Efficient View Maintenance Using Version Numbers

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

Abstract

Maintaining a materialized view in an environment of multiple, distributed, autonomous data sources is a challenging issue. Given that the view site does not control the transactions at the data sources, results of incremental computation is affected by interfering updates and compensation process is required to resolve such problem. The compensation algorithm in our previous work handles the resolving of interfering updates without making any assumption on the first-sent-first-received and non-lost delivery of messages over the communication network.

In this paper, we improve the incremental computation process by cutting down unnecessary maintenance queries and thus their corresponding query results, through using the knowledge of referential integrity constraints and functional dependencies of the base relations. We reduce the number of times of sending sub-queries to a data source with multiple base relations, as well as avoiding the execution of cartesian products, by using the join graph of the view to determine the access path of querying these base relations. We also provide a compensation algorithm that makes the accessing of multiple base relations from the same data source within a single sub-query possible. Overall performance is improved by cutting down the sending of unnecessary view maintenance queries and results, sizes of results are reduced drastically as cartesian products are avoided, as well as the number of times of issuing sub-queries to each data source.

1 Introduction

Providing integrated access to information from different data sources, of which each could be located at site of different areas, has received recent interest from both the industries and research communities. Analysis can then be made on this integrated data to provide useful information for the business concerned. Two methods to this data integration are the *on-demand* and the *in-advance* approaches.

In the on-demand approach, information is gathered and integrated from the various data sources only when requested by users. This is suitable when it is not possible to predict beforehand what is the required information. The disadvantage of this method is the delay in generating the information to the user as the data involved are usually large.

To provide fast access to the integrated information, the in-advance approach is preferred instead. Information is extracted and integrated from the data sources, and then stored in a central site as a materialized view. Users access this materialized view directly, and thus queries are answered immediately as compared to the on-demand approach. Noting that information at the data sources do get updated as time progresses, this materialized view will have to be *refreshed* accordingly to be consistent with the data sources. This refreshing of the materialized view due to changes at the data sources is called *materialized view maintenance*. The view can be refreshed either by recomputing the integrated information from scratch, or through incrementally changing only the affected portion of the view.

Recomputing the view from scratch is inefficient due to the size of the data involved, as well as the fact that the site housing the view and the data sources could be located far away from one another. It is also not feasible to provide up-to-date information because doing so requires frequent view recomputation. The deriving of the relevant portion of the view and then incrementally changing it is a preferred approach as a smaller set of data are involved. It is then possible to allow for a smaller differences between the state of the view with that of the data sources. Applications such as credit card billing or stock market software which require up-to-date information would require such method to maintain the view.

Many incremental materialized view maintenance algorithms [CGL$^+$96, GL95, GLT97, GK98, QW91, Qua96] have been developed for centralized database systems. In such systems, problems of interfering updates on the results of incremental computation do not occur, and the main consideration is to efficiently compute the incremental changes rather than to recompute the view from scratch in response to the updates at the data sources. [GL95, GLT97, QW91] deal with immediate approach to view maintenance, where the changes in the base relations are immediately propagated to the view relation, while [CGL$^+$96] handles the deferred approach to view maintenance. [GK98] extends the works on the immediate approach [GL95, GLT97, QW91], which cover the common operators, to include the semijoin and the outerjoin operators. [Qua96] covers the environment where aggregation is involved.

The view maintenance in an environment of distributed, autonomous data sources is considered in [ZGMHW95, ZGMW96, AASY97, CM97b, LS99]. Issues of view self-maintenance for a single view are discussed in [Huy96b, Huy96c, Huy96a, Huy97a]. [Huy97b] considers the situation of multple-view self-maintenance, and thus provide more opportunities for self-maintenance as compared to treating individual view separately.

In Section 2, we explain the concept of incremental computation used in maintaining the materialized view, and its associated problems. We propose our solution to these problems in Sections 3, 4 and 5. We compare related works with our approach in Section 6, and conclude our discussion in Section 7.

2 Background

In this section, the idea behind the *incremental computation* approach to incrementally refresh the materialized view and the levels of consistency of maintaining this view is discussed. We also discuss the problems affecting this incremental computation approach, which results in the *view maintenance anomalies.*

2.1 Incremental Computation

We consider the scenario of select-project-join view V, with n numbers of base relations $\{R_i\}_{1 \leq i \leq n}$, with each base relation housed in one of the data sources and a separate site for the view. The view definition is given in Equation 1. A *count* attribute is appended to the view attributes if it does not contain the key of the join relations $R_1 \bowtie ... \bowtie R_n$ to indicate the number of ways the same view tuple could be derived from the base relations. There are multiple, distributed, autonomous data sources, each with one or more base relations. There is communication between the view site and the data sources, but no assumption is made with regard to the communication between individual data sources. Thus a transaction can involve one or more base relations of the same data source, but not between different data sources. The view site does not control the transactions at the data sources. No assumption is made regarding the reliability of the network, i.e., messages sent could be lost or could arrive at the destination in a different order from what was originally sent out.

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

The data sources will send notifications to the view site on the updates that have occurred. To incrementally refresh the view with respect to an update $\triangle R_i$ of base relation R_i requires the computation of $R_1 \bowtie ... \bowtie \triangle R_i \bowtie ... \bowtie R_n$ through the issuing of sub-queries to the data sources concerned. The result is then applied to the view relation by either inserting, deleting or modifying the relevant view tuples, or the count attribute of the affected tuples.

Existing methods handle the incremental computation by executing 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 results 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 as $\mu R_1 \bowtie ... \bowtie \mu R_n$, and this gives the incremental change for the view relation. Drawbacks of using this approach include (1) high possibility of executing a cartesian product, (2) parallelism is limited to a left and a right scan, and (3) only one base relation is accessed at any one time in either one of the scan despite the fact that a data source usually has more than one relation. However, this approach of querying the base relations is needed by most of the compensation algorithms [ZGMW96, AASY97, CM97a, CM97b] to function, with the exception of [ZGMHW95] since it only deals only with one data source of multiple base relations. Our proposed solution does not have such requirement and thus multiple base relations residing at the same data source can be accessed within the same query.

2.2 Levels of Consistency

Four levels of consistency of the view with respect to the updates at the data sources are defined in [ZGMW96]. We look at two of these levels which will be used in our subsequent discussion.

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 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 is also in strong consistency with a data source.

