Efficient View Maintenance at Data Warehouses*

D. Agrawal A. El Abbadi A. Singh T. Yurek
Department of Computer Science
University of California
Santa Barbara, CA 93106

Abstract

We present incremental view maintenance algorithms for a data warehouse derived from multiple distributed autonomous data sources. We begin with a detailed framework for analyzing view maintenance algorithms for multiple data sources with concurrent updates. Earlier approaches for view maintenance in the presence of concurrent updates typically require two types of messages: one to compute the view change due to the initial update and the other to compensate the view change due to interfering concurrent updates. The algorithms developed in this paper instead perform the compensation locally by using the information that is already available at the data warehouse. The first algorithm, termed SWEEP, ensures complete consistency of the view at the data warehouse in the presence of concurrent updates. Previous algorithms for incremental view maintenance either required a quiescent state at the data warehouse or required an exponential number of messages in terms of the data sources. In contrast, this algorithm does not require that the data warehouse be in a quiescent state for incorporating the new views and also the message complexity is linear in the number of data sources. The second algorithm, termed Nested SWEEP, attempts to compute a composite view change for multiple updates that occur concurrently while maintaining strong consistency.

1 Introduction

Data warehousing is used for reducing the load of on-line transactional systems by extracting and storing the data needed for analytical purposes (e.g., decision support, data mining). A materialized view of the system is kept at a site called the data warehouse, and user queries are processed using this view. The view has to be maintained to reflect the updates done against the base relations stored at the various data sources. The efficient incremental maintenance of materialized views has become an important research issue since the update efficiency of the warehouse view is counterbalanced by the query overhead at the data sources. Several approaches have focused on the problems associated with incremental view maintenance. Such problems include dealing with the anomalies resulting from the order of processing events, the levels of consistency in reflecting the source states in the view and the current validity of the view.

Previous work [ZGMHW95, HZ96a, ZGMW96, HZ96b, BCP96, GJM96, QGMW96, GMS93, HGMW$^+$95, GM95, CGL$^+$96, BLT86, CW91, GL95, QW91, SI84] on this issue has formed a spectrum of solutions ranging from a fully virtual approach at one end where no data is materialized and all queries are answered by interrogating the base relations to a full replication at the other end where the whole data in the base relations is copied so that the updates can be handled locally at the data warehouse. The two extreme solutions are inefficient in terms of communication in the former case, and storage in the latter. A more efficient solution is to materialize the relevant subsets of the data at the base relations pertaining to the actual view and process user queries against this local data. The updates done against the base relations are propagated to the materialized view and the view is maintained incrementally. However, this approach may necessitate a solution in which the sources have to be contacted for additional information to ensure correctness. Example work includes the ECA algorithm [ZGMHW95] designed for a system with a central database site, the Strobe algorithms [ZGMW96] handling multiple, distributed sites, a hybrid system [HZ96a] combining different regions of the spectrum. Self-maintainable views which lie towards the fully materialized end of the spectrum [GJM96, QGMW96], or full replication [HZ96b] do not require additional querying of the sources. These different approaches come with varying overheads and costs as discussed later in the paper.

Our algorithms are designed for a system in which there are multiple distributed autonomous sources and data at a source may be updated as a result of local update transactions, which are independent with respect to the update transactions at other sources. Several consistency notions have been associated with the views at the data warehouse, viz., complete consistency, strong consistency, weak consistency, and convergence [ZGMHW95, ZGMW96]. We develop two efficient algorithms in this paper. The first algorithm ensures complete consistency (the most stringent of the requirements) of the view by ordering the updates as they are delivered at the data warehouse and guarantees that the state of the view at the data warehouse preserves the delivery order of updates. Unlike some previous algorithms for incremental view maintenance, this algorithm does not

*This research is partially supported by NSF under grant numbers IRI94-11330, CDA94-21978, and CCR95-05807, by CALTRANS under grant number 65V250A and by LANL under grant number 686V0016-3A.

require that the data warehouse be quiescent for incorporating the new view. Also, the complexity of the number of messages involved in processing an update is linear with respect to the number of individual data sources. The second algorithm ensures strong consistency and in the worst case has linear message complexity (message complexity may be amortized over several updates if they are concurrent).

The rest of the paper is organized as follows. Section 2 presents the data warehouse model. In Sections 3 and 4, a framework is developed for analyzing and characterizing view maintenance algorithms and different approaches for compensating the effects of concurrent updates are described. The two algorithms SWEEP and Nested SWEEP are presented in Sections 5 and 6. The paper concludes with a summary of the results in Section 7.

2 The Data Warehouse Model

We adopt the data warehouse model developed in [ZGMW96, HZ96a, HZ96b]. In this model, updates occurring at the data sources are classified into three categories:

1. Single update transactions where each update is executed at a single data source.
2. Source local transactions where a sequence of updates are performed as a single transaction. However, all of the updates are directed to a single data source.
3. Global transactions where the updates involve multiple data sources.

For the purpose of this paper, we assume that the updates being handled at the data warehouse are of types 1 and 2. The approaches described in [ZGMW96] can be used to extend the algorithms presented in this paper for type 3 updates.

The updates at the data sources can be handled at the data warehouse in different ways. Depending on how the updates are incorporated into the view at the data warehouse, different notions of consistency of the view have been identified in [ZGMW96, HZ96a]. These are defined as follows:

- *Convergence* where the updates are eventually incorporated into the materialized view.
- *Strong consistency* where the order of state transformations of the view at the data warehouse corresponds to the order of the state transformations at the data sources.
- *Complete consistency* where every state of the data sources is reflected as a distinct state at the data warehouse, and the ordering constraints among the state transformations at the data sources are preserved at the data warehouse.

In this paper, we first describe an algorithm that ensures complete consistency of the views. Later we relax this requirement to achieve strong consistency. Commercially available data warehouse products such as Red Brick systems [RBS96] only ensure convergence.

The architecture of the data warehouse is as shown in Figure 1 [HGMW$^+$95]. The underlying system used for the data warehouse consists of n sites for data sources and another site for storing and maintaining the materialized view of the data warehouse. The communication between each data source and the data warehouse site is assumed to be reliable and FIFO, i.e., messages are not lost and are delivered in the order in which they are sent. No assumption is made about the communication between the different sites maintaining the data sources. In fact, they may be completely independent and may not be able to communicate with each other. The underlying database model for each data source is assumed to be a relational data model. Each data source may store any number of base relations, but conceptually we assume a single base relation R_i at data source i. Each data source is assumed to be completely autonomous in that the updates on different data source are not related, and therefore not synchronized. However, we assume that for a given data source, updates are executed atomically. As shown in Figure 1, the updates at each source are monitored. As updates occur, they are transmitted asynchronously from the source to the data warehouse. All the updates performed atomically at a data source are sent as a single unit from the source to the data warehouse [ZGMW96].

