**SUBMISSION to Information Systems**
Submission Date: July 19, 1999
Associated Editor: Professor Amr El Abbadi
Resubmission Date: Dec 20, 2000
2nd Resubmission Date: Aug 10, 2001

# Integrating the Maintenance and Synchronization of Data Warehouses Using a Cooperative Framework *

Xin Zhang and Elke A. Rundensteiner

Department of Computer Science
Worcester Polytechnic Institute
Worcester, MA 01609-2280
xinz | rundenst@cs.wpi.edu
phone: (508) 831-5857
fax: (508) 831-5815

**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 remains an unexplored problem however. This paper 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. The overhead of the SDCC solution beyond the costs of the known view maintenance algorithms it incorporates is shown to be negligible. Lastly, a refined hierarchy of consistency levels for the state of a data warehouse with respect to its underlying dynamic environment is presented, now incorporating both dynamicity of the data and the schema. The SDCC solution is shown to reach a semi-concurrency level of consistency, not reached by any prior DW system.

**Keywords:** Data Warehousing, View Maintenance, Concurrency of Updates, Data Updates and Schema Changes, View Consistency, Distributed Information Sources.

---

*This work was supported in part by several grants from NSF, namely, the NSF NYI grant #IRI 97–96264, the NSF CISE Instrumentation grant #IRIS 97–29878, and the NSF grant #IIS 99–88776. Dr. Rundensteiner would like to thank our industrial sponsors, in particular, IBM for the IBM partnership award and Verizon Communications for partial support of Xin Zhang.

# 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 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 warehouse [AESY97], [ZWGM97], [ZGMW96], [ZGMHW95], [BLT86], [GJM96], [Wid95], [CGL+96]. Two exceptions are Gupta et al. [GMR97] and Mohania et al. [MD96] instead focus on the problem and 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, LNR97, NR98a, NLR98, Nic99] is the first effort of tackling the problem of how to preserve the view definition itself under schema 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 semi-autonomous. This implies they will update their data and schemas independently and possibly without any concern for how this may affect the DW defined upon them. They will not be aware of or 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 [AESY97], they may do the IS updates in any order and at any time. In that case, any new data update at an IS may interfere with the process of view maintenance for the previous data updates. Current view maintenance algorithms only handle the

concurrency of data updates at ISs, while our work 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 [AESY97, GMR97, ZGMW96] would fail to handle this problem of concurrent data updates (DUs) and schema changes (SCs). Assume we have two information sources IS1 and IS2 with relations R and S, respectively. The upper region of Figure 1 shows the initial extent of R and S, with a view V defined by the SQL query Q-(1)[1]:

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

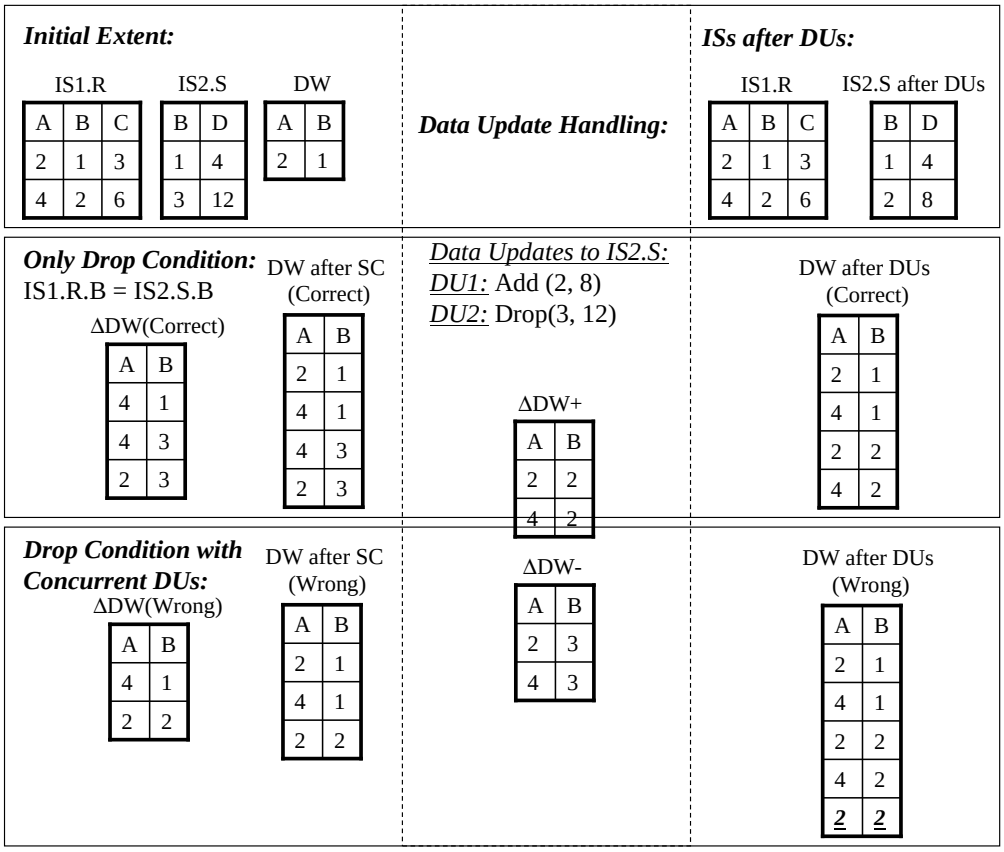


**Figure 1:** Example of Two Different Orderings of Schema Changes and Data Updates.

[1] We use Q-(i) to denote a query in Equation (i). We use QR-(i) to denote the query result of Q-(i).

We assume there is one schema change SC at R of IS1 to drop the attribute IS1.R.B from IS1.R. Before IS1 drops the attribute IS1.R.B it will nodify the middle layer to 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 [LNR01], would create the following view rewriting Q-(2):

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

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 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 Q-(3)[2] to IS1.R and IS2.S to get the additional tuples for V'

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

We assume here (as done in Gupta et al. [GMR97] ) that the incremental maintenance query Q-(3) can be processed before the actual drop of IS1.R.B is committed to IS1. Now assume two data updates at IS2.S that occur concurrently with this SC change at IS1: DU1 to drop tuple $< 3, 12 >$ from S, and DU2 to insert tuple $< 2, 8 >$ into S.

First, we assume the quiescence between the SC and two DUs is long enough for the middle layer to be able to complete the handling of the SC. As illustrated in the middle region of Figure 1, the query result of query Q-(3), denoted by QR-(3) (and visually depicted as table $\Delta DW$*(Correct)* in Figure 1), correctly reflects the state of the IS space of the SC. After the two data updates have been handled next by the middle layer (say the SWEEP View Maintenance Algorithm [AESY97]),

[2]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 Q-(3) in our distributed system. See Section 6.4 for more details on this.

tuples $\{< 2,3 >, < 4,3 >\}$ would be dropped from DW, and tuples $\{< 2,2 >, < 4,2 >\}$ would be inserted to the DW. As the result, the final DW (shown as table *DW-after-DUs(Correct)* in Figure 1) will be consistent with the ISs after handling those two data updates.

