & \$R \end{array} \quad (11)$$

From the template "Q3-Template" in Equation (12), we get the compensation query ΔQ given in Equation (12):

$$\begin{array}{ll} \text{SELECT} & R\_LC.B \\ \text{FROM} & R\_LC \end{array} \quad (12)$$

The query result ΔQR of the compensating query are the tuples $\{<+,\ 3,\ 12>,\ <-,\ 4,\ 16>\}$. Because the data update DU is an insertion, we need to remove tuples corresponding to ΔQR from the QR: $QR' = QR - \Delta QR$. As result, we remove tuple $<3,12>$ from and insert tuple $<4,16>$ to the query result QR. Now QR is in the correct result.

7 Proof of Correctness of the LC Algorithm

In order to give out the proof, we first introduce a more formal example of **maintenance-concurrent** data updates.

7.1 A General Example For Delete-Condition VDC

In general, the problem occurs when a data update occurred at the IS at a time, during or before which the IS was still processing a maintenance query from the middle layer. There are four adaptation queries (namely, TAQ, CTAQ, AAQ and MAAQ) that require remote IS access and hence could cause that **maintenance-concurrent** problem. We select the most simplest of the adaptation queries, namely TAQ[13], to illustrate the problem.

We now study the case of del-condition VDC, which can be caused by the drop-relation or the drop-attribute schema change (SC). We assume two relations R1 and R2 in IS1 and IS2, respectively. The data warehouse DW is defined by: $DW = \sigma_{C1} R1 \bowtie_{JC} R2$, where $\sigma_{C1} R1$ denotes the subset of R1 that satisfies condition C1 and $\bowtie_{JC}$ denotes the join between the R1 and R2 on join condition JC. By definition, we have:

$R1 = \sigma_{C1} R1 + \sigma_{\overline{C1}} R1$.

Let DU2 denote some data update of R2, i.e., a tuple that is either added or deleted from R2. Then $R2'$, the updated R2 after the data update DU2, is defined by $R2' = R2 + DU2$, with the "+" operator as defined in Section 3.

First, if we drop C1 from the DW, we have to apply the $TAQ = \sigma_{\overline{C1}} R1 \bowtie_{JC} R2$ to the DW1:

$DW1 = DW + \sigma_{\overline{C1}} R1 \bowtie_{JC} R2 = \sigma_{C1} R1 \bowtie_{JC} R2 + \sigma_{\overline{C1}} R1 \bowtie_{JC} R2 = R1 \bowtie_{JC} R2$

Then, if DU2 happens after the data warehouse handles the drop condition C1 rewriting request, we have following data warehouse DW2:

$DW2 = DW1 + R1 \bowtie_{JC} DU2 = R1 \bowtie_{JC} R2 + R1 \bowtie_{JC} DU2 = R1 \bowtie_{JC} (R2 + DU2) = R1 \bowtie_{JC} R2'$

The extent of DW2 is what the user expected, i.e., it is consistent with the current state of the ISs.

Second, if DU2 happens before data warehouse handles the drop condition C1 request, the current $TAQ = \sigma_{\overline{C1}} R1 \bowtie_{JC} R2'$ is affected by the data update DU2. So, we have DW1':

$DW1' = DW + \sigma_{\overline{C1}} R1 \bowtie_{JC} (R2 + DU2) = \sigma_{C1} R1 \bowtie_{JC} R2 + \sigma_{\overline{C1}} R1 \bowtie_{JC} R2' = \sigma_{C1} R1 \bowtie_{JC} R2 + \sigma_{\overline{C1}} R1 \bowtie_{JC} (R2 + DU2) = (\sigma_{C1} R1 + \sigma_{\overline{C1}} R1) \bowtie_{JC} R2 + \sigma_{\overline{C1}} R1 \bowtie_{JC} DU2 = R1 \bowtie_{JC} R2 + \sigma_{\overline{C1}} R1 \bowtie_{JC} DU2 = DW1 + \sigma_{\overline{C1}} R1 \bowtie_{JC} DU2$.