2.3 View Maintenance Anomalies and Compensation

There are three problems in using the incremental approach to maintain the view. They are (1) the presence of interfering updates in the query results of incremental computation, (2) the misordering of messages in the communication network, and (3) the lost of these messages in the network. In the rest of this subsection, we explain how these three problems affect the view maintenance process.

2.3.1 Interfering Updates

If updates are separated from one another by a sufficient large amount of time, incremental computation will not be affected by interfering updates and the view would be maintained correctly. The following example shows the maintenance of view without 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 attributed 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 (a2,b1) $\rtimes R_2$ is formulated for the incremental computation. Since R_2 is empty, an empty results is returned and there is no change to the view. Suppose another insertion update of R_2(b1,c1) occurs and now the view sent the following query $R_1 \ltimes$ (b1,c1) for the incremental computation. The tuples (a1,b1) and (a2,b1) are returned from R_1. Therefore, the overall results (without projection) is {(a1,b1,c1),(a2,b1,c1)}. Projecting this over the view attributes add two tuples of (c1) to the view relation, and this is indicated by the count value of 2 in the view relation. The final base and view relations are as follow.

$R_1(\underline{A}, B)$	$R_2(\underline{B}, C)$	$V(C, count)$
a1,b1	b1,c1	c1,2
a2,b1		

■

Since we are dealing with an environment of autonomous data sources, there is no way to enforce the above constraint where updates do not interfere with each others' incremental computation. This problem was first identified by [ZGMHW95]. The next example shows the problem in maintaining the view due to the presence of interfering updates, when another update from the other base relation occurred before the completion of the incremental computation of the earlier update.

Example 2 Consider the same base relations and view as in Example 1. However, now we let the insertion update R_2(b1,c1) occurs just before the view maintanance query for insertion update R_1(a2,b1) reach the relation R_2. The tuple (b1,c1) would be returned as the answer for the incremental computation. The overall result (without projection) is the single tuple (a2,b1,c1). The single projected tuple (c1) is added to the view to give (c1,1).

$V(C, count)$
c1,1

The result of the second view maintenance query is the same as in Example 1, and this adds two projected tuples of (c1) to give (c1,3). The final base and view relations are as follow.

$R_1(\underline{A}, B)$	$R_2(\underline{B}, C)$	$V(C, count)$
a1,b1	b1,c1	c1,3
a2,b1		

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. ■

2.3.2 Misordering of Messages

Compensation is the removal of the effect of interfering updates from the query results of incremental computation. Most of the existing compensation algorithms that remove the effect of interfering updates and thus achieve complete consistency are based on the first-sent-first-received delivery assumption of the messages over the network, and thus will not work correctly when messages are misordered. They include the method by [CM97a] and the SWEEP Algorithm of [AASY97]. A studies carried out by [CM97b] has shown that 1 percent of the messages delivered over the network are misordered. 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. The ECA Algorithm of [ZGMHW95] and the Strobe Algorithm of [ZGMW96], which handles the compensation by issuing compensating queries and duplicate tuples removal respectively, and thus only achieve strong consistency, are also based on the first-sent-first-received assumption. These algorithms do not maintain the view correctly when messages are misordered.

Example 3 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 for the incremental computation of update R_1(a2,b1). Thus, we know that the tuple (b1,c1) in the query result 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 network congestion for example, then using this method of identifying interfering updates will fail to treat R_2(b1,c1) as an interfering update. ■

2.3.3 Lost Messages

The third problem is the lost of messages. Although the lost of network packets can be detected and resolved at the network layer, the lost of messages due to the disconnection of the network link (machine reboot, network failure, etc .) have to be resolved by the application itself after the re-establishing of the link. The lost of the view maintenance queries messages or its corresponding results can be detected by the view site by setting a timer whenever such queries are issued. However, the lost of an update notification messages from the data sources 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 4 Using the same scenario as in Example 2, 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 results of update R_1(a2,b1) is carried out. The overall results 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 tuple, and insertion R_2(b1,c1) should have placed the tuple (c1,2) in the view relation. ■

3 Version Numbers

In our previous work [LS99], we propose the use of version numbers and a compensation algorithm to handle the problem of interfering updates that maintain the view correctly when messages are misordered or lost in the communication network. In this paper, we improve on this algorithm by cutting down the number of the view maintenance queries and reducing the sizes of these maintenance query results significantly. This improvement is the result of (1) involving the view relation in the maintenance process using the knowledge provided by the referential integrity constraints and functional dependencies of the attributes, (2) combining the querying of the multiple base relations within the same data source to a single sub-query to reduce both the size of the results and the overall round-trip time per view maintenance query, and (3) avoiding of cartesian products. The improvement is done by extending the concept of version numbers, as well as the working of the compensation algorithm, and using the information provided by the referential integrity constraints, functional dependencies and the join graph of the view. In this section, we discuss our concept of version numbers which will be used in the incremental computation of updates (Section 4) and their corresponding compensation (Section 5).

This idea of version numbers is proposed in [LS99], and we further extend it in this paper to allow for *partial self-maintenance.* Instead of using solely the base relations for the incremental computation, the use of the view relation (since the view contains information of the base relations) in the maintenance process to cut down the number and size of view maintenance queries is called partial self-maintenance. Self-maintenance is the extreme case of using only the view relation and the updates (and maybe some auxiliary views) to maintain the view, without the need to query the base relations stored at the data source. This comes with the expense of providing extra storage space at the view site for the auxiliary views or the duplicated base relations, which can be quite significant, as well as the need to maintain such relations in addition to the view.

We use the concept of version numbers to identify the states of a data source, as well as its base relations (a data source can have one or more base relations), instead of depending on the order of arrival of messages at the view site. The use of these version numbers also allow for the concurrent handling of the incremental computation of multiple updates, rather than processing them one at a time, as in the case of [CM97a, AASY97, CM97b]. Also, with the use of these version numbers, multiple base relations involved in the view within the same data source can be queried together within the same sub-query of the incremental computation, instead of accessing one base relation at a time. This reduces the overall computation time and the network traffic.

The compensation process simply look at these version numbers to identify the interfering updates on the query result of the incremental computation of an update, and then proceed to undo the effect of these interfering updates. For instance, if the incremental computation of an update is supposed to access the base relations of some version numbers (some particular states), and the view maintenance query result is generated by the base relations of some other version numbers (some other states), then those updates that cause the differences in the version numbers (states) of the base relations are the interfering updates whose effect has to be removed.

