Materialized View Maintenance Using Version Numbers

Tok Wang Ling Eng Koon Sze
School of Computing
National University of Singapore
Lower Kent Ridge Road, Singapore 119260
{lingtw,szeengko}@comp.nus.edu.sg

Abstract

A data warehouse stores materialized views over data from one or more sources in order to provide fast access to the integrated data, regardless of the availability of the data sources. In this paper, we define a new compensation algorithm that is used in removing the anomalies, caused by interfering updates at the base relations, of incremental computation for updating the view. Unlike existing methods on view maintenance, our algorithm does not assume that messages from a data source will reach the view maintenance machinery in the same order as they are generated, and we are also able to detect update notification messages that are lost in their transit to the view, which would otherwise cause the view to be updated incorrectly. These are achieved with the use of version numbers that reflect the states of the base relations. Our algorithm also does not require that the system be quiescent before the view can be refreshed.

1. Introduction

Data warehousing is used for reducing the load of online transactional systems by extracting and storing the data needed for analytical purposes. A materialized view of the system is kept at a site called the *data warehouse*, and user queries are processed using this view. As this view is an integration of data from many sources that undergo constant updating, algorithm for efficient maintenance of this materialized view is an important current research area. Simple recomputing of the view relation from scratch in response to each source update will not work well as the base relations are generally large. An algorithm is needed for the view to decide how to refresh its relation with respect to this update. Existing works on this approach include the Eager Compensating Algorithm (ECA) [10], the Strobe algorithm [11], the SWEEP algorithm [1], and the algorithm of [2].

Our algorithm is designed to work for a materialized view that integrates data from multiple distributed autonomous sources, where the base relations of these sources are not materialized at the view. Unlike other methods, this algorithm does not require that the system be quiescent before the materialized view can be refreshed. We do not assume that messages from the data sources are delivered in the same order to the view as they are generated, and we are able to handle update notification messages that are lost in their transit to the view maintenance machinery, both through the use of version numbers that reflect the states of the base relations. We also handle modification as one update in certain cases, rather than simply treating it as a deletion update, followed by an insertion update. This is more efficient since the splitting of a modification update into 2 updates would possibly involve the removal of tuples, together with their indexes, from the view relation and the subsequent adding in of the same tuples (but with the relevant updating of the modified attributes) as well as the re-building back of their indexes. This would create a lot of processing overhead for applications that deal mainly with modification updates.

In Section 2, we show a general model for a data warehouse. Section 3 describes some existing materialized view maintenance algorithms. We discuss our algorithm in Section 4, together with an example. Section 5 concludes the work of this paper.

2. Data warehouse model

In this section, we describe the general architecture of a data warehouse. There are m sites for the data sources and another site for housing and maintaining the materialized view. There is communication between each data source and the site storing the view, but no assumption is made with regard to any communication between individual data sources. Thus, the data sources are treated as completely independent and may not be able to communicate with one another. No assumption is made with regard to the reliability of the network connection between the view and each data sources, i. e., messages sent could be lost. We also do not assume that messages will reach the view maintenance machinery in the same order as they are generated at the data sources.

The underlying database model for each data source and the view is considered to be a relational data model. Each data source can store any number of base relations. Updates to different data sources are assumed to be not related at all, but a transaction can involve multiple base relations residing at the same data source. For each update transaction, the data source will send an update notification message to the view after the transaction is committed.

We consider the case where the view relation is defined by selection-projection-join expression. Hence, given n base relations, $\{R_1, ..., R_i, ..., R_n\}$ with $1 \leq i \leq n$, and each data source can hold one or more of these relations, the view V is given as:

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

If the materialized view V does not contain any of the keys of $R_1 \bowtie ... \bowtie R_i \bowtie ... \bowtie R_n$, then each tuple in V will have a *count* value which indicates in how many different ways the same tuple can be derived from the given view definition and the base relations.

3. Previous works

The view maintenance problems for a centralized relational databases are tackled in [3], [4], [5] and [8]. [6] and [7] examine when is a view self-maintainable with respect to an update, while [9] derives auxiliary views of parts of the base relations, using key and referential integrity constraints, to make a view self-maintainable. Our work is closer to that of [10], [11], [1] and [2].