Then, after VM handles the DU2 of IS2, we have DW2':

$DW2' = DW1' + R1 \bowtie_{JC} DU2 = DW1 + R1 \bowtie_{JC} DU2 + \sigma_{\overline{C1}} R1 \bowtie_{JC} DU2 = DW2 + \sigma_{\overline{C1}} R1 \bowtie_{JC} DU2$

This $DW2'$ is different from the previous $DW2$. It has the extra item, i.e., $\sigma_{\overline{C1}} R1 \bowtie_{JC} DU2$. In the second case, the data update DU2 happened after the schema change, but the query sent by the DW for the schema change is executed later than the data update DU2 in the information source. So, here DU2 is a **maintenance-concurrent** data update.

From this example, we can see that under the drop-condition VDC, **maintenance-concurrent** data updates will

[13] TAQ is the adaptation query designed to fix the extent after a del-condition VDC.

result in an incorrect extent of the data warehouse. That problem really occurs because the IS returns the affected query result to the middle layer at a time when the middle layer is not yet aware of the conflicting data update.

7.2 Proof

From the previous discussions, we can now derive the following two theorems to prove the correctness of the LC algorithm.

Theorem 1 *If an update X(n)[j] is* **maintenance-concurrent** *with an update (SE or DU) Y(m)[i] as defined by Definition 4, that could only be Y(m)[i] had sent a Q(m)[j] to IS[j].*

This theorem can be explained by the definition of **maintenance-concurrent** . From the second condition of Definition 4, we know that the **maintenance-concurrent** data update $X(n)[j]$ must received before the $QR(m)[j]$. The query result $QR(m)[j]$ means there must be a query Q(m)[j] that had been send to IS[j] for the update $Y(m)[i]$.

Theorem 2 *Local Compensation as given in Section 6 can solve the* **maintenance-concurrent** *problem.*

Proof: Recall from Section 4 that the view adaptation query can be either AAQ, MAAQ, TAQ, or CTAQ, as all have the possibility to cause maintenance concurrent DUs. In this proof, we now focus on the correctness for TAQ and CTAQ, while AAQ and MAAQ could be treated very similarly. We notice that the problems happens with delete-relation (where) VDC or delete-condition VDC that are handled by CTAQ or TAQ, respectively. The **maintenance-concurrent** problem could only happen in TAQ or CTAQ, and CTAQ can be reduced to TAQ. Hence TAQ is the only technique we need to use when adapting the delete-condition VDC. So, we can prove the correctness of Theorem 2 by proving the correctness of LC for the delete-condition VDC.

The problem can be described as follows. Assume we have the relations $R_1, R_2, ..., R_n$. The current extent of the data warehouse is $DW = \sigma_C R_1 \bowtie_{JC_1} R_2 \bowtie_{JC_2} ... \bowtie_{JC_{k-1}} R_k$[14]. $\sigma_C R_1$ denotes the subset of R_1 that satisfies condition C with C is a local condition imposed on R_1 in the data warehouse. By definition, we have:

$R_1 = \sigma_C R_1 + \sigma_{\overline{C}} R_1$.

$D_{i_1}, ..., D_{i_k}$ are the data updates on the relations $R_{i_1}, ..., R_{i_k}$ for $2 \leq i_j \leq n$ and $j = 1, ..., k$. The updated relations $R_{i_1}, ..., R_{i_k}$ after applying the sequence of data updates $D_{i_1}, ..., D_{i_k}$ are defined by:

$R'_{i_j} = R_{i_j} + D_{i_j}, j = 1, 2, ..., k$, for some $k \geq 1$.

Let VDC(a)(b) denote the view definition change that will drop the condition C in the data warehouse definition. VDC(a)(b) is caused by SC(a) that denotes a schema change at some IS. "a" denotes the sequential number of schema change. "b" denotes the subscript of the view definition change VDC(a)(b).

DW_s, the result of the data warehouse after VA handles VDC(a)(b), is defined by:

$DW_s = DW + \sigma_{\overline{C}} R_1 \bowtie_{JC_1} ... \bowtie_{JC_{k-1}} R_k$. The extent of the relation $\sigma_{\overline{C}} R_1 \bowtie_{JC_1} ... \bowtie_{JC_{k-1}} R_k$ is generated by the corresponding TAQ (see Appendix A for a full definition).