We assume that the view function used at the data warehouse for the materialized view is defined by the SPJ-expression (selection-projection-join). That is, if $\{R_1, \ldots, R_i, \ldots, R_n\}$ are the n base relations at corresponding data sources, the view denoted V, is as follows:

$$\prod_{\mathrm{ProjAttr}} \sigma_{\mathrm{SelectCond}} (R_1 \bowtie \ldots \bowtie R_i \bowtie \ldots \bowtie R_n)$$

It is possible to model the data warehouse using more complex view functions such as aggregates, but for simplicity we restrict the model in this paper to SPJ expressions. The updates to the base relations are assumed to be inserts and deletes of tuples. Furthermore, a modify is modeled as a delete followed by an insert. Although some earlier data warehouse models assume that the view definition includes the key attribute of each base relations, no such assumption is made here. However, it is assumed that the multiplicity of a tuple is maintained in terms of a control field that maintains the occurrence of each tuple [GMS93]. In particular, each tuple in the materialized view has a *count* value which indicates in how many different ways the same tuple can be derived from the given view definition and the source relations.

3 Maintaining Consistency of Incremental View Computations

The main problem that arises in the context of a data warehouse is to maintain the materialized view at the data warehouse in the presence of updates to the data sources. A simple approach of recomputing the view as a result of each update is unrealistic. A more appropriate solution would be to update the data warehouse incrementally in response to the updates arriving from the data sources. If any two updates are sufficiently far apart in terms of time such that the incremental recomputation of the view is not interfered by these updates, the view maintenance can be carried out in a straightforward manner. In this case, when an update ΔR_i, i.e., an update to the base relation R_i, is received at the data warehouse, the incremental changes are computed by querying the other data sources by sending appropriate parts of the following query [HJ91, HZ96a, GM95, GHJ96]:

$$\prod_{\mathrm{ProjAttr}} \sigma_{\mathrm{SelectCond}} (R_1 \bowtie \cdots \bowtie \Delta R_i \bowtie \cdots \bowtie R_n).$$

We illustrate the concept of incremental view maintenance with the following example. Assume two relations R_1 and

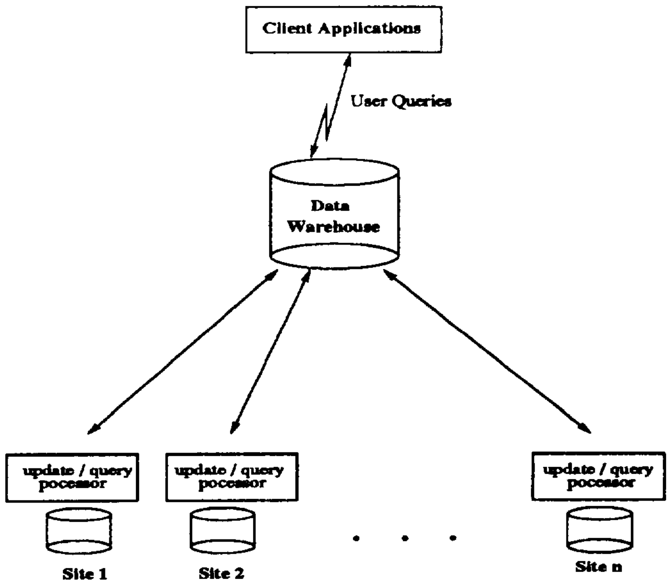

Figure 1: An Architecture of a Data Warehouse

R_2 and a view defined as a simple join over the two relations $V = R_1 \bowtie R_2$. As a result of an update ΔR_1 the new view should be as follows:

$$(R_1 + \Delta R_1) \bowtie R_2 \equiv R_1 \bowtie R_2 + \Delta R_1 \bowtie R_2$$

Since the data warehouse already has $R_1 \bowtie R_2$, it only needs to compute a query $Q_1 = \Delta R_1 \bowtie R_2$ at the data source with R_2 and incorporate the results into the materialized view. As discussed in the model, the updates are in the form of inserts and deletes of a tuple. Using the above notation, ΔR for an insert will carry a positive sign and hence will have the effect of adding tuples to the materialized view. Deletes, on the other hand, will result in ΔR having a negative sign and therefore will result in removing the appropriate tuples from the materialized view. In this way the views can be maintained incrementally as long as the updates are separated far enough for completing the incremental computation of the view. Unfortunately, such a separation between updates is not always guaranteed and can not be enforced due to the autonomous nature of the data sources with respect to the data warehouse. Hence, the data warehouse problem is to maintain the views incrementally in the presence of concurrent updates occurring at the data sources. The ECA [ZGMHW95] and Strobe [ZGMW96] algorithms are examples of two approaches for addressing this problem.

In the case of ECA, the data warehouse model is restricted in that the number of data sources is limited to a single data source. However, the data source may store several base relations. In order to understand the ECA algorithm, let us extend the above example to involve a join of three relations, $R_1 \bowtie R_2 \bowtie R_3$. When an update ΔR_1 arrives at the data warehouse it composes the incremental query $Q_1 = \Delta R_1 \bowtie R_2 \bowtie R_3$ and sends it to the data source. While Q_1 is in transit from the data warehouse to the data source, further updates, e.g. ΔR_2, may occur at the data source and may be delivered to the data warehouse. As a result of the two updates, the view should change to the following:

$$(R_1+\Delta R_1) \bowtie (R_2+\Delta R_2) \bowtie R_3 \equiv \left(\begin{array}{c} (R_1 \bowtie R_2 \bowtie R_3) \\ + \\ (\Delta R_1 \bowtie R_2 \bowtie R_3) \\ + \\ (R_1 \bowtie \Delta R_2 \bowtie R_3) \\ + \\ (\Delta R_1 \bowtie \Delta R_2 \bowtie R_3) \end{array} \right)$$

The answer to the incremental query Q_1 will include the effects of ΔR_1 and ΔR_2 and hence will be $A_1 = (\Delta R_1 \bowtie R_2 \bowtie R_3) + (\Delta R_1 \bowtie \Delta R_2 \bowtie R_3)$. We refer to $\Delta R_1 \bowtie \Delta R_2 \bowtie R_3$ as the *error term*[1] in the incremental answer due to concurrent update ΔR_2. Incorporating A_1 into the materialized view will not reflect all the changes that should have occurred after the two updates, i.e., $R_1 \bowtie \Delta R_2 \bowtie R_3$, is missing. A blind formulation of the incremental query Q_2 as $R_1 \bowtie \Delta R_2 \bowtie R_3$ will result in an incorrect answer since Q_1 has partially incorporated the effects of update ΔR_2. The ECA protocol is based on this idea and uses the notion of "compensation" to formulate a query Q_2 to offset the error term introduced in A_1. In particular, Q_2 in this example will be formulated as:

$$(R_1 \bowtie \Delta R_2 \bowtie R_3) - (\Delta R_1 \bowtie \Delta R_2 \bowtie R_3)$$