For refreshing a materialized view through incremental computation in response to an update, queries are issued to the various base relations to retrieve those tuples that join with this updated tuple to form the affected tuples of the view relation. However, other updates could occur between the time the view first receives this update and the final receiving of the query results from the last base relation that the view queried. Consider the view defined in Equation (1), in which the view first receives the insertion update of $\triangle R_i$ and proceeds to query the base relations. Before the view queries the relation R_j with $j > i$, another insertion update of $\triangle R_j$ occurs. Thus, for $\triangle R_i$ the answer that the view has is $R_1 \bowtie ... \bowtie \triangle R_i \bowtie ... \bowtie (R_j \cup \triangle R_j) \bowtie ... \bowtie R_n$ (note that in all our discussion, unless stated explicitly in the expression by prefixing the relation with "$\triangle$" to denote a group of updated tuples, any other relations indicated by "..." refers to the whole relation), while that of $\triangle R_j$ gives $R_1 \bowtie ... \bowtie (R_i \cup \triangle R_i) \bowtie ... \bowtie \triangle R_j \bowtie ... \bowtie R_n$. Thus the term $R_1 \bowtie ... \bowtie \triangle R_i \bowtie ... \bowtie \triangle R_j \bowtie ... \bowtie R_n$ is computed and inserted into the view relation twice, and the redundant set of tuples is known as an *error term*, causing *anomaly* to the maintenance of the materialized view.

The Eager Compensating Algorithm (ECA) [10], designed to work for only one source with multiple base relations, attempts to solve this interfering updates anomaly problem by means of *compensating queries*. This algorithm assumes that messages are delivered to the view in the same order as they are sent out by the source. An update notification received by the view triggers a query processing event for incremental computation, and those updates that arrive before this update but have not completed their query processing will be used in generating the compensating queries for this update. Hence in the above scenario in which update $\triangle R_j$ arrives after the issue of the query $R_1 \bowtie ... \bowtie \triangle R_i \bowtie ... \bowtie R_n$ to the source for update $\triangle R_i$ but before the view receives back this answer, there is a need to generate a corresponding compensating query of $R_1 \bowtie ... \bowtie \triangle R_i \bowtie ... \bowtie \triangle R_j \bowtie ... \bowtie R_n$ for $\triangle R_j$, and the *compensated answer* of $\triangle R_j$ will be given as $[R_1 \bowtie ... \bowtie (R_i \cup \triangle R_i) \bowtie ... \bowtie \triangle R_j \bowtie ... \bowtie R_n] - [R_1 \bowtie ... \bowtie \triangle R_i \bowtie ... \bowtie \triangle R_j \bowtie ... \bowtie R_n]$.

The Strobe algorithm [11] is designed for an environment with multiple data sources, thus overcoming the single source restriction imposed by the ECA algorithm. This algorithm handles the compensation of interfering updates locally by requiring that the keys of all base relations be included in the view relation. Besides the restriction on the view definition, the materialized view cannot be updated until there is quiescence in the system. The correct working of this algorithm also requires that messages are delivered in the same order to the view as they are generated at the data sources.

The SWEEP algorithm [1] is another view maintenance method designed for a system with multiple data sources. The definition of the view relation is more flexible as compared to the Strobe algorithm in that there is no requirement for the keys of all base relations to be kept in the view. This algorithm also differs from the Strobe algorithm in that it does not require that the system be quiescent before the view can be refreshed. Messages are again assumed to be delivered to the view in the same order as they are generated at the data sources. This algorithm works by removing immediately the effect of interfering updates from the intermediate answer of the querying process of an update. Thus using the same scenario as above, if insertion update $\triangle R_j$ reaches the view before the view receives the intermediate answer from R_j for update $\triangle R_i$, there is a need to remove $\triangle R_j$ from this answer (if it is present) before it is used for querying the next base relation.

The concept of using counters to identify the states of a base relation in [2] overcomes the assumption of the delivery order of messages. However, the need to update the counter at the view only after each update has completed both its query processing and compensation forces the maintenance machinery to handle the incremental computation of one update at any one time.

4. View maintenance solution

In this paper, we propose a compensating algorithm that removes the interfering updates anomalies found in the answers of the incremental computation. Unlike the 3 algorithms mentioned in the previous section, our proposed view maintenance solution takes into consideration the situation where the delivery of messages from a data source to the site housing the view is delayed, resulting in messages reaching the view in a different order from what was sent out at the data source. Our algorithm can also detect update notification messages that are lost in their transit to the view maintenance machinery, which if left unattended might cause the view to be maintained incorrectly.

4.1. Motivation

We achieve the above through the use of *version numbers* to order the updates occurring at a base relation. A group of updates of a transaction occurring at a base relation will increment its version number by one, and the base relation notifies the view maintenance machinery on the updates involved, together with the new version number of this base relation. The version numbers not only allow the view to arrange the updates from the same base relation in the same order as they have occurred, instead of the order that they arrived at the view, they also provide a means for the view to detect update notification messages that are lost by keeping track of the highest version number received and the set of version numbers that should have reached the view but have not done so.

