A Model for Evaluating and Comparing Materialized View Maintenance Algorithms

Eng Koon Sze and Tok Wang Ling
School of Computing
National University of Singapore
S16, 3 Science Drive 2, Singapore 117543
email:{szeek,lingtw}@comp.nus.edu.sg

Abstract

Providing integrated access to information from multiple, distributed, autonomous data sources has received much attention from both the research communities and industries in recent years. Having a materialized view is a straightforward solution to provide fast access to such information, but maintaining such a view to reflect the changes at the data sources is not an easy task. Numerous algorithms have been proposed to handle the materialized view maintenance. The problems faced in this view maintenance include the presence of *interfering* updates, *misordering* of messages and the *loss* of these messages. In this paper, we provide a framework for comparing the merits of each of these approaches. We classify our criteria under four main categories in terms of the *environment* that they function in, the *correctness* of the algorithms, their *efficiency*, and the *application requirements* that these algorithms provide. The environment criteria includes the number of data sources the algorithm supports, and the nature of the compensation process. The correctness criteria looks at how accurate is the identification of interfering updates, and the proper working of the view maintenance process when messages are misordered or lost during their transmission through the network. Efficiency consideration includes the number of base relations accessed per sub-query of the incremental computation, the sequential or parallel handling of incremental computation both within the same update and between different updates, the use of partial self-maintenance in the incremental computation and how is modification handled. Application requirement involves the flexibility of the view definition, degree of consistency provided, and whether is quiescent state of the system necessary before the view can be refreshed. Based on these criteria, we evaluate and compare the existing algorithms in detail.

1 Introduction

Providing integrated access to information from different data sources, possibly at locations away from one another, has received strong interest both from the industry and research communities in recent years. This is the result of better networking facilities, higher processing power of computers, as well as the drop in prices of storage media. Analysis can then be done on this integrated data to provide useful information to aid in future planning, for instance, in deciding on new business strategies. This integrated information is a *view* of the various information at the data sources.

Two methods to this integration of data are the *on-demand* and the *in-advance* approaches. In the on-demand approach, information from the different data sources are gathered and integrated only when requested by users. This approach suffers from delay in presenting the required integrated data to the user, due to the huge amount of information to be processed. Nevertheless, it is the only method applicable if the needs of the users cannot be predicted in advance. On the other hand, information are gathered, integrated and stored in a central repository even before being explicitly requested by users in the in-advance approach. Thus, the integrated information is a *materialized view* of the information at the data sources. In this case, users' queries can be answered faster as compared to the previous approach. Since the information at the data sources are not static, i.e., the data are updated, the materialized view stored at the central repository needs to be refreshed accordingly. This refreshing of the materialized view in response to updates at the data sources is called *materialized view maintenance*. The view can either be recomputed from scratch, or incrementally refreshed, so as to keep up-to-date with respect to the data sources.

Recomputing the view from scratch is not efficient when the base data involved is large and only a small portion of this data is updated. Much time and resources would be required to do this recomputation, and if this is carried out during the normal operation hours of the organization, the overall performance of the database would be affected. Deferring such recomputation of the view to a convenient downtime outside the normal operation hours of the organization is not necessary a good solution as many companies have operations at different timezones of the world, thus such convenient downtime might not be sufficient for the whole recomputation process. Hence, *incremental change* to the view in response to data sources updates is preferred. Up-to-date information of the view is then readily available, or if the view needs only be refreshed during a convenient downtime, this downtime can be kept to a minimum as compared to the approach of recomputing the view from scratch.

Many incremental materialized view maintenance algorithms [CGL+96, GL95, GLT97, GK98, QW91, Qua96] have been developed for centralized database systems. Incremental view maintenance in an environment of independent, autonomous data sources presents new problems not found in the centralized database systems. Such maintenance algorithms must be able to take care of the problems of interfering updates affecting the *incremental computation* process, misordering of messages (update notifications, view maintenance queries, or results of these queries) as well as the loss of update notification messages during their transmission through the communication network. We will look at these problems in greater detail in the next section.

The focus of our survey paper is on the view maintenance algorithms that are designed for non-centralized database systems. In Section 2, we look at the data warehouse model, and explained the incremental computation process which derives the necessary data for incrementally changing the view in response to data sources updates. We also look at the problems affecting the answers of incremental computation, which results in the *view maintenance anomalies*. We proposed a model for evaluating existing materialized view maintenance algorithms for the case of independent, autonomous data sources in Section 3. We compare and constrast the various algorithms that have been proposed so far in Section 4, and look at the merits of each based on our model. We summarize the comparison in Section 5.

2 Background

In this section, we first look at the data warehouse model. Next, we explain the method of incremental change of view maintenance and the levels of consistency in the view maintenance scenario, followed by the problems faced in this approach of maintaining the view.

2.1 Data Warehouse Model

Figure 1 shows the general architecture of a data warehouse. There are m autonomous data sources, each with one or more base relations, and a separate site for housing the materialized view. Network communication exists between each data source and the view site, but no assumption is made with regard to any communications between different data sources. Thus, an update transaction can involve one or more base relations at a single data source, but not between base relations of different data sources. Also, the view site does not control the transactions at the data sources, and thus we have an environment of multiple, distributed, autonomous data sources. Client applications can make use of the view through the issuing of users' queries.

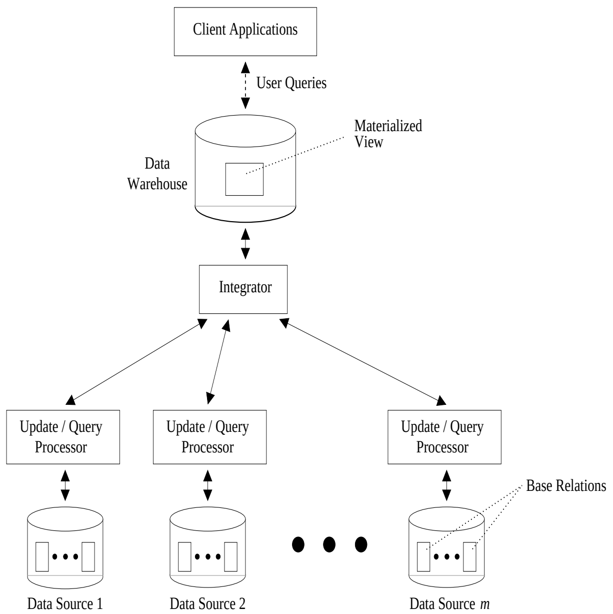

Figure 1: Model of a data warehouse

The underlying data model for both data sources and the view is assumed to be relational data model. We consider the case of select-project-join (SPJ) view. Hence, given the set of base relations $\{R_i\}_{1\leq i\leq n}$, where a data source can contain one or more of these base relations, but the same base relation is not found in two different data sources, the definition of the view V is as given in Equation 1. If none of the keys of the joined relation $R_1 \bowtie ... \bowtie R_i \bowtie ... \bowtie R_n$

is found in V due to projection, then a *count* attribute is appended to indicate the number of ways the same tuple in V could be derived from the base relations.

$$V = \prod_{proj\_attr} \sigma_{sel\_cond} R_1 \bowtie ... \bowtie R_i \bowtie ... \bowtie R_n \tag{1}$$

The *update processor* at the data sources handle the notification of updates that have occurred through the sending of update notification messages to the view site. The *integrator* at the view site process this notification messages by sending out the appropriate view maintenance queries (for incremental computation) to the various data sources. Whenever the *query processor* at a data source receives such queries, the necessary data will be retrieved from its base relations and the result is sent back to the integrator at the view site. Once the integrator receives back all the answers or results of the view maintenance queries, the view relation can be incrementally changed to reflect the new state with respect to the data source update.

2.2 Incremental Computation

To incrementally refresh the view V defined in Equation 1 with respect to an update $\triangle R_i$ on base relation R_i, we first need to compute $R_1 \bowtie ... \bowtie \triangle R_i \bowtie ... \bowtie R_n$ through the issuing of the view maintenance queries from the view site to the other data sources. If only one data source is involved, then the above can be issued as a single query.

However, when considering the case involving multiple data sources, then this view maintenance query needs to be broken down into multiple sub-queries to be sent to the different data sources, where the data sources (or base relations, depending on the working of the view maintenance algorithms) are queried one at a time, using the result of the previous query to generate the next query. Most of the methods handle such incremental computation by doing a left and a right scan of the relations as defined by the relation algebra expression for the view, querying one relation at a time. In the left scan, a query $R_{i-1} \ltimes \triangle R_i$ is sent to R_{i-1} to retrieve those tuples from R_{i-1} that join with $\triangle R_i$. Using the result μR_{i-1} returned from R_{i-1}, query

$R_{i-2} \ltimes \mu R_{i-1}$ is sent to R_{i-2}. This continues until relation R_1 is reached. The same occurs for the right scan. After all the base relations have been queried, the final result is given by $\mu R_1 \bowtie ... \bowtie \triangle R_i \bowtie ... \bowtie \mu R_n$, and this gives the incremental change for the view relation.