Then, as illustrated in the lower part of Figure 1, we assume the quiescence between the SC and two DUs is too short to complete query Q-(3) first. As a result, DU1 and DU2 are committed at IS2 before query Q-(3) is being handled by IS2. The query result QR-(3) (shown as table $\Delta DW$*(Wrong)* in Figure 1) is affected by both DU1 and DU2 and thus contains some incorrect tuples. Results in refreshing the DW with this wrong $\Delta DW$, and thus in an incorrect DW extent. Next the two data updates are handled by the middle layer. As a response to this, we still need to drop tuples $\{< 2,3 >, < 4,3 >\}$ from DW, and insert $\{< 2,2 >, < 4,2 >\}$ to the DW. As a result, the DW denoted as *DW (wrong after DUs)* will have the redundant tuple $\{< 2,2 >\}$, 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, we utilize SWEEP [AESY97] for handling data updates of the ISs (view maintenance), because ECA [ZGMHW95] is for centralized database systems, and Strobe [ZGMW96] requires that the views include the keys of all of the relations involved. We select View Adaptation [GMR97] for extent updates after view definitions changes (view adaptation). The only available view synchronization solution thus far, which has been developed by the EVE-Project [LNR01], is selected for handling schema changes of the ISs (view synchronization). Our integration solution approach is however generic, and we expect that other algorithms could be 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 coordinate the execution of the algorithms. For this purpose, we develop an overall control strategy to handle both messages and data exchanges between these maintenance modules. 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, the Update Message Queue (UMQ) stores and orders the data and schema update messages from the ISs and associates appropriate identifications with all messages so to support the detection of faulty query results due to concurrency conflicts. Two, the Central Control (CC) unit incorporates protocols for the coordination of the execution of VS, VA and VM. The protocols will postpone the SC to be done on one IS while the other ISs can still continue to commit DUs and request to do SCs freely. We introduce a strategy for the detection of faulty query results based on a local DW-resident time stamp scheme. This is then complemented by the development of an algorithm for fault-correction. 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 negligible overhead on the performance of DW maintenance.

### 1.4 Contributions

Preliminary ideas related to this problem of concurrent data and schema updates are reported in a conference [ZR99c] and a one-page poster [ZR99b], in particular, they include:

- a characterization of the problem of data warehouse maintenance under concurrent schema and data updates;
- a strategy for the *detection of faulty data warehouse extent updates* in the context of this enhanced view maintenance process; and
- the design and development of the SDCC framework as a general *solution approach* for this problem. The key idea of SDCC is to exploit existing techniques for view maintenance, view adaptation, and view synchronization by incorporating protocols for the integration of these existing technologies.

Our current work makes the following contributions beyond these basic first steps:

- We investigate the state transactions to define the *dynamics of the SDCC framework* for the run-time processing of concurrent SCs and DUs. Then we identify the place to have the concurrency detection and analyze the concurrency behavior of the system.

- We develop an algorithm for the correction of the problem of maintenance-current updates. 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.

- We prove SDCC and in particular the LC algorithm *to be correct*. That means the data warehouse extent is proven to be consistent with the underlying information sources after the SDCC system maintains the data warehouse under concurrent SCs and DUs.

- We conduct a preliminary 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. The incorporating protocols increase the network communication (in terms of the number of messages) for SCs. The reminder of overhead is local CPU cost for concurrency detection. They are all shown to be minimal compared to the base costs.

- We propose a refined hierarchy of *consistency levels* for the state of a data warehouse with respect to its underlying dynamic environment, now incorporating both dynamicity of the data as well as the schema. We categorize SDCC as well as existing maintenance strategies utilizing this classification, and show the SDCC solution to reach the highest level of concurrency handling thus far, namely, the semi-concurrency level.

### 1.5 Outline of Paper

In the next section, we briefly review related research. We introduce 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. We give a proof of SDCC and particular the LC algorithm in Section 7. Section 8 discusses the overhead of the SDCC system. Section 9 identifies four levels of concurrency handling for data warehouse maintenance and classifies SDCC and other existing strategies. In Section 10, we conclude and discuss future directions of our work.

## 2 Related Work

Data warehousing has been recognized as a suitable technology 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], [AESY97]). They all are concerned with how to maintain the view extent at the data warehouse in the context of data updates at ISs. 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. [AESY97] propose the SWEEP-algorithm that can ensure consistency of the data warehouse in a larger number of cases compared to the Strobe [ZGMW96] 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 [LNR01, LKNR99, NR98a, NLR98] 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 [LNR97] 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. Such preferences indicate, 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 view rewriting triggered by schema changes of ISs is called view synchronization. Concurrency of schema or data updates have not been considered in EVE, they are however the focus of this current paper.

Gupta et. al. 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 for each type of view definition change. 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 problem of data maintenance after IS changes. Their techniques are not designed to handle concurrent schema changes or the interleaving of data and schema changes, rather they assume a completely static IS environment.

# 3 Background Material

Our system handles two types of data updates: insertions and deletions. For convenience, we propose an approach similar to [BLT86] of using a *counter* to indicate duplicates of tuples. A *positive* number denotes newly inserted or existing tuples, and a *negative* number denotes deleted tuples. The fact that a relation $r$ has $c$ duplicates of tuple $t$ is denoted as $t[c]$, with c an integer.

The propagation rules for tuple counters are as follows:

- Selection: Assume relation $r$ contains $t[c]$. The result relation $r'$ after a selection will have the tuple $t[c]$ if the tuple $t$ satisfies the select condition with $c$ preserved as in the original relation $r$, otherwise $t$ will not appear in $r'$ at all.
- Projection: Assume tuple $t'$ is the tuple $t$ after a projection. If the relation $r$ has $t[c]$, then the relation $r'$ after the projection will have the tuple $t'[c]$. If multiple different tuples $t_i[c_i]$ $i = 1..n$ project to one and the same tuple $t'$, then the relation $r'$ will have the tuple $t'$ with the counter values $\sum_{i=1}^{n} c_i$.
- Cartesian Product: Assume relation $r_1$ has a tuple $t_1[c_1]$, and relation $r_2$ has a tuple $t_2[c_2]$. The Cartesian product on relations $r_1$ and $r_2$ will have a tuple $t_1 \times t_2$ with counter $c_1 * c_2$.

SQL queries of type SPJ (select-project-join) can be represented as an expression composed of *selection*, *projection*, and *Cartesian product* operators. If we apply SQL on tuples with counters, the query result will follow the propagation rules described above.

Following [ZGMHW95], we define two binary operators, called + and -, that operate on relations with counters. For two relations $r_1$ and $r_2$ conforming to the same relational schema, let $t[c_1]$ and

$t[c_2]$ denote the identical tuple $t$ in $r_1$ and $r_2$ with counters $c_1$ and $c_2$ respectively. Then $r_1 + r_2$ and $r_1 - r_2$ will contain the tuple $t[c_1 + c_2]$ and $t[c_1 - c_2]$, respectively. If the counter $c_1 + c_2$ or $c_1 - c_2$ is zero, the tuple $t$ will be removed from the relation $r_1 + r_2$ or $r_1 - r_2$, respectively.