It is interesting to note that with the above formulation, it is easy to identify an optimization for ECA [ZGMHW95]. In particular, after an initial update ΔR_1, if there is a sequence of concurrent updates $\Delta_1 R_2, \Delta_2 R_2, \cdots, \Delta_k R_2$ that occurred while $Q_1 = \Delta R_1 \bowtie R_2 \bowtie R_3$ was in transit, only one incremental query Q_2 formulated for all updates $\Delta_i R_2$ is needed. There is no need to send individual incremental queries corresponding to each update $\Delta_* R_2$. In ECA the size of query messages is quadratic in the number of interfering updates.

Instead of generalizing ECA for a distributed data warehouse model, Zhuge et al. [ZGMW96] proposed a new algorithm called *Strobe* for maintaining views incrementally

[1]Hull and Zhou [HZ96a] refer to this as the problem of "missing" contribution.

in an environment with multiple data sources. In the basic version of Strobe, the error term due to concurrent updates is handled by making the following assumptions about the base relations and the materialized view. The materialized view in Strobe is assumed to be an SPJ-expression where the projection list contains the key attributes from each relation. As a result of this assumption the handling of the error term is localized at the data warehouse. The concurrent updates are handled by recognizing the type of updates being a *delete* or an *insert* of a tuple. If the update corresponds to a delete of a tuple, the update is handled locally by inserting a delete marker for all ongoing incremental queries as well as in an accumulated answer list for subsequent incorporation in the materialized view. On the other hand, if an update is an insert, a query is initiated based on the definition of the view and evaluated at all of the sources. The error term resulting from concurrent inserts may result in duplicates. Strobe handles this by suppressing all duplicates before merging the results of the query into the materialized view. This is always possible since the view definition assumes that the view always includes key attributes of all base relations.

We illustrate Strobe using the above framework. As before, the view is defined as a join of three relations $R_1 \bowtie R_2 \bowtie R_3$ and consider the case when both ΔR_1 and ΔR_2 are inserts of a tuple. Assume the same scenario in which ΔR_2 overlapped with the computation of $Q_1 = \Delta R_1 \bowtie R_2 \bowtie R_3$. Unlike in ECA, the query Q_2 does not compensate for offsetting the error term in the answer of Q_1 and instead is formulated as follows: $R_1 \bowtie \Delta R_2 \bowtie R_3$. The answers to the queries are:

$$A_1 = \Delta R_1 \bowtie R_2 \bowtie R_3 + \Delta R_1 \bowtie \Delta R_2 \bowtie R_3$$

and

$$A_2 = R_1 \bowtie \Delta R_2 \bowtie R_3 + \Delta R_1 \bowtie \Delta R_2 \bowtie R_3$$

This results in the term $\Delta R_1 \bowtie \Delta R_2 \bowtie R_3$ being included twice in the materialized view. Since the tuples are assumed to have key attributes from each base relation, by suppressing duplicates this error term can be easily eliminated in Strobe. The problem with the Strobe algorithm is that the materialized view cannot be updated until there is quiescence in the system, i.e., the algorithm waits until all updates subside and only then the answers resulting from all updates are merged in the view. Until then the materialized view trails the updated state of the data sources. In fact, the materialized view will never get updated if there is no period of quiescence in the system. Strobe ensures strong consistency but not complete consistency since it incorporates the effects of several updates collectively.

Zhuge et al. [ZGMW96] propose another algorithm called *C-strobe,* to circumvent the necessity of quiescence in the Strobe algorithm. In C-strobe, each update is handled completely at the data warehouse before handling subsequent updates. In this sense, C-strobe provides complete consistency. That is the state of the materialized view reflects every state transition of the data sources. Basically, in C-strobe an answer to a given update is evaluated by contacting all the sources. Due to concurrent updates, this answer may contain errors. Further queries are generated to compensate for this error and further errors may arise due to concurrent updates during the compensation queries. We illustrate this problem by using our framework.

Consider a view definition involving multiple relations and an update ΔR_1. In response, the data warehouse dispatches a query $Q_1 = \Delta R_1 \bowtie R_2 \bowtie R_3 \bowtie \cdots$ to the data sources corresponding to the base relations $R_2, R_3, \cdots$. If no concurrent updates arise then the resulting answer of Q_1 can be merged into the materialized view which will reflect the correct state after ΔR_1. However, if a concurrent update[2] ΔR_2 occurs during the computation of the answer to Q_1 then the answer will have an error term $\Delta R_1 \bowtie \Delta R_2 \bowtie R_3 \bowtie \cdots$. The data warehouse now has to send a query, which is $\Delta R_1 \bowtie \Delta R_2 \bowtie R_3$, to $R_3, \cdots$ to subtract out the effects of the error term. During this evaluation, if a concurrent update ΔR_3 occurs then we will again get a second order error term $\Delta R_1 \bowtie \Delta R_2 \bowtie \Delta R_3 \bowtie \cdots$. As a result, a further compensation query needs to be sent to the remaining base relations. In the above example, we illustrated the problem of nested compensation by using a single concurrent update. If there are at most K concurrent updates that can arrive between the time a query is sent and its answer is received, then the authors show that at most K^{n-2} queries need to be sent for a single update [ZGMW96].

Zhuge et al. [ZGMW96] optimize the C-strobe algorithm by exploiting the unique key attribute in SPJ expression and by classifying updates into ***inserts*** and ***deletes***. When the initial update is a delete of a tuple it is immediately incorporated locally at the data warehouse. This is possible due to the unique key assumption. On the other hand, if the initial update is an insert then a query is sent to the data sources. The effects of all concurrent updates, during the time the query is being evaluated, need to be offset. For all concurrent deletes new queries are sent to insert the tuples that may be missing in the initial answer. Concurrent inserts are handled locally by deleting the corresponding tuples from the initial answer. All subsequent queries initiated due to concurrent deletes are handled in the same manner. By grouping queries that are sent to the same base relation, the complexity of C-strobe can be made $(n-1)!$ instead of K^{n-2}. However, this is still a very high value, and renders the algorithm unscalable for a large number of data sources.

