A Model for Evaluating Materialized View Maintenance Algorithms

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

Abstract

Many algorithms have been proposed in the area of materialized view maintenance. They provide different capabilities and features, and have different complexity in their implementation. Each of these algorithms is suited for some types of applications. In this paper, we propose a model for evaluating the merit of these algorithms. This model would allow a user to choose an appropriate maintenance algorithm according to his application's needs. It can also be used as a benchmark where new maintenance algorithms can be evaluated.

1. Introduction

The rapid growth of internet technology, increase in processor speed, and drop in prices of storage devices in the recent years have propelled many new development and research areas in the database community. One of these is the integration of data from multiple data sources. With this integrated data, many useful information can be derived to allow the users to make better decisions. In this age where information is key to the survival of businesses, a lot of researches has been focused on the works of this data integration.

Unlike the case of a view in a single database, this integrated data reads in information from data sources which potentially could be far away from one another, since it is not uncommon for businesses to have operations at different parts of the world. On top of that, the base data is usually in the range of gigabytes or terabytes. Hence, using the traditional approach of a virtual view, where integrated data is produced *on-demand*, would not be appropriate in this aspect. Instead, the *in-advance* method, where base data is integrated and stored or materialized well before the users access it, is adopted.

As information stored at the data sources is not static and users require updated data to make better decisions, this integrated data must also be kept or *maintained* up-to-date. *Recomputing* the *materialized view*, or physical integrated data, provides an easy solution at the expense of efficiency. Understanding the fact that typically only a small percentage of the overall data changes, and coupled with the knowledge of the schema information of the integrated data called for an *incremental* approach to refresh this materialized view.

Numerous algorithms have been proposed to solve the materialized view maintenance problems. In this paper, we propose a model for evaluating the merits of each of these algorithms. This model also serves as a benchmark where new maintenance algorithms can be weighed upon.

In Section 2, we explain the general method of maintaining a materialized view. All incremental maintenance algorithms are based on this method, the main differences being in the execution of the incremental computation and the deployment of *compensation* algorithm. This compensation algorithm is necessary due to the *autonomous* nature of the data sources, resulting in view maintenance *anomalies* which we describe in Section 3. In Section 4, we give the model for evaluating the various algorithms. We conclude our work in Section 5.

2. View Maintenance

We first look at the general architecture of a data warehouse, before explaining the concept of incremental computation in maintaining the materialized view. We also explain two levels of consistency which are relevant to our discussion in Section 4.

2.1. Data Warehouse Model

Figure 1 shows the general architecture of a data warehouse. There are m data sources and a separate site for storing the materialized view. We consider the case of relational database where each data source can house one or more base relations, and select-project-join view V defined as in

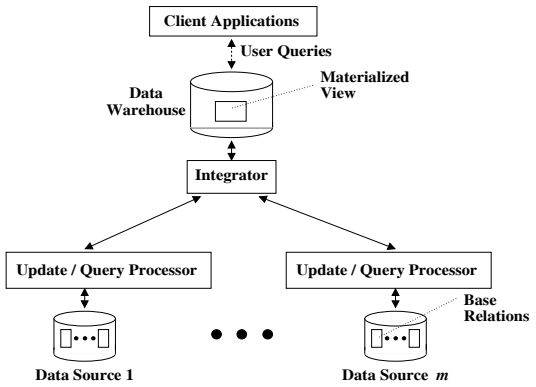

Figure 1. General architecture of a data warehouse

Equation 1 for n number of base relations. Each of these n base relations is found in only one of the data sources, but a data source can have more than one base relations. Thus, we have $n \geq m$. For the incremental view maintenance to work correctly, V also stores a *count* attribute that keep track of the number of duplicate tuples for the case where none of the keys of $R_1 \bowtie ... \bowtie R_n$ is found in V, i.e., they are completely or partially projected out.

$$V = \sigma_{sel\_cond} \prod_{proj\_attr} R_1 \bowtie ... \bowtie R_n \qquad (1)$$

Each update transaction at the data sources is communicated to the view site by sending an update notification message. The view site refresh its view accordingly by first issuing *view maintenance queries* to the various data sources to derive the affected view tuples, before using these queries' results to incrementally update the view.