Let us first look at how the result of the view maintenance query is used to maintain the view for the case where any of the key of the joined relation $R_1 \bowtie ... \bowtie R_i \bowtie ... \bowtie R_n$ is retained in the view V. Assuming that the tuples in $\triangle R_i$ is made up of one type of update of insertion, deletion or modification, then each tuple (after applying selection condition and projection) in $\mu R_1 \bowtie ... \bowtie \triangle R_i \bowtie ... \bowtie \mu R_n$ corresponds to inserting, deleting of modifying the same tuple in the view V.

For the case where none of the keys of the joined relation $R_1 \bowtie ... \bowtie R_i \bowtie ... \bowtie R_n$ is retained in the view V, we need to take into account the fact that the same tuple in V could be derived from different ways from the base relations as indicated by the value of count. For example, if a particular tuple of the view has a count value of 4, then a deleted tuple (after applying selection condition and projection) in $\mu R_1 \bowtie ... \bowtie \triangle R_i \bowtie ... \bowtie \mu R_n$ would mean decrementing the count value of the same tuple in V from 4 to 3, instead of dropping the tuple completely. If it had been an insertion instead, the count value would then become 5.

2.3 Levels of Consistency

Let us look at two of the levels of consistency of the view with respect to the updates at the data sources as defined in [ZGMW96, ZGMW98]. Each of the algorithms discussed in this paper achieves one of these levels of consistency.

Definition 1 The view is in **complete consistency** with respect to a data source if each update transaction at the data source is incorporated into the view in the same order as they have occurred. The order of update transactions from different data sources are not relevant since we do not consider the case of transactions involving multiple data sources. ■

Definition 2 If some of the consecutive update transactions of a data source are incorporated into the view in a single combined step, with the different steps still preserving the order of data source actions, then the view is in **strong consistency** with respect to the data source.

■

Thus, a view that is in complete consistency with respect to a data source will also be in strong consistency with respect to the data source, while the reverse is not necessary true.

2.4 View Maintenance Anomalies and Compensation

The incremental change approach in materialized view maintenance as outlined earlier works well when no two updates occur close to each other with respect to time, and when the network is working in ideal situation. However, this is not the case in the real world applications. The three problems which we will look at here are (1) the effect of *interfering updates* on the result of incremental computation, (2) the misordering of messages as they travel through the network, and (3) the loss of these messages in the network.

2.4.1 Interfering Updates

In an environment of autonomous data sources, there is no way to enforce the constraint that updates do not interfere with each others' incremental computation, i.e., we cannot force an update of a data source to occur only after the completion of incremental computation of updates from other data sources. The following example shows the problem in maintaining the view due to the presence of interfering updates.

Example 1 Consider 2 base relations $R_1(\underline{A}, B)$ and $R_2(\underline{B}, C)$. Let the view V be defined as $V = \prod_C R_1 \bowtie R_2$, with a count attribute added for the proper working of the incremental computation. Let R_1 contains a single tuple (a1,b1) and let R_2 be empty. Hence the view is also empty. An insertion update of R_1(a2,b1) occurs. The view receives this update notification and the query $Q_1 = [\prod_B R_1(a2, b1)] \rtimes R_2$ is formulated for the incremental computation. Let insertion update R_2(b1,c1) occurs just before the query Q_1 reaches the relation R_2. The tuple (b1,c1) would then be returned as the answer for Q_1. The overall result (without projection) is the single tuple (a2,b1,c1), and the single projected tuple (c1) is added to the view to give (c1,1).

$V(C, count)$
c1,1

For insertion R_2(b1,c1), the query $Q_2 = R_1 \ltimes [\prod_B R_2(b1, c1)]$ is sent to R_1. The result {(a1,b1),(a2,b1)} returned for Q_2 adds two projected tuples of (c1) to give (c1,3). The final base and view relations are as follow.

$R_1(\underline{A}, B)$
a1,b1
a2,b1

$R_2(\underline{B}, C)$
b1,c1

$V(C, count)$
c1,3

Clearly, the view here is not consistent with the base relations. The presence of interfering updates in the incremental computation of insertion update R_1(a2,b1) results in this view maintenance anomalies as the tuple (a1,b1,c1) appears in the results of the incremental computation of both updates. This adds an extra tuple of (c1) to the view relation, giving it a count value of 3 instead of 2. ■

2.4.2 Misordering of Messages

Compensation is the removal of the effect of interfering updates from the result of incremental computation so that the view can be maintained correctly. The compensation process of some of the view maintenance algorithms identifies the presence of interfering updates based on the first-sent-first-received assumption of delivery of messages through the network. A studies carried out by [CM97b] showed that 1 percent of the messages delivered over the network are misordered. Thus, those algorithms which placed the assumption on the first-sent-first-received delivery of messages will not work correctly.

The idea behind the first-sent-first-received assumption of identification of interfering updates is as follow. Consider two updates u_i and u_j such that u_i is received by the view earlier than u_j. Let u_j comes from relation R_j. If the result of incremental computation of update u_i from R_j reaches the view site later than that of u_j, then u_j is an interfering update on this result. The working of the compensation step is to remove the effect of update u_j from this result.

Example 2 Applying the method for identifying the interfering updates using the order of arrival of messages at the view site to the previous example, it is obvious that insertion update R_2(b1,c1) is an interfering update on the incremental computation of update R_1(a2,b1). Thus, we know that the tuple (b1,c1) in the result of the query to R_2 should be dropped (to eliminate the effect caused by insertion R_2(b1,c1)). This is because if update insertion R_2(b1,c1) has not occurred, then (b1,c1) would not be returned in the result (to undo the effect of insertion R_2(b1,c1)). Now the view is refreshed correctly.

However, if the order of arrival of notification for update R_2(b1,c1) and the query result of incremental computation of update R_1(a2,b1) is swapped due to the misordering of messages, then using this method of identifying interfering updates will fail to detect R_2(b1,c1) as an interfering update. ∎

2.4.3 Loss of Messages

The third problem is the loss of messages. The loss of the view maintenance query messages or their corresponding results can be detected by the view site by setting a timer whenever such queries are issued. However, the loss of an update notification message from a data source cannot be detected by the view site directly. Thus, the view would not be refreshed with respect to this update, and the compensation of other updates would not take this update into account if it is an interfering update.

Example 3 Using the same scenario as in Example 1, if the update notification of insertion R_2(b1,c1) is lost during its transmission to the view site, then no compensation of the query result of update R_1(a2,b1) is carried out. The overall result of insertion R_1(a2,b1) is (a2,b1,c1), and this adds one tuple of (c1) to the view to give (c1,1). Since the update notification of insertion R_2(b1,c1) is not received by the view site, there is no change to the view for this update. The final base and view relations are as follow.

$R_1(\underline{A}, B)$
a1,b1
a2,b1

$R_2(\underline{B}, C)$
b1,c1

$V(C, count)$
c1,1

Insertion R_1(a2,b1) should not have contributed to the above view tuple, and insertion R_2(b1,c1) should have placed the tuple (c1,2) in the view relation. ■

3 Model for Materialized View Maintenance Algorithms

Having discussed the general architecture of a data warehouse, how the maintenance of the materialized view is carried out, and the various problems faced in maintaining such view, we now proposed a model for evaluating materialized view maintenance algorithms based on some criteria. These criteria defined can be grouped under four categories, namely, environment, correctness, efficiency and application requirement. The following subsections examine these criteria in greater detail.

3.1 Environment

Criteria	Type	Ideal Case
Data source	Single	Multiple
	Multiple	
Compensation Process	Compensation queries	Local compensation
	Local compensation	

Table 1: Environment criteria

Table 1 summaries the criteria under the environment category. View maintenance algorithms defined can be used for either a data warehouse of single data source or multiple data sources. Those algorithms designed for multiple data sources are better suited to the data warehouse environment where there tend to be more than one data source, and where analysis are usually performed on the collective data from these data sources. Also, an algorithm designed for a multiple data sources environment can be used for a single data source case, but not vice versa.

Compensation is used to resolve the anomalies of the incremental computation of an update caused by interfering updates. Compensation can either be carried out by issuing compensation queries to some of the data sources, on top of the usual view maintenance queries, or it can be resolved locally at the view site. The sending of compensating queries to data sources, though it simplifies the complexity of the compensation algorithms, generates more network traffic. Also, these compensating queries are subject to the effect of further interfering updates, resulting in more compensating queries being generated. Handling the compensation at the view site locally, without the sending of any compensating queries, is a better approach to the view maintenance problems. Of course, this come with the penalty of either having a more sophiscated algorithms, or extra information has to be stored on top of the required view relation.