4 On-line Error Correction Of Incremental View Computations

All of the above algorithms, ECA [ZGMHW95], Strobe, and C-Strobe [ZGMW96], completely evaluate the answer to a query before doing any compensation. As a consequence, all updates that are received at the data warehouse between the time when the query is initiated up to the time it is fully evaluated are considered concurrent updates. In the case of ECA with a single data source, these updates are indeed concurrent updates that interfered with the evaluation of the query. However, in a distributed setting, an update from a data source will only interfere with a query if the update occurs between the sending of the query to and the receiving of the answer from that data source. If, on the other hand, the update occurs after the query has been evaluated at that source, then this update does not interfere with the answer. Hence, the answer should not be compensated for such updates. Strobe and C-strobe assume that even such updates must be compensated. This, however, does not result in any inconsistency due to the unique key assumption. If the unique key assumption is dropped then compensation for non-interfering updates will result in compensation for an error term that was non-existent resulting in an inconsistency.

[2]We assume that an update ΔR_2 is concurrent if it is received at the data warehouse after sending of the query to the data source R_2 and before receiving the answer from R_2. If ΔR_2 arrives after the query has been evaluated at R_2, it is deemed not concurrent.

Waiting to do the compensation until the query has been completely evaluated at all data sources also results in the loss of the opportunity of subtracting the error term locally at the data warehouse itself. That is, the compensation has to be done by sending compensating queries to eliminate the effects of concurrent updates. In a distributed setting, the answer to a query cannot be evaluated atomically. As shown in Figure 2, a query $R_1 \bowtie \cdots R_{i-1} \bowtie \Delta R_i \bowtie R_{i+1} \bowtie \cdots \bowtie R_n$ needs to be computed iteratively as follows: first in the left direction from R_i and then proceed rightward from R_i. The left side of the query proceeds iteratively by performing the computation at R_{i-1}, getting an answer A_{i-1}, followed by performing the computation at R_{i-2} and so on as shown in the figure. Similarly for the right direction.

While this query is being evaluated updates may occur at any of these sources, and as a result an error term may be introduced into the answers from the individual sites. For example, when the query initiated as a result of ΔR_i is in progress, an update ΔR_{i-1} occurs before $R_{i-1} \bowtie \Delta R_i$ is evaluated at R_{i-1}. As a result of the FIFO property of communication channels, the data warehouse must receive ΔR_{i-1} before it receives the answer to the query $R_{i-1} \bowtie \Delta R_i$. From this the data warehouse can conclude that the answer includes the error term $\Delta R_{i-1} \bowtie \Delta R_i$, which can be evaluated locally and its effects can be eliminated from the answer set resulting in the desired answer $R_{i-1} \bowtie \Delta R_i$. We refer to this as an *on-line error correction* since it eliminates the effects of concurrent updates as soon as they are detected at the data warehouse. In contrast, in the Strobe/C-strobe algorithm, this error accumulates until the entire query is completely evaluated by querying all data sources. Under this approach, the local information at the data warehouse is not sufficient to eliminate this accumulated error and hence the data sources need to be queried to compensate for this error.

In the general case, the warehouse may receive a concurrent update ΔR_j, $j < i$, while it is evaluating the incremental query resulting from ΔR_i. As before, when the answer arrives from the data source R_j, it is $(R_j + \Delta R_j) \bowtie R_{j+1} \bowtie \cdots \bowtie R_{i-1} \bowtie \Delta R_i$ instead of $R_j \bowtie R_{j+1} \bowtie \cdots \bowtie R_{i-1} \bowtie \Delta R_i$. The error term included in the answer is $\Delta R_j \bowtie R_{j+1} \bowtie \cdots \bowtie R_{i-1} \bowtie \Delta R_i$, which can be evaluated locally at the data warehouse since both components, ΔR_j and $R_{j+1} \bowtie \cdots \bowtie R_{i-1} \bowtie \Delta R_i$ are available at the data warehouse. Note that the latter term is the partially evaluated answer from R_{j+1} for the query initiated on account of ΔR_i. The case of a concurrent update ΔR_j, $j > i$, is symmetric.

5 Independent Updates

In this section, we develop an algorithm based on on-line error correction to update the materialized view at the data warehouse incrementally for every update. The updates occurring at different data sources are totally ordered based on the order in which the updates are delivered to the data warehouse. The materialized view, therefore, is updated in the order of these updates. Thus for an update u, the algorithm ensures that the effects of all the updates that arrived at the data warehouse before u will be reflected in the materialized view but none of the effects of the updates that arrived after u will be included. Hence, the algorithm ensures complete consistency. The algorithm presented here is motivated from the brief description of an incremental view update approach presented in [HZ96a].

5.1 The SWEEP algorithm

We begin by describing the update and query server component employed at each data source to facilitate incremental view maintenance at the data warehouse. Figure 3 shows the code for the update and query server at a data source. The server essentially provides services to handle updates occurring at the base relation at the data source and to answer incremental queries initiated by the data warehouse.

As shown in Figure 3, the server module basically consists of two process threads corresponding to the two types of input events that may occur at the data sources:

1. An update ΔR at the relation R.

2. A request to compute new incremental view, ΔV

We assume that the server processes requests sequentially, i.e., a request is completely serviced before servicing the next request. The server processes the internal event ΔR by forwarding the update to the data warehouse. A query request from the data warehouse is computed by performing the join of the local base relation R with the partially computed answer ΔV included in the query. The result of the join is then sent back to the data warehouse. Note that the join at the data source must be synchronized with the local update transactions occurring at the data source.

Figure 4 depicts the software module that is employed at the data warehouse for view maintenance. The main function in this module is $ViewChange$ which is invoked for every update $(\Delta R, i)$ received at the data warehouse. As mentioned before, $ViewChange$ ensures that the change in view computed will reflect all the updates at the data sources that are delivered at the data warehouse up to $(\Delta R, i)$ but none of the updates that are delivered after it. The view change computation proceeds as follows. Initially, the change in the view V, denoted ΔV, is initialized to ΔR. Next, the view computation is carried out by querying data sources to the left of i one at a time. Then ΔV is computed incrementally by contacting sources to the right of i one at a time. The two for loops correspond to this iterative computation or sweep (the reason for the name SWEEP). In each iteration, ΔV is expanded by querying data source j and the answer ΔV is compensated at the data warehouse for any errors that arose due to a concurrent update $(\Delta R, j)$. The detection of a concurrent update is done by checking if the $UpdateMessageQueue$ has ΔR_j that arrived before receiving the answer ΔV from the data source j. If there are multiple interfering updates found in the $UpdateMessageQueue$ coming from R_j, they can be merged into a single ΔR_j at this point. After querying all data sources except i, the change in the view $\Delta V = R_1 \bowtie \cdots \bowtie \Delta R_i \bowtie \cdots R_n$ is returned.