We do not assume the data sources know the existence of other data sources, and transactions do not span across multiple data sources. Also data sources are autonomous in that the view site does not control the transactions at the data sources. As in any network communication, messages sent could be lost or arrive in a different order from that sent out. Messages here include update transaction notifications, view maintenance queries and their results.

2.2. Incremental Computation

Using the view definition given in Equation 1, let $\triangle R_i$, where $1 \leq i \leq n$, be a set of updates on a base relation of a transaction. The transaction can involve more than one base relation, of which the incremental computation will handle each separately. The purpose of incremental computation is to compute $\sigma_{sel\_cond} R_1 \bowtie ... \bowtie \triangle R_i \bowtie ... \bowtie R_n$ (to simplify the discussion, we assume that projection is carried out only during updating of the view). How the view maintenance queries are issued to handle this incremental computation is dependent on the compensation algorithms and we will discuss this when we look at the various works in Section 4.

For the case where one of the keys of $R_1 \bowtie ... \bowtie R_n$ is retained in V, (1) insertion update $\triangle R_i$ corresponds to adding $Ans = \prod_{proj\_attr} \sigma_{sel\_cond} R_1 \bowtie ... \bowtie \triangle R_i \bowtie ... \bowtie R_n$ to V, (2) deletion update $\triangle R_i$ requires the dropping of the tuples in Ans from V, and (3) modification update $\triangle R_i$ means updating the old values of the tuples in Ans to their respective modified values in Ans (assuming Ans stores both the before and after images of the modification).

If none of the keys of $R_1 \bowtie ... \bowtie R_n$ is found in V, then the above actions are similarly carried out, but with the corresponding updating of the count attribute due to the different ways the same view tuple can be derived from the base relations.

2.3. Levels of Consistency

We are interested in two of the levels of consistency of the view with respect to the updates at the data sources defined in [7].

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.

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.

3. View Maintenance Anomalies and Compensation

The incremental view maintenance approach outlined in the previous section overcomes the need to recompute the view whenever any of the base relations changes. However, the fact that multiple, autonomous data sources are involved means that this approach will not work correctly all the time due to the interfering effect of other concurrent updates.

3.1. Interfering Updates

Example Consider two base relations $R_1(\underline{A}, B)$ and $R_2(\underline{B}, C)$, with view $V = \prod_{BC} R_1 \bowtie R_2$. Initially, both relations are empty.

$R_1(\underline{A}, B)$	$R_2(\underline{B}, C)$	$V(B, C, count)$

A transaction update involving the insertion of tuple $(a1, b1)$ into R_1 occurs, and a notification message is sent to the view site. At about the same time, another transaction update involving the insertion of tuple $(b1, c1)$ into R_2 occurs. The view site first handles the incremental computation of the tuple (a1,b1), and the query of $\{(a1, b1)\} \bowtie R_2$ will return the result $\{(a1, b1, c1)\}$. The view is refreshed to give $\{(b1, c1, 1)\}$, where 1 is the value of the count attribute.

$R_1(\underline{A}, B)$	$R_2(\underline{B}, C)$	$V(B, C, count)$
$a1, b1$	$b1, c1$	$b1, c1, 1$

Next the view site proceeds to handle the incremental computation of $(b1, c1)$. The view maintenance query of $R_1 \bowtie \{(b1, c1)\}$ will give the result $\{(a1, b1, c1)\}$. Another tuple of $(b1, c1)$ will be added to the view relation to give $\{(b1, c1, 2)\}$.

$R_1(\underline{A}, B)$	$R_2(\underline{B}, C)$	$V(B, C, count)$
$a1, b1$	$b1, c1$	$b1, c1, 2$

The view is now not consistent with the two base relations. ■

The insertion update on R_2 is an *interfering* update on the incremental computation of insertion update on R_1, causing the above view maintenance *anomalies*.