**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:*

$$\begin{array}{ll} \text{CREATE VIEW} & V \text{ AS} \\ \text{SELECT} & A_1,\ A_2,\ ...,\ A_n \\ \text{FROM} & R_1\ \&\ R_2\ \&\ ...\ \&\ R_m \\ \text{WHERE} & C_1\ AND\ C_2\ ...\ AND\ C_k \end{array} \tag{4}$$

*where the ampersands in the FROM clause means the relations $R_1, ..., R_m$ are combined by natural join, $C_1, ..., C_k$ are local conditions, and $A_1, ..., A_n$ are a subset of attributes of the relations $R_1, ..., R_m$, 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-attr, add-rel, change-attr-name, change-rel-name, drop-attr, and drop-rel.*
- *A* **view definition change (VDC)** *denotes a primitive change of one of the view definitions in the warehouse. It could be add-attr, add-rel, add-cond, delete-attr, delete-rel, and delete-cond.*

The VDCs can be explicitly requested by a user [GMR97] for a given view. They could also be automatically generated by a view evolution system like EVE [LNR01] 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 1. For each VDC, there is a corresponding adaptation query that adapts the extent of the view after the VDC[3]. Based on Table 1, we can see that a set of adaptation queries may be required in order to fix the views affected by a single SC.

[3]Please see Section A and Table 8 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.

| Schema Change | View Definition Change (VDC) |
|---|---|
| add-rel | None |
| change-rel-name | None |
| drop-rel | delete-rel (delete-attr \| delete-cond)<br>[ add-rel (add-attr \| add-cond)] |
| add-attr | add-attr |
| change-attr-name | None |
| drop-attr | (delete-attr \| delete-cond) [add-attr \| add-cond] |

**Table 1:** Relationship between SC and VDC (View Definition Changes)

In Table 1 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 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 positive or negative counters respectively ( see Section 3 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].*

| **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 for 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] 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. |
| 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 $i$. |
| $\Delta$Q(n)(m)(p)[i] | Local compensating query generated by our system addressing the concurrency of QR(n)(m)[i] and concurrent X(p)[i]. |
| $\Delta$QR(n)(m)(p)[i] | Query result of $\Delta$Q(n)(m)(p)[i]. |

**Table 2:** Notations and Their Meanings.

Table 2 defines the main notations that will be used in this paper. A sequence number $n$,

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 (See Table 2).

# 4 Definition of the Maintenance Concurrency Problem

A **maintenance-concurrent** update is an update that unexpectedly affects the query result returned by a data source to the middle layer to handle other updates.

**Definition 4** *Let $X(n)[j]$ and $Y(m)[i]$ denote either data updates or schema changes on IS[j] and IS[i] respectively, and $n$ and $m$ are the time stamps assigned to the updates respectively. 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 the update Y(m)[i] is received at DW.*

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

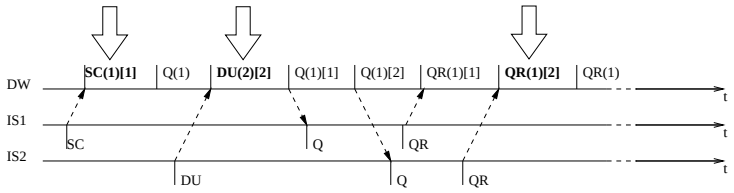


**Figure 2:** Time Line for a Maintenance Concurrent Data Update.

Figure 2 depicts 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, 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 of above are pointed out by arrows on top of Figure 2. So, here the DU is maintenance concurrent with SC by Definition 4. In other words, the DU is a maintenance concurrent DU by Definition 5.

Any remote query sent down to the information source space has the possibility of causing the **maintenance-concurrency** 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 two of several simplifying assumptions also made by the previous DW maintenance approaches in the literature [ZGMHW95, ZWGM97, ZGMW96, AESY97].

**Assumption 1** *Each IS only has one relation.*

This first assumption can be released [DZR99]. However, this extension is beyond the focus and scope of this current work.

**Assumption 2** *Network communication between an IS and the DW is FIFO.*

This second assumption guarantees us a key property exploited by our and also by previous fault detection and also correction techniques. Namely, that if one message (say a new change) from an IS is received before another one (say a query result), then the later (the query result) will incorporate the effects of the former (the new change).

Finally, we now add a new assumption of semi-cooperating ISs that represents a cornerstone for the overall SDCC framework. This then allows us to exploit existing DW maintenance related strategies for our new concurrency problem.

**Assumption 3** *Information sources are semi-cooperative. For data updates, ISs will first commit the update and then only notify the middle layer. However, if there is a schema change planned request at an IS, this IS must first notify the DW of this impending change and then await the acknowledgment from the DW before executing the SC.*

This assumption is reasonable because schema evolution often may involve some major restructuring, typically carefully planned by a system administrator. They also 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:

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

Then assume that 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 rewrite V into the view definition V', which is:

$$\begin{array}{ll} \text{CREATE VIEW} & V' \text{ AS} \\ \text{SELECT} & R.A, R.B \\ \text{FROM} & R \end{array} \tag{6}$$

In this case, we need to send the following adaptation query [GMR97] down to the IS:

$$\begin{array}{ll} \text{SELECT} & R.A, R.B \\ \text{FROM} & R \\ \text{WHERE} & R.C \neq 5 \end{array} \tag{7}$$

However, without our assumption, the IS will not keep the R.C information in its schema (and thus database extent), and hence this query Q-(7) will fail.