3.2 Correctness

Criteria	Type	Ideal Case
Detection of interfering updates	Not precise	Precise
	Precise	
Communication assumption	First-sent-first-received	No assumption
	No assumption	
	Non-loss	No assumption
	No assumption	

Table 2: Correctness criteria

Next, we look at the criteria under the correctness catergory (Table 2). The precise detection of interfering updates allow for complete consistency of the view, since identifying the interfering updates allow the view to correctly reflect the various states of the data sources. Nevertheless, it is still possible to achieve complete consistency when interfering updates are not precisely detected, by taking other constraints of the database into consideration. For instance, the C-Strobe algorithm [ZGMW96, ZGMW98] uses the key constraints of the base relations to eliminate duplicate tuples that are erroneously added to the query result when non-interfering deletion updates are identified as interfering updates. In this way, the error is removed before being applied to the view, and thus complete consistency is still maintained.

The second criterion is the assumption on the network communication. Placing assumption on the first-sent-first-received, or non-loss delivery of messages over the communication network will cause the view to be maintained incorrectly when such assumption fails. Taking into consideration that the view and a data source might be located far away from one another, then it is possible for messages delivered to be misordered when they take different paths to reach the destination, or messages might fail to reach the destination due to misrouted or damaged packets, etc.

3.3 Efficiency

Criteria	Type	Idea Case
View maintenance query nature	One query for one relation	One query for all relations
	One query for one data source	
	One query for all relations	
Incremental computation for one update	Sequential	Parallel
	Left and right scan	
	Parallel	
Incremental computation between updates	Sequential	Parallel
	Parallel	
Self-maintenance	None	Combined
	Identify irrelevant updates	
	Use keys	
	Use functional dependencies	
Handle modification	Consider as deletion and insertion	Handle as modification
	Handle as modification	

Table 3: Efficiency criteria

Since data warehouse deals with huge volume of data, efficiency is another important consideration (Table 3). A view maintenance query could be broken up into many sub-queries, and each sub-query is sent to one or a few base relations. The ideal case is for all base relations to be accessed by a single query, but this is only applicable in an environment of single data source. For multiple data sources environment, base relations within the same data source should be ideally queried together. The worst case is when each base relation has to be accessed by one view maintenance sub-query. Not only is more network traffic being generated in this case, the time spent in the incremental computation of an update becomes longer because the cummulative round-trip time is now greater.

The incremental computation of an update is carried out differently depending on the view maintenance algorithms defined. It could be the case where only one base relation is queried at any one time, and the next base relation is only accessed after the result is returned from the previous sub-query. Some algorithms handle the querying of the base relations based on doing a left and right scan of the base relations as defined in the relational algebra of the view relation, using the relation where the update has occurred as the starting point. The ideal case would be to access the base relations based on the join graph so that as many base relations from the same data source can be accessed via a single query, and multiple data sources can be queried in parallel. Independent path can be handled in parallel, and cartesian product would be avoided to prevent the transmission of huge volume of data across the network.

Some algorithms handle the incremental computation between different updates sequentially, while some handle them in parallel. Executing the incremental computation one at a time simplifies the maintenance algorithms, but requires a longer time for the maintenance process to finish. If too many updates occur, then in the worst case, the number of updates waiting for incremental computation might never decrease.

The examples shown in the previous section maintain the view by using the updates and querying the base relations directly. *Partial self-maintenance* is the maintenance of the view through using a combination of the updates, the base relations, as well as the view relation, since the view contains information of the base relations. On the other hand, *self-maintenance* is the maintenance of the view through using the updates and the view relation, and possibly some auxiliary views, without the need to query the base relations. In this paper, we will not deal with self-maintenance algorithms [Huy96a, Huy96b, Huy97a, Huy97b, QGMW96]. If the incremental computation of an update does not take issues of partial self-maintenance into consideration, then view maintenance queries must be issued to the data sources in all cases. By incorporating partial self-maintenance, incremental computation can be handled by querying only some of the base relations, with the remaining information gathered from the view relation, or in the extreme case, the view itself might provide all the information necessary for incremental computation. Partial self-maintenance can come in various types, from detecting irrelevant updates to using keys and functional dependencies in the algorithms. Since using the view relation in the maintenance algorithms cut down the number of queries sent, all the above types of partial self-maintenance considerations should be incorporated in the algorithm.

Some view maintenance algorithms treat modification as a deletion update, followed by an insertion update. Although this does simplify the maintenance algorithms by having to consider two types of updates instead of three, it introduced inefficiency into the maintenance process and also prevent the use of partial self-maintenance issues which would otherwise be possible if the modification update is not being splitted. Handling modification update as deletion and insertion creates more network traffic for the maintenance process as two updates are to be considered now instead of one. If indexes are being built for the view, then these indexes have to be modified (caused by deletion and insertion of the view tuples) even though the modified attributes might not have an index built on it.

3.4 Application Requirements

We now look at the criteria under the application category (Table 4). Some view maintenance algorithms limit the view definition to a fixed number of base relations, for instance, two base relations in order to handle outerjoin. There are also algorithms that require that the key of each base relation be retained in the view relation. It would be preferred if none of these constraints need to be applied, and the users are free to defined their views base on their needs.

Criteria	Type	Ideal Case
View definition	Number of base relations limited to 2	Flexible
	Retain key of base relation	
	Flexible	
Consistency	Strong	Complete
	Complete	
Quiescence state	Required	Not required
	Not required	

Table 4: Application requirements criteria

Some applications require that the view to be in strong consistency with the data sources, while some might demand the view to be in complete consistency. Maintenance algorithms that cater for complete consistency can also be used by applications requiring strong consistency only, but not vice versa.

If the view can be refreshed only when there is quiescence in the system, then the view might not be able to get a chance to be updated if the data sources keep changing. Also, the number of different states reflected by the view depends on the frequency of the occurrence of this quiescent state. This also means that a view maintenance algorithm that require a quiescent state of the system before the view can be refreshed can only provide for strong consistency. Algorithms that do not require the system to be quiesence before refreshing the view are more suitable for an environment where there are frequent updating.

Having look at the criteria necessary for an ideal materialized view maintenance algorithm, we proceed to look at the existing algorithms proposed and evaluate them based on these.

4 Evaluation of Materialized View Maintenance Algorithms

In this section, we look at the workings of the various view maintenance algorithms that have been proposed. They are the Eager Compensating Algorithm (ECA and ECA^K) [ZGMHW95], the Strobe Algorithm (T-Strobe and C-Strobe) [ZGMW96, ZGMW98], the work of [CM97a], the SWEEP Algorithms (SWEEP and Nested SWEEP) [AASY97], the work of [CM97b], and the approach of [LS99, SL00].

4.1 Eager Compensating Algorithm

The Eager Compensation Algorithm (ECA) [ZGMHW95] is designed for an environment with a single data source, with a separate site for housing the view. The data source can contain one or more base relations.

Incremental computation is handled by the view site through the issuing of view maintenance queries to the data source. In ECA, the effect of interfering updates is taken care of by the sending of extra queries called compensating queries.

With only one data source involved, a single view maintenance query is sufficient to be sent to the data source. Incremental computation of different updates could be handled concurrently. Since it takes time for the query result to be returned after the issuing of the view maintenance query, those updates with pending query results are temporary placed in an *unanswered query set* (UQS). Once its query result is received back at the view site, this update is removed from UQS and its result placed in $COLLECT$.

For each view maintenance query of an update, there is also a need to generate a compensating query based on the presence of other updates in UQS. For each insertion update in UQS, the result of the compensation query is to be substracted away from the result of the view maintenance query. On the other hand, for each deletion update in UQS, the result of the compensation query is to be added to the result of the view maintenance query. In ECA, modification update is treated as a deletion update, followed by an insertion update. To achieve strong consistency, the view can only be refreshed when UQS is empty. The view is refreshed by applying the results in $COLLECT$ to the view through the adding or deleting of tuples, or updating of the count attribute. $COLLECT$ is then initialized to empty for the next cycle of view maintenance.