3.2. Misordering of Messages

The process of removing the effect of interfering updates from the results of view maintenance queries is called *compensation*. Before the effect of these interfering updates can be resolved, their presence has to be detected first. One of the approaches to identify these interfering updates is to consider the order of the arrival of messages at the view site. In the scenario depicted in the previous subsection, since the update notification message of insertion on R_2 arrives at the view site in between the arrival of the update notification message of insertion R_1 and its result of view maintenance query, it is identified as the interfering update on this query result. Compensation handles this by undoing the effect caused by the insertion of the tuple $(b1, c1)$. Hence, it is deduced that $\{(a1, b1, c1)\}$ should not be found in the view maintenance query result of the insertion update on R_1. Thus, there is no change to the view relation with respect to this update on R_1. Only the insertion update on R_2 affects the view relation to give $\{(b1, c1, 1)\}$.

The above compensation process works well as long as messages are delivered in the same order as they have sent out. However, in the internet environment where congestion is frequent and multiple routes exist, such assumption does not hold and thus either the presence of interfering updates are not detected or non-interfering updates are wrongly identified as interfering updates. [3] shows that 1 percent of all messages are delivered and received in different orders when a local area network is heavily loaded.

3.3. Loss of Messages

The third problem is the loss of the messages sent between the data sources and the view site. Although the loss of network packets within a network connection can be handled and resolved at the network layer, this loss cannot be recovered between different connections. This happens when a connection is broken and another new connection has to be re-established. This is not an uncommon problem in the internet environment due to network congestion or hardware failure. The loss of messages will similarly cause the view to be inconsistent with the base relations when the effect of the updates are not incorporated into the view, or when compensation is not carried out because the view site is not aware of the existence of those updates.

4. Model for Materialized View Maintenance Algorithms

We have looked at the incremental computation approach in maintaining the materialized view and its associated problems. In this section, we propose a model for evaluating the merits of the existing works in this area based on a set of criteria. These works include ECA and ECA^K [6], the Strobe and C-Strobe Algorithm [7], the work of [2], SWEEP and Nested SWEEP Algorithm [1], the work of [3], the work of [4], and the work of [5]. We group the various criteria under four categories of *environment*, *correctness*, *efficiency* and *application requirements*. In the following subsections, we will discuss each of these in detail.

4.1. Environment

We define two criteria for evaluating the view maintenance algorithms under the environment category. They are (1) the number of data sources and (2) the handling of the compensation process.

Env(1). Number of data sources Some maintenance algorithms are meant for a single data source environment, while others are designed for a multiple data sources scenario. View maintenance algorithms in an environment of multiple data sources need to split the view maintenance query into multiple sub-queries for the various data sources. As such, such algorithms must be able to handle the presence of interfering updates in the intermediate results of

Criteria	Cases	Ideal Case
Number of data sources	Single	Multiple
	Multiple	
Compensation approaches	Compensation queries	Local compensation
	Local compensation	

Table 1. Environment

these sub-queries, whereas those algorithms meant for a single data source environment need only take care of the effect of interfering updates in the overall result. Thus, maintenance algorithms designed for multiple data sources can also be used in a single source environment but not vice versa.

ECA and ECA^K [6] are designed for single data source environment. The algorithm of [2] is limited to an environment of two base relations because it is designed mainly for a view that involves outerjoin. The other algorithms cater to multiple data sources.

Env(2). Compensation approaches The compensation of the results of view maintenance queries to remove the effect of interfering updates can be carried out in two ways. (1) First, it can be resolved by sending extra compensating queries on top of the view maintenance queries, and the view is refreshed by considering the results of both view maintenance and compensation queries. (2) The second method involves the resolving of the effect of the interfering updates locally at the view site by undoing the action caused by these updates. The second method is preferred over the first for the following reasons. The need for compensating queries means that more traffic has to be added to the netork, and since these compensating queries are also subjected to other interfering updates as well, more compensating queries have to be generated.

Example ECA issues compensating queries to resolve the view maintenance anomalies. Consider the scenario described in Section 3.1. When the view site receives the update notification message of the insertion update on R_2, it generates the view maintenance query $R_1 \bowtie \{(b1, c1)\}$. At this moment, since the previous maintenance query of $\{(a1, b1)\} \bowtie R_2$ is still outstanding, the view site generates a corresponding compensating queries of $\{(a1, b1)\} \bowtie \{(b1, c1)\}$. Thus, the overall effect of the second query is given by $[R_1 \bowtie \{(b1, c1)\}] - [\{(a1, b1)\} \bowtie \{(b1, c1)\}]$. This gives $\{\}$ and thus the view will not be changed with respect to this update. Only the view maintenance query result of insert $R_1(a1, b1)$ is applied to the view relation.