Updates (updates of the same transaction for a relation are grouped together) from the same relation need to be ordered based on their version numbers, otherwise we might get the wrong results in the incremental computation if the updates are on the same tuple, for instance modifying a tuple before inserting it. As for updates from different base relations, we can just order them arbitrary based on the order that we process them, since updates from different relations can never involve the same tuple. However, the order of refreshing the view relation should follow the same order as the updates that are picked for processing. In the compensation of the answer of the query of an update, we will eliminate the effects of those updates (interfering updates) that are ordered to occur after this particular update, but leaving behind the effects of those updates that are ordered to occur before it.

4.2. Compensation algorithm

We first consider the working of the data source in support of our maintenance machinery, followed by that of the view.

For each base relation, three types of events could occur. They are *update*, *query* (query here refers to the query from the view for incremental computation) and *re-send*. The update event handles all the updates involved in this base relation of a transaction, increments its version number and pass this information to the data source, so that the data source can put the information from all base relations involved in this transaction into one update notification message before sending it to the view. For a query event on a base relation, the required tuples will be retrieved through the computation of the join operation, and they are then sent back to the view, together with the current version number (no incrementation required here) of this base relation known as the *queried version number*. A re-send event occurs when the view maintenance machinery detects that an update notification message is lost. This re-send event will request that the base relation re-transmit the update notification of a particular version number. The base relation responds by retrieving the information from its log file.

We now describe the working of the view maintenance machinery for a materialized view defined by Equation (1). For each of the base relation R_i, the view keeps track of the highest version number, α_i, of the updates sent for processing. Processing here refers to the querying of the base relations for incremental computation. Note that α_i might not coincide with the highest version number that has already reached the view for two reasons. First, the view might not be able to process the updates as fast as they have reached the view. Secondly, the next version of update notification after the previous version of update notification sent for incremental computation might not have reached the view yet.

Definition 1 *The collection of the highest version numbers of all the base relations, $<\alpha_1, ..., \alpha_n>$, that the view has picked for incremental computation (processing) are called the* ***highest processing version numbers****.*

Whenever the view receives an update transaction notification from a data source, it is first placed in a *pending queue*, one for each data source and ordered according to their version numbers. Whenever the view is ready to pick an update transaction for incremental computation, it can do so from any pending queue as long as the version number of this update transaction is equal to the highest processing version number plus one for that base relation. For an update transaction that involves more than one base relation, the version numbers of the updates from the various base relations in this transaction have to be equal to the highest processing version number plus one for each of the base relation concerned (this ensures that the atomic updates of a transaction are processed and applied to the view together). The highest processing version number for the base relation is then incremented by one to reflect the current processing state.

Definition 2 *The storing of the highest processing version numbers of all base relations, $< \alpha_1, ..., \alpha_n >$, to a group of updates at the start of its incremental computation are called its **initial version numbers**.*

Several consistency notions have been associated with the views at the data warehouse ([10], [11]). They are complete consistency (most stringent), strong consistency, weak consistency, and convergence (least stringent). The arbitrary picking of updates from any base relation can ensure complete consistency (complete order-preserving mapping between the states of the view and the states of the data sources) if the data source contains only one base relation. Otherwise, only convergence (the view is consistent with the data sources after the last update and after all activity has ceased) is ensured. To achieve complete consistency for the case of a data source with multiple base relations, there is a need to add another level of version number to this data source, and this version number is incremented for each update transaction occurring at the data source. The update transactions in the pending queue for such data source will be ordered according to this version number. With this we would be able to tell, for example, the exact order of 2 update transactions from the same data source, but involving a different base relation in each case.

Assuming that we have $R_1 \bowtie ... \bowtie \triangle R_i \bowtie ... \bowtie R_n$ as an optimized query operation for update $\triangle R_i$, queries are sent to the various base relations on both sides of R_i, for the relations on the left, $\{R_1, ..., R_{i-1}\}$, starting with R_{i-1}, and for the relations on the right, $\{R_{i+1}, ..., R_n\}$, starting with R_{i+1}. The query processing on both sides of R_i can occur in parallel, but for each side query will be sent to one relation at a time, using the results of the previous query for the next query. The query for each side will stop either when relation R_1 (or R_n for the other side) has been queried, or when an empty results is returned for a query. Although the view is defined by a natural join operation, outerjoin (denoted as "$*^r$" for right outerjoin and "$*^l$" for left outerjoin) of the tuples returned through the querying process is used for storing the answer in order to support the compensation process. Thus we store the results temporary as $R_1 *^r ... *^r \triangle R_i *^l ... *^l R_n$. Note that right outerjoin is used for those relations on the left of R_i and left outerjoin for those relations on the right. The idea here is to keep the incomplete tuples of the view relation temporary for applying the compensation process in case they are actually complete tuples of the view, due to the effect of interfering deletion updates.