Example 4 We look at how ECA is used to resolve the view maintenance anomalies of Example 1, assuming that the first-sent-first-received of delivery of messages hold. The view maintenance query for insertion R_1(a2,b1) is $Q_1 = [\prod_B R_1(a2, b1)] \rtimes R_2$. Since when the view site receives the notification of insertion R_2(b1,c1), the incremental computation of insertion R_1(a2,b1) is still outstanding (in UQS), there is a need to generate a corresponding compensating query of $R_1(a2, b1) \ltimes R_2(b1, c1)$. Thus, the view maintenance and compensating queries of insertion R_2(b1,c1) is $Q_2 = [R_1 \ltimes \prod_B R_2(b1, c1)] - [R_1(a2, b1) \ltimes \prod_B R_2(b1, c1)]$. The tuple (b1,c1) would be returned as the answer for query Q_1. The overall result (without projection) is the single tuple (a2,b1,c1). However, the view cannot be refreshed at this moment because UQS is not empty as Q_2 is still outstanding. The result is temporary placed in $COLLECT$. Once the tuple (a1,b1) is received back at the view site for query Q_2 (overall result is the single tuple (a1,b1,c1), indicating one projected tuple of (c1) is to be added to the view), the view relation can be refreshed because UQS is now empty. The resultant view is as follow.

$V(C, count)$
c1,2

Note that although there are two separate update transactions at the data sources, the view is only capable of showing them as a combined transactions, i.e., strong consistency. ■

Evaluating ECA using the criteria defined earlier, ECA is designed for a single source environment, with view maintenance anomalies handled by the sending of compensating queries on top of the usual view maintenance queries. The working of ECA does not require the identification of interfering updates, and messages are assumed to be delivered in a first-sent-first-received manner, with no loss of any of these messages. A single query is sufficient to access all base relations for the incremental computation of an update because all the base relations are in one data source. Incremental computation of multiple updates can be handled concurrently. Partial self-maintenance issues are not incorporated into the algorithm, and modification is treated as deletion and insertion. View definition is flexible, but only strong consistency is achieved and quiescent state of the system is required before the view can be refreshed.

A variation of the ECA is the ECA-Key Algorithm (ECA^K) [ZGMHW95], designed for the case where the projection list of the view definition contains the key attributes of each of the base relations. In this algorithm, $COLLECT$ is initialized to be the same as the current copy of the materialized view at the start of a new maintenance cycle. For a deletion update, no view maintenance query is required as the key of the tuple is sufficient to identify the view tuples in $COLLECT$ that need to be deleted. For an insertion update, the usual view maintenance query as in the case of the standard ECA is generated, but no compensating queries will be issued. Results of the view maintenance queries for insertion updates are applied to $COLLECT$, without adding in duplicate tuples, effectively dropping the error terms caused by interfering insertion updates. Whenever UQS is empty, the materialized view is replaced with $COLLECT$. Unlike the case of the standard ECA, $COLLECT$ will not be initialized to an empty set. The next cycle of view maintenance will work on this copy of $COLLECT$ (which is also the current copy of the materialized view).

As in the case of ECA, ECA^K is designed for a single data source environment. However, unlike ECA, compensating queries are not required as compensation is handled locally at the view site. The criteria under the correctness category is the same as that of ECA. All, but the partial self-maintenance criteria, under the efficiency category is also identical to that of ECA. In ECA^K, partial self-maintenance is applicable for the case of deletion update due to the retaining of the keys of the base relations in the view. Thus, the view definition here differs from that of ECA, which have no such requirement. Strong consistency is achieved and quiescent state of the system is needed before the view can be refreshed.

4.2 Strobe Algorithm

The Strobe family of algorithms [ZGMW96, ZGMW98] are designed for an environment with multiple, distributed, autonomous data sources, thus overcoming the single source restriction imposed by ECA. In these algorithms, the key of each base relation is retained in the view.

This family of algorithms consists of Strobe for transactions made up of atomic updates, Transaction-Strobe (T-Strobe) for transactions on base relations within a data source, and Global-Strobe (G-Strobe) for global transactions involving multiple data sources. The workings of the various algorithms are similar, with slight modifications to ensure that the view does not reflect any state that show any intermediate steps of a transaction. In this paper, we will only discuss Transaction-Strobe, and for simplicity, we will just refer to it as the Strobe Algorithm. All these algorithms achieve strong consistency. We will also look at the Complete Strobe Algorithm (C-Strobe) [ZGMW96, ZGMW98], which is designed to achieve complete consistency.

Update transactions notification messages are sent by the data source to the view site, as well as results of view maintenance queries issued by the view site.

At the view site, an *action list* (AL) is initially set to empty. Upon the receipt of an update transaction notification from any data source, the deletion updates are processed first, followed by the insertion updates. For each deletion update of a transaction, it is added to the *pending* set of each insertion update currently in the *unanswered query set* (UQS). Since the key attribute of each base relation is retained in the view, there is no need to issue any view maintenance queries for deletion updates as in the case of ECA^K. Thus, these deletion updates can be immediately placed in the AL.

For each insertion update of the transaction, it is initialized with an empty *pending* set. Since multiple data sources are involved in this case, the incremental computation is carried out by sending various sub-queries to the different data sources as it is not possible to do it within one query, unlike the case of ECA. This insertion update is placed in UQS while its view maintenance queries are executing. Similar to ECA, when the final result of an insertion update is received back at the view site, this insertion update is removed from UQS. Before placing the result of the insertion update in AL, those deletion updates in its *pending* set are used to remove those tuples from its result using the value of the key. After this, the remaining result is then placed in AL.

The view can only be refreshed when UQS is empty. The results in AL are applied to the materialized view. For each deletion update, the key value is used to remove the appropriate view tuples. For each insertion update, its result is added to the view, without putting in any duplicate view tuples, and thus effectively dropping the error terms.

Strobe defined the incremental computation of an update by sending query to one base relation at a time, using the result of the previous query (the update itself is used for the first query) to generate the next query. It requires that the next base relation queried should be joined with the base relation whose query result is used to generate the query.

Example 5 Example 4 resolves the view maintenance anomalies using ECA. In this example, we look at how Strobe is used to resolve the same problem. The view V integrates data from base relations $R_1(\underline{A}, B)$ and $R_2(\underline{B}, C)$, with the view definition given by $V = \prod_{A,B,C} R_1 \bowtie R_2$ (note that the attribute A and B have to be retained in V, although we are only interested in the attribute C in our application). Assume that the delivery of the messages obey the first-sent-first-received constraint. The view maintenance query for insertion R_1(a2,b1) is $Q_1 = [\prod_B R_1(a2, b1)] \rtimes R_2$, and the view maintenance query of insertion R_2(b1,c1) is $Q_2 = R_1 \ltimes [\prod_B R_2(b1, c1)]$. The tuple (b1,c1) would be returned as the answer for query Q_1. The overall result (without projection) is the single tuple (a2,b1,c1). However, the view cannot be refreshed at this moment because UQS is not empty as Q_2 is still outstanding. The result is temporary placed in AL. Once the tuples {(a2,b1),(a1,b1)} are received back at the view site for query Q_2 (overall results are the tuples {(a2,b1,c1), (a1,b1,c1)}), the view relation can be refreshed. The result of query Q_1 is added to the view relation. As for the query result of Q_2, only the tuple (a1,b1,c1) is added to the view because the tuple (a2,b1,c1) is already present in the view. The final view relation is as follow.

$V(\underline{A}, B, C)$
a1,b1,c1
a2,b1,c1

■

Strobe differs from ECA and ECA^K in that it is designed for a multiple data sources environment instead of a single data source environment. As in ECA^K, compensation is handled locally at the view site without the need to issue any compensating queries. The criteria under the correctness category is the same as that of ECA and ECA^K. Unlike ECA and ECA^K which access all base relation using a single query, Strobe uses one sub-query for each base relation of the view because of the multiple data sources environment. Incremental computation of an update is handled by querying each base relation in a sequential manner, but multiple updates can be undergoing incremental computation concurrently. As in the case of ECA^K, partial self-maintenance is applicable in the case of deletion update since the key of each base relation is retained in the view. Modification is treated as a deletion update, followed by an insertion update. The criteria under the application category is the same as that of ECA^K.

Next, we look at the workings of C-Strobe [ZGMW96, ZGMW98], which achieves complete consistency and overcomes the quiescent state requirement in refreshing the view in the standard Strobe Algorithm. In C-Strobe, updates are processed for incremental computation one at a time, in order of their arrival at the view site.

Initially, the set *Delta* is empty. For a deletion update, no view maintenance queries are necessary and the view can be maintained immediately using the key value of the updated tuples to remove the appropriate view tuples.

As for insertion update, view maintenance queries are generated as in the standard Strobe Algorithm, and the final result is added to *Delta*. For all deletion updates that arrived after the receiving of this insertion update by the view site, but before the final result of its view maintenance query, compensating query is generated to incorporate the tuples of the deletion update which were missed previously (this handle the compensation for interfering deletion updates). The query result of these compensating queries could also be affected by interfering updates, and thus might also require further compensation. For all other insertion updates that arrived after the receiving of this insertion update by the view site, but before the final result of its view maintenance query, drop those tuples from *Delta* that have the same key value as the tuples in these other insertion updates (this handle the compensation for interfering insertion updates). After all this necessary compensation, the tuples from *Delta* are added to the materialized view, and *Delta* is initialized to an empty set for the next cycle of view maintenance process.