[14] Here we only concentrate on the TAQ, and thus we only show one condition in the relation R_1, and omit the conditions in the rest of the relations

Let DW_s^k denote the result of the data warehouse after VA handles VDC(a)(b) with **maintenance-concurrent** $D_{i_1}, ..., D_{i_k}$. DW_s^k is defined by:

$DW_s^k = DW + \sigma_{\overline{C}} R_1 \bowtie_{JC_1} ... \bowtie_{JC_{i_1-1}} R'_{i_1} \bowtie_{JC_{i_1}} ... \bowtie_{JC_{i_k-1}} R'_{i_k} \bowtie_{JC_{i_k}} ... \bowtie_{JC_{n-1}} R_n$.

We denote TAQ by Q(a)(b). Then the result returned QR(a)(b) is:

$QR(a)(b)[1] \bowtie_{JC_1} ... \bowtie_{JC_{i_1-1}} QR(a)(b)[i_1] \bowtie_{JC_{i_1}} ... \bowtie_{JC_{i_k-1}} QR(a)(b)[i_k] \bowtie_{JC_{i_k}} ... \bowtie_{JC_{n-1}} QR(a)(b)[n]$.

where $QR(a)(b)[1] = \sigma_{\overline{C}} R_1 \bowtie_{JC_1}$, and $QR(a)(b)[i_j] = R'_{i_j}, j = 1...k$, and $QR(a)(b)[j] = R_j, j = 2...n$ except $i_1...i_k$.

The LC algorithm will adjust each affected QR(a)(b)[d] query result by subtracting ΔQR(a)(b)(c)[d] from it, here d is the subscript of the information source. If we abbreviate the query ΔQR(a)(b)(c)[d] by D_d, then we can write $LC(QR(a)(b)[d]) = QR(a)(b)[d] - D_d$.

We now will prove that after we apply the LC query to all the **maintenance-concurrent** D_j, $j = i_1..i_k$, then the result of data warehouse will be equal to DW_s.

If we apply $LC(DW_s^i, j)$ to all the **maintenance-concurrent** D_{i_k}, we get:

$DW_s^k = DW + \sigma_{\overline{C}} R_1 \bowtie_{JC_1} ... \bowtie_{JC_{i_1-1}} LC(QR(a)(b)[i_1]) \bowtie_{JC_{i_1}} ... \bowtie_{JC_{i_k-1}} LC(QR(a)(b)[i_k]) \bowtie_{JC_{i_k}} ... \bowtie_{JC_{n-1}} R_n$

$= DW + \sigma_{\overline{C}} R_1 \bowtie_{JC_1} ... \bowtie_{JC_{k-1}} R_k$

$= DW_s$

Q.E.D.

8 Discussion of SDCC Overhead

Three major algorithms are required to cooperate in an integrated fashion in our environment, namely, VM, VA and VS. Thus, a natural issue to explore is the overhead of our SDCC system beyond the fixed costs of these conventional algorithms.

View synchronization algorithms as employed by the EVE system [LNR97b] only evolve the view definitions at the data warehouse without sending down any query to remote information sources. Hence, there is no cost in terms of network communication or disk I/O access for view synchronization. So that we don't need to consider the VS algorithm further.

The VM component for which we employ the SWEEP algorithm[15] is independent from the LC algorithm. If all updates are data updates, then the SDCC system works just like the conventional VM system. So, the number of messages transfered between DW and IS is the same as that for SWEEP. The same also holds for the data sizes shipped across the network.

Because LC performs only local compensation, no network action is required. There only two more messages transferred for each SC. Namely, ACK is sent from the DW to the IS, and a Re-ACK is received by the DW. If we

[15]ECA [ZGMHW95], Strobe [ZGMW96] and SWEEP [AAS97]are all algorithms developed in the VM area, the EVE algorithm is developed in VS area, and the View Adaptation algorithm is developed in VA area. ECA is for a centralized database system, and thus would not fit in our framework. Strobe needs the assumption that views include the keys of all of the relations involved. In order to discuss the **maintenance-concurrent** problem in a more general case, we want to get rid of such a strong assumption, so the current version of SDCC does not select the Strobe algorithm as the VM algorithm.

assume the number of messages send by the VA is $msg(VA)$, then the number of messages of SDCC is $msg(VA)+2$. Because the size of ACK and Re-ACK messages is very small, the data sizes transferred between DW and IS in SDCC are effectively the same as the data sizes transfered between DW and IS for VA.