We define the following types of version numbers and their functions or uses in the working of the incremental computation and compensation of the view maintenance query results of the updates. Points (1) to (5) have been introduced in [LS99], and point (6) is proposed in this paper to support partial self-maintenance.

(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 results.

(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, we define another level of version number 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 is 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 us to handle the incremental computation of update transactions according to the order that they have occurred, instead of their order of arrival at the view site which is network traffic dependent. 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 has 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. For an update transaction involving multiple base relations, the incremental computation of the updates

from the individual base relations have to be sent for incremental computation one after another, and should be not interlinked with updates from another transaction. 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 we do not consider the case of global transaction. 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, where each state of a base relation is identified by its base relation version number, is called the initial version numbers of this update. The highest processing version numbers are assigned to an update as its initial version numbers when it is sent for incremental computation. 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 result is generated by the base relations with different base relation version numbers from the initial version numbers of this update. The base relation version number of a relation where the result is generated is called the queried version number and this is returned together with the query result to the view site. We associate this queried version number of the base relation 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

approach 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. In the case of querying the base relations in the usual incremental computation approach, compensation is necessary to handle the effect of interfering updates, identified through the use of queried version numbers taken from the base relation version numbers where the result tuples are generated. Similarly, there would be problem of interfering updates when the view relation is used in the incremental computation. 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 initial 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 number of a base relation may be lower than the initial version number of the same relation for the incremental computation of an update, because the result 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. Thus, the refreshed version number is analogus to the base relation version number of a base relation when it is being queried.

In summary, each data source will have a data source version number, and a base relation version number for each of the base relation that it stored. The view site will have n highest processing version numbers and n refreshed version numbers for the n base relations that are involved in the view relation. The initial version numbers associated with the incremental computation of an update, as well as the queried version numbers of the tuples in its result are only needed during the processing of its incremental computation. These will be discarded together with the result when the view is refreshed, and thus do not pose a storage problem as they do not persist in the system. Example 6 in the next section will show the use of these version numbers in the view maintenance process.

4 Improved Incremental Computation

The concepts of version numbers discussed in the previous section, besides being used for identifying the interfering updates instead of the first-sent-first-received approach as in other existing algorithms, it also allows for more efficient incremental computation. In this paper, we propose a more efficient way of doing the incremental computation. This improvement is made possible because our compensation algorithm does not use the order of delivery of messages to identify interfering updates. Due to space constraint, we are not able to go into detail the full workings of this improved incremental computation. However, what we would like to demonstrate is how can this use of version numbers open up the opportunity for better view maintenance processing as compared to the approaches in existing methods.

We first explain the general approach of the incremental computation applicable to all updates, before introducing specific approaches used for insertion, modification and deletion updates.

4.1 General

Existing view maintenance methods [AASY97, CM97b] can only access **one** base relation per sub-query (of the view maintenance query), despite of the fact that a data source usually has more than one base relation that are involved in the view definition. The algorithm of [AASY97] uses the first-sent-first-received assumption of delivery of messages through the network in the compensation algorithm, thus it is not easy to generalize it to accessing multiple base relations within a single view maintenance sub-query and still be able to detect interfering updates. The disadvantages of accessing one base relation at a time are (1) the extra network traffic generated by the many sub-queries and their corresponding results, (2) longer time required to complete the incremental computation for each update because the total round-trip time is longer in comparison to the case where multiple base relations in the data source are queried together, (3) larger number of tuples are returned in the results since all the join operations have to be performed at the view site, as compared to the case of querying multiple base relations together in which case some of these join operations can be performed at the data sources.

Although [CM97b] does not assume the first-sent-first-received delivery of messages, it is still not able to access multiple base relations in a single sub-query because of its limited use of version numbers (called counters in [CM97b]). We overcome this problem by temporarily storing the state of the base relation (queried version number) where the query result is generated.

Since our algorithm (Section 5) does not detect the interfering updates base on the order of arrival of messages, and we store extra information temporarily in the form of queried version numbers with the view maintenance query results, our algorithm is able to access **multiple** base relations residing at the same data source with a single sub-query. This queried version number is associated with each tuple of the query results, and require only a small amount of extra temporary storage space. This extra storage space does not persist as it is not required once the results of the incremental computation is applied to the view. To avoid cartesian product, which would otherwise generates large volume of unnecessary data, we limit such access within the single sub-query to only those relations that have join attributes in common.

Briefly, the incremental computation is handled as such. Consider the join graph of the base relations of a view, the view maintenance query starts with the base relation R_i, $1 \leq i \leq n$, where the update has occurred. A sub-query is sent to a set of relations S, where $S \subset \{R_j\}_{1 \leq j \leq n}$, the relations in S comes from the same data source, R_i and the relations in S form a connected sub-graph with R_i as the root of this sub-graph. If there are more than one such set of base relations S, multiple sub-queries can be sent out in parallel. For each sub-query sent, we marked the relations in S as "queried". R_i is also marked as "queried". Whenever a result is returned from a data source, another sub-query is generated using the similar approach. Let R_k, $1 \leq k \leq n$, be one of the relations that have been queried, a sub-query is sent to a set of relations S, where $S \subset \{R_j\}_{1 \leq j \leq n}$, the relations in S comes from the same data source, none of the relations in S are marked "queried", R_k and the relations in S form a connected sub-graph with R_k as the root of this sub-graph. Again, if there are more than one such set of base relations S, multiple sub-queries can be sent in parallel. The incremantal computation for this update is completed when all the base relations are marked "queried".

If the view definition is such that there are cartesian products, i.e., the join graph has two or more disconnected sub-graphs, sub-queries have to be sent to the base relations in the other disconnected sub-graphs even though there is no link between these relations. However, note that these sub-queries between disconnected sub-graphs need only be sent when the view is empty at the initial stage. Since there are cartesian products between disconnected sub-graphs, the records in the view can be used in the incremental computation when it is not empty. This is done by doubling the number of records from the base relations in the other disconnected sub-graphs for each tuple of the query result added from the base relations in the sub-graph where the update has occurred.

Example 5 Referring to Figure 1 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 of 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. Let A_2 and A_3 be the query results of Q_2 and Q_3 respectively, and assuming that the necessary compensation has been applied, the overall incremental change is given by $A_1 \bowtie A_2 \bowtie A_3$.