We now evaluate the C-Strobe Algorithm using the model we defined. C-Strobe is designed for a multiple data source environment. Compensating queries are required for the case of insertion update with interfering deletion updates (insertion update with interfering insertion update is handled locally by dropping the appropriate tuples from the query result, and deletion update does not have the problem of interfering updates since no querying of the base relations is involved). The method for identifying the interfering updates is not precise in that some non-interfering deletion updates are mistakenly detected as interfering updates. Nevertheless, no anomalies is introduced because the algorithm ensures that no duplicate tuples are added to the view (identified using the keys of the base relations). Messages delivered through the network are assumed to be first-sent-first-received and non-loss. Efficiency consideration is identical to that of Strobe, except that the incremental computation between different updates is handled sequentially in C-Strobe, whereas that of Strobe handles it in parallel. It differs from Strobe in the application category by being in complete consistency, instead of strong consistency, with the data sources. It also does not require that the system be quiescence before the view can be refreshed.

4.3 Algorithm by [CM97a]

The algorithm by [CM97a] is designed for a view defined by outerjoin operator (can also be used for natural join operator), and thus the number of base relations is limited to two. It is not applicable to the general data warehousing environment. Nevertheless, it is discussed here because it is the first approach that correctly identifies the presence of interfering updates and thus achieve complete consistency.

Unlike the natural join operation where the dangling (unmatched) tuples in the operand relations are eliminated, outerjoin keeps those dangling tuples and fills unmatched fields with null values. To handle the correct maintenance of view defined by outerjoin without the sending out of extra queries on top of the view maintenance queries, [CM97a] defined the maintaining of an additional table in the front-end System Catalogue. In this table, each row contains a distinct join attribute value in $\prod_J R_1 \cup \prod_J R_2$, where J is the join attribute between relations R_1 and R_2, and the numbers of tuples in R_1 and R_2 which have the corresponding J value.

We will focus our discussion on the maintenance for the view defined by natural join operator. As in the case of C-Strobe, updates are processed for incremental computation one at a time, in order of their arrival at the view site. Consider the incremental computation of an update u_1, those updates from the other base relation (not the same base relation as that of update u_1) that arrive after u_1 but before the arrival of the result of view maintenance query of update u_1 are the interfering updates on the result of update u_1.

After the identification of the interfering updates, compensation is carried out locally at the view site. For an interfering insertion update, those tuples of the result containing this interfering insertion update is dropped. As for a deletion interfering update, the resultant joined tuples of update u_1 and the interfering deletion update are added to the result of view maintenance query. Again, in this algorithm, modification is handled as a deletion and an insertion updates. After applying the compensation steps, the result can be applied to the materialized view.

Example 6 We use the scenario as in Example 1. The corresponding System Catalogue is as shown below.

Attribute	Count in R_1	Count in R_2
b1	1	0

When the view receives the notification of insertion R_1(a2,b1), it will not query the base relation because it knows from the System Catalogue that there is no tuple from R_2 that will join with this tuple. There is no change to the view, but the System Catalogue is updated as follow.

Attribute	Count in R_1	Count in R_2
b1	2	0

For insertion R_2(b1,c1), it will have to query base relation R_1 using the query $R_1 \ltimes [\prod_B R_2(b1, c1)]$. The tuples {(a1,b1),(a2,b1)} are returned, and therefore two tuples of (c1) are added to the view as follow.

Attribute	Count in R_1	Count in R_2
b1	2	1

$V(C, count)$
c1,2

In this example, compensation is not necessary because the System Catalogue eliminates the irrelevant updates. We leave the case where compensation is required to the next example when we discuss the SWEEP Algorithm because the resolving of the anomalies is the same. ■

As this algorithm is designed to cater to views defined by outerjoin operator, the base relations involved is limited to two. Compensation is handled locally at the view site by removal of the interfering updates immediately when the result of the view maintenance query is returned from the base relations. [CM97a] correctly identifies the interfering updates. Messages delivered over the network are assumed to be first-sent-first-received, and they are also assumed to be non-loss. As only two base relations are involved, the incremental computation of an update only requires the query to be sent to one base relation. Incremental computation between different updates are handled sequentially. Some irrelevant updates (updates that will not affect the view) can be identified from the front-end System Catalogue, and thus unnecessary view maintenance queries and its corresponding compensation can be eliminated as shown in the previous example. Modification is handled as deletion and insertion. Complete consistency is achieved, and the view can be refreshed without requiring that the system be quiescent.

4.4 SWEEP Algorithms

Like the Strobe Algorithm, the SWEEP Algorithm [AASY97] is designed for an environment with multiple, distributed, autonomous data sources. However, its view definition is more flexible in that it does not require that the key of each base relation be retained in the view definition. Complete consistency is achieved through the correct identification of interfering updates using the approach similar to [CM97a], but extending it to accomodate more than two base relations. This maintenance algorithm caters to source local transaction.

At the view site, updates are processed one at a time for incremental computation, in the same order as they have arrived at the view site, similar to the approach taken by [CM97a].

The incremental computation of an update requires the splitting of the view maintenance query into multiple sub-queries since more than one data source is involved. The SWEEP Algorithm handled this by defining a left and a right sweeps in doing the computation. Consider a view V defined by Equation 2. For an update $\triangle R_i$ on base relation R_i, the incremental computation proceeds by first doing the left sweep, sending first the query $R_{i-1} \ltimes \triangle R_i$ to base relation R_{i-1}. When the query result μR_{i-1} is returned from base relation R_{i-1}, another query $R_{i-2} \ltimes \mu R_{i-1}$ is sent to R_{i-2}. This continues until base relation R_1 is queried, after which the incremental computation process is transferred to the right sweep using the same approach as in the left sweep.

$$V = R_1 \bowtie ... \bowtie R_i \bowtie ... \bowtie R_n \tag{2}$$

Interfering updates from base relation R_j, where $1 \leq j \leq n$ and $j \neq i$, on the query result of incremental computation of this update $\triangle R_i$ are those updates that arrived at the view site later than $\triangle R_i$, but before the arrival of the query result μR_j at the view site.

The compensation of the query result of $\triangle R_i$ with the interfering deletion and insertion updates is similar to the approach taken by [CM97a]. Compensation with interfering deletion update $\triangle R_j$ adds to the query result those tuples from $\triangle R_j$ that could join with the existing result, i.e., $\mu R_j = \mu R_j \cup (\triangle R_j \ltimes \mu R_{j-1})$ (assuming we are currently executing the right sweep). While compensation with interfering insertion update $\triangle R_j$ drops those tuples from $\triangle R_j$ that are present in μR_j, i.e., $\mu R_j = \mu R_j - \triangle R_j$. After carrying out this compensation, querying of the next relation (R_{j+1} in this case since we are considering the right sweep) can then be carried out. When the final query result is received back from R_n, the next update in the queue can be processed for incremental computation. The compensated result $\mu R_1 \bowtie ... \bowtie \triangle R_i \bowtie ... \mu R_n$ are applied to the view based on the update transaction so that the view does not reflect intermediate states of a transaction.

Example 7 Again, considering the scenario in Example 1. When the view receives the notification of insertion R_1(a2,b1), it generates the query $Q_1 = [\prod_B R_1(a2, b1)] \rtimes R_2$ and sends it to base relation R_2. Before the query result of Q_1 is returned, it receives the notification for insertion R_2(b1,c1). Thus, this update is an interfering insertion update on the query result of Q_1. On receiving back the tuple (b1,c1) from R_2 for Q_1, compensation is carried out with the interfering insertion update of R_2(b1,c1) and this made the query result of Q_1 empty. There is no change to the view for insertion R_1(a2,b1). As for insertion R_2(b1,c1), the query $Q_2 = R_1 \ltimes [\prod_B R_2(b1, c1)]$ is sent to R_1, with {(a1,b1),(a2,b1)} returned as the result. This adds two tuples of (c1) to the view relation.

$V(C, count)$
c1,2

■

SWEEP Algorithm executes the incremental computation of the updates one at a time, even though multiple updates could have occurred and are currently queueing up at the view site, waiting to be processed. Noting that such updates may be able to share components of incremental query result, this fact is exploited by the Nested SWEEP Algorithm [AASY97] through piggybacking queries for the new updates on top of the queries initiated for the current update. Thus, multiple updates could be processed for incremental computation concurrently. It also 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.