So, in general, the overall performance of SDCC is equal to the number of messages: $m \times msg(SWEEP) + n \times (msg(VA) + 2)$, where m denotes the number of DUs and n denotes the number of SCs. If we have the number of information sources large enough, so that $msg(VA) >> 2$, then we can ignore the constant 2. Then, we get the $msg(SDCC) = m \times msg(SWEEP) + n \times (msg(VA) + 2)$, with m data updates and n schema changes. In short, we have determined that the network communication costs are the same for SDCC as that for the combination of SWEEP and VA. However, there is some additional local cost in SDCC beyond the local costs in SWEEP and VA in isolation. Assuming a sufficiently large enough memory in the DW to store all the delta query results, as often assumed by other approaches like ECA [ZGMHW95], so there are no additional IO costs for the SDCC at the DW site. In conclusion, the performance of SDCC is roughly comparable to that of the simple combination of those of SWEEP and VA.

9 Concurrency Handling Level Criteria

In order to compare our system to others, we introduce a "concurrency handling level criteria" in this section. First, we give the definitions of the terms and then we discuss our system and compare it with other systems using the new terms.

9.1 Definitions of Concurrency Handling Level Criteria

In this section, we assume two kinds of updates of the IS space, namely, data updates (DU) and schema changes (SC). All the "updates" in the following definitions can be either DU or SC. First, we give basic definitions.

Definition 6 *We say that two updates are* **independent** *from one another if the updates are not* **maintenance-concurrent** *with one other. For more than two updates, if every pair of them is independent, we say these updates are independent.*

Definition 7 *The state of the IS space, denoted by ss, is the snapshot of the extents of all ISs at a given time.*

Definition 8 *The states of DW, denoted by ws, correspond to the extents of DW if the extent of DW changed after the effect of an update from the underlying IS space is propagated up.*

Definition 9 *The state of the DW (or the state of the IS space) is a* **final state** *if the next update is an update independent from all the previous updates.*

Definition 10 *We say a view maintenance algorithm is* **correct***, if the final state of the DW generated by the algorithm is same as the view recomputed over the final state of the corresponding IS space* [16].

Definition 11 *Equation* $\underline{ws_i < ws_j}$ *(or* $\underline{ss_i < ss_j}$*) means that the state of DW* $\underline{ws_i}$ *(or a state of IS space* $\underline{ss_i}$*) happened prior than the state of DW* $\underline{ws_j}$ *(or a state of IS space* $\underline{ss_j}$*).*

Depending on these definitions, the four levels of concurrency handling are defined below.

Definition 12 *We say an algorithm is* **data concurrency handling***, if the algorithm is* correct *under* **maintenance-concurrent** *data updates with the term "correct" defined as in Definition 10*

Definition 13 *We say an algorithm is* **semi-schema concurrency handling***, if the algorithm is* correct *when* **maintenance-concurrent** *data updates happened after one schema change SC and no more schema changes are* **maintenance-concurrent** *with SC.*

Definition 14 *We say an algorithm is* **schema concurrency handling***, if the algorithm is* correct *under* **maintenance-concurrent** *schema changes.*

Definition 15 *We say an algorithm is* **fully concurrency handling***, if the algorithm is* correct *when* **maintenance-concurrent** *data updates and* **maintenance-concurrent** *schema changes happened in any order.*

9.2 Comparison of Existing Maintenance Algorithms

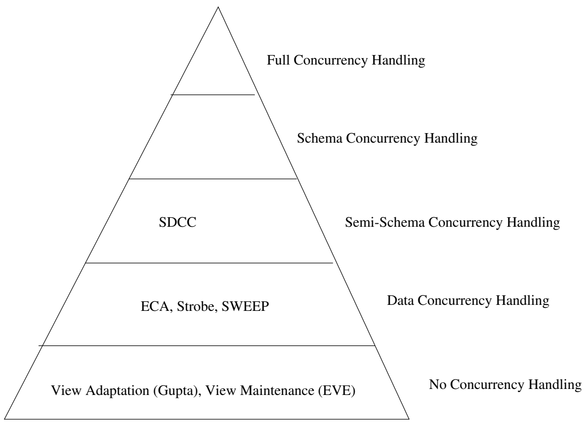

Figure 5: Algorithm Comparition Diagram

Figure 5 shows the levels of current existing data warehouse maintenance algorithms. As we can see, the view adaptation algorithm provided by Gupta et. al [GMR97]. and the view synchronization algorithm provided by the EVE DSRG group have not been designed to handle any kind of concurrency. Current view maintenance algorithms like SWEEP, Strobe and ECA can handle data concurrency. The system we described in this paper can handle semi-schema concurrency.