$R_1(\underline{A}, B)$	$R_2(\underline{B}, C)$	$V(B, C, count)$
$a1, b1$	$b1, c1$	$b1, c1, 1$

The view is consistent with the base relations. ■

In C-Strobe algorithm, compensation queries is required if the interfering update is a deletion update. Consider the view defined in Equation 1. If deletion update $\triangle R_j$ occurs during the incremental computation of update $\triangle R_i$, then compensating query $\sigma_{sel\_cond} R_1 \bowtie ... \bowtie \triangle R_i \bowtie ... \bowtie \triangle R_j \bowtie ... \bowtie R_n$ will be sent out, and its result added to that of the view maintenance query of $\triangle R_i$ to compensate those tuples that are missing due to the absence of $\triangle R_j$ in R_j.

The other algorithms handle the compensation process locally at the view site. ECA^K and Strobe (also C-Strobe) require that the key of each base relations be retained in the view, thus interfering insertion updates is resolved through duplicate tuples removal. Deletion can be handled without the need to issue any view maintenance queries. The rest of the algorithms handle the compensation by undoing directly the effect of the interfering updates. The next example shows how this is generally carried out.

Example Using the scenario of Section 3.1, the result $\{(a1, b1, c1)\}$ returned by the query $\{(a1, b1)\} \bowtie R_2$ should have the effect of the inserted tuple $(b1, c1)$ removed, giving $\{\}$. Thus the view is not affected by insert $R_1(a1, b1)$.

$R_1(\underline{A}, B)$	$R_2(\underline{B}, C)$	$V(B, C, count)$
$a1, b1$	$b1, c1$	

The view maintenance query result of insert $R_2(b1, c1)$ gives $\{(a1, b1, c1)\}$ and the tuple $(b1, c1, 1)$ is added to the view.

$R_1(\underline{A}, B)$	$R_2(\underline{B}, C)$	$V(B, C, count)$
$a1, b1$	$b1, c1$	$b1, c1, 1$

Similarly, in this example, the view is consistent with the base relations. ■

Table 1 summarizes the criteria under the environment category.

4.2. Correctness

Next we look the two criteria which we define under the correctness category. (1) First, the precise detection of the presence of interfering updates can affect the correctness of the maintenance algorithm. (2) The second criteria examines the correct working of the maintenance algorithms when messages are misordered or lost during transmission over the network.

Cor(1). Precise detection of interfering updates Before the compensation of the effect of the interfering updates on the results of view maintenance queries can be carried out, the presence of interfering updates have to be identified. Incorrect detection of these interfering updates can

Criteria	Cases	Ideal Case
Precise detection of interfering updates	Yes	Yes
	No	
Network communication assumption	Assume first-sent-first-received and non-loss	No assumption
	No assumption	

Table 2. Correctness

lead to the problem of view maintenance anomalies even if the compensation is done correctly.

The workings of ECA, ECA^K and Strobe Algorithms do not require the identification of interfering updates. As shown in the previous subsection, ECA compensates the result of the incremental computation of insertion update on R_2 although it is the query result of update on R_1 that has the view maintenance anomaly. Thus, refreshing of the view with respect to the query result of insertion on R_1 gives an inconsistent state. It is only after the next refresh that the view gives a consistent state, and the users are only presented with this state but not the earlier inconsistent state. Hence, such algorithms provide a weaker level of consistency for the view. It is possible for the C-Strobe Algorithm to identify non-interfering deletion updates as interfering updates. Nevertheless, the view can still be maintained correctly through duplicate tuples removal since the key of each base relation is retained in the view, but at the expense of sending unnecessary compensating queries for this non-interfering deletion update.