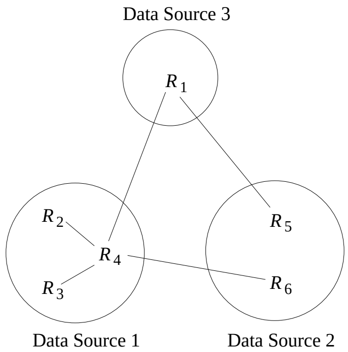

Figure 1: Join graph of base relations for view $V = \bowtie_{i=1}^{6} R_i$

Comparing this approach with that of doing a left and a right scan, with one relation accessed for each sub-query, our approach only needs to issue 3 sub-queries instead of 5. Assuming that the transfer time between any 2 sites to be 1 unit, then 6 units of time will be elapsed before the final result is returned to the view site instead of 10. This not only reduces the maintenance time per update, it also makes the compensation process more efficient since the faster it takes to do the incremental computation, the less interfering updates there are.

On top of that, using the join graph to determine the access path of querying the base relations, instead of doing a left and a right scans of the relations based on their arrangement in the relation algebra of the view definition, cuts down the size of the query results sent through the network by avoiding cartesian products. For instance, if the arrangement of the 6 relations is $R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4 \bowtie R_5 \bowtie R_6$, then the method of left and right scans of the relations would result in 2 cartesian products (between R_1 and R_2, and between R_2 and R_3) for an update in R_1. Assuming that each base relation has 1,000,000 of records, with an average record size of 100 bytes, 100MB of data would be retrieved from R_2, and another 100MB of data would be retrieved from R_3, although only a few of these records would be inside the final result of the incremental change. This creates unnecessary burden on the network and extra processing for the view site. This problem is made worse if the view maintenance algorithm computes the intermediate result of the incremental change each time a result is returned. In this case, the view site would compute the cartesian product between R_2 and R_3 to give a result of 200TB data. Note that rearranging the base relations does not help in this example (as well as many cases) because R_4 is joined to more than 2 base relations, thus a specific arrangement might avoid cartesian product for update of some base relations, but not every relation.

■

4.2 Insertion

We describe how the information of referential integrity constraint is used to eliminate irrelevant updates, i.e., updates that does not affect the view. We give a brief idea on how functional dependencies is used to involve the view in the incremental computation.

If a data source enforces the referential integrity constraint that each tuple in R_j must refer to a tuple in R_i, then we know that an insertion update on R_i will not affect the view and thus can be ignored by our view maintenance process.

The use of the view relation in the partial self-maintenance of the incremental computation of an update is as follows. When the incremental computation of an update needs to query base relation R_j, (1) if the key of R_j is found in the view, and (2) if R_j functionally determines the rest of the relations that are to be queried based on the query result of R_j, then the view can be accessed for this incremental computation and its refreshed version numbers are taken as the queried version numbers for the result. When both conditions are satisfied, the view is first used for the incremental computation. If the required tuples are not found in the view, the base relations are next accessed. This cuts down the number of queries as compared to using purely the base relations for the view maintenance process. Example 6 shows the use of the view for incremental computation, i.e., partial self-maintenance.

Example 6 Let the three base relations be $R_1(\underline{A}, B, C)$, $R_2(\underline{C}, D, E)$ and $R_3(\underline{E}, F, G)$, with the view defined as $V = \prod_{B,C,F} R_1 \bowtie R_2 \bowtie R_3$. Again, the count attribute is required for the correct working of the maintenance algorithm. The following shows the initial states of the base and view relations.

$R_1(\underline{A}, B, C)$	$R_2(\underline{C}, D, E)$	$R_3(\underline{E}, F, G)$	$V(B, C, F, count)$
a1,b1,c1	c1,d1,e1	e1,f1,g1	b1,c1,f1,1
	c2,d2,e2	e2,f2,g2	

Let R_1 resides in data source 1, and R_2 and R_3 reside in data source 2. The base relation version numbers of R_1, R_2 and R_3 are each 1, and the data source version numbers of data sources 1 and 2 are also 1. The refreshed version numbers at the view site are $\langle 1, 1, 1\rangle$ for R_1, R_2 and R_3 respectively. Thus the view is in consistent with both data sources. The highest processing version numbers at the view site are also $\langle 1, 1, 1\rangle$ for the three base relations.

Update transaction with data source version number 2 occurs at data source 1, and the updates involved are insertion R_1(a2,b2,c2) and R_1(a3,b3,c1), which now has its base relation version number changed to 2. The view site receives this notification and proceed to handle the incremental computation. The highest processing version number of R_1 is updated with the base relation version number of R_1, which is 2. Thus, the initial version numbers for the incremental computation of this update are $\langle 2,1,1\rangle$, taken from the highest processing version numbers. R_2 and R_3 are to be queried. Since the key of R_2 is in the view, and R_2 functionally determines the other base relations to be queried (only R_3 in this case), the view is first accessed for this incremental computation using the query $\{(a2,b2,c2),(a3,b3,c1)\} \rtimes V[R_2,R_3]$. It is found that the tuple (a3,b3,c1) can join with the tuple (c1,-,-) from R_2 and (-,f1,-) from R_3. Note the use of "-" for the unknown attributes values. The queried version numbers for both tuples are 1, taken from the refreshed version numbers for R_2 and R_3. Projecting the overall result over the view attributes adds one tuple of (b3,c1,f1) to the view. Thus the tuple (b3,c1,f1,1) is inserted into the view relation, with the count of 1 to indicate the single tuple added. The tuple (a2,b2,c2) that cannot retrieve any results from the view relation will have to do so by sending the view maintenance query $[\prod_C\{(a2,b2,c2)\}] \rtimes (R_2 \bowtie R_3)$ to data source 2. The result returned consists of the tuple (c2,d2,e2) from R_2 and (e2,f2,g2) from R_3, each with queried version number 1. Since the queried version numbers here correspond with the initial version numbers of both R_2 and R_3, there is no interfering update and compensation is not required. The tuple (b2,c2,f2,1) is inserted into the view to reflect the addition of one view tuple (b2,c2,f2). The final view is as follow, and its refreshed version numbers are $\langle 2,1,1\rangle$ since the result of the incremental computation of update from R_1 of base relation version number 2 has been incorporated into the view.