The data warehouse module employs two processes to handle view changes. The process, $LogUpdates$, receives the updates from various data sources over the communication channel and appends them to a message queue for later processing. The other process, $ViewUpdate$, removes an update message $(\Delta R, i)$, from the message queue and invokes the function $ViewChange$ for ΔR_i. The $ViewUpdate$ process incorporates the change in the view, ΔV, returned by the function into the view, V, and proceeds to remove the next message in the message queue. If there are no new messages, the process blocks until the queue becomes non-empty. We assume that the view V is initialized to the correct value.

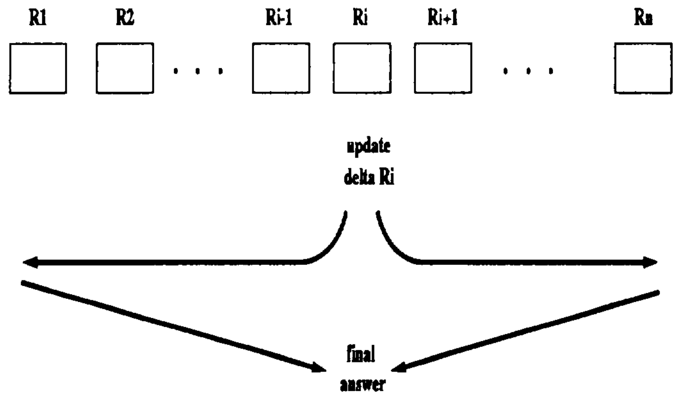

Figure 2: On-line Incremental View Computation

```
MODULE Update&QueryServer;

    CONSTANT
        MyIndex = i;

    PROCESS SendUpdates;
    BEGIN
        LOOP
            RECEIVE $\Delta R$ FROM $R$;
            SEND( $\Delta R$, MyIndex) TO DataWarehouse;
        FOREVER;
    END SendUpdates;

    PROCESS ProcessQuery;
    BEGIN
        LOOP
            RECEIVE $\Delta V$ FROM DataWarehouse;
            $\Delta V = ComputeJoin(\Delta V, R)$;
            SEND $\Delta V$ TO DataWarehouse;
        FOREVER;
    END ProcessQuery;

BEGIN /* Initialization */
    StartProcess(SendUpdates);
    StartProcess(ProcessQuery);
END Update&QueryServer.
```

Figure 3: The Update and Query Server at the Data Source i

5.2 Example

In this section, we illustrate the details of SWEEP by using the following examples. Consider a data warehouse that is defined over three data sources with base relations $R_1[A,B], R_2[C,D]$ and $R_3[E,F]$. The notation $R[X,Y]$ signifies relation R with two attributes X and Y. Let the materialized view be defined by the following SQL query:

$$\text{SELECT } R_2.D,\ R_3.F \text{ WHERE } R_1.B = R_2.C \text{ AND } R_2.D = R_3.E$$

which is equivalent to the following SPJ expression:

$$V(R_1, R_2, R_3) = \prod_{[D,F]} \left(R_1[A,B] \overset{B=C}{\bowtie} R_2[C,D] \overset{D=E}{\bowtie} R_3[E,F]\right).$$

The initial configuration of each relation is as shown in Figure 5. The initial state of the data warehouse is $\{(7,8)[2]\}$ where the integer value in the square brackets represents the number of ways in which tuple $(7,8)$ is materialized.

We now consider three updates $\Delta R_2, \Delta R_3$, and ΔR_1 that occurred at various data sources. Assuming that each update occurred sequentially, i.e., the new view was computed before the next update occurred. The resulting state transformation is illustrated in Figure 5. Any algorithm that ensures complete consistency must reflect all of the above state transformations even if the updates are concurrent (but maintain the relative order). We now illustrate this property of SWEEP by considering the case when the above three updates occur concurrently. Since ΔR_2 is the first to be delivered at the data warehouse the *ViewChange* function is initiated on its behalf and therefore $\Delta R_2 = \{+(3,5)\}$ is removed from the message queue (see process *UpdateView* in Figure 4). As a result of the left sweep, $\Delta V = \{+(3,5)\}$ is sent to the data source with relation R_1. Before the answer to this query is returned from the data source R_1, both updates ΔR_3 and ΔR_1 are delivered to the data warehouse. Thus, as per the code in Figure 4, the answer from R_1 which is $\{(1,3,5)\}$ must be compensated to eliminate the effects of the concurrent update ΔR_1. The compensation will be $\{-(2,3)\} \bowtie \{(3,5)\}$ which evaluates to be $\{-(2,3,5)\}$. The compensation $\{-(2,3,5)\}$ that was deleted as a result of ΔR_1 is added to the partial view change making ΔV to be $\{(1,3,5),(2,3,5)\}$. The partially computed view change ΔV is forwarded to the data source with R_3 during the

```
MODULE DataWarehouse;
    GLOBAL DATA
        $V$: RELATION; /* Initialized to the correct view */
        $UpdateMessageQueue$: QUEUE initially $\emptyset$;

    FUNCTION $ViewChange$( $\Delta R$: RELATION; $UpdateSource$: INTEGER): RELATION
    VAR
            $\Delta V$, $TempView$: RELATION;
            $j$: INTEGER;
    BEGIN
        $\Delta V = \Delta R$;
        /* Compute the left part of the incremental view resulting from $\Delta R$ */
        FOR ($j = UpdateSource - 1$; $j \geq 1$; $j --$) DO
            $TempView = \Delta V$;
            SEND $\Delta V$ TO Data Source $j$;
            RECEIVE $\Delta V$ FROM Data Source $j$;
            /* Remove the error due to concurrent update if any */
            IF $\exists \Delta R_j \in UpdateMessageQueue$ THEN $\Delta V = \Delta V - \Delta R_j \bowtie TempView$; ENDIF;
        ENDFOR;
        /* Compute the right part of the incremental view resulting from $\Delta R$ */
        FOR ($j = UpdateSource + 1; j \leq n; j ++$) DO
            $TempView = \Delta V$;
            SEND $\Delta V$ TO Data Source $j$;
            RECEIVE $\Delta V$ FROM Data Source $j$;
            /* Remove the error due to concurrent update if any */
            IF $\exists \Delta R_j \in UpdateMessageQueue$ THEN $\Delta V = \Delta V - \Delta R_j \bowtie TempView$; ENDIF;
        ENDFOR;
        RETURN($\Delta V$);
    END $ViewChange$;

    PROCESS $LogUpdates$;
    BEGIN
        LOOP
            RECEIVE $\Delta R$ FROM Data Source $i$;
            APPEND ($\Delta R, i$) TO $UpdateMessageQueue$;
        FOREVER;
    END $LogUpdates$;

    PROCESS $UpdateView$;
    BEGIN
        LOOP
            REMOVE ($\Delta R, i$) FROM $UpdateMessageQueue$; /* block if queue empty */
            $V = V + ViewChange(\Delta R, i)$
        FOREVER;
    END $UpdateView$;

BEGIN /* Start DataWarehouse Processes */
    $StartProcess(LogUpdates)$;
    $StartProcess(UpdateView)$;
END $DataWarehouse$.
```