### 5.2 The Overall Architecture of SDCC System

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

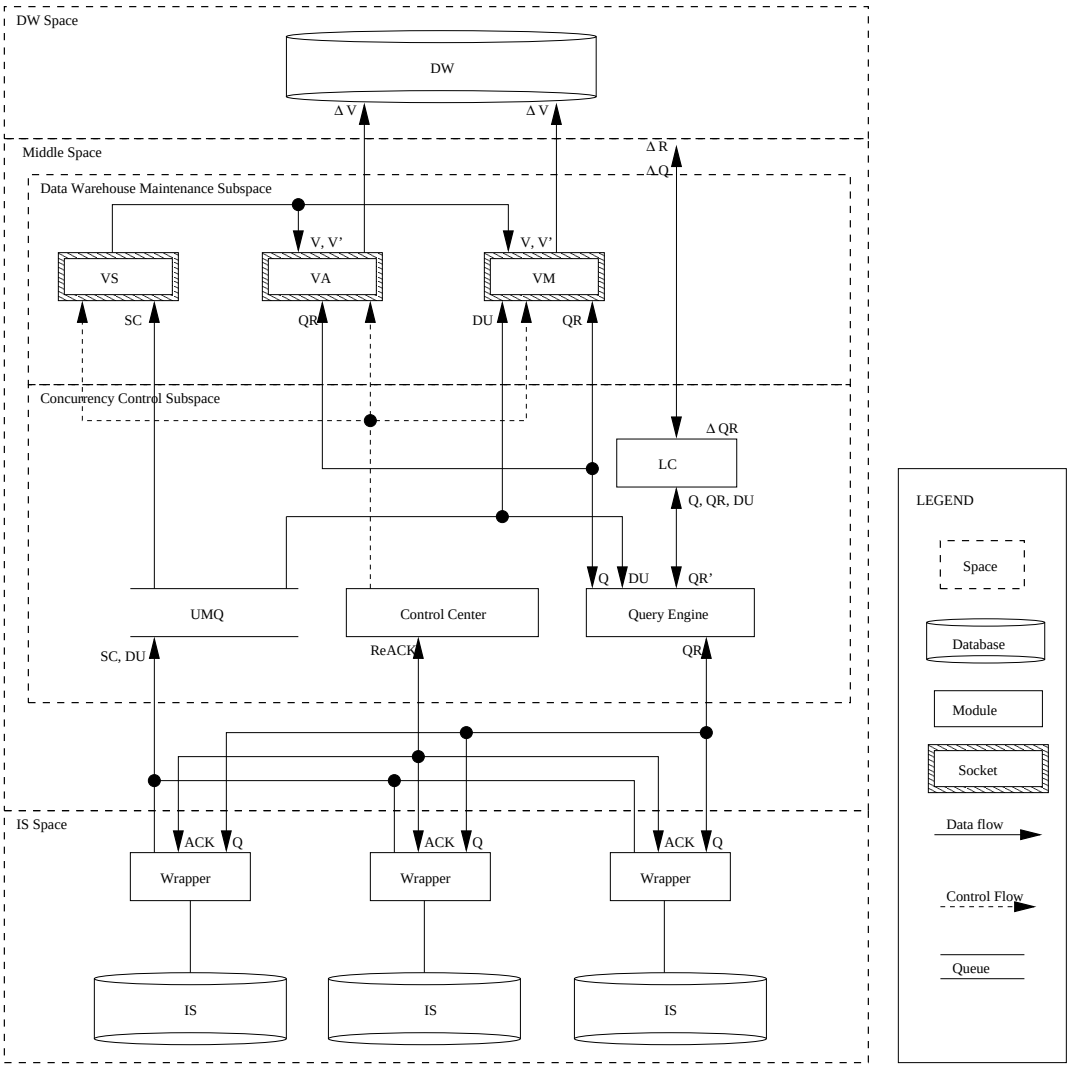


**Figure 3:** Framework of Local Compensation/SDCC System

- The View Maintainer (VM) maintains the extent of a view under IS data updates. The VM component should take a view definition and a DU as input and should generate view maintenance queries as output.

- The View Synchronizer (VS) rewrites the view definition in response to IS schema changes. The VS component should take a view definition and one SC as input and should generate a new view definition and a list of view definition changes (VDCs) as output.

- The View Adaptor (VA) adapts the extent of a view after its definition has been changed. The VA component should take a view definition and a list of view definition changes (VDCs) as input, and should generate view adaptation queries as output.

SDCC is general in the sense that it is not restricted to the use of a particular view management algorithm as long as it has the input and output as described above. In addition, the VM modules cannot have their independent database connections rather they will submit their queries via the connection by SDCC, for the reason of concurrency detection and correction. Also, they must be able to handle the SCs or DUs one by one. In order to fit existing incremental view maintenance algorithms into the SDCC framework, they must be able to 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 [AESY97] assume a static environment with respect to schema and thus are not equipped with this capability. However, this extension is relatively straightforward[4]. Figure 3 shows the modules of our framework and the data flow between them[5]. Table 3 lists the meaning of each symbol that appears in the framework figure.

To coordinate 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 processes queries sent 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

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

[5]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.

| 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. |
| $\Delta V$ | incremental view extent of data warehouse corresponding to tuples that will be inserted into or removed from extent of view V. |
| $\Delta Q$ | Local Compensation (LC) query. |
| $\Delta R$ | 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 3:** Notation used for Framework of SDCC System in Figure 3.

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 QE will have the following attributes as part of the message: QR(n)[i] (or QR(n)(m)[i], where 'm' denotes the sub-query of QR(n)[i]), where "n" denotes the update that generated this query, i.e., the update with the sequence 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** DU by Definition 4.

Each data warehouse maintenance module connects to the SDCC system by a socket that is responsible for assigning a number to the query generated by the module and for sending it down to the QE for processing. This number is always set to be identical to the sequence number of the update that is currently being processed. From the sequence number, we can easily know which update is handled by this query. QE will assign the same sequence number to the query result as that associated with 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 in Figure 4.

```
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)[i])
    END IF
    RETURN QR(n)[i];
END PROCEDURE
```

**Figure 4:** Description of QE 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 of the accomplishment of the maintenance of the data warehouse for a SC.`
- `IS will send back Re-ACK to middle space to notify it of the schema update completion at the IS.`

The cooperation protocol is created based on the assumption of semi-cooperating ISs, i.e., Assumption 3 that is justified in Section 5.1. 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

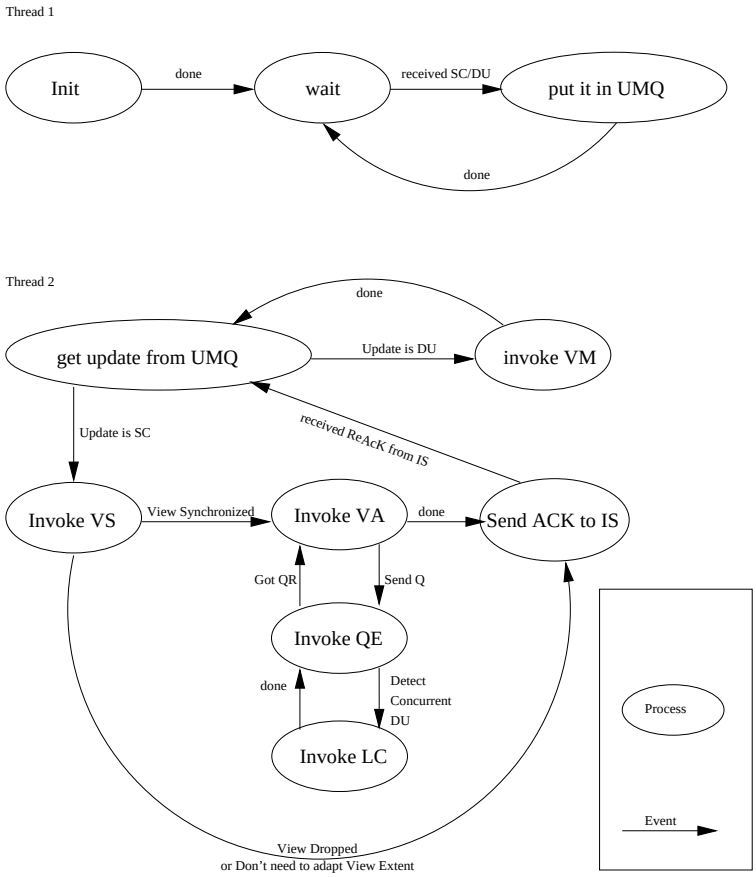


**Figure 5:** Event-Process Diagram of SDCC System

There are two major threads running in parallel in the system. The first thread depicted in Figure 5 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 5 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 View Sychonization (VS) [LNR01, NR98a, NLR98, Nic99]. If the synchronization is successful, VS will then invoke View Adaption (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, Query Engine (QE) will call Local Compensation (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 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 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. The algorithm used for VS could be any one of the view synchronization algorithms of the EVE system [LNR01]. The algorithm used for VM could be Strobe [ZGMW96], SWEEP [AESY97] 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 VA algorithm or Nica [Nic99] VA algorithm, which more closely matches the concurrent environments. In this paper, in particular, 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

## 6.1 Four Types of Concurrency Problems

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 Assumption 3, the IS will not change its schema based on the SC 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 context of VM algorithms [AESY97, 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 several concurrent data updates at other ISs. Below, we present the Local Compensation (LC) algorithm of the SDCC system, for which we assume 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.