$V(B,C,F,count)$
b1,c1,f1,1
b3,c1,f1,1
b2,c2,f2,1

■

4.3 Modification

In our paper, modification update that involves the change of any of the view's join attributes will be handled as a deletion and an insertion updates because these update will join with different tuples of the other relations after the modification. Otherwise, the modification update will be handled as one type of update by our view maintenance algorithm.

Referential integrity constraint is not applicable in the incremental computation of modification update because it cannot be used to eliminate irrelevant updates. The use of the view relation for partial self-maintenance of the incremental computation of an update through the use of functional dependencies as in the case of insertion update can also be used in the case of modification update. On top of this, for a modification update $\triangle R_i$, if the key of R_i is in the view, then the view can be refreshed using the key value of $\triangle R_i$, without the need to compute the view tuple. Even for the case where the key of R_i is not in the view, but if there exists another relation R_j such that R_j functionally determines R_i and the key of R_j is in the view, then it is sufficient to query the base relations until R_j is accessed, without the need to derive the complete view tuple. This is because the view can be refreshed using the key value of the tuples of R_j that joins with the modification update $\triangle R_i$. The following example shows how this is achieved.

Example 7 Using the base and view relations of Example 6 after the result of incremental computation of the insertion update from R_1 is applied to the view, consider the case where the tuple (e1,f1,g1) of R_3 is modified to (e1,f2,g1). Since the modification is not on the view's join attributes, although the key of R_3 is not in the view relation, but the key of R_2 is in the view, and R_2 functionally determines R_3, then it is sufficient to query R_2 using $[\prod_E\{(e1, f1, g1)\}] \rtimes R_2$. The tuple (c1,d1,e1) returned in the query result from R_2 would change the view tuples from (b1,c1,f1,1) to (b1,c1,f2,1), and (b3,c1,f1,1) to (b3,c1,f2,1), through the SQL statement "UPDATE V SET F = f2 WHERE C = c1". ■

This explains why it is not efficient to treat modification simply as deletion and insertion updates as in the existing works. Not only more work is needed to refresh the view because unnecessary re-construction of the indexes could result, all base relations have to be accessed for incremental computation because we need to deal with insertion.

In the case of **deletion update**, all the above three improvements can be used. They are (1) the using of referential integrity constraints avoid unnecessary incremental computation, using the functional dependencies of the attributes and information of the keys of the relations to (2) involve the view in the incremental computation, and to (3) query a subset of the base relations instead of all of them.

5 Compensation

If each update is separated from one another for a sufficient large period of time such that the incremental computation of an update is allowed to complete before the next update occurs, then the problem of view maintenance anomalies would not occur. The initial version numbers of the incremental computation of an update, taken from the highest processing version numbers, which are updated whenever an update is sent for incremental computation, give an arbitrary ordering of the occurrence of the updates. In the ideal case, the queried version numbers correspond to the initial version numbers of an update. Since this is not possible in an environment of autonomous data sources, the job of the compensation process is to resolve the differences of the query results between the required states (initial version numbers) and the accessed states (queried version numbers) of the base relations. It does so by undoing the effect of the updates from the query result, if the queried version number is higher than the initial version number because the results has the unnecessary effect of the interfering updates. The compensation algorithm for these interfering updates are proposed in [LS99]. In Section 4, we propose the use of the view relation in the incremental computation through partial self-maintenance. If the view relation is used instead in the incremental computation, then the queried version number could be lower than the initial version number. In this case, the query result does not contain the actual change of the view relation, resulting in the view being maintained incorrectly. Thus, the effect of the *missing updates* have to be added to the result in order to give the correct incremental change. Note that missing updates refer to those updates that have occurred, but are not reflected in the query results because of the delay in refreshing the view, and is different from lost updates, which refer to those update transaction notifications that fail to reach to the view site due to network problems. In this paper, we propose the compensation method for resolving the anomalies caused by missing deletion, insertion and modification updates.

To compensate the query result with the interfering updates, the general idea is to undo the effect of these interfering updates, so as to bring the result which was generated based on the incorrect state (queried version number) to the required state (initial version number).

When using the view relation in the incremental computation for partial self-maintenance, we have to consider the fact that the view might not have been refreshed to the state of the update transaction processed for incremental computation prior to the update under consideration. In this case, the queried version numbers would be lower than the initial version numbers of an update. Compensation in this case is to incorporate the effect of these missing updates into the query result, so that the result reflect the presence of these updates.

Lemma 1 *Given that a tuple of the query result from R_j has queried version number β_j, and the initial version number of R_j for the incremental computation of $\triangle R_i$ is α_j.*

(1) If $\beta_j > \alpha_j$, then this tuple requires the compensation with updates from R_j of base relation version numbers β_j down to $\alpha_j + 1$. These are the ***interfering updates*** *on the tuple.*

(2) If $\beta_j < \alpha_j$, then this tuple (taken from the view relation through partial self-maintenance)

requires the compensation with updates from R_j of base relation version numbers $\beta_j + 1$ to α_j. These are the ***missing updates*** *on the tuple.*

(3) *If $\beta_j = \alpha_j$, then compensation on the query result from R_j is not required.*

■

Proof Let r_j from R_j be a tuple in the query result of incremental computation of update $\triangle R_i$. Also let the queried version number of r_j be β_j, and let the initial version number of R_j for the incremental computation of $\triangle R_i$ be α_j, i.e., its required state. If $\beta_j = \alpha_j$, then the accessed state is the required state and no compensation is necessary. If $\beta_j > \alpha_j$, then the query results have the effect of updates from R_j of base relation version number β_j down to $\alpha_j + 1$. Compensation is needed to undo this effect. Similarly, if $\beta_j < \alpha_j$, then the query results do not have the effect of updates from R_j of base relation version number $\beta_j + 1$ to α_j. Compensation is needed to incorporate this effect. ■

Example 8 Consider the same base and view relations of Example 6, as well as the same initial states. $R_1(\underline{A}, B, C)$ resides in data source 1, $R_2(\underline{C}, D, E)$ and $R_3(\underline{E}, F, G)$ reside in data source 2. The data source version numbers of both data sources are 1, so are the base relation version numbers of the base relations. The highest processing version numbers (and also the refreshed version numbers) at the view site are $\langle 1, 1, 1 \rangle$ for R_1, R_2, and R_3 respectively.