[16]This definition conrresponds to "convergence" defined in Zhuge et. al.'s [ZGMW96]

10 Conclusions

Data warehousing is a widely used technique for gathering and integrating data from heterogeneous and autonomous information sources. There are a lot of different data warehouse maintenance algorithms developed to keep the data warehouse up-to-date. In particular, we have three kinds of maintenance, view maintenance maintains the extent of the data warehouse when the data of ISs is changed [AAS97]; view synchronization maintains the view definition when the schema of the ISs changed [LNR97b]; view adaptation maintains the extent of the view when the view definitions changed [GMR97]. These three algorithms were proposed independently. However, because concurrent data updates and schema changes could be happening at the information sources at any time, we cannot simply put them together and expect the data warehouse to now be maintained consistently under concurrent data and schema updates. Consequently, previous algorithms may need to be either reexamined or new algorithms may need to be developed.

We have presented a new framework for making those algorithms work together without requiring modification of these three known solutions. Our SDCC system with the support of the LC algorithm successfully solves the problem of concurrent DUs and SCs. The SDCC system is shown to correctly maintain the data warehouse. Our initial evaluation analyzing the two cost factors of the number of messages and data traffic suggests that the SDCC system adds no overhead to the cost of integrating data warehouse maintenance algorithms. In short, it successfully solves the problem without affecting the overall system performance.

The SDCC system currently doesn't aim to achieve high performance but rather correctness under concurrency on one hand and reuse of known techniques for data warehouse maintenance on the other hand. In the future we plan to explore optimization techniques to reduce the overall system performance.

Acknowledgments. The authors would like to thank students at the Database Systems Research Group at WPI for their interactions and feedback on this research. In particular, we are grateful to Andreas Koeller and Yong Li for helping to implement the EVE system. We also thank Amy Lee and Anisoara Nica from the University of Michigan for interactions on this EVE project.

References

[AAS97] D. Agrawal, A. El 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.

[CTL+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.

[FMZ94] F. Ferrandina, T. Meyer, and R. Zicari. Implementing lazy database updates for an object database system. In *Proc. of the 20th Int'l Conf. on Very Large Databases*, pages 261–272, 1994.

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

[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.

[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.

[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.

[LNR97a] 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 CASCON97, Best Paper Award*, pages 1–14, November 1997.

[LNR97b] A. J. Lee, A. Nica, and E. A. Rundensteiner. The EVE Framework: View Synchronization in Evolving Environments. Technical Report WPI-CS-TR-97-4, Worcester Polytechnic Institute, Dept. of Computer Science, 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.

[Nic98] A. Nica. *View Evolution Support for Information Integration Systems over Dynamic Distributed Information Spaces*. PhD thesis, University of Michigan in Ann Arbor, July 1998.

[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.

[NR98a] 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.

[NR98b] A. Nica and E. A. Rundensteiner. Using Complex Substitution Strategies for View Synchronization. Technical Report WPI-CS-TR-98-4, Worcester Polytechnic Institute, Dept. of Computer Scien ce, 1998.

[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.

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

[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.

[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 View Adaptation of View Extent after View Definition Change

A.1 Introduction

View adaptation is the process that computes the extent of modified view definition V' by utilizing the previously materialized extent of the view V [GMR97]. This process is attended to be used to change the view extent after the view definition is changed. In this paper, we only use a subset of Gupta's view adaptation algorithms [GMR97] as suitable for basic SELECT-FROM-WHERE views. In order to integrate the VA with our distributed environment system, we had to modify the centralized view adaptation algorithm to work in this loosely-coupled environment.