Definition 3 *At the end of the querying process for $\triangle R_i$, the view will have $R_1(s_1) *^r ... *^r \triangle R_i *^l ... *^l R_j(s_j) *^l ... *^l R_n(s_n)$, where s_j denotes the **queried version number** of base relation R_j. The collection of the queried version numbers is also denoted as $< s_1, ..., s_j, ..., s_n >$.*

The answer of an update after the query processing will have to be compensated with all the interfering updates first before they can be applied to the view relation. The identification of the interfering updates is stated in Lemma 1 and the method of compensation is described in Figure 1. Since there is a time lag between the sending out of a query to a base relation, and the subsequent receiving back of the results at the view, the maintenance machinery should proceed to handle other messages (update notification or query results).

Lemma 1 *Given updates $\triangle R_i$ (same version number from relation R_i) with $< \alpha_1, ..., \alpha_j, ..., \alpha_n >$ as its initial version numbers, and $< s_1, ..., s_j, ..., s_n >$ as its queried version numbers. There is a need to compensate with all interfering updates from R_j, where $1 \leq j \leq n$ and $j \neq i$, with version numbers from s_j down to $\alpha_j + 1$ if $s_j > \alpha_j$.*

Proof Considering that the initial version numbers $< \alpha_1, ..., \alpha_j, ..., \alpha_n >$ as the state of all the base relations when $\triangle R_i$ occurs, any base relation R_j with queried version number $s_j > \alpha_j$ means that the state of R_j that was queried on has the interfering effect of updates of version numbers $\alpha_j + 1, ..., s_j$.

Figure 1 shows the method of removing the effect of the interfering insertion, deletion and modification updates. The compensation starts with all update transactions which are interfering updates on the relations on both sides of R_i, i. e., for the left of R_i starting with R_{i-1} and progressing down to R_1, and for the right starting with R_{i+1} and progressing up to R_n. Compensation with the interfering updates of R_j, where $1 \leq j \leq n$ and $j \neq i$, starts with the largest version number s_j and proceeds down to the lowest version number $\alpha_j + 1$. This order of processing allows the compensation to work correctly in case any of the interfering updates are on the same tuple.

How the compensation is carried out depends on the update type (insertion, deletion or modification) of the interfering update to be compensated with. Our algorithm will split a modification update into a deletion update, followed by an insertion update, only if the modified attributes involve at least one join attributes. The checking and splitting of such modification updates is carried out by the view maintenance machinery when the update notifications reach the view. Also if the modified attributes are not on the join attributes and also do not appear in the view relation, then this modification update can be ignored.

```
Compensation Algorithm
INPUT : Group of updates $\triangle R_i$ of version number $\alpha_i$,
        initial version numbers $< \alpha_1, ..., \alpha_n >$,
        uncompensated answer $A_i$ with queried
        version numbers$< s_1, ..., s_n >$
OUTPUT : Compensated answer $A_i$

FOR $j = i + 1$ TO $n$ DO /* if $i \neq n$ */
  IF $s_j > \alpha_j$ THEN
    /* Note that as a result of compensating with deletion
       updates, some newly added tuples could have a
       different queried version number $s'_j$ (or initial
       version number if the compensated tuples are used
       instead of the uncompensated tuples). If $s'_j > s_j$, we
       will need to apply this algorithm on these tuples to
       bring them to the state of version $s_j$ first. If $s'_j < s_j$,
       we will only apply the compensation algorithm on
       those tuples with queried version number $s'_j$ when $k$
       reaches $s'_j$ in the below FOR loop. */
    Let $\triangle R_{jk}$ be those groups of updates from $R_j$ with
      version $k$, where $\alpha_j + 1 \leq k \leq s_j$
    FOR $k = s_j$ DOWNTO $\alpha_j + 1$ DO
      Let $u_I$, $u_M$ and $u_D$ be the set of insertion, modification
        and deletion updates in $\triangle R_{jk}$
      Compensation with insertion updates $u_I$:
        $A_i \leftarrow A_i - (\sigma_{R_j \in \triangle R_j} A_i)$
      Compensation with modification updates $u_M$:
        Let $u^{old}$ denotes one tuple of $u_M$ before modification
          and $u^{new}$ denotes the corresponding tuple after
          modification
        Replace all occurrences of $u^{new}$ in $A_i$ with $u^{old}$
        Do the above for all tuples of $u_M$
      Compensation with deletion updates $u_D$:
        From $R_1(s'_1) *^r ... *^r R_{j-1}(s'_{j-1}) *^r$
          $u_D *^l R_{j+1}(s'_{j+1}) *^l ... *^l R_n(s'_n)$
          of incremental computation of $u_D$, use
          $u_D *^l R_{j+1}(s'_{j+1}) *^l ... *^l R_n(s'_n)$ to compute
          $A_i \leftarrow A_i[R_1, ..., R_{j-1}]$
              $*^l (A_i[R_j, ..., R_n] \cup \triangle R_j *^l R_{j+1} *^l ... *^l R_n)$
    END-FOR
  ELSE do nothing
END-FOR
The above is repeated using a FOR loop for
  $j = i - 1$ down to 1 if $i \neq 1$
RETURN $A_i$ /* compensated answer $A_i$ */
```

Figure 1. Compensation algorithm.