$R_1(\underline{A}, B, C)$	$R_2(\underline{C}, D, E)$	$R_3(\underline{E}, F, G)$	$V(B, C, F, count)$
a1,b1,c1	c1,d1,e1	e1,f1,g1	b1,c1,f1,1
	c2,d2,e2	e2,f2,g2	

The following update transactions occur at the data sources.

(1) Update transaction from data source 2 of data source version number 2 with delete R_2(c1,d1,e1). The base relation version number of R_2 is now 2.

(2) Update transaction from data source 1 of data source version number 2 with insertion R_1(a3,b3,c1) and R_1(a2,b2,c2). The base relation version number of R_1 is now 2.

(3) Update transaction from data source 2 of data source version number 3 with delete R_3(e2,f2,g2). The base relation version number of R_3 is now 2.

We process the incremental computation of the update transactions in the same order as above. Note that item (1) can also be processed for incremental computation either after item (2) or (3), while item (2) cannot be processed before item (3). This is because update transactions from the same data source have to be handled for incremental computation in the same sequence as their data source version numbers to achieve complete consistency.

The initial version numbers of delete R_2(c1,d1,e1) is $\langle 1,2,1\rangle$ for R_1, R_2 and R_3 respectively. Its query result is as follow ("qvn" denotes queried version number). There is no need to query any relation for this incremental computation because the key of R_2 is in the view. We forced it to query R_1 and R_3 so that we can demonstrate the working of the compensation algorithm in subsequent examples. The query sent to data source 1 is $[\prod_C\{(c1,d1,e1)\}] \rtimes R_1$, and that sent to data source 2 is $[\prod_E\{(c1,d1,e1)\}] \rtimes R_3$. The query result is as below.

$\mu R_1(\underline{A},B,C,qvn)$	$\triangle R_2(\underline{C},D,E)$	$\mu R_3(\underline{E},F,G,qvn)$
a1,b1,c1,2 a3,b3,c1,2	c1,d1,e1	e1,f1,g1,2

Table 1: Query result for deletion R_2(c1,d1,e1)

Comparing the queried version numbers of the result with the initial version numbers, there is a need to undo the effect of updates of base relation version number 2 from R_1 and R_3 as described in Section 5.2.

The initial version numbers of the two insertion updates from R_1 are $\langle 2,2,1\rangle$. Using the query $\{(a3,b3,c1),(a2,b2,c2)\} \rtimes V[R_2,R_3]$ to access the view relation, we obtain (c1,-,_e99) from R_2 of refreshed version number 1 and (_e99,f1,-) from R_3 of refreshed version number 1. "_e99" is used as a dummy value to link up the two tuples to make the result clearer. They are placed in the query result as (c1,-,_e99,1) and (_e99,f1,-,1) respectively. Since the tuple (a2,b2,c2) does not retrieve any results from the view relation, a query $[\prod_C\{(a2,b2,c2)\}] \rtimes [R_2 \bowtie R_3]$ is sent to data source 2. The tuple (c2,d2,e2) is returned from R_2 with queried version number 2, and it is placed in the query result μR_2 as (c2,d2,e2,2). An empty result is returned from R_3 with base relation version number 2, and it is placed in μR_3 as (-,-,-,2). The tuple (c1,-,_e99,1) needs compensaion with missing update from R_2 of base relation version number 2, while that of (-,-,-,2) needs compensation with interfering updates from R_3 of base relation version number 2. They are discussed in Sections 5.1 and 5.2 respectively.

$\triangle R_1(\underline{A},B,C)$	$\mu R_2(\underline{C},D,E,qvn)$	$\mu R_3(\underline{E},F,G,qvn)$
a3,b3,c1	c1,-,_e99,1	_e99,f1,-,1
a2,b2,c2	c2,d2,e2,2	-,-,-,2

Table 2: Query result for insertion R_1(a3,b3,c1) and R_1(a2,b2,c2)

The initial version numbers of delete R_3(e2,f2,g2) is $\langle 2,2,2\rangle$. The query $[\prod_E\{(e2,f2,g2)\}] \rtimes$

R_2 is sent to data source 2 and the the result of its incremental computation is as follow. Note that it does not need to query R_1 because using the key value of R_2 of the result in Table 3 is sufficient to maintain the view. Also, compensation is not required for this update since its initial version numbers are the same as the queried version numbers of the query result.

$\mu R_2(\underline{C}, D, E, qvn)$	$\triangle R_3(\underline{E}, F, G)$
c2,d2,e2,2	e2,f2,g2

Table 3: Query result for deletion R_3(e2,f2,g2)

■

5.1 Resolving Missing Updates

Now that we have identified what are the intefering and the missing updates, we look at how we go about resolving the effect of these updates. In this subsection, we propose the compensation method for resolving the missing deletion, insertion and modification updates. The purpose of compensating with a missing deletion update is to remove the tuples of this deletion from the query result.

Lemma 2 *Let μR_j be the query result from R_j (retrieved from the view through partial self-maintenance) for the incremental computation of $\triangle R_i$. To compensate the effect of* ***missing deletion update*** *$\triangle R_j$ for the result of incremental computation of $\triangle R_i$, all tuples of $\triangle R_j$ that are found in μR_j are dropped, together with those tuples from the other base relations that are retrieved due to the original presence of $\triangle R_j$ in μR_j.* ■

Example 9 Continuing from Example 8, the result of incremental computation of the updates from R_1 requires the compensation with the missing deletion update R_2(c1,d1,e1). The tuples (c1,-,_e99,1) and (_e99,f1,-,1) are dropped from the result in Table 2.

$\triangle R_1(\underline{A}, B, C)$	$\mu R_2(\underline{C}, D, E, qvn)$	$\mu R_3(\underline{E}, F, G, qvn)$
a3,b3,c1 a2,b2,c2	c2,d2,e2,2	-,-,-,2

Table 4: Compensation of result in Table 2 with deletion R_2 (c1,d1,e1)

■

Likewise, the compensation of a missing insertion update is to add the tuples of this insertion into the query result.

Lemma 3 *Let μR_j be the query result from R_j (retrieved from the view through partial self-maintenance) for the incremental computation of $\triangle R_i$, and μR_k be the query result from R_k. To compensate the effect of* ***missing insertion update*** *$\triangle R_j$ on the result of incremental computation of $\triangle R_i$, and assuming that μR_j is queried using the result from μR_k, all tuples of $\triangle R_j$ that can join with μR_k are added to μR_j, together with those tuples from the other base relations that should be retrieved due to the inclusion of $\triangle R_j$ in μR_j.* ■

The compensation with a missing modification update is simply to update the tuple from the unmodified state to the modified state.