Figure 4: The Incremental View Construction Algorithm

Event	Source 1 $R_1[A,B]$	Source 2 $R_2[C,D]$	Source 3 $R_3[E,F]$	Warehouse $V(R_1,R_2,R_3)$
Initial State	(1,3) (2,3)	(3,7)	(5,6) (7,8)	(7,8)[2]
$\Delta R_2 = +(3,5)$ ("+" means *insert*)	(1,3) (2,3)	(3,7) (3,5)	(5,6) (7,8)	(5,6)[2] (7,8)[2]
$\Delta R_3 = -(7,8)$ ("−" means *delete*)	(1,3) (2,3)	(3,7) (3,5)	(5,6)	(5,6)[2]
$\Delta R_1 = -(2,3)$	(1,3)	(3,7) (3,5)	(5,6)	(5,6)[1]

Figure 5: An example illustrating the effects of updates on the data sources and the materialized view (the value in the square brackets represents the tuple count)

right sweep of *ViewChange* on behalf of ΔR_2. Once again, when the answer is obtained from R_3 it includes the effects of ΔR_3 and is $\{(1,3,5,6),(2,3,5,6)\}$. This result is compensated for the error term $\{(1,3,5),(2,3,5)\} \bowtie \{-(7,8)\}$ which is $\emptyset$. After incorporating ΔV into the materialized view, we get $\{(5,6)[2],(7,8)[2]\}$ which is indeed the desired result as shown in Figure 5.

Assuming that the next update message in the message queue is ΔR_3, the view change process proceeds as follows. In the left sweep R_2 is queried and the answer returned is $\{-(3,7,8)\}$ which is a correct answer since ΔR_2 is not concurrent with ΔR_3. The left sweep continues and the partial view change is computed further by querying R_1 resulting in the answer $\{-(1,3,7,8)\}$, which reflects the effects of ΔR_1 that is indeed concurrent with respect to ΔR_3. As a result of local compensation which is $\{-(2,3)\} \bowtie \{-(3,7,8)\} \equiv \{-(2,3,7,8)\}$, the deleted tuple is added into ΔV making it $\{-(1,3,7,8),-(2,3,7,8)\}$. When this view change is incorporated into the materialized view we get $V = \{(5,6)[2]\}$ which is the desired state of the data warehouse after ΔR_2 and ΔR_3 as shown in Figure 5. It can be easily verified that after executing *ViewChange* on behalf of the last update ΔR_1, the final state will indeed be as shown in Figure 5.

5.3 Discussion

We conclude this section with a brief analysis of the independent update based view maintenance algorithm, SWEEP, as well as its comparison with other view maintenance algorithms. One of the main properties of SWEEP is that it ensures complete consistency. Hence, all queries executed at the data warehouse are guaranteed a consistent view of the distributed database. Another property of this algorithm is that in contrast to the earlier approaches for view maintenance, the error compensation is completely localized at the data warehouse. As a consequence, the cost of computing the view change per update in this algorithm is linear in the number of messages, i.e., only $(n-1)$ messages are needed where n is the number of data sources. This is significantly cheaper than C-strobe which supports the same notion of consistency as SWEEP but has a message complexity of $(n-1)!$ in the worst case. Although Strobe may be more efficient since it piggybacks the view change for multiple updates, it may not terminate if there is no quiescence at the data warehouse. In that case, the view at the data warehouse may trail significantly compared to the state of the data sources. ECA cannot be compared directly to SWEEP since the former has been designed for a single data source and cannot be used with multiple data sources. However, the size of the compensation queries in ECA increases quadratically in terms of the number of concurrent updates. Furthermore, none of these algorithms have been designed to perform compensation locally. Finally, both Strobe and C-strobe are very restrictive in that they both assume that the view function includes the key attributes of every base relation in the view.

We conclude this section by making several observation to optimize the performance of the SWEEP algorithm. In particular, the two for loops, i.e., the left and right sweeps, in the *ViewChange* function are independent and therefore can be executed in parallel. The only requirement will be that the two partial views obtained after the two sweeps complete, should be merged, i.e., $\Delta V = \Delta V_{left} \bowtie \Delta V_{right}$. Another optimization that is possible in the SWEEP algorithm is to pipeline the view construction for multiple updates. This will introduce some complexity in the data warehouse software module but will result in a rapid installation of view changes at the data warehouse. To maintain consistency, the view changes should be incorporated in the order of the arrival of the updates and a more elaborate mechanism will be needed to detect concurrent updates.

6 Cumulative Updates

In this section, we extend SWEEP so that the view maintenance for multiple updates can be carried out in a cumulative manner. Since a view change due to multiple updates is incorporated into the materialized view, the algorithm no longer ensures complete consistency. However, the algorithm does ensure strong consistency, thus ensuring correctness of user queries at the materialized view.

6.1 The Nested SWEEP Algorithm

The SWEEP algorithm described in the previous section incrementally incorporates the effects of each source update one at a time. Hence, when a set of concurrent updates occur, they are handled sequentially, incorporating the results of the new update into the data warehouse before proceeding to handle the next update. Thus, even though a concurrent update may have occurred, the algorithm, conservatively, eliminates the effects of its error term and keeps it in the update message queue to be handled later. We first note that such subsequent updates may be able to share components of incremental query results of the current update. This fact can be exploited by piggybacking queries for the new updates on top of the queries initiated for the current update. However, queries resulting from a concurrent update cannot simply be tagged on to an existing query since such a query already incorporates the partial evaluation of the updates resulting from the current update.