By our subset, we defined six primitive VDCs:

- Addition or deletion of an attribute in the SELECT clause.
- Addition or deletion of a relation in the FROM clause, with associated addition or deletion of equijoin conditions in the WHERE clause and attributes in the SELECT clause.
- Addition or deletion of a condition in the WHERE clause.

We use five kinds of adaptation techniques to adapt the view extent for the six VDCs. They are:

- Local adaptation Query (LQ) which will only query within the data warehouse.
- Attribute Additive adaptation Query (AAQ) which will add a new attribute to the data warehouse.
- Multiple Attribute Additive adaptation Query (MAAQ) which will add more than one new attribute to the data warehouse.
- Tuple Additive adaptation Query (TAQ) which will add more tuples to the data warehouse due to dropping of one condition.
- Complex Tuple Additive adaptation Query (CTAQ) which will add more tuples to the data warehouse due to dropping more than one condition.

The adaptation queries AAQ, MAAQ, TAQ, and CTAQ handle the VDCs add-attribute, add-relation with adding attributes, delete-condition, and delete-condition with deleting conditions respectively. The rest VDCs, including delete-attribute, add-condition, add-relation with adding conditions and delete-relation with deleting-attributes are handled by LQ adaptation query. Table 5 shows the relationship between VDCs and the corresponding adaptation techniques.

	Relation		Attribute	Condition
	(Select)	(Where)		
Add	MAAQ	LQ_4	AAQ	LQ_2
Delete	LQ_3	CTAQ	LQ_1	TAQ

Table 5: Relationship between View Definition Change and Adaptation Query

Figure 6 shows a more general representation of the relationships between VDCs and adaptation techniques. In Figure 6, the rectangle labeled "data warehouse relation" represents the extent of the view. The rectangle labeled "AAQ" (MAAQ) indicates to add one (multiple) column to the view by using an AAQ (MAAQ) adaptation query. The rectangle labeled "TAQ" (CTAQ) indicates addition of tuples due to dropping one (multiple) condition from the view by using a TAQ (CTAQ) adaptation query.

The adaptation queries for all view definition changes are defined in Tables 6 and 7. Table 6, called the "VDC_Adapt" table, stores which adaptation query is needed for which specific VDC. Table 7, called the "Adapt_Query" table, stores the type and template of each view adaptation query.

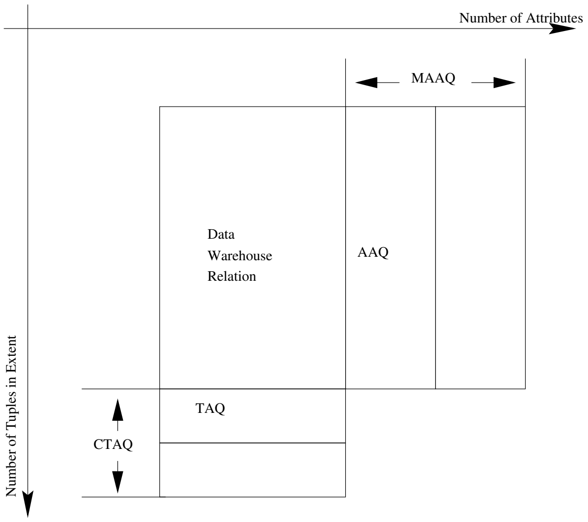

Figure 6: Relationship between View Definition Changes and Adaptation Queries

A.2 Example of View Adaptation

Assume we define a view V as:

$$\begin{array}{ll} \text{CREATE VIEW} & V \text{ AS} \\ \text{SELECT} & R.A \\ \text{FROM} & R \end{array} \tag{14}$$

Suppose we wish to add the $R.B$ attribute to the view. The updated view definition is:

$$\begin{array}{ll} \text{CREATE VIEW} & V \text{ AS} \\ \text{SELECT} & R.A, R.B \\ \text{FROM} & R \end{array} \tag{15}$$

Based on Tables 6 and 7, the adaptation query for this case now should be:

$$\begin{array}{lllllll} \text{ALTER TABLE} & V & \text{ADD} & R.B & & & \\ \text{UPDATE} & V & \text{SET} & R.B & = (\text{SELECT} & R.B & \\ & & & \text{FROM} & R & & \\ & & & \text{WHERE} & R.K = V.K) & & \end{array} \tag{16}$$

This adaptation strategy makes several assumptions including the fact that a key value exists in the view.