Lemma 4 *Let μR_j be the query result from R_j (retrieved from the view through partial self-maintenance) for the incremental computation of $\triangle R_i$. To compensate the effect of* ***missing modification update*** *$\triangle R_j$ (which does not involve any change to the view's join attributes), for the result of incremental computation of $\triangle R_i$, each old tuple (before the modification) of $\triangle R_j$ that occurs in μR_j has its values changed to the corresponding new tuple (after the modification).* ■

Note that for both missing deletion or modification update $\triangle R_j$, if the key of R_j is not in the view, then the relation R_k with its key that functionally determines the attributes of R_j will be used in applying Lemmas 2 and 4.

5.2 Resolving Interfering Updates

The compensation method for resolving the interfering deletion, insertion and modification updates are proposed in [LS99]. The purpose of compensating with an interfering deletion update is to add this update into the query result, because it would be present in this result if the deletion has not occurred. Note that Lemmas 5, 6 and 7 are already given in [LS99], and are provided below for completeness of the compensation algorithm.

Lemma 5 *Let μR_j be the query result from R_j for the incremental computation of $\triangle R_i$, and μR_k be the query result from R_k. To compensate the effect of* ***interfering deletion update*** *$\triangle R_j$ for the result of incremental computation of $\triangle R_i$, and assuming that μR_j is queried using μR_k, all tuples of $\triangle R_j$ that can join with μR_k are added to μR_j, together with those tuples from the other base relations that should be retrieved due to the inclusion of $\triangle R_j$ in μR_j.* ■

Example 10 From Example 9, the tuple (-,-,-,2) of the query result (Table 4) require the compensation with interfering deletion update R_3(e2,f2,g2). Since this tuple can join with the tuple (c2,d2,e2,2), it is added to the query result.

$\triangle R_1(\underline{A}, B, C)$	$\mu R_2(\underline{C}, D, E, qvn)$	$\mu R_3(\underline{E}, F, G, qvn)$
a3,b3,c1 a2,b2,c2	c2,d2,e2,2	e2,f2,g2,1

Table 5: Compensation of result in Table 4 with deletion R_3 (e2,f2,g2)

■

Compensation with an interfering insertion update removes such tuples from the query result because they should not be present when the incremental computation accesses the base relation.

Lemma 6 *Let μR_j be the query result from R_j for the incremental computation of $\triangle R_i$. To compensate the effect of* ***interfering insertion update*** *$\triangle R_j$ for the result of incremental computation of $\triangle R_i$, all tuples of $\triangle R_j$ that are found in μR_j are dropped, together with those tuples from the other base relations that are retrieved due to the original presence of $\triangle R_j$ in μR_j.* ■

Example 11 From Example 8, the compensation with interfering insertion update from R_1 on the query result of delete R_2(c1,d1,e1) drops the tuple (a3,b3,c1,2) from the query result in Table 1. Note that compensation with the interfering deletion update R_3(e2,f2,g2) has no effect on its query result.

$\mu R_1(\underline{A}, B, C, qvn)$	$\triangle R_2(\underline{C}, D, E)$	$\mu R_3(\underline{E}, F, G, qvn)$
a1,b1,c1,1	c1,d1,e1	e1,f1,g1,1

Table 6: Compensation of result in Table 1 with insertion R_1(a3,b3,c1) and R_1(a2,b2,c2) and deletion R_3(e2,f2,g2)

■

The compensation with an interfering modification update is the reverse action for the compensation with a missing modification update.

Lemma 7 *Let μR_j be the query result from R_j for the incremental computation of $\triangle R_i$. To compensate the effect of* ***interfering modification update*** *$\triangle R_j$ (which does not involve any change to the view's join attributes) for the result of incremental computation of $\triangle R_i$, each new tuple (after the modification) of $\triangle R_j$ that occurs in μR_j has its values changed to the corresponding old tuple (before the modification).* ■

Having looked at how are the interfering and missing updates identified using Lemma 1, and also how are the effect of these updates on the query result resolved using Lemmas 2 to 7, Theorem 1 gives a systematic way to handle the compensation process. Since the compensation accesses the results from the base relations in the same order as they are queried, the compensation can be carried out immediately when the result is returned from a data source, before the next data source is queried.

Theorem 1 *Given α_1,...,α_n as the initial version numbers of incremental computation of update $\triangle R_i$, the compensation starts with those relations that are linked to R_i in the join graph, and proceed recursively to the rest of the relations in the same sequence as they are been queried.*

(1) The compensation on the query result from R_j proceeds by first compensating with the missing deletion, insertion and modification updates of base relation version number $\beta_j^{min}+1$, and progresses to the updates of base relation version number α_j, where β_j^{min} is the minimum queried version number of the tuples in the result from R_j, using Lemmas 2, 3 and 4.

(2) This is followed by the compensation with the interfering deletion, insertion and modification updates of base relation version number β_j^{max}, and progresses down to the updates of base relation version number α_j+1, where β_j^{max} is the maximum queried version number of the tuples in the query result from R_j, using Lemmas 5, 6 and 7.

■

The detection of lost update notification messages (note that this is the lost of messages in the network and is different from the missing updates, which is another type of interfering updates) is through the detection of the lost of an update notification message of a particular data source version number when the update transactions of higher data source version numbers have already been received by the view site for a sufficient long period of time. It can also be detected through the lost of an update of a particular base relation version number during the compensation process [LS99]. For instance, the initial version number for R_j is α_j for the incremental computation of $\triangle R_i$, and the queried version number of a tuple from R_j is β_j. If $\beta_j > \alpha_j$, but the view site does not receive one of the updates from R_j of base relation version number γ_j, where $\beta_j \geq \gamma_j > \alpha_j$, then this update is probably lost during the transmission over the network.

In [LS99], an update transaction is assumed to involve only one base relation. In this paper, we extend this to allow for an update transaction to involve any number of base relations within the same data source. The following theorem states the condition for achieving complete consistency of the view with respect to the update transactions at the data sources.

Theorem 2 *If the following two conditions hold, then complete consistency of the view is achieved.*

(1) Update transactions of the same data source have to be sent for incremental computation and their compensated results applied to the view in ascending order of data source version numbers.