### 6.2 Query Template For Plugging in Any View Adaption Query

The LC algorithm is generic, in the sense that it works with the query generated by the VA but does not care how the query has been generated. This is achieved with the help of the query template construct that is used to generate the LC query. A query template is a parameterized SQL query where relation names are variables that can be instantiated to specific relations. For the LC algorithm, 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 6.2**: In this example, assume we have the query:

$$\begin{array}{ll} \text{SELECT} & R.A \\ \text{FROM} & R \\ \text{WHERE} & R.B > 5 \end{array} \tag{8}$$

We can abstract a query template from the query given in Q-(8) by replacing the relation name "R" with the variable "$R", resulting in the template in Q-(9):

$$\begin{array}{ll} \text{SELECT} & \$R.A \\ \text{FROM} & \$R \\ \text{WHERE} & \$R.B > 5 \end{array} \tag{9}$$

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:

$$\begin{array}{ll} \text{SELECT} & S.A \\ \text{FROM} & S \\ \text{WHERE} & S.B > 5 \end{array} \tag{10}$$

### 6.3 General Description of Local Compensation Strategy

Local compensation is concerned with the query that has been sent to the IS and is affected by **maintenance-concurrent** data updates of that IS. LC tries to erase any faults, caused by the

**maintenance-concurrent** data update, from the query result. 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.

**PROCEDURE:** LC
**Input:**
Sub-query result QR(m)(n)[j] from IS[j],
Sub-query Q(m)(n)[j],
**maintenance-concurrent** data updates DU(p)[j] from IS[j] stored in UMQ,
where $p > m$, $1 \leq j \leq k$, with $k$ number of ISs.
**Output:**
Adjusted QR'(m)(n)[j] with effect of all **maintenance-concurrent** DU(p)[j], where $p > m$, erased.
**Algorithm:**
1. Create one temporary relational table $\Delta R_j$ for all **maintenance-concurrent** DU(p)[j].
2. Create one local compensating query $\Delta$Q(m)(n)[j] from query template of Q(m)(n)[j] and the relational table $\Delta R_j$ created in step 1.
3. Get $\Delta$QR(m)(n)[j] by executing LC query $\Delta$Q(m)(n)[j] on the table $\Delta R_j$.
4. Get QR'(m)(n)[j] by subtracting $\Delta$QR(m)(n)[j] from QR(m)(n)[j].

**END PROCEDURE**

**Figure 6:** Description of LC Procedure

The LC procedure defined in Figure 6 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 counter 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 $\Delta$Q(m)(n)[j], from the original adaptation query Q(m)(n)[j] by using query template techniques described in Section 6.2. In step three, LC executes the compensation query on the temporary table and gets the faulty tuples denoted as $\Delta$QR(m)(n)[j]. In the last step, LC separates out the faulty tuples from the affected query result, denoted by $\Delta$QR(m)(n)[j], by subtracting the inserted faulty tuples and adding the deleted faulty tuples.
We observe from the description given above that the Local Compensation (LC) algorithm:

- sends no (remote) 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.

### 6.4 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. First, the view definition V (Q-(1)) is evolved to view definition V' (Q-(2)) for the schema evolution SC of dropping the attribute IS1.R.B from relation IS1.R. Then the adaptation query Q-(3) 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 after adapting the schema change (*DW after SC (Wrong)* in Figure 1). In particular the tuple $\{ < 2, 2 > \}$ is extra and tuples $\{< 4, 3 >, < 2, 3 >\}$ are missing, and thus need to be detected and compensated for.

In order to execute the query over distributed information sources, we break down the query Q-(3) for two information sources: query Q-(11) is sub-query for IS1, query Q-(12) is sub-query for IS2, and query Q-(13) is the assembly query.

$$\begin{array}{lll} \text{CREATE tempTABLE} & TempIS1 & \\ \text{SELECT} & IS1.R.A,\ IS1.R.B & \\ \text{FROM} & IS1.R & \end{array} \qquad (11)$$

$$\begin{array}{lll} \text{CREATE tempTABLE} & TempIS2 & \\ \text{SELECT} & IS2.S.B & \\ \text{FROM} & IS2.S & \end{array} \qquad (12)$$

$$\begin{array}{ll} \text{SELECT} & TempIS1.A,\ TempIS2.B \\ \text{FROM} & TempIS1,\ TempIS2 \\ \text{WHERE} & TempIS1.B \neq TempIS2.B \end{array} \qquad (13)$$

Only the query result, denoted by QR-(12), of query Q-(12) is affected by the **maintenance-concurrent** data updates. The QR-(12) should be the extent $\{< 1 >, < 3 > \}$ instead of the extent $\{< 1 >, < 2 >\}$. 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 counters, e.g., the tuple $< 2, 8 > [+1]$ and the tuple $< 3, 12 > [-1]$. Then, we abstract the query template Q-(14) from the query Q-(12):

$$\begin{array}{ll} \text{SELECT} & \$R.B \\ \text{FROM} & \$R \end{array} \qquad (14)$$

From the template query Q-(14), we get the compensation query Q-(15):

$$\begin{array}{ll} \text{SELECT} & R\_LC.B \\ \text{FROM} & R\_LC \end{array} \qquad (15)$$

The query result QR-(15) of the compensating query Q-(14) consists of the tuples $\{< 2 > [+1], < 3 > [-1]\}$. Then, we compensate the QR-(12) by getting rid of the effect in QR-(15), which will drop the tuple $\{< 2 >\}$ and add the tuple $\{< 3 >\}$. Now, the QR-(12) is $\{< 1 >, < 3 >\}$.

Using the Q-(13) to assemble the results QR-(11) and QR-(12), we generate the query result QR-(3) is $\{< 4,1 >, < 4,3 >, < 2,3 >\}$ (shown as $\Delta DW$ *(Correct)* in Figure 1). This then results in a correct DW state after the view adaptation.

# 7 Correctness of the LC Algorithm

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 (SC or DU) Y(m)[i] as defined by Definition 4, then there exists some mentenance query Q(m)[j] that has been sent to IS[j] for handling the maintenance of the Y(m)[i] update.*

This theorem can be easily explained based on the **maintenance-concurrent** properties given in Definition 4. From the second condition of Definition 4, we know that the **maintenance-concurrent** data update $X(n)[j]$ must be received by the data warehouse before the query result $QR(m)[j]$ from the same source $IS_j$. The query result $QR(m)[j]$ represents the answer to a query Q(m)[j] that must have been send to IS[j] for the update $Y(m)[i]$.

**Theorem 2** *Local Compensation as described in Section 6 solves the* **maintenance-concurrent** *problem defined in Definition 4.*

**Proof**: We want to prove that LC correctly compensates the view adaptation queries (detailed in Appendix A) if they are affected by **maintenance-concurrent** DUs, i.e., for the relevant $SC \vdash DU$ case.

Assume we have $m$ information sources $IS_1, ..., IS_m$, with one relation $R_1, ..., R_m$ respectively, going to be queried by a certain view adaptation query. Each information source $IS_j$ has its own **maintenance-concurrent** data updates, denoted as $DU_j$ for its relation $R_j$ $(j = 1..m)$. If there is no data update, then the $DU_j$ is null. We denote the relations of the information sources after have completed their respective data updates as $R'_j$, $j = 1..m$.

Each view adaptation query has a remote part that is sent to the other information sources to get additional information needed to be able to determine the effect on the data warehouse. The

$$Q(p)(q) = \Pi_{A_1,...,A_n}\sigma_{C_1,...,C_k}(R_1 \bowtie ... \bowtie R_m), \tag{16}$$

where $p$ is the sequence number of SC that triggered this view adaptation query, $q$ is the subscript of this view adaption query, allowing us to distinguish between multiple view adaptation queries caused by the same SC(p). $n$ is number of attributes, $k$ is number of local conditions, and $m$ is number of information sources.

remote part of each view adaptation query is an SPJ query. So, we can represent the remote part in the SPJ format in the query Q-(16).