A.3 Detailed Algorithm of View Adaptation

In general, view adaption is composed of the following three steps:

1. receive a set of VDCs from the VS module.
2. generate VDCs from V and V'.
3. apply appropriate adaptation depending on the VDCs (Tables 6 and 7).

Here is a more detailed description of the third step.

Original View Definition

$$\begin{array}{ll} \text{CREATE VIEW} & V \text{ AS} \\ \text{SELECT} & A_1,\ A_2,\ ...,\ A_n \\ \text{FROM} & R_1,\ R_2,\ ...,\ R_n \\ \text{WHERE} & C_1\ AND\ C_2\ ...\ AND\ C_m \end{array} \tag{13}$$

View Definition Change	Redefined View V'	Adapt Query Type
Add Attribute A from Relation $R_i, 1 \leq i \leq m$	$SELECT\ A,\ A_1,\ A_2,\ ...,\ A_n$ $FROM\ R_1\ \&\ ...\ \&\ R_m$ $WHERE\ C_1\ AND\ ...\ AND\ C_k$	AAQ
Delete Condition C_1	$SELECT\ A_1,\ A_2,\ ...,\ A_n$ $FROM\ R_1\ \&\ ...\ \&\ R_m$ $WHERE\ C_2\ AND\ ...\ AND\ C_k$	TAQ
Add Relation(SELECT) $R(B_1,\ ...,\ B_j)$	$SELECT\ A_1,\ A_2,\ ...,\ A_n,\ B_1,\ ...,\ B_j$ $FROM\ R\ \&\ R_1\ \&\ ...\ \&\ R_m$ $WHERE\ C_1\ AND\ ...\ AND\ C_k$	MAAQ
Delete Relation(WHERE) $R_1(C_1,\ ...,\ C_j)$	$SELECT\ A_1,\ A_2,\ ...,\ A_n$ $FROM\ R_2\ \&\ ...\ \&\ R_m$ $WHERE\ C_{j+1}\ AND\ ...\ AND\ C_k$	CTAQ
Delete Attribute A_1	$SELECT\ A_2,\ ...,\ A_n$ $FROM\ R_1\ \&\ ...\ \&\ R_m$ $WHERE\ C_1\ AND\ ...\ AND\ C_k$	LQ_1
Add Condition C	$SELECT\ A_1,\ A_2,\ ...,\ A_n$ $FROM\ R_1\ \&\ ...\ \&\ R_m$ $WHERE\ C\ AND\ C_1\ AND\ ...\ AND\ C_k$	LQ_2
Delete Relation(SELECT) $R_1(A_1,\ ...,\ A_j)$	$SELECT\ A_{j+1}, ...,\ A_n$ $FROM\ R_2\ \&\ ...\ \&\ R_m$ $WHERE\ C_1\ AND\ ...\ AND\ C_k$	LQ_3
Add Relation(WHERE) $R(D_1,\ ...,\ D_j)$	$SELECT\ A_1, ...,\ A_n$ $FROM\ R\ \&\ R_1\ \&\ ...\ \&\ R_m$ $WHERE\ C_1\ AND\ ...\ AND\ C_k\ AND\ D_1\ AND\ ...\ AND\ D_j$	LQ_3

Table 6: View Definition Changes and View Adaptation Queries

PROCEDURE: `VA`
Input: `VDC(m) a set of view definition changes passed from VS module.`
Output: `void`
Algorithm:

```
FOR each VDC(m)[i] in Vector VDC(m)
    Q(m) = Adapt_Query.getQuery(V, VDC(m)[i])
    IF (Q(m).isLocal())
        localAdapt(Q(m));
    ELSE
        remoteAdapt(Q(m));
    END IF
END FOR
```