Insertion. Suppose we have the answer $A_i \equiv R_1 *^r ... *^r \triangle R_i *^l ... *^l R_j *^l ... *^l R_n$ at the end of the incremental computation for update $\triangle R_i$. Let $\triangle R_j$ from R_j be an interfering insertion update that has caused an anomaly to answer A_i. Since $\triangle R_j$ occurs before the querying of R_j by the view for update $\triangle R_i$, it is true that $\triangle R_j \subseteq R_j$, therefore $(R_1 *^r ... *^r \triangle R_i *^l ... *^l \triangle R_j *^l ... *^l R_n) \subseteq (R_1 *^r ... *^r \triangle R_i *^l ... *^l R_j *^l ... *^l R_n)$.

Lemma 2 *The compensation of the answer A_i of $\triangle R_i$ with an **interfering insertion update** $\triangle R_j$ can be handled by deleting those tuples in A_i that contain $\triangle R_j$, without the need to issue any new query to the base relations (computed locally) using:*

$$A_i = A_i - (\sigma_{R_j \in \triangle R_j} A_i) \tag{2}$$

Proof The effect on the view V by update $\triangle R_i$ should be with respect to the state of base relation R_j without insertion $\triangle R_j$, i. e., $(R_j - \triangle R_j)$. Thus, we need to remove $\triangle R_j$ from A_i if it is in A_i, together with those tuples that are in A_i due to the inclusion of $\triangle R_j$.

Modification. For the case of interfering modification update $\triangle R_j$, we denote the values of the tuple before modification as $\triangle R_j^{old}$, and the values of the tuple after modification as $\triangle R_j^{new}$. Since a modification update from a base relation that changes at least one join attributes would have been separated into a deletion update, followed by an insertion update by our view maintenance machinery, thus any modification update $\triangle R_j$ that remains as such satisfies $\prod_{(V-R_j)\cup C} R_1 *^r ... *^r \triangle R_i *^l ... *^l \triangle R_j^{old} *^l ... *^l R_n \equiv \prod_{(V-R_j)\cup C} R_1 *^r ... *^r \triangle R_i *^l ... *^l \triangle R_j^{new} *^l ... *^l R_n$, where C is the set of join attributes between R_j and all relations that it joins with.

Lemma 3 *The compensation of the answer A_i of $\triangle R_i$ with an **interfering modification update** that does not involve the change of values of its join attributes, with $\triangle R_j^{old}$ as the tuples before modification and $\triangle R_j^{new}$ as the tuples after modification, can be computed locally by replacing the tuples in $\triangle R_j^{new}$ in the answer A_i with the corresponding tuples in $\triangle R_j^{old}$, keeping the tuples from the other base relations intact.*

Proof As in the case of insertion.

Deletion. The situation for interfering deletion update is more complicated than the other 2 types of updates. The adding in of interfering deletion update $\triangle R_j$ to A_i (assuming $j > i$) means that more tuples from the other base relations might also have to be added because we now have to consider those tuples in $\{R_{j+1}, ..., R_n\}$ that could join with $\triangle R_j$ on top of the original tuples in R_j (without $\triangle R_j$).

Lemma 4 *The compensation of the answer A_i of $\triangle R_i$ with an* ***interfering deletion update*** $\triangle R_j$ *is computed locally as (for $i < j$):*

$$\begin{aligned} A_i &= A_i[R_1, ..., R_{j-1}] \\ &\quad *^l (A_i[R_j, ..., R_n] \cup \triangle R_j *^l R_{j+1} *^l ... *^l R_n) \end{aligned} \tag{3}$$

or (for $j < i$):

$$\begin{aligned} A_i &= (R_1 *^r ... *^r R_{j-1} *^r \triangle R_j \cup A_i[R_1, ..., R_j]) \\ &\quad *^r A_i[R_{j+1}, ..., R_n] \end{aligned} \tag{4}$$

The projection and union operations here do not remove duplicate tuples, so that the count field in the view relation can be updated correctly.