Consider two insertion updates of $\triangle R_i$ from R_i and $\triangle R_j$ from R_j, where $j > i$ and $\triangle R_i$ reaches the view site before $\triangle R_j$, and let $\triangle R_j$ be an interfering insertion update on the query result of $\triangle R_i$. In the SWEEP Algorithm, those tuples in $\triangle R_j$ that are found in the query result of $\triangle R_i$ are dropped before querying the next base relation. However, in the Nested SWEEP Algorithm, instead of dropping $\triangle R_j$ from the query result of incremental computation of $\triangle R_i$, it re-evaluates the terms between base relations R_i and R_j by computing $(R_i \cup \triangle R_i) \bowtie R_{i+1} \bowtie ... \bowtie R_{j-1} \bowtie (R_j \cup \triangle R_j)$. After this, it proceeds to query the other base relations, i.e., R_{i-1} down to R_1 for the left sweep and R_{j+1} to R_n for the right sweep, giving the result as stated in Equation 3. Note that the same result would have been evaluated if we consider the incremental computation of $\triangle R_i$ (less the interfering insertion update $\triangle R_j$), followed by the incremental computation of $\triangle R_j$ in the case of the SWEEP Algorithm. Also in the SWEEP Algorithm, these result would be incorporated into the view as two separate steps giving complete consistency, while Nested SWEEP incorporate them into the view as a single step to achieve only strong consistency.

$$R_1 \bowtie ... \bowtie R_{i-1} \bowtie (R_i \cup \triangle R_i) \bowtie R_{i+1} \bowtie ... \bowtie R_{j-1} \bowtie (R_j \cup \triangle R_j) \bowtie R_{j+1} \bowtie ... \bowtie R_n \qquad (3)$$

Both SWEEP and Nested SWEEP are designed for an environment with multiple data sources. SWEEP resolves the view maintenance anomalies by local compensation through the immediate removal of interfering updates from the view maintenance queries results when they are received back at the view site, while Nested SWEEP merge view maintenance queries of interfering updates. The identification of interfering updates is precise in both algorithms, and both also place the assumption that messages sent through the network are delivered in a first-sent-first-received manner and no messages are ever lost. Each view maintenance sub-query is only sent to one base relation, and incremental computation of an update is limited to a left and a right sweep of the base relations as given by the relation algebra of the view definition. No partial self-maintenance issues are taken into consideration in the two algorithms, and modification is treated as a deletion and an insertion. Incremental computation between different updates are handled sequentially in SWEEP, while Nested SWEEP handles them in parallel. The view definition is flexible in both cases. SWEEP achieves complete consistency and does not require the system to be quiescent before the view can be refreshed, while Nested SWEEP achieves strong consistency and can only refresh the view when the system is quiescence.

4.5 Algorithm by [CM97b]

The maintenance algorithm of [CM97b], like the SWEEP Algorithm, is designed for an environment with multiple, distributed, autonomous data sources, and with no requirement that the view needs to retain the key of each base relation. It differs from the SWEEP Algorithm in that it does not assume that messages reach the destination in the same order as what was sent out. However, it still assumes that messages sent are never lost.

The workings of the maintenance machinery at the data source here is slightly different from the other maintenance algorithms. Each data source keeps a counter. For an update at a data source, the update notification message besides storing this update, also stores the current value of the counter at the data source. This update notification message is then sent to the view site, and the value of the counter is next incremented by one. For a view maintenance query received from the view, the required result is generated and the current value of the counter retrieved, after which both informations are sent back to the view site.

At the view site, the maintenance machinery process the updates for incremental computation one at a time. However, unlike the algorithm of [CM97a] and the SWEEP Algorithm, it need not necessary process them in the same order as their arrival at the view site because messages could be misordered during their transmission in the network. The view site maintains a global counter $gcounterR_i$ and a maintenance index $indexR_i$. $gcounterR_i$ is used to tell which maintenances for local updates on R_i have been processed. When $gcounterR_i = n$, it means that all maintenances for local updates on R_i, whose corresponding notifications have local counter values smaller than n, have been processed completely for incremental computation. On the other hand, $indexR_i$ is used to indicate which maintenance for a local update on R_i can be initiated. When $indexR_i = m$, it means that the view site can initiate the maintenance for the local update on R_i whose notification have a local counter value equal to m. Both $gcounterR_i$ and $indexR_i$ can only be incremented by one at any one time, and their values cannot be decremented. $gcounterR_i$ is used for the identification of interfering updates, while $indexR_i$ ensures that update notification messages are processed based on the order that they have occurred, and not on their order of arrival.

The processing of the incremental computation is the same as that of the SWEEP Algorithm, where one base relation is queried at a time using the approach of executing a left and a right scan (called backward and forward phase in [CM97b]). For the query result of $\triangle R_i$, the interfering updates from R_j, where $j \neq i$, are those updates from R_j with counter value a such that $gcounterR_j \leq a < C_j$ and C_j is the counter value returned by the query result from R_j. Compensation for the interfering deletion and insertion updates are carried out using the same approach as the SWEEP Algorithm, and it is executed before the querying of the next base relation.

The algorithm of [CM97b] is designed for an environment with multiple data sources. Compensation is handled locally at the view site by removing the interfering updates from the view maintenance sub-queries results when they are returned from a base relation. Interfering updates are correctly identified. No assumption is made with regard to the first-sent-first-received of the messages sent through the network. However, messages are assumed to be non-loss during transmission. Each view maintenance sub-query only accesses one base relation. Incremental computation of an update is limited to doing a left and a right scan on the base relations as given by the relation algebra definition for the view. Incremental computation of different updates are handled sequentially. Partial self-maintenance issues are not considered in this algorithm, and modification is handled as deletion and insertion. The view definition is flexible, complete consistency is achieved and quiescence is not required for the refreshing of the view.

4.6 Algorithm by [LS99, SL00]

View maintenance using version numbers [LS99, SL00], like the algorithm of [CM97b], is designed for an environment with multiple, distributed, autonomous data sources. It differs from [CM97b] in a few aspects. Messages are not assumed to be non-loss during their transmission through the network, multiple base relations within the same data source can be queried together, and incremental computation of different updates can be executed concurrently.

The working of the view maintenance algorithm of [LS99, SL00] is centred around the concept of version numbers. The following types of version numbers are defined to support the working of the algorithm.

1. **Base relation version number** (of a base relation). The base relation version number identifies the state of a base relation. It is incremented by one when there is an update transaction on this base relation, after the transaction is committed. This base relation version number is used by the compensation algorithm to identify the interfering updates of the view maintenance query result.

2. **Data source version number** (of a data source). Since a data source usually has more than one base relation, it is not sufficient to determine the exact sequence of two update transactions involving non-overlapping base relations using the base relation version number alone. Thus, another level of version number is defined called the data source version number which identifies the state of a data source. It is incremented by one when there is an update transaction on the data source, after the transaction is committed. The updates involved in this transaction, the base relation version numbers of their respective base relations, and the current data source version number are sent to the view site as an update notification message. This data source version number is used for ordering the processing of the update transactions for incremental computation and subsequent

refreshing of the view relation. It enables the incremental computation of update transactions to be handled according to the order that they have occurred, instead of their order of arrival at the view site. One data source version number is associated with each data source.

3. **Highest processing version number** (of a base relation, and this number is stored at the view site). The base relation version number of the latest update of a relation sent for incremental computation becomes the highest processing version number. This number is updated with the base relation version number of the update when it is sent for incremental computation (and not at the end of the incremental computation, which would result in the sequential processing of the incremental computation for the various updates). Note that to achieve complete consistency, the incremental computation of update transactions from the same data source have to be processed in the same order that they have occurred, determined through their data source version numbers. This will cause the highest processing version number to be changed in an increasing order only. Hence, keeping this highest processing version number is sufficient to tell which updates of a base relation has been processed for incremental computation. This highest processing version number is then used to assign the initial version numbers (as described next) of the incremental computation of an update. There are n highest processing version numbers for the n base relations.

4. **Initial version numbers** (of an update). The incremental computation of an update should see the effect of those updates (from the other base relations) that have occurred before it, but not those updates that have occurred after it. The ordering here is arbitrary for updates from different data sources since global transaction is not considered. The use of highest processing version numbers to track those updates that have been processed for incremental computation provide the means to determine what are those updates that

have occurred before, as well as those that should occur after. The incremental computation of an update is supposed to access the base relations of certain states. The collection of these states (the state of a base relation is indicated by its base relation version number) is called the initial version numbers of this update. The highest processing version number of the base relation where the update is sent for incremental computation is first updated with the base relation version number of this update. The highest processing version numbers of all the base relations then become the initial version numbers of this update. Since there are n base relations, there are n initial version numbers associated with the incremental computation of an update.

5. **Queried version number** (of a tuple of the result from a base relation). Due to the autonomous nature of the data sources, the query result of the incremental computation of an update generated by the base relations need not necessary be from those states of the base relations as required by this update. In other words, the results are generated by the base relations with different base relation version numbers from the initial version numbers of this update. The queried version number of a result returned by a base relation is the state of the base relation, indicated by its base relation version number, when this result is generated. It is returned as part of the result to the view site. This queried version number is associated with each tuple of the result returned to the view site. Note that if the view relation is accessed instead of the base relation, then the refreshed version number (explained next) of the view is taken as the queried version number instead.