We want to show that $QR(p)(q)$ **is** $\Pi_{A_1,...,A_n}\sigma_{C_1,...,C_k}(R_1 \bowtie ... \bowtie R_m)$ **after maintenance-concurrent** $DU_j, j = 1..m$ **have been compensated for by LC.**

During the view adaptation, query Q-(16) will be sent to the distributed IS space that contains relations $R_1, ..., R_m$. This distributed query will be first decomposed into $m$ sub-queries, one for each information source, as given in query Q-(17)

$$Q(p)(q)[i] = \Pi_{A_{n_{i,1}},...,A_{n_{i,a_i}}}\sigma_{C_{k_{i,1}},...,C_{k_{i,c_i}}}R_i, \tag{17}$$

for $i = 1..m$, with $\sum_{i=1}^{m} a_i \geq n$, and $\sum_{i=1}^{m} c_i \geq k$.

For any $i = 1..m$, query Q-(17) has only local attributes and conditions relevant to $IS_i$. Query Q-(18) contains the assembly of the query results from those $IS_i$s. This global assembly query is shown in query Q-(18), where the superscript $asm$ of query Q-(18) indicates that the purpose of this query is for "assembly".

$$Q(p)(q)^{asm} = \Pi_{A_1,...,A_n}\sigma_{C_1,...,C_k}(QR(p)(q)[1] \bowtie ... \bowtie QR(p)(q)[m]). \tag{18}$$

Before the $Q(p)(q)[i]$ is executed by the $IS_i$, the $DU_i$ has already been executed at $IS_i$. Thus at this point the relation $R_i$ has already become $R_i'$ with $R_i'$ containing the effectgs of $DU_i$. Hence the affected query result $QR(p)(q)[i]^{affect}$ is:

$$QR(p)(q)[i]^{affected} = \Pi_{A_{n_{i,1}},...,A_{n_{i,a_i}}}\sigma_{C_{k_{i,1}},...,C_{k_{i,c_i}}}(R_i + DU_i). \tag{19}$$

We know $DU_i$ is **maintenance-concurrent** update. Then the **maintenance-concurrent** $DU_i$ is detected by the data warehouse's query engine, and a local compensation query is issued for $DU_i$ of the form:

$$Q(p)(q)[i]^{LC} = \Pi_{A_{n_{i,1}},...,A_{n_{i,a_i}}} \sigma_{C_{k_{i,1}},...,C_{k_{i,c_i}}}(DU_i) \tag{20}$$

That is, this LC query reapplies the projection and predicates of query Q-(17) directly on the $DU_i$ instead of on the whole relation, as done above.

The details of the generation of query Q-(20) are described in step 2 of the LC procedure in Figure 6. The superscript $LC$ indicates that the query Q-(20) is a "local compensation" query. The query result of the local compensation query Q-(20) is:

$$\Delta QR(p)(q)[i] = Q(p)(q)[i]^{LC} = \Pi_{A_{n_{i,1}},...,A_{n_{i,a_i}}} \sigma_{C_{k_{i,1}},...,C_{k_{i,c_i}}}(DU_i) \tag{21}$$

By compensation, which is to remove $\Delta QR(p)(q)[i]$ from the affected query result $QR(p)(q)[i]^{affected}$, we finally get:

$$\begin{aligned} QR(p)(q)[i] = \; & QR(p)(q)[i]^{affected} - \Delta QR(p)(q)[i] \\ = \; & \Pi_{A_{n_{i,1}},...,A_{n_{i,a_i}}} \sigma_{C_{k_{i,1}},...,C_{k_{i,c_i}}}(R_i + DU_i) \\ & -\Pi_{A_{n_{i,1}},...,A_{n_{i,a_i}}} \sigma_{C_{k_{i,1}},...,C_{k_{i,c_i}}}(DU_i) \\ = \; & \Pi_{A_{n_{i,1}},...,A_{n_{i,a_i}}} \sigma_{C_{k_{i,1}},...,C_{k_{i,c_i}}}(R_i) \end{aligned} \tag{22}$$

Then, we put the $QR(p)(q)[i]$ back into one integrated query result via the $Q(p)(q)^{asm}$ (Q-(18)). Thus we get:

$$\begin{aligned} QR(p)(q) = \; & Q(p)(q)^{asm} \\ = \; & \Pi_{A_1,...,A_n} \sigma_{C_1,...,C_k}(QR(p)(q)[1] \bowtie ... \bowtie QR(p)(q)[m]) \\ = \; & \Pi_{A_1,...,A_n} \sigma_{C_1,...,C_k}( \\ & \Pi_{A_{n_{1,1}},...,A_{n_{1,a_1}}} \sigma_{C_{k_{1,1}},...,C_{k_{1,c_1}}}(R_1) \\ & \bowtie \Pi_{A_{n_{2,1}},...,A_{n_{2,a_2}}} \sigma_{C_{k_{2,1}},...,C_{k_{2,c_2}}}(R_2) \\ & ... \\ & \bowtie \Pi_{A_{n_{m,1}},...,A_{n_{m,a_m}}} \sigma_{C_{k_{m,1}},...,C_{k_{m,c_m}}}(R_m)) \\ = \; & \Pi_{A_1,...,A_n} \sigma_{C_1,...,C_k}(R_1 \bowtie ... \bowtie R_m) \end{aligned} \tag{23}$$

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 three conventional algorithms. We note however that the three algorithms without the SDCC coordination strategy could not at all address the concurrency problem that SDCC tackles.

View synchronization algorithms as employed by the EVE system [LNR01] 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, rather only CPU costs. Hence, we now no longer consider the VS algorithm below, since current cost models typically do ignore such CPU costs.

The VM component for which we employ the SWEEP algorithm 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 transferred between DW and IS is the same as that for SWEEP. The same also holds for the size of data shipped across the network.

Because LC performs only local compensation, no network action is required. Only two more messages are 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 assume the number of messages sent 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 transferred between DW and IS in SDCC is effectively the same as the data transferred 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 $msg(SDCC) = m \times msg(SWEEP) + n \times msg(VA)$, 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 of SWEEP and of VA respectively. Assuming a sufficiently large

enough local memory in the DW to store all the delta query results, as typically assumed by other approaches like ECA [ZGMHW95], then there are no additional IO costs for the SDCC at the DW site. In conclusion, the performance of SDCC is roughly comparable to the cost gotten by simply summing the cost of SWEEP and VA.

# 9 Concurrency Handling Level Criteria

In order to compare our system to others, we introduce a hierarchy of levels of *concurrency handling* in this section. First, we give the definitions of the terms and then we compare our system with other systems based on this hierarchy.

## 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 updates in the following definitions can be either DU or SC, unless otherwise specified. First, we give basic definitions.

**Definition 6** *We say that two updates are* **independent** *from one another if the updates are not* **maintenance-concurrent** *to one other by Definition 4. For more than two updates, if every pair in this set is* independent*, then we say the updates in this set are* **independent** *from one another.*

**Definition 7** *The* **state of the IS space at any given time** $t$*, denoted by* $\underline{ss_t}$*, is the snapshot of the extents of all ISs at this given time.*

*The* **state of the DW at any given time** $t$*, denoted by* $\underline{ws_t}$*, corresponds to the extent of the DW after the non-null effect of one update from the underlying IS space is propagated up and the DW has been updated accordingly.*

**Definition 8** *The state of the DW ( or the state of the IS space) is a* **terminal state** *if the next update is an update* independent *from all the previous updates with the term* independent *as defined in Definition 6.*

**Definition 9** *Given a sequence of updates U over the ISs, then we say a view management algorithm is* **correct***, if the* terminal state *of the DW generated by the algorithm for this set of updates is same as the view computed directly over the* terminal state *of the corresponding IS space.*

Based on these definitions, the four levels of concurrency handling can now be defined.