(2) Two update transactions from two different data sources (assuming that they are both the next update transactions to be processed for their respective data sources) can be sent for incremental computation in whichever order, but the order of applying their compensated results to the view must follow the same order as they are sent for incremental computation.

■

Proof Consider two update transactions u_1 and u_2 such that u_1 occurs earlier than u_2. If u_1 and u_2 come from the same data source, then the data source version number of u_1 has to be smaller than that of u_2. If they come from different data sources, then this ordering is arbitrary. Processing the incremental computation of u_1 earlier than u_2 ensures that the compensated view maintenance query result of the incremental computation of u_1 does not see the effect of u_2, while that of u_2 sees the effect of u_1. The compensated results need to be applied to the view in the same order to show such effect. ■

Example 12 Using Example 8, and consider its compensated query result in Table 6, deletion update R_2(c1,d1,e1) will remove the only tuple (b1,c1,f1) with a count of 1 from the view.

$V(B, C, F, count)$

Using the result in Table 5, the insertion update from R_1 will add one tuple (b2,c2,f2) to the view.

$V(B, C, F, count)$
b2,c2,f2,1

From the result in Table 3, the deletion update R_3(c2,f2,g2) drops the single tuple of (b2,c2,f2) from the view (DELETE FROM V WHERE C = c2). The view is now empty. ■

6 Comparison

Related works in this area include that of Eager Compensation Algorithm (ECA) [ZGMHW95], the Strobe and C-Strobe Algorithms [ZGMW96], the work of [CM97a], the SWEEP and Nested SWEEP Algorithms [AASY97], the work of [CM97b], and the work of [LS99].

All the approaches, except ECA and [CM97a], cater to multiple data sources. ECA works for an environment of single data source with multiple base relations, while [CM97a] is limited to two base relations because it is designed for a view relation with outerjoin.

The compensation process of ECA and Strobe is not through the detection of interfering updates, and hence they only achieved strong consistency. The algorithm of C-Strobe results in the detection of some interfering deletion updates which are actually not interfering updates. Nevertheless, the view is still maintained correctly through duplicate tuples removal. The rest of the algorithms work by correctly detecting for the presence of interfering updates when messages are not misordered or lost.

All the approaches, except our methods (both [LS99] and the work of this paper), assume that messages are never lost and first-sent-first-received delivery of messages are guaranteed. [CM97b] also does not assume the first-sent-first-received delivery of messages.

Most of the approaches can only work by querying one base relation at a time because querying multiple base relations is not supported by their compensation algorithm. ECA can query all base relations of their single data source together, but they can only achieve strong consistency. We are able to query multiple base relations within the same data source through our use of version numbers and the compensation process that we define.

Existing methods base their view maintenance querying on a left and right scan approach, and thus limit their parallelism to the two scans. In this paper, we use the join graph to guide the accessing of the base relations, and thus provide more parallelism. More significantly, the use of the join graph to determine the query path of the incremental computation avoids the execution of cartesian products. Data transmitted through the network are reduced significantly, and the view site does not have to process the extra data returned by the base relations. Together with our proposed compensation algorithm that allows for the querying of more than one base relation with a single sub-query, multiple base relations within the same data source can be queried together. This reduces the overrall round-trip time of the incremental computation for each update, and also further reduces the number of records transmitted through the network as the join operation can also be performed at the data source instead of at the view site only.

ECA, Strobe and Nested SWEEP are able to process the incremental computation of different updates concurrently, but they are only able to achieve strong consistency. The rest of the methods have to process the incremental computation of different updates sequentially because of the need to keep track of which are the interfering updates. Our methods can process the incremental computation concurrently and at the same time achieve complete consistency.

Strobe and C-Strobe have a limited form of self-maintenance in that deletion update need not be processed for incremental computation because of the retainment of the keys of all base relations in the view. [CM97a] can detect irrelevant updates, i.e., updates that will not affect the view, without the need to issue any view maintenance queries at the expense of keeping a system catalogue. In this paper, we provide for more opportunity of self-maintenance through the use of referential integrity constraint and functional dependencies.

C-Strobe, [CM97a], SWEEP, [CM97b] and our approach achieve complete consistency, and also do not require a quiescent state before the view can be refreshed. This does not apply to ECA, Strobe and the Nested SWEEP Algorithm.

From the above discussion, it is noted that none of the other approaches cater to all the criteria, although each can cater to some of these, while we are able to achieve all of them. This criteria are view maintenance in an environment of multiple data sources, correct identification of interfering updates, handling misordered and lost messages, querying multiple base relations together, parallel incremental computation within the same update and between different updates, self-maintenance, flexible view definition, complete consistency and no quiescence requirement.

7 Conclusion

We extend our work of [LS99] to make the incremental computation of materialized view maintenance more efficient by improving the view maintenance query process. The algorithm of [LS99] handles the compensation of interfering updates correctly when messages are misordered in the communication network, or when update notification messages are lost. Incremental computation of different updates can be done concurrently, quiescent state is not required before the view can be refreshed, and complete consistency is achieved.

In this paper, the querying of the base relations for incremental computation are based on the join graph to avoid the executing of cartesian products. This gives much better performance and reduces the network traffic significantly. Using the join graph to determine the query path of the incremental computation also results in more parallelism, and multiple base relations residing at the same data source can be queried together as it is supported by our compensation algorithm. Knowledge of referential integrity constraint imposed by the data source are used to eliminate irrelevant updates. Also functional dependencies, together with the support of our compensation algorithm, are used to involve the view relation (partial self-maintenance) in the maintenance process to cut down the number and size of the queries and their corresponding results. This is possible because besides compensating for interfering updates, our new compensation algorithm can also handle missing updates.

In summary, we have achieved all the criteria of catering to an environment of multiple data sources, correct identification of interfering updates, handling misordered and lost messages, querying multiple base relations together, parallel incremental computation within the same update and between different updates, partial self-maintenance, flexible view definition, complete consistency and no quiescence requirement. Performance is improved significantly by reducing the amount of network traffic and the time to handle individual incremental computation. All other existing algorithms can only achieve some of these criteria, but none of them can achieve all the criteria within the same algorithm.

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 self-maintenance of materialized views. Technical report, Stanford University, 1996.

[Huy96b] Nam Huyn. Efficient view self-maintenance. In *Proceedings of the ACM Workshop on Materialized Views: Techniques and Applications*, pages 17–25, June 1996.

[Huy96c] 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.

[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.

[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.