6. **Refreshed version number** (of a base relation, and this number is stored at the view site). Without the involvement of the view in the incremental computation, the base relations have to be accessed in all situations. The use of view in this partial self-maintenance process can greatly reduce the amount of network traffic. To allow for the partial self-maintenance issues to be incorporated into the view maintenance process at the view site,

we need to know the states of the base relations that the view is currently reflecting. This is to allow for the compensation process associated with these partial self-maintenance issues. Define refreshed version number to indicate the base relation version number (state) of a base relation that the view is currently reflecting. When the result of the incremental computation of an update is applied to the view relation, the refreshed version number of the particular base relation is updated with the base relation version number of this update. Since there are n base relations, there are n refreshed version numbers kept at the view site. Note that the refreshed version numbers of the base relations may be lower than the corresponding initial version numbers of the incremental computation of an update, because the results of the incremental computation of the earlier updates have not been incorporated into the view. This refreshed version number will be returned as the queried version number of the result tuples if the view is used in the incremental computation instead of the base relation.

The algorithms for multiple data sources environment discussed in the previous subsections can only access **one** base relation per sub-query (of the view maintenance query), despite the fact that a data source usually has more than one base relation that are involved in the view definition. The detection of interfering updates in this algorithm is not based on the order of arrival of messages, instead it achieves this by storing extra information temporary in the form of queried version numbers with the view maintenance query result. Thus this algorithm is able to access **multiple** base relations residing at the same data source with a single sub-query. The following example shows how the join graph is used to handle the incremental computation for an update. Note that multiple base relations within the same data source are accessed together if they have some join attributes in common as defined in the view.

Example 8 Referring to Figure 2 for the join graph of 6 base relations, where the view is given as $V = \bowtie_{i=1}^{6} R_i$. Consider the incremental computation of update $\triangle R_1$. The first query

$Q_1 = [\prod_{J_a} \triangle R_1] \rtimes [R_2 \bowtie R_3 \bowtie R_4]$, where J_a is the attributes in R_1 required to evaluate the join for $R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4$, is sent to data source 1. Concurrently, the second query $Q_2 = [\prod_{J_b} \triangle R_1] \rtimes R_5$, where J_b is the attributes in R_1 required to evaluate the join for $R_1 \bowtie R_5$, can be sent to data source 2, independent on the completion of the first query. Let A_1 be the result returned by data source 1. The queried version numbers of the tuples from each of the relation R_2, R_3 and R_4 are also temporary stored for purpose of compensation. Ignoring the compensation step of A_1, the next query $Q_3 = [\prod_{J_c} A_1] \rtimes R_6$ is sent to data source 2, again, independent of the completion of the second query.

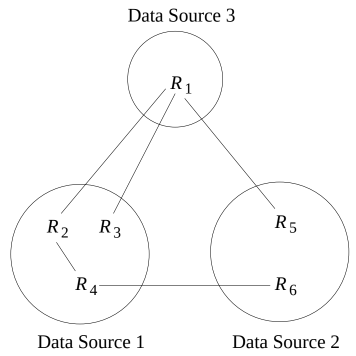

Figure 2: Join graph of base relations for view $V = \bowtie_{i=1}^{6} R_i$

■

Consider the case of an insertion update, knowledge of referential integrity constraint between the base relations of a data source is used to cut down unnecessary view maintenance queries. For instance, if it is enforced by the data source that each tuple in R_j must refer to a tuple in R_i, then insertion updates to R_i can be ignored. Functional dependencies between the attributes are used to decide when can the view be accessed for incremental computation. Suppose we have the following functional dependencies of $R_i \rightarrow R_j$ and $R_j \rightarrow R_k$, and the key of R_j is also in the view, then for insertion update $\triangle R_i$, we can query the view using $\triangle R_i \rtimes V[R_j, R_k]$. Only those tuples in $\triangle R_i$ that cannot retrieve any tuples from the view relation, i.e., $\triangle R_i \bar{\ltimes} V[R_j, R_k]$, need to be sent in the query to the data sources.

In the case of modification update, besides using information on functional dependencies to access the view for incremental computation, it is also used together with the presence of the key of a base relation to help in cutting down the number of base relations that need to be accessed. Supposed it is given that $R_j \rightarrow R_i$, and the key of R_j is in the view, then for a modification update $\triangle R_i$, it is sufficient to query relation R_j, and the intermediate relations that linked R_j and R_i in the join graph. This is because the view can be updated based on the key value of R_j, without the need to explicitly identify the complete view tuples through the full incremental computation process. If the key of R_i is in the view, then $R_j = R_i$ and no querying to any base relation is required.

For deletion, information of referential integrity constraint can be used to eliminate unnecessary queries, functional dependencies can be used to access the view for incremental computation, and together with the knowledge of the key of some base relation being retained in the view identifies a subset of the base relations that are needed to be queried. These are summarized in Table 5.

Improvement	Insertion	Modification	Deletion
Referential integrity constraint	Applicable		Applicable
Functional dependencies to access view for incremental computation	Applicable	Applicable	Applicable
Functional dependencies and key of base relation to query a subset of all base relations		Applicable	Applicable

Table 5: Ways to cut down number and size of queries to data source

Next, we look at how are the presence of interfering and missing updates identified. Consider the incremental computation of update $\triangle R_i$ with initial version number α_j for base relation R_j, and a tuple in the query result from base relation R_j with queried version number β_j. If $\beta_j > \alpha_j$, then updates from R_j with base relation version numbers of β_j down to $\alpha_j + 1$ are the interfering updates. If $\beta_j < \alpha_j$, then updates from R_j with base relation version numbers $\beta_j + 1$ to α_j are the missing updates.

As in the case of the algorithms of [CM97a] and [CM97b], as well as the SWEEP Algorithm, compensation works to undo the effect of the interfering updates. The compensation in [LS99, SL00] also does the same thing. However, besides undoing the effect of the interfering updates, it also undo the effect of missing updates if they are present. On top of that, since multiple base relations could be queried at a time, this undoing operation has to work on more than one base relation. This does not cause any problem since the presence of initial version numbers and queried version numbers serve to identify the presence of any interfering or missing updates, instead of basing it on the order of arrival of messages. For instance, referring to the query result A_1 returned from data source 1 in Example 8. Although A_1 is made up of query result from R_2, R_3 and R_4, there is no problem in identifying the interfering or missing updates because these are done through comparing the initial version numbers α_2, α_3 and α_4 of $\triangle R_1$ with the corresponding queried version numbers β_2, β_3 and β_4 of the tuples in the query result.

The loss of update notification messages is identified by either the missing of an update transaction of some data source version number if the view site already received other transactions of higher data source version number from the same data source. Otherwise, it can also be identified during the compensation of the view maintenance queries results of the updates from the other data sources.

To summarize, this algorithm is designed for an environment with multiple data sources. Compensation is carried out locally at the view site. The identification of the interfering updates is precise, and no assumption is made with regard to the first-sent-first-received nature of delivery of messages, and these messages are also not assumed to be non-loss. Each view maintenance sub-query can access multiple base relations within the same data source. This is possible because the compensation algorithm supports this. More parallelism is exploited in the querying of the base relations for the incremental computation of an update because these relations are accessed based on the join graph of the view. Incremental computation between the different updates are also handled in parallel as this is made feasible through the use of the different types of version numbers. More partial self-maintenance issues are incorporated, and modification is handled as one type of update when none of its join attributes are changed. The view definition is flexible, complete consistency is achieved, and the refreshing of the view does not require a quiescent state of the system.

5 Summary and Conclusion

In this survey paper, we looked at the three problems that materialized view maintenance in an environment of multiple, distributed, autonomous data sources faced. They are the problems of interfering updates, misordering of messages and the loss of these messages.

We proposed a model for evaluating the various materialized view maintenance algorithms, and discussed the workings of these algorithms. Tables 6, 7, 8 and 9 summaries the evaluation of the various algorithms based on the category of environment, correctness, efficiency and application requirement respectively.

From Table 6, we see that most algorithms cater to an environment with multiple data sources, and also handle the compensation locally at the view site without the need to issue any compensation queries.

Table 7 shows that most algorithms either do not identify individual interfering updates (and thus only achieve strong consistency), or identify the individual interfering updates correctly (achieve complete consistency). Only one algorithm fails to precisely identify the interfering updates, but nevertheless the view is still maintained correctly through removal of duplicate tuples. Most of the algorithms place the assumptions that messages sent through the network are delivered in a first-sent-first-received manner and are non-loss. [CM97b] and [LS99, SL00] do not assume the first-sent-first-received delivery of messages. On top of that, [LS99, SL00] also do not assume that messages sent are never loss in the communication network.