Proof The effect on the view V by update $\triangle R_i$ should be with respect to the state of base relation R_j without deletion $\triangle R_j$. Thus, we need to add $\triangle R_j$ into A_i (if it can join with some tuples from R_{j-1} of A_i assuming $i < j$), together with those tuples from $R_{j+1}, ..., R_n$ that could join with $\triangle R_j$. Although the term $\triangle R_j *^l R_{j+1} *^l ... *^l R_n$ was not found in A_i, there is no necessity to send new queries to base relations $R_{j+1}, ..., R_n$. This is because the update $\triangle R_j$ itself will cause the view machinery to query the respective base relations during its incremental computation. In effect, we have the answer A_j due to the query processing for update $\triangle R_j$:

$$A_j \equiv R_1 *^r ... *^r R_i *^r ... *^r \triangle R_j *^l R_{j+1} *^l ... *^l R_n$$

Note that the base relations that are seen by deletion update $\triangle R_j$ during its querying process could be different from what was seen by update $\triangle R_i$ due to some other interfering updates that could have taken place, and this will be indicated by the differences in their queried version numbers. The compensation of the answer of $\triangle R_i$ has to take this into account.

In doing the above interfering deletion update compensation, we can use both the uncompensated or compensated term $\triangle R_j *^l R_{j+1} *^l ... *^l R_n$ of A_j. Using the compensated term would be more efficient since less compensation needs to be done for the answer A_i. In this case, the initial version numbers of the term added in should be used, instead of the queried version numbers, for subsequent compensation of these newly added tuples in A_i of update $\triangle R_i$. Also since deletion update $\triangle R_j$ can only be an interfering update for those updates that are ordered to occur before it, there is no need to keep the answer A_j of $\triangle R_j$ once it is applied to the view. This is because those updates that are ordered to occur after $\triangle R_j$ will never use it for compensation (as mentioned earlier, the order of refreshing the view has to follow the order of picking the updates for processing to achieve complete consistency).

At the end of the compensation of the answer of an update, the natural join of the individual tuples in the answer is taken, effectively dropping those dangling tuples of the outerjoin operation, before this answer is applied to the view relation.

Theorem 1
Given an update $\triangle R_i$ with $< \alpha_1, ..., \alpha_j, ..., \alpha_n >$ as its initial version numbers, let its queried version numbers be $< s_1, ..., s_j, ..., s_n >$. The interfering updates are removed from its answer by first checking for the presence of such updates in R_j of its answer for j from $i - 1$ to 1 if $i \neq 1$, and then for j from $i + 1$ to n if $i \neq n$. For each R_j in the answer, if $s_j > \alpha_j$, then updates of version numbers $\alpha_j + 1$ to s_j from R_j are the interfering updates (Lemma 1). The effects of these interfering updates (given that none of them are lost and have been received by the view) are removed by first compensating through Lemma 2, 3 or 4 with the updates of version number s_j, and progressing down to the updates of version number $\alpha_j + 1$.

Proof Let $< \alpha_1, ..., \alpha_j, ..., \alpha_n >$ be the initial version numbers for update $\triangle R_i$ where $i \neq j$, and $< s_1, ..., s_j, ..., s_n >$ as the corresponding queried version numbers for its answer A_i. Assuming $s_j > \alpha_j$, compensation of A_i with the updates from R_j of version number s_j brings A_i to the state of $< s_1, ..., s_j - 1, ..., s_n >$. Continuing this process of compensation with all interfering updates will bring A_i to the state as its initial version numbers, i. e., $< \alpha_1, ..., \alpha_j, ..., \alpha_n >$.

We can make the above compensation process more efficient by combining the interfering updates for the compensation of the answer of an update, i. e., collecting all interfering updates into 3 sets of insertion, modification and deletion updates. The version numbers of the interfering updates allow us to combine the updates correctly if there are a few updates on the same tuple. For example, a tuple of value $(1, 2)$ with version number 1 that is inserted, and subsequently modified to $(1, 3)$ with version number 2 can be treated as an interfering insertion update of $(1, 3)$.

So far, our discussion assumes that the next update version of a base relation is available (not lost) for incremental computation or compensation. The detection of potential lost updates is stated in the next Theorem.

Theorem 2 *An update notification of a particular version number from a base relation which has not been received by the view is a potential lost notification if the view has received another update notification with a higher version number from the same base relation, or the compensation process of another update requires the compensation with this version of update notification which has not been received by the view.*

Whenever such possibility is detected, a timer is set of which the expiration will trigger the re-send request to be sent to the base relation concerned. Should this lost update appeared at the view again at a later time, the maintenance machinery can detect such duplicate messages since they would have the same version number.