The rest of the algorithms correctly identify the presence of interfering updates. Given that update $\triangle R_j$ occurs after update $\triangle R_i$ (how the updates are ordered is dependent on the algorithms used), where $i \neq j$, $\triangle R_j$ is an interfering update on the result of incremental computation of $\triangle R_i$ if $\triangle R_j$ occurs before the view maintenance sub-query of $\triangle R_i$ is processed by R_j.

Cor(2). Network communication assumption The workings of ECA, ECA^K, Strobe, C-Strobe, [2], SWEEP, and Nested SWEEP assume that messages sent through the network are delivered in the same order to the destination as they are sent out, and that these messages are never lost. These maintenance algorithms will not refresh the view correctly when these assumptions are violated. [3] is able to maintain the view correctly when messages are misordered as they are delivered through the network. [4] and [5] work correctly when either messages are misordered or lost during tranmission through the network. They achieve this by using version numbers to order the transactions, instead of depending it on the order of arrival of messages at the view site.

Table 2 summarizes the criteria under the correctness category.

4.3. Efficiency

We identify 5 criteria under the efficiency category. They are (1) the number of base relations accessed per view maintenance sub-query, (2) sequential or parallel processing of the incremental computation of an update, (3) sequential or parallel processing of incremental computation between different updates, (4) the use of partial self-maintenance, and (5) the handling of modification update.

Criteria	Cases	Ideal Cases
Number of base relations accessed per sub-query	One	Multiple
	Multiple	
Incremental computation of an update	Sequential	Parallel
	Parallel	
Incremental computation between updates	Sequential	Parallel
	Parallel	
Use of partial self-maintenance	Yes	Yes
	No	
Handling of modification	Deletion and insertion	Consider as one type of update
	Consider as one type of update	

Table 3. Efficiency

Eff(1). Number of base relations accessed per sub-query For a data source with multiple base relations involved in the view definition, it is not efficient to query one base relation at a time for the incremental computation. More network traffic has to be generated if the maintenance sub-query only accesses one base relation at a time. The extra round-trip time incurred will make the overall delay time for the incremental computation of an update longer, and this translates to higher number of interfering updates for this incremental computation, resulting in more processing for the compensation stage. A better approach would be to query multiple base relations residing at the same data source together.

Eff(2). Handling of incremental computation of an update The view maintenance and compensation queries of ECA and ECA^K are issued to all the base relations since only one data source is involved and compensation is not

processed locally at the view site (although the compensation queries are issued by the view site). The maintenance processing of Strobe, C-Strobe, SWEEP, Nested SWEEP, [3], and [4] query one base relation at a time. Since [2] deals only with two base relations, its view maintenance query needs only be applied to one base relation. [5] is able to access multiple base relations from the same data source within a single view maintenance sub-query. This is possible because of two reasons. Firstly, version numbers are used to determine the actual sequence of actions at the data sources instead of the order of delivery. And secondly, by defining a compensation algorithm that does not require the immediate resolving of interfering updates.

Example Consider a view defined by $V = \prod_D R_1 \bowtie R_2 \bowtie R_3$, where $R_1(\underline{A}, B)$, $R_2(\underline{B}, C)$ and $R_3(\underline{C}, D)$. Given that R_1 resides in data source 1, and R_2 and R_3 reside in data source 2. Initially, R_1 has one tuple $(a1, b1)$, R_2 has one tuple $(b1, c1)$, R_3 has one tuple $(c1, d1)$, and thus the view has one tuple $(d1, 1)$ with 1 as the value of the count attribute.

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

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

$R_3(\underline{C}, D)$
$c1, d1$

$V(D, Count)$
d1,1

The view site receives the notification update of delete $R_1(a1, b1)$, followed by $R_2(b1, c1)$. The view maintenance query for delete $R_1(a1, b1)$ is $\{(a1, b1)\} \bowtie R_2 \bowtie R_3$. Those algorithms that access one base relation at a time will first send the query $\{(a1, b1)\} \bowtie R_2$ to data source 2 first (for simplicity, we use natural join in the discussion, but note that semijoin could be employed to cut down the size of the messages). An empty intermediate result is returned. Compensation acts on this empty result to give the tuple $\{(a1, b1, c1)\}$. Next, $\{(b1, c1)\} \bowtie R_3$ is sent to data source 2 again, and the intermediate result $\{(b1, c1, d1)\}$ is returned. This gives the overall result of $\{(a1, b1, c1, d1)\}$. Thus, the tuple $(d1, 1)$ is removed from V.