Algorithm	Data source	Compensation process
ECA	Single data source	Compensating queries incorporated into maintenance queries
ECA^K	Single data source	Local compensation through key delete and duplicate tuples removal
Strobe	Multiple data sources	Local compensation through key delete and duplicate tuples removal
C-Strobe	Multiple data sources	Compensating queries issued after view maintenance queries, local compensation through key delete and duplicate tuples removal
[CM97a]	Two base relations	Local compensation by immediate removal of interfering updates
SWEEP	Multiple data sources	Local compensation by immediate removal of interfering updates
Nested SWEEP	Multiple data sources	Merge queries of interfering updates
[CM97b]	Multiple data sources	Local compensation by immediate removal of interfering updates
[LS99, SL00]	Multiple data sources	Local compensation by removal of interfering updates, both immediate and at the end of view maintenance queries

Table 6: Comparison of environment of different compensation algorithms

Algorithm	Precise identification of interfering updates	Communication assumption
ECA	N.A.	First-sent-first-received and non-loss
ECA^K	N.A.	First-sent-first-received and non-loss
Strobe	N.A.	First-sent-first-received and non-loss
C-Strobe	Not precise	First-sent-first-received and non-loss
[CM97a]	Precise	First-sent-first-received and non-loss
SWEEP	Precise	First-sent-first-received and non-loss
Nested SWEEP	Precise	First-sent-first-received and non-loss
[CM97b]	Precise	Non-loss
[LS99, SL00]	Precise	No first-sent-first-received or non-loss assumption

Table 7: Comparison of correctness of workings of different compensation algorithms

Algorithm	Query nature as supported by compensation method	Sequential or parallel incremental computation of one update	Sequential or parallel incremental computation between updates	Self-maintenance	Handle Modification
ECA	One query for all relations	N.A.	Parallel	No	No
ECA^K	One query for all relations	N.A.	Parallel	Limited to deletion	No
Strobe	One query for one relation	Sequential (one relation at a time, next relation queried must be joined with current relation)	Parallel	Limited to deletion	No
C-Strobe	One query for one relation	Sequential (one relation at a time, next relation queried must be joined with current relation)	Sequential	Limited to deletion	No
[CM97a]	One query for one relation	N.A.	Sequential	Only identify irrelevant updates	No
SWEEP	One query for one relation	Limited parallelism (left and right scan, next relation queried dependent on relation algebra expression defined for view)	Sequential	No	No
Nested SWEEP	One query for one relation	Limited parallelism (left and right scan, next relation queried dependent on relation algebra expression defined for view)	Parallel	No	No
[CM97b]	One query for one relation	Limited parallelism (left and right scan, next relation queried dependent on relation algebra expression defined for view)	Sequential	No	No
[LS99, SL00]	One query for one data source	Parallel (querying based on joined graph where independent paths are processed in parallel)	Parallel	Yes	Yes

Table 8: Comparison of efficiency of different compensation algorithms

Algorithm	View definition	Consistency	Quiescence requirement
ECA	Flexible	Strong	Yes
ECA^K	Keys of base relations need to be retained	Strong	Yes
Strobe	Keys of base relations need to be retained	Strong	Yes
C-Strobe	Keys of base relations need to be retained	Complete	No
[CM97a]	Limited to 2 base relations	Complete	No
SWEEP	Flexible	Complete	No
Nested SWEEP	Flexible	Strong	Yes
[CM97b]	Flexible	Complete	No
[LS99, SL00]	Flexible	Complete	No

Table 9: Comparison of application requirements of different compensation algorithms

From Table 8, we see that all algorithms that support multiple data sources environment, except the algorithm of [LS99, SL00], query one base relation at a time, although most data sources would contain more than a single base relation. Also, most algorithms handle the querying of the base relations by doing a left and a right scan of the base relations as defined by the relation algebra expression for the view. Only [LS99, SL00] provides for more parallelism in this process by accessing these base relations based on the join graph for the view. Also cartesian product would be avoided in this case. The incremental computation between different updates are either handled one at a time in a sequential manner, or multiple updates could be handled concurrently. However, only the algorithm of [LS99, SL00] are able to handle the concurrent incremental computation of different updates and achieve complete consistency at the same time. Partial self-maintenance issues are more fully exploited by [LS99, SL00] in doing the incremental computation, and it also handle modification as one type of update instead of treating it simply as deletion and insertion as in the other algorithms.

Table 9 shows that those algorithms that do not provide for flexible view definition require that the key of each base relation be retained in the view, while that of [CM97a] limit the number of base relations to two. Those methods that provide only strong consistency also requires that the system be quiescent before the view can be refreshed.

In conclusion, an ideal materialized view maintenance algorithm should be able to support an environment of multiple data sources. Compensation should preferably be carried out locally at the view site without the need to generate compensation queries to the data sources. Individual interfering updates should be correctly detected to achieve complete consistency. The algorithm should not assume that messages are always delivered in a first-sent-first-received manner over the network, or are never lost during transmission. Multiple base relations within the same data source should be queried together if they are related to each other through the join conditions, rather than to query one base relation at a time. Incremental computation between different updates and within the same update should be handled in parallel to make the process more efficient. Partial self-maintenance should be incorporated into the view maintenance process whenever feasible, and modification should not be simply treated as deletion and insertion in all cases. The view definition should be flexible. Algorithm that achieves complete consistency is more generic, and no quiescence requirement of the system should be necessary before the view can be refreshed. Finally, the criteria presented in this paper can be used to serve as a benchmarking tools for both existing and future materialized view maintenance algorithms to weigh their advantages and disadvantages.

References

[AASY97] D. Agrawal, A. El Abbadi, A. Singh, and T. Yurek. Efficient view maintenance at data warehouses. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 417–427, May 1997.

[CGL+96] Latha S. Colby, Timothy Griffin, Leonid Libkin, Inderpal Singh Mumick, and Howard Trickey. Algorithms for deferred view maintenance. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 469–480, June 1996.

[CM97a] Rongquen Chen and Weiyi Meng. Efficient view maintenance in a multidatabase environment. In *Proceedings of the Fifth International Conference on Database Systems for Advanced Applications*, pages 391–400, April 1997.

[CM97b] Rongquen Chen and Weiyi Meng. Precise detection and proper handling of view maintenance anomalies in a multidatabase environment. In *Second IFCIS Internation Conference on Cooperative Information Systems*, June 1997.

[GK98] Timothy Griffin and Bharat Kumar. Algebraic change propagation for semijoin and outerjoin queries. *SIGMOD Record*, 27(3), September 1998.

[GL95] Timothy Griffin and Leonid Libkin. Incremental maintenance of views with duplicates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 328–339, May 1995.

[GLT97] Timothy Griffin, Leonid Libkin, and Howard Trickey. An improved algorithm for the incremental recomputation of active relational expressions. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):508–511, May 1997.

[Huy96a] Nam Huyn. Efficient view self-maintenance. In *Proceedings of the ACM Workshop on Materialized Views: Techniques and Applications*, pages 17–25, June 1996.

[Huy96b] Nam Huyn. Efficient view self-maintenance. Technical report, Stanford University, 1996.

[Huy97a] Nam Huyn. Exploiting dependencies to enhance view self-maintainability. Technical report, Stanford University, 1997.

[Huy97b] Nam Huyn. Multiple-view self-maintenance in data warehousing environments. Technical report, Stanford University, 1997.

[LS99] Tok Wang Ling and Eng Koon Sze. Materialized view maintenance using version numbers. In *Proceedings of the Sixth International Conference on Database Systems for Advanced Applications*, pages 263–270, April 1999.

[QGMW96] Dallan Quass, Ashish Gupta, Inderpal Singh Mumick, and Jennifer Widom. Making views self-maintainable for data warehousing. In *Proceedings of the Conference on Parallel and Distributed Information Systems*, December 1996.

[Qua96] Dallan Quass. Maintenance expressions for views with aggregation. In *Proceedings of the ACM Workshop on Materialized Views: Techniques and Applications*, June 1996.

[QW91] Xiaolei Qian and Gio Wiederhold. Incremental recomputation of active relational expressions. *IEEE Transactions on Knowledge and Data Engineering*, 3(3):337–341, September 1991.

[SL00] Eng Koon Sze and Tok Wang Ling. Efficient view maintenance using version numbers. *Submitted for Review*, 2000.

[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 Conference on Parallel and Distributed Information Systems*, December 1996.

[ZGMW98] Yue Zhuge, Hector Garcia-Molina, and Janet L. Wiener. Consistency algorithms for multi-source warehouse view maintenance. *Journal of Distributed and Parallel Databases*, 6(1):7–40, January 1998.