4.3. Example

We show an example of the working of our compensation algorithm. Consider 3 base relations, $R_1(\underline{A}, B)$, $R_2(\underline{C}, D)$ and $R_3(\underline{E}, F)$. Let R_1 and R_2 reside at the same data source, so that an update transaction can invoke both the relations. The view V is defined as $V \equiv \prod_{A,F} R_1 \bowtie_{B<C} R_2 \bowtie_{D<E} R_3$. We use $< x, y, z >$ to denote the highest processing, initial or queried version numbers of x, y and z for R_1, R_2 and R_3 respectively. Note that queries issued to the base relations are through the natural join operation, but the storing of the answer before it is been applied to the view relation uses the outerjoin operation.

Initial states for the 3 base relations, and the view with highest processing version numbers $< 1, 1, 1 >$ (the *Count* field in V indicates in how many different ways the same tuple can be derived from the given view definition and the base relations):

$R_1(\underline{A}, B)$	$R_2(\underline{C}, D)$	$R_3(\underline{E}, F)$
(1,2)	(3,4)	(5,6)
	(4,4)	(6,6)
Version 1	Version 1	Version 1

$V(A, F, Count)$
(1,6,4)

Update transaction involving deletion of (1,2) from R_1, and deletion of (4,4) from R_2:

$R_1(\underline{A}, B)$	$R_2(\underline{C}, D)$	$R_3(\underline{E}, F)$
	(3,4)	(5,6)
		(6,6)
Version 2	Version 2	Version 1

The view receives the above update transaction notification, separates the updates into 2 groups according to the 2 base relations, and formulates the queries (a) $(1, 2) \bowtie_{B<C} R_2$ to be sent to R_2 (for the purpose of discussion, we call this query $Q1a$) and its initial version numbers are $< 2, 1, 1 >$; (b) Query $Q2a$ with $R_1 \bowtie_{B<C} (4, 4)$ to be sent to R_1 and $Q2b$ with $(4, 4) \bowtie_{D<E} R_3$ to be sent to R_3. Initial version numbers are $< 2, 2, 1 >$.

The view receives the results (1,2,3,4), queried version number 2, from R_2 for query $Q1a$. It formulates another query $Q1b$ of $(3, 4) \bowtie_{D<E} R_3$ to be sent to R_3. For $Q1b$, the view receives (3,4,5,6) and (3,4,6,6), queried version number 1. Thus the view has $A_1 = \{(1, 2, 3, 4, 5, 6), (1, 2, 3, 4, 6, 6)\}$ after the completion of queries $Q1a$ and $Q1b$, with $< 2, 2, 1 >$ as the queried version numbers.

Update transaction involving modification of (6,6) to (6,7), version number 2, on R_3 occurs:

$R_1(\underline{A}, B)$	$R_2(\underline{C}, D)$	$R_3(\underline{E}, F)$
	(3,4)	(5,6)
		(6,7)
Version 2	Version 2	Version 2

For query $Q2a$, the view receives zero tuple from R_1 with queried version number 2. For query $Q2b$, the view receives (4,4,5,6) and (4,4,6,7) from R_3 with queried version number 2. Putting them together, the view has $A_2 = \{(null, null, 4, 4, 5, 6), (null, null, 4, 4, 6, 7)\}$ (using outerjoin), with $< 2, 2, 2 >$ as the queried version numbers.

Update transaction involving insertion of (3,2) into R_1 occurs with version number 3:

$R_1(\underline{A}, B)$	$R_2(\underline{C}, D)$	$R_3(\underline{E}, F)$
(3,2)	(3,4)	(5,6)
		(6,7)
Version 3	Version 2	Version 2

The view receives the insertion update notification of (3,2) from R_1. Query $Q3a$ of $(3, 2) \bowtie_{B<C} R_2$ is sent to R_2. Initial version numbers are stored as $< 3, 2, 1 >$. The view receives (3,2,3,4), queried version number 2, from R_2. It formulates another query $Q3b$ of $(3, 4) \bowtie_{D<E} R_3$ to be sent to R_3. The view receives (3,4,5,6) and (3,4,6,7), queried version number 2, from R_3. Putting the results for $Q3a$ and $Q3b$ together, the view has $A_3 = \{(3, 2, 3, 4, 5, 6), (3, 2, 3, 4, 6, 7)\}$, queried version numbers $< 3, 2, 2 >$.

The view receives the modification update notification from R_3 and formulates the query $Q4a$ of $R_2 \bowtie_{D<E} (6, 6)$ to be sent to R_2. Initial version numbers are $< 3, 2, 2 >$. For $Q4a$, the view receives (3,4,6,6), queried version number 2, from R_2. It formulates query $Q4b$ of $R_1 \bowtie_{B<C} (3, 4)$ to be sent to R_1. The view receives the tuple (3,2,3,4), queried version number 3, from R_1 for query $Q4b$. Putting the results of $Q4a$ and $Q4b$ together, we have $A_4 = \{(3, 2, 3, 4, 6, 6)\}$ with queried version numbers $< 3, 2, 2 >$. No compensation is required as its initial version numbers are also $< 3, 2, 2 >$.