We illustrate this with an example involving base relations $R_1, \cdots R_j, \cdots, R_i, \cdots R_k, \cdots R_n$. As in the earlier algorithm, when ΔR_i is received at the data warehouse, a set of queries are formulated and sent first in the left directional sweep and then in the right directional sweep. Consider the situation when a concurrent update, ΔR_j $(j < i)$ occurs and is received at the data warehouse before receiving the answer to its query from R_j. In this case, when the data warehouse does receive the answer from R_j, it first eliminates the error term caused by ΔR_j, resulting in the partial answer $R_j \bowtie \cdots \bowtie \Delta R_i$. Once this is evaluated, and assuming no more updates, the previous algorithm proceeds past R_j to incorporate the effect of ΔR_i on all relations to the left of R_j resulting in the answer set $R_1 \bowtie \cdots \bowtie R_{j-1} \bowtie R_j \bowtie R_{j+1} \bowtie \cdots \bowtie \Delta R_i$ and finally all the relations to the right of R_i, i.e., $A_i = R_1 \bowtie \cdots R_{j-1} \bowtie R_j \bowtie R_{j+1} \bowtie \cdots \bowtie \Delta R_i \bowtie R_{i+1} \bowtie \cdots \bowtie R_n$.

When the data warehouse later decides to incorporate the effects of ΔR_j, a new sweep of the relations is made resulting in an answer set which can be expressed as follows: $A_j = R_1 \bowtie \cdots R_{j-1} \bowtie \Delta R_j \bowtie R_{j+1} \bowtie \cdots \bowtie R_i^{new} \bowtie R_{i+1} \bowtie \cdots \bowtie R_n$ where R_i^{new} is $R_i + \Delta R_i$. We note that both A_i and A_j share several terms in common, which are $R_1 \bowtie \cdots \bowtie R_{j-1}$ and $R_{i+1} \bowtie \cdots \bowtie R_n$. Furthermore, by the time the data warehouse is evaluating these terms on behalf of ΔR_i, it is already aware of the new concurrent update ΔR_j and has in fact used it to eliminate the error term for relation R_j. This fact can be exploited by dovetailing the remainder computation for ΔR_i with ΔR_j. However, before this can be done we must compute the missing view change components for ΔR_j. To correctly dovetail the evaluation of ΔR_j with

ΔR_i, we must first evaluate the terms where they differ, i.e., $\Delta R_j \bowtie R_{j+1} \bowtie \cdots \bowtie R_i^{new}$. Once this term is evaluated, the algorithm can proceed with the original evaluation of ΔR_i, however, incorporating the effects of both updates ΔR_i as well as ΔR_j in the queries. Hence, for example, the query to R_{j-1} is $R_{j-1} \bowtie (\Delta R_j \bowtie \cdots \bowtie R_i^{new} + R_j \bowtie \cdots \bowtie \Delta R_i)$ which is equivalent to the following:

$$\begin{gathered} R_{j-1} \bowtie \Delta R_j \bowtie \cdots \bowtie R_i \\ + \\ R_{j-1} \bowtie \Delta R_j \bowtie \cdots \bowtie \Delta R_i \\ + \\ R_{j-1} \bowtie R_j \bowtie \cdots \bowtie \Delta R_i \end{gathered}$$

In Figure 6, we present the Nested SWEEP algorithm that recursively incorporates all concurrent updates encountered during the evaluation of an update. The algorithm recursively evaluates a concurrent update by suspending the current evaluation. It recursively incorporates all the missing terms and then returns to the original query, after modifying it to reflect all relevant concurrent updates. The algorithm is similar in structure to the previous algorithm in that it first sweeps left and then right. On the left sweep of an update ΔR_i, whenever it encounters a new concurrent update ΔR_j, the algorithm recursively evaluates the effect of ΔR_j on all relations from R_{j+1} to R_i. On the other hand, on the right directional sweep, when an update ΔR_k ($k > i$) is encountered, the algorithm recursively incorporates the effects of ΔR_k on all relations $R_1 \cdots R_i \cdots R_{k-1}$ before proceeding to the right of R_k.

The overall structure of the Nested SWEEP algorithm shown in Figure 6 remains the same as SWEEP. The only change that occurs is that in the two for loops, when a concurrent update is detected, it is removed from the message queue, its effects on the ΔV of the current update is compensated, its missing effects are evaluated by a recursive call to the *ViewChange* function, and after that call is completed, the process to compute the remaining terms on behalf of the current update continues by incorporating the concurrent update. A detailed analysis of this algorithm appears in [Yur97].

6.2 Discussion

Although Nested SWEEP does not guarantee complete consistency, it ensures that the state transformations occur in the order of the arrival of the updates. Hence, in that sense Nested SWEEP guarantees strong consistency of the materialized view at the data warehouse. The message complexity of Nested SWEEP is bounded in the worst case by that of SWEEP and hence it is linear. This is because if there is only one update Nested SWEEP is identical to SWEEP. However, if there are multiple updates, Nested SWEEP constructs the view change collectively for all the updates. Thus the message cost is amortized over these multiple updates. We have developed an analytical model to characterize the performance of Nested SWEEP [Yur97]. Unlike Strobe which also performs cumulative view change for multiple updates, Nested SWEEP does not require absolute quiescence where all updates must cease. It, however, does require that there not be a sequence of alternating updates which interfere with each other. In such a case, the algorithm will recursively oscillate between the two source relations until the alternating sequence of interfering updates from these source relations is broken. If this becomes a problem, Nested SWEEP can be easily modified to force termination. In all other cases the process will eventually terminate and the view will be updated.

7 Conclusion

In this paper, we developed a framework to analyze, characterize, and classify view maintenance algorithms for data warehouses. We then use this framework to design two algorithms, called SWEEP and Nested SWEEP, for incremental view maintenance. SWEEP processes one update at a time on the warehouse and constructs the changes in the materialized view for that update. Any interference caused by concurrent updates during this view construction is offset by using local information. SWEEP guarantees complete consistency and has linear message complexity in terms of the number of data sources for each update. This is significantly better than other algorithms that guarantee complete consistency but have exponential message complexity. Also, SWEEP does not require quiescence to incorporate the view changes into the materialized view. Table 1 compares the properties of SWEEP and Nested SWEEP with respect to some of the known algorithms for incremental view maintenance.

We then extend SWEEP to design a recursive view maintenance algorithm called Nested SWEEP that computes view changes for multiple updates collectively. Nested SWEEP guarantees strong consistency and it also has linear message complexity in terms of the number of data sources involved in the data warehouse. Although Nested SWEEP does not require absolute quiescence, it does need a period during which interfering updates should subside for termination. However, the algorithm can be easily modified to guarantee termination by periodically switching to the SWEEP algorithm.

References

[BCP96] Elena Baralis, Stefano Ceri, and Stefano Paraboschi. Conservative Timestamp Revisited for Materialized View Maintenance in a Data Warehouse. In *Workshop on VIEWS'96*, 1996.