**Definition 10** *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 9.*

**Definition 11** *We say an algorithm is* **semi-concurrency handling**, *if the algorithm is* data-concurrency handling *and is also* correct *under* independent *schema changes.*

**Definition 12** *We say an algorithm is* **full concurrency handling**, *if the algorithm is* semi-concurrency handling *and is also* correct *under* **maintenance-concurrent** *schema changes and a mix of* **maintenance-concurrent** *data updates and schema changes.*

## 9.2 Comparison of Existing DW Maintenance Algorithms

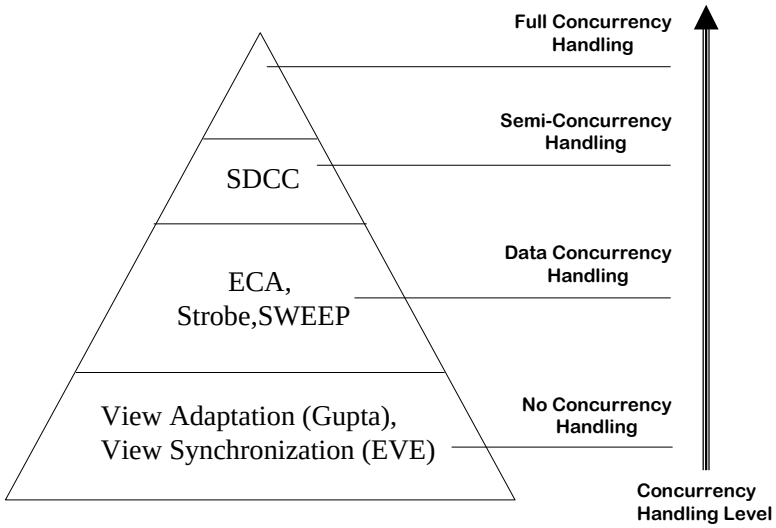


**Figure 7:** Algorithm Comparison Diagram

Figure 7 shows the levels of data warehouse maintenance algorithms from the literature. As we can see, the view adaptation algorithm provided by Gupta et al. [GMR97] and the view synchronization algorithm provided by the EVE project [NR98b, NLR98, KRH98] have not been designed to handle any kind of concurrency. Current view maintenance algorithms in the literature like ECA [ZGMHW95], SWEEP [AESY97], PSWEEP [ZRD01], and Strobe [ZGMW96] can handle data concurrency. The SDCC system we have described in this paper can handle semi-concurrency.

# 10 Conclusions

Data warehousing is a widely used technique for gathering and integrating data from heterogeneous and autonomous information sources. There are many 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 [AESY97]; view synchronization maintains the view definition when the schema of the ISs changed [LNR01]; and 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 happen 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 SDCC 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 significant overhead to the cost of integrating data warehouse maintenance algorithms. In short, it successfully solves the problem without affecting the overall system performance. Note that the SDCC strategies are being incorporated into the EVE data warehousing system that has been demonstrated at ACM SIGMOD'99 [RKZ+99].

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, techniques to optimize the overall system performance could be explored. We are also interested in studying alternate DW solutions that can release our semi-cooperative assumption [ZR99a].

**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 thank Prof. N. Hachem for providing advice to Xin Zhang as the reader of his Master's thesis. We also thank Amy Lee and Anisoara Nica from the University of Michigan for collaborations and interactions on the joint EVE project.

## References


[AESY97] D. Agrawal, A. El Abbadi, A. Singh, and T. Yurek. 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.

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

[DZR99] Lingli Ding, Xin Zhang, and E. A. Rundensteiner. The MRE Wrapper Approach: Enabling Incremental View Maintenance of Data Warehouses Defined On Multi-Relation Information Sources. In *Proceedings of the ACM First International Workshop on Data Warehousing and OLAP (DOLAP'99)*, pages 30–35, November 1999.

[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)*, pages 140–144, 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.

[KRH98] A. Koeller, E. A. Rundensteiner, and N. Hachem. Integrating the Rewriting and Ranking Phases of View Synchronization. In *Proceedings of the ACM First International Workshop on Data Warehousing and OLAP (DOLAP'98)*, pages 60–65, November 1998.

[LKNR99] A. J. Lee, A. Koeller, A. Nica, and E. A. Rundensteiner. Data Warehouse Evolution: Trade-offs between Quality and Cost of Query Rewritings. In *Proceedings of IEEE International Conference on Data Engineering*, Special Poster Session, page 255, March, Sydney, Australia 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.

[LNR01] A. J. Lee, A. Nica, and E. A. Rundensteiner. The *EVE* Approach: View Synchronization In Dynamic Distributed Environments. *IEEE Transaction on Knowledge and Data Engineering*, Accepted 2001. To Appear.

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

[Nic99] A. Nica. *View Evolution Support for Information Integration Systems over Dynamic Distributed Information Spaces.* PhD thesis, University of Michigan in Ann Arbor, in progress 1999.

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

[RKZ+99] E. A. Rundensteiner, A. Koeller, X. Zhang, A. Lee, A. Nica, A. VanWyk, and Y. Li. Evolvable View Environment. In *Proceedings of SIGMOD'99 Demo Session*, pages 553–555, May 1999.

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

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

[Zha99] Xin Zhang. Data Warehouse Maintenance Under Concurrent Schema and Data Updates. Master's thesis, Worcester Polytechnic Institute, May 1999.

[ZR99a] X. Zhang and E. A. Rundensteiner. DyDa: Dynamic Data Warehouse Maintenance in a Fully Concurrent Environment. Technical Report WPI-CS-TR-99-20, Worcester Polytechnic Institute, Dept. of Computer Science, July 1999.

[ZR99b] X. Zhang and E. A. Rundensteiner. Flexible Data Warehouse Maintenance Under Concurrent Schema and Data Updates. In *Proceedings of IEEE International Conference on Data Engineering*, Special Poster Session, page 253, March, Sydney, Australia 1999.

[ZR99c] X. Zhang and E. A. Rundensteiner. The SDCC Framework for Integrating Existing Algorithms for Diverse Data Warehouse Maintenance Tasks. In *International Database Engineering and Application Symposium*, pages 206–214, Montreal, Canada, August, 1999.

[ZRD01] Xin Zhang, Elke A. Rundensteiner, and Lingli Ding. PVM: Parallel View Maintenance Under Concurrent Data Updates of Distributed Sources. In *Data Warehousing and Knowledge Discovery, Proceedings*, Munich, Germany, September 2001. to be appeared.

[ZWGM97] Y. Zhuge, J. L. Wiener, and Héctor García-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 SPJ 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) that will only query within the data warehouse.
- Attribute Additive adaptation Query (AAQ) that will add a new attribute to the data warehouse.
- Multiple Attribute Additive adaptation Query (MAAQ) that will add more than one new attribute to the data warehouse.
- Tuple Additive adaptation Query (TAQ) that will add more tuples to the data warehouse due to dropping of one condition.
- Complex Tuple Additive adaptation Query (CTAQ) that will add more tuples to the data warehouse due to dropping more than one condition.