$R_1(\underline{A}, B)$

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

$R_3(\underline{C}, D)$
$c1, d1$

$V(D, Count)$

The query result of delete $R_2(b1, c1)$ gives $\{\} \bowtie \{(b1, c1)\} \bowtie \{(c1, d1)\}$ and thus there is no change to the view. In the case of [5], only one query $\{(a1, b1)\} \bowtie R_2 \bowtie R_3$ is needed to be sent to data source 2 for delete $R_1(a1, b1)$. Again, an empty result is returned to the view site. The algorithm of [5] is able to detect the presence of interfering deletion update $R_2(b1, c1)$ in this result by checking its expected version numbers and the actual version numbers. It is also able to compensate this result to give $\{(a1, b1, c1, d1)\}$, by combining $\{(a1, b1)\} \bowtie \{\} \bowtie \{\}$ with $\{\} \bowtie \{(b1, c1)\} \bowtie \{(c1, d1)\}$, the view maintenance query result of delete $R_2(b1, c1)$. ■

As multiple base relations and data sources are involved, the single view maintenance query for the incremental computation of one update has to be broken down into many sub-queries. This sub-queries can be issued one at a time, i.e., sequentially. Alternatively, parallelism could be employed by accessing multiple base relations or data sources concurrently. Although the sequential approach is easy to implement, it takes longer time to process than the parallel counterpart.

As ECA and ECA^K deal with only one data source, and [2] is limited to two base relations, this criteria is not applicable in such algorithms. Strobe and C-Strobe access one base relation at a time, while the parallelism approach of SWEEP, Nested SWEEP, [3] and [4] are limited to a left scan and a right scan. [5] uses the join graph to determine the order of querying the base relations. Multiple path could be exploited in accessing the base relations, instead of limiting it to two, and cartesian product is avoided which otherwise can be disastrous.

Eff(3). Handling of incremental computation between updates The incremental computation between different updates can also be handled sequentially or in a parallel manner. Handling the incremental computation in a parallel manner requires a more sophiscated algorithm for the proper detection of interfering updates, but is more efficient when the transmission time of messages between a data source and the view site is much longer than local processing time. Less delay will be incurred before the view can be refreshed.

ECA, ECA^K and Strobe handle the incremental computation of multiple updates in parallel. However, the workings of these algorithms do not require the detection of individual interfering updates. Results of the incremental computation of the different updates that are processed in parallel are combined to give a weaker consistent state for the view. C-Strobe, [2], the SWEEP Algorithm and [3] handle the incremental computation of each update sequentially for ease of keeping track of the interfering updates. In Nested SWEEP, incremental computation of different updates are handled in parallel by recognizing that interfering updates share some intermediate results. However, a weaker level of consistency can only be achieved by this algorithm. [4] and [5] process the incremental computation of different updates in parallel. This is achieved by implementing version number counters at the data sources, and by tagging information of expected and actual version numbers with the view maintenance query results. This does not add much burden to the materialized view relation as the query results

will be discarded once the view has been refreshed.

Eff(4). Use of partial self-maintenance *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, *full 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. While self-maintenance is an intentional view design consideration to do away with the need to depend on the data sources, partial self-maintenance is employed to minimize this querying whenever possible, through making use of the existing information provided by the view. Thus, the use of partial self-maintenance can cut down the number or size of messages sent through the network and can be especially useful when communication cost is high.

ECA, SWEEP, Nested SWEEP, [3] and [4] do not consider partial self-maintenance issues. ECA^K, Strobe and C-Strobe can maintain the view without the need to issue any view maintenance query for deletion updates because of the retainment of the key of each base relations in the view relation. [2] is able to identify certain updates that will not affect the view, and thus no incremental computation is required for such updates, through the keeping of a system catalog that records the number of tuples in each of the base relations with each of the join attribute value. [5] provides more opportunities for partial self-maintenance by taking key constraints and functional dependencies of the view attributes into consideration.