[BLT86] J. A. Blakeley, P. A. Larson, and F. W. Tompa. Efficiently Updating Materialized Views. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 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 the ACM SIGMOD International Conference on Management of Data*, pages 469–492, 1996.

[CW91] S. Ceri and J. Widom. Deriving Production Rules for Incremental View Maintenance. In *Proceedings of the International Conference on Very Large Data Bases*, pages 577–589, 1991.

[GHJ96] S. Ghandeharizadeh, R. Hull, and D. Jacobs. Heraclitus: Elevating Deltas to be First-Class Citizens in Database Programming Languages. *ACM Transactions on Database Systems*, 21(3):370–426, September 1996.

```
MODULE DataWarehouse;
    GLOBAL DATA
        $V$ : RELATION; /* Initialized to the correct view */
        $UpdateMessageQueue$ : QUEUE;

    FUNCTION $ViewChange(\Delta R$ : RELATION, $Left, UpdateSource, Right$: INTEGER): RELATION
    VAR
        $\Delta V, TempView$: RELATION;
        $j$: INTEGER;
    BEGIN
        $\Delta V = \Delta R$;
        /* Compute the left part of the recursive incremental view resulting from $\Delta R$ */
        FOR ($j = UpdateSource - 1$; $j \geq Left$; $j --$) DO
            $TempView = \Delta V$;
            SEND $\Delta V$ TO Data Source $j$;
            RECEIVE $\Delta V$ FROM Data Source $j$;
            /* Recursively compute the missing effects of concurrent updates, if any */
            IF $\exists \Delta R_j \in UpdateMessageQueue$ THEN
                Remove $\Delta R_j$ from $UpdateMessageQueue$;
                $\Delta V = \Delta V - \Delta R_j \bowtie TempView$;
                $\Delta V = \Delta V + ViewChange(\Delta R_j, j, j, UpdateSource)$;
            ENDIF;
        ENDFOR;
        /* Compute the right part of the recursive incremental view resulting from $\Delta R$ */
        FOR ($j = UpdateSource + 1$; $j \leq Right$; $j ++$) DO
            $TempView = \Delta V$;
            SEND $\Delta V$ TO Data Source $j$;
            RECEIVE $\Delta V$ FROM Data Source $j$;
            /* Recursively compute the missing effects of concurrent updates, if any */
            IF $\exists \Delta R_j \in UpdateMessageQueue$ THEN
                Remove $\Delta R_j$ from $UpdateMessageQueue$;
                $\Delta V = \Delta V - \Delta R_j \bowtie TempView$;
                $\Delta V = \Delta V + ViewChange(\Delta R_j, Left, j, j)$;
            ENDIF;
        ENDFOR;
        RETURN($\Delta V$);
    END $ViewChange$;

    PROCESS $LogUpdates$;
        /* Same as $LogUpdates$ in Figure 4 */
    END $LogUpdates$;

    PROCESS $UpdateView$;
    BEGIN
        LOOP
            REMOVE ($\Delta R, i$) FROM $UpdateMessageQueue$; /* block if queue empty */
            $V = V + ViewChange(\Delta R, 1, i, n)$
        FOREVER;
    END $UpdateView$;

BEGIN /* Start DataWarehouse Processes */
    $StartProcess(LogUpdates)$;
    $StartProcess(UpdateView)$;
END $DataWarehouse$.
```

Figure 6: The Recursive Incremental View Construction Algorithm

Algorithm	Architecture	Consistency	Message Cost per Update	Comments
ECA	Centralized	Strong	$O(1)$	Remote Compensation Quadratic Message size Requires Quiescence
Strobe	Distributed	Strong	$O(n)$	Unique key assumption Requires Quiescence
C-strobe	Distributed	Complete	$O(n!)$	Unique key assumption Not scalable
SWEEP	Distributed	Complete	$O(n)$	Local compensation
Nested SWEEP	Distributed	Strong	$O(n)$	Local compensation Requires non-interference

Table 1: Comparison of various view maintenance algorithms

[GJM96] Ashish Gupta, H.V. Jagadish, and Inderpal Singh Mumick. Data Integration using Self- Maintainable Views. In *Proceedings of the International Conference on Extending Data Base Theory*, 1996.

[GL95] T. Griffin and L. Libkin. Incremental Maintenance of Views with Duplicates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 328–339, 1995.

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

[GMS93] A. Gupta, I. S. Mumick, and V. S. Subramanian. Maintaining Views Incrementally. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 157–166, May 1993.

[HGMW$^+$95] J. Hammer, H. Garcia-Molina, J. Widom, W. Labio, and Y. Zhuge. The Stanford Data Warehousing Project. *IEEE Bulletin of the Technical Committee on Data Engineering*, 18(2):41–48, June 1995.

[HJ91] R. Hull and D. Jacobs. Language Constructs for Programming Active Databases. In *Proceedings of the International Conference on Very Large Databases*, pages 455–468, September 1991.

[HZ96a] Richard Hull and Gang Zhou. A Framework for Supporting Data Integration Using the Materialized and Virtual Approaches. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 481–492, June 1996.

[HZ96b] Richard Hull and Gang Zhou. Towards the Study of Performance Trade-offs Between Materialized and Virtual Integrated Views. In *Workshop on VIEWS'96*, 1996.

[QGMW96] Dallan Quass, Ashish Gupta, Inderpal Mumick, and Jennifer Widom. Making Views Self-Maintainable for Data Warehousing. In *Proceedings of the International Conference on Parallel and Distributed Information Systems*, December 1996.

[QW91] X. Qian and G. Wiederhold. Incremental Recomputation of Active Relational Expressions. *IEEE Transactions on Knowledge and Data Engineering*, 3(3):337–341, 1991.

[RBS96] RBS. *Data Warehouse Applications*. Red Brick Systems, 1996.

[SI84] O. Shmueli and I. Itai. Maintenance of Views. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1984.

[Yur97] T. Yurek. Efficient View Maintenance at Data Warehouses. Master's thesis, University of California at Santa Barbara, Department of Computer Science, UCSB, Santa Barbara, CA 93106, 1997.

[ZGMHW95] Yue Zhuge, Hector Garcia-Molina, Joachim Hammer, and Jennifer Widom. View Maintenance in a Warehousing Environment. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 316–327, May 1995.

[ZGMW96] Yue Zhuge, Hector Garcia-Molina, and Janet L. Wiener. The Strobe Algorithms for Multi-Source Warehouse Consistency. In *Proceedings of the International Conference on Parallel and Distributed Information Systems*, December 1996.