END PROCEDURE

Adaption Query Type	Adaption Query Expened in SQL	Local Query
AAQ	*ALTER TABLE V ADD A* *UPDATE V SET A = (SELECT A* *FROM S* *WHERE S.K = V.K)*	false
TAQ	*INSERT INTO V* *SELECT* A_1, ... , A_n *FROM* R_1 & ... & R_m *WHERE NOT* C_1 *AND* C_2 *AND ... AND* C_k	false
MAAQ	*ALTER TABLE V ADD* D_1, ..., D_j *UPDATE V* *SET* D_1, ..., D_j = (*SELECT* $R_{m+1}.D_1$, ..., $R_{m+1}.D_j$ *FROM* R_{m+1} *WHERE* $R_{m+1}.A$ = *V.B*)	false
CTAQ	*INSERT INTO V* *SELECT* A_1, ... , A_n *FROM* R_1 & ... & R_m *WHERE NOT* C_1 *AND* C_2 *AND ... AND* C_k ... *INSERT INTO V* *SELECT* A_1, ... , A_n *FROM* R_1 & ... & R_m *WHERE NOT* C_j *AND* C_{j+1} *AND ... AND* C_k	false
LAQ_1	*ALTER TABLE V DROP A*	true
LAQ_2	*DELETE FROM V* *WHERE NOT C*	true
LAQ_3	*ALTER TABLE V DROP* A_1, ..., A_j	true
LAQ_4	*DELETE FROM V* *WHERE NOT* D_1, ..., *OR NOT* D_j	true

Table 7: View Adaptation Queries

PROCEDURE: `remoteAdapt`
Input: `Q(m) as Adaptation Query`
Output: `void`
Algorithm:

```
SubQueryVector = Q(m).breakDown();
FOR each subquery Q(m)[j] in SubQueryVector{
    /* Send Query to and Receive Query Result from Query Engine */
    QR(m)[j] = QE(Q(m)[j]);
END FOR QR(m) = reAssemble(a Vector of sub-query results QR(m)[j]);
updateVE(QR(m);
```

END PROCEDURE

Table 8 shows all the functions and their meaning.

A.4 Justification of the Major Assumption

Assumption: IS are semi-cooperating. If there are data updates scheduled for the IS, they can do the updates immediately, but they will then also send the update notification to data warehouse. However, if there is a schema change request at an IS, they will first notify the DW of this impending change, and then await the ACK before

Method	of Class	Meaning
breakDown()	Q(m)	break down query Q(m) for each IS.
getQuery(V, VDC(m)[i])	Adapt_Query	get adaptation query for VDC from Adapt_Query table (Table 7).
isLocal()	Q(m)	check if query Q(m) that can be processed inside data warehouse.
localAdapt(Q(m))		adapt the extent of view within data warehouse.
QE(Q(m)[j])		execute query Q(m)[j] through Query Engine (QE).
reAssembly(a Vector of sub-query results QR(m)[j])		re-assemble sub-query results QR(m)[j] to QR(m).
remoteAdapt(Q(m))		adapt the extent of view from data of ISs.
updateVE(QR(m))		update VE by using QR(m).

Table 8: Functions and Their Meanings

executing the SC. This assumption is reasonable because schema evolution are often major restructuring, typically carefully planned by a system administrator, and do require significant execution time [FMZ94].

The reason for this assumption is to allow time for processing the view adaptation query so that the later can get proper information from the IS in order to adapt the view extent. If the information at the ISs has been modified or removed via an SC before the middle layer handles it, the adaptation queries — proposed by Gupta et. al. [GMR97] for example — will fail as they cannot be understood by the ISs. Here is an example of this problem.

Assume we define a view like:

$$\begin{array}{ll} \text{CREATE VIEW} & V \text{ AS} \\ \text{SELECT} & R.A, R.B \\ \text{FROM} & R \\ \text{WHERE} & R.C = 5 \end{array} \tag{17}$$

Then, the IS that contains R is going to delete attribute R.C. This will cause the condition $R.C = 5$ to be dropped from the view V. We now get view definition V', which is:

$$\begin{array}{ll} \text{CREATE VIEW} & V' \text{ AS} \\ \text{SELECT} & R.A, R.B \\ \text{FROM} & R \end{array} \tag{18}$$

In this case, we need to send the following adaptation query down to the IS:

$$\begin{array}{ll} \text{SELECT} & R.A, R.B \\ \text{FROM} & R \\ \text{WHERE} & R.C \neq 5 \end{array} \tag{19}$$

However, without our assumption, the IS will not keep the R.C information in its schema (and thus database extent), and hence this query (19) will fail. The SDCC system would not work correctly.