Eff(5). Handling of modification Treating modification as a deletion followed by an insertion update simplifies the view maintenance process at the expense of a less efficient algorithm. It is possible for a modification of a single tuple from a base relation to be translated into multiple tuples for the joined relation, and unnecessary work might also need to be carried out in rebuilding the view indexes although the modification might not be on the attributes involved in the indexes.

Except for [4] and [5] that consider modification as one type of update other than deletion and insertion, the rest of the algorithms simply treat it as a deletion and an insertion updates.

Table 3 summarizes the criteria under the efficiency category.

4.4. Application Requirements

In the application requirements category, the criteria we are looking at are (1) the flexibility of the view definition, (2) the level of consistency achieved by the maintenance algorithms, and (3) the need for quiescent state.

App(1). Flexibility of view definition A flexible view definition would provide more freedom in designing the view, while a more restrictive one might need further processing during querying to give what the users require. ECA^K, Strobe and C-Strobe require that the key of each base relation be retained in the view. Since [2] is intended for an outerjoin view, the number of base relations is limited to two. The other algorithms do not impose any restriction on the view definition.

Criteria	Cases	Ideal Case
Flexibility of view definition	Flexible	Flexible
	Not flexible	
Consistency level	Complete	Complete
	Strong	
Quiescence requirement	Yes	No
	No	

Table 4. Application Requirements

App(2). Consistency level A view maintenance algorithm that gives complete consistency can be used for an application that only requires strong consistency but not vice versa. ECA, ECA^K, Strobe and Nested SWEEP Algorithms achieve strong consistency while the rest provide complete consistency.

App(3). Quiescence requirement If a view maintenance algorithm needs a quiescent state in the system before the view can be refreshed, then a continuous stream of update notifications arriving at the view site will cause delay in the refreshing of the view. ECA, ECA^K, Strobe and Nested SWEEP Algorithms require a quiescent state in the system before the view can be updated, while the other algorithms do not.

Table 4 summarizes the criteria under the application requirements category.

5. Conclusion

In this paper, we proposed a model for evaluating materialized view maintenance algorithms. Various criteria are used in our model to weigh the merits of these algorithms, and they are grouped under four main categories. (I) Under the environment category, we look at (Env(1)) the number of data sources handled by the algorithms, and whether (Env(2)) the compensation process is handled locally at the view site or requires the sending of compensating queries. (II) For the correctness category, we are interested in whether (Cor(1)) the detection of the interfering updates is precise, and (Cor(2)) the correctness of the algorithms when messages transmitted through the network are misordered or lost. (III) The efficiency category exam-

ines (Eff(1)) the number of base relations accessed per view maintenance sub-query, the sequential or parallel handling of incremental computation within (Eff(2)) the same update and (Eff(3)) between different updates, (Eff(4)) the use of partial self-maintenance, as well as (Eff(5)) the handling of modification updates. (IV) In the application requirements category, (App(1)) the flexibility of the view definition, (App(2)) levels of consistency provided and (App(3)) the need for quiescent state are evaluated.

To summarize, the ideal algorithms should be able to handle multiple data sources, resolve compensation locally at the view site, identify accurately the presence of interfering updates, do not assume that messages are never misordered or lost, access multiple base relations within the same data source together for a view maintenance sub-query, parallel processing of incremental computation within the same update and between updates, use of partial self-maintenance, treat modification update as one type of update, flexibility of view definition, achieve complete consistency and no quiescence requirement.

References

[1] D. Agrawal, A. E. 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.

[2] R. Chen and W. 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.

[3] R. Chen and W. Meng. Precise detection and proper handling of view maintenance anomalies in a multidatabase environment. In *Proceedings of the Second International Conference on Cooperative Information Systems*, June 1997.

[4] T. W. Ling and E. K. 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.

[5] E. K. Sze and T. W. Ling. Efficient view maintenance using version numbers. Technical report, TRA2/00, School of Computing, National University of Singapore, 2000.

[6] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 316–327, May 1995.

[7] Y. Zhuge, H. Garcia-Molina, and J. L. Wiener. The strobe algorithms for multi-source warehouse consistency. In *Proceedings of the Conference on Parallel and Distributed Information Systems*, December 1996.