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

**Figure 8:** Relationship between View Definition Change and Adaptation Query

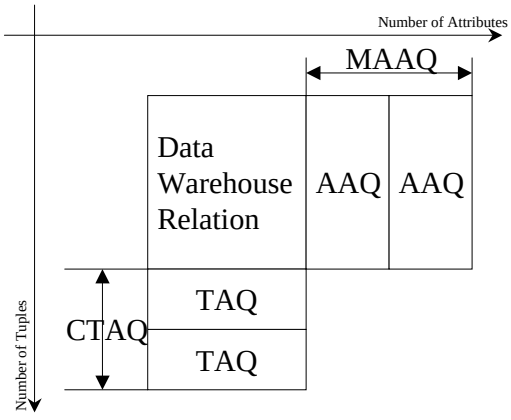


**Figure 9:** Relationship between View Definition Changes and Adaptation Queries

The adaptation queries AAQ, MAAQ, TAQ, and CTAQ handle the add-attr VDC, the add-rel VDC with attributes, the delete-cond VDC, and the delete-rel VDC with conditions, respectively. The rest VDCs, including the delete-attr VDC, the add-cond VDC, the add-rel VDC with conditions, and the delete-rel VDC with attributes are handled by local adaptation query. Figure 8 shows the relationship between VDCs and the corresponding adaptation techniques.

Figure 9 shows a more general representation of the relationships between VDCs and adaptation techniques. In Figure 9, 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 4 and 5. We repeat the original view definition given in Section 3 as Q-(24).

$$\begin{array}{ll} \text{CREATE VIEW} & V \text{ AS} \\ \text{SELECT} & A_1,\ A_2,\ ...,\ A_n \\ \text{FROM} & R_1\ \&\ R_2\ \&\ ...\ \&\ R_m \\ \text{WHERE} & C_1\ AND\ C_2\ ...\ AND\ C_k \end{array} \tag{24}$$

| View Definition Change | Abbreviation | Redefined View V' | Adapt Query Type |
|---|---|---|---|
| Add Attribute A from Relation $R_i, 1 \leq i \leq m$ | add-attr $R_i.A$ | $SELECT\ A,\ A_1,\ A_2,\ ...,\ A_n$ $FROM\ R_1\ \&\ ...\ \&\ R_m$ $WHERE\ C_1\ AND\ ...\ AND\ C_k$ | AAQ |
| Delete Condition $C_1$ | del-cond $C_1$ | $SELECT\ A_1,\ A_2,\ ...,\ A_n$ $FROM\ R_1\ \&\ ...\ \&\ R_m$ $WHERE\ C_2\ AND\ ...\ AND\ C_k$ | TAQ |
| Add Relation(SELECT) $R(B_1,\ ...,\ B_j)$ | add-rel(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) $R_1\{C_1,\ ...,\ C_j\}$ | del-rel(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 $A_1$ | del-attr $A_1$ | $SELECT\ A_2,\ ...,\ A_n$ $FROM\ R_1\ \&\ ...\ \&\ R_m$ $WHERE\ C_1\ AND\ ...\ AND\ C_k$ | $LQ_1$ |
| Add Condition $C$ | add-cond $C$ | $SELECT\ A_1,\ A_2,\ ...,\ A_n$ $FROM\ R_1\ \&\ ...\ \&\ R_m$ $WHERE\ C\ AND\ C_1\ AND\ ...\ AND\ C_k$ | $LQ_2$ |
| Delete Relation(SELECT) $R_1(A_1,\ ...,\ A_j)$ | del-rel(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) $R\{D_1,\ ...,\ D_j\}$ | add-rel(WHERE) $R\{D_1,\ ...,\ D_j\}$ | $SELECT\ A_1,...,\ A_n$ $FROM\ R\ \&\ R_1\ \&\ ...\ \&\ R_m$ $WHERE\ C_1\ AND\ ...\ AND\ C_k$ $AND\ D_1\ AND\ ...\ AND\ D_j$ | $LQ_4$ |

**Table 4:** View Definition Changes and View Adaptation Queries

Table 4 stores which adaptation query is needed for which specific VDC. Table 5 stores the type and template of each view adaptation query.

The assumptions required in Table 5 are given below:

- (1) Attribute $A$ is from relation $R_i$ and the key $K$ for $R_i$ is in view V.

| Adaption Query Type | Adaption Query Expened in SQL | Local Query | Assumption |
|---|---|---|---|
| AAQ | $ALTER\ TABLE\ V\ ADD\ A$<br>$UPDATE\ V\ SET\ A\ =\ (\ SELECT\ A$<br>$FROM\ R_i$<br>$WHERE\ R_i.K = V.K)$ | false | (1) |
| TAQ | $INSERT\ INTO\ V$<br>$SELECT\ A_1,\ ...\ ,\ A_n$<br>$FROM\ R_1\ \&\ ...\ \&\ R_m$<br>$WHERE\ NOT\ C_1\ AND\ C_2\ AND\ ...\ AND\ C_k$ | false | (2) |
| MAAQ | $ALTER\ TABLE\ V\ ADD\ B_1,\ ...,\ B_j$<br>$UPDATE\ V$<br>$SET\ B_1,\ ...,\ B_j\ =\ (\ SELECT\ B_1,\ ...,\ B_j$<br>$FROM\ R$<br>$WHERE\ R.A\ =\ V.B)$ | false | (3) |
| CTAQ | $INSERT\ INTO\ V$<br>$SELECT\ A_1,\ ...\ ,\ A_n$<br>$FROM\ R_1\ \&\ ...\ \&\ R_m$<br>$WHERE\ NOT\ C_1\ AND\ C_2\ AND\ ...\ AND\ C_k$<br>$OR\ NOT\ C_2\ AND\ C_3\ AND\ ...\ AND\ C_k$<br>...<br>$OR\ NOT\ C_j\ AND\ C_{j+1}\ AND\ ...\ AND\ C_k$ | false | (2) |
| $LQ_1$ | $ALTER\ TABLE\ V\ DROP\ A$ | true | N/A |
| $LQ_2$ | $DELETE\ FROM\ V$<br>$WHERE\ NOT\ C$ | true | (2) |
| $LQ_3$ | $ALTER\ TABLE\ V\ DROP\ A_1,\ ...,\ A_j$ | true | N/A |
| $LQ_4$ | $DELETE\ FROM\ V$<br>$WHERE\ NOT\ D_1\ OR\ ...\ OR\ NOT\ D_j$ | true | (2) |

**Table 5:** View Adaptation Queries

- (2) Attribute of conditions are either the attributes of the view, or of a wider augmented stored view.
- (3) $B_1$, ..., $B_j$ and $A$ are attributes of $R$, and the join condition is $A = B$; $B$ is an attribute of $V$; $A$ is a key for relation $R$.

## A.2 Detailed Algorithm of View Adaptation

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

1. receive a set of VDCs from the VS module.
2. apply appropriate adaptation depending on the VDCs (Tables 4 and 5).

   Figure 10 and 11 give out a more detailed description of the third step.
   Table 6 shows all the functions and their meaning.

```
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
```

**Figure 10:** Description of VA Procedure

```
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
```

**Figure 11:** Description of remoteAdapt Procedure

| **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 5). |
| 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). |
| reAssemble (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 6:** Functions and Their Meanings