Compensation of answer A_1 with deletion (4,4) from R_2 using $A_1[A, B] *^l A_2[C, D, E, F]$ gives the additional tuples in number 3 and 4 of the table (QVN denotes queried version numbers):

Number	Tuple	QVN
1	$(1, 2, 3, 4, 5, 6)$	$< 2, 2, 1 >$
2	$(1, 2, 3, 4, 6, 6)$	$< 2, 2, 1 >$
3	$(1, 2, 4, 4, 5, 6)$	$< 2, 2, 2 >$
4	$(1, 2, 4, 4, 6, 7)$	$< 2, 2, 2 >$

Tuple number 3 and 4 require further compensation with update version number 2 of R_3. For this compensation with modification update of tuple (6,6) to (6,7), the tuple (1,2,4,4,6,7) (tuple number 4) is changed to (1,2,4,4,6,6), while tuple number 3 is not affected. Using the compensated answer of A_1, 4 tuples of (1,6) need to be removed from the view V, and the view is now empty.

Compensation of answer A_2 with the modification update of (6,6) to (6,7) in R_3 (version 2) changes the tuple $(null, null, 4, 4, 6, 7)$ to $(null, null, 4, 4, 6, 6)$. The compensated answer of A_2 are the tuples $(null, null, 4, 4, 5, 6)$ and $(null, null, 4, 4, 6, 6)$. Since both tuples in A_2 are dangling tuples, the view need not be refreshed.

The compensation of the answer A_3 is similar to that of A_2 and its tuples are $(3, 2, 3, 4, 5, 6)$ and $(3, 2, 3, 4, 6, 6)$ after compensation. Thus A_3 adds 2 tuples of (3,6) to the view. The answer A_4 indicates that 1 tuple of (3,6) in the view should be changed to (3,7). Thus the final view is:

$V(A, F, Count)$
(3,6,1)
(3,7,1)

4.4. Discussion

Unlike the Strobe algorithm, our compensation algorithm does not require a quiescent state of the system before the view can be refreshed. Consider an update $\triangle R_i$ on base relation R_i, with the view definition given in Equation (1).

The initial version numbers of $\triangle R_i$, together with its queried version numbers (before the compensation process begins), give a set of interfering updates for the compensation of its answer. From Lemma 2 and Lemma 3, it is noted that the compensation with both interfering insertion and modification updates does not add new tuples to the answer of $\triangle R_i$. Thus the queried version numbers for all tuples contributed by all relations remain the same.

The case is different for compensation with an interfering deletion update. New tuples could be added into the answer of $\triangle R_i$ as shown in Equations (3) and (4), possibly with a new set of queried version numbers that are larger than the existing corresponding version numbers, implying that extra interfering updates have to be included for consideration over the existing ones. However, such increase will not continue without bound, which would otherwise require a quiescent state of the system before the view can be refreshed.

Consider a deletion update $\triangle R_j$, with $j > i$. Suppose the answer of $\triangle R_i$ needs compensation with $\triangle R_j$. Then $\triangle R_j$ can only add in new set of updates to be considered for compensation for the answer of $\triangle R_i$ from base relations $R_{j+1}, ..., R_n$. Since after compensating with all updates which are from R_j, the next relation to be considered is R_{j+1}, any new set of updates added in for compensation by deletion update $\triangle R_{j+1}$ will have to come from base relations $R_{j+2}, ..., R_n$. Thus when we reach the stage of compensating with updates from R_n, no new update transactions can be added in for compensation. The situation for $j < i$ will be similarly bounded when the compensation process reaches relation R_1.

5. Conclusion

In this paper, we developed a compensation algorithm for resolving the anomalies found in the result of incremental computation used in refreshing the materialized view derived from multiple distributed autonomous data sources. This algorithm caters to the situation where no assumption could be made with regard to the order of arrival of messages at the view maintenance machinery to that of generated at the data sources. Neither do we assume the reliability of the communication network that guarantees the successful delivery of messages. The algorithm does not require a quiescent state of the system before it can be used in updating the view relation. The algorithm handles a modification update as a deletion update, followed by an insertion update, only when the modified attributes involve at least one join attribute, else it is treated as one modification update which is more efficient during the incremental computation and subsequent updating of the materialized view.

References

[1] D. Agrawal, A. E. Abbadi, A. Singh, and T. Yurek. Efficient view maintenance at data warehouse. In *SIGMOD*, May 1997.

[2] R. Chen and W. Meng. Precise detection and proper handling of view maintenance anomalies in a multidatabase environment. In *2nd IFCIS International Conference on Cooperative Information Systems*, June 1997.

[3] L. Colby, T. Griffin, L. Libkin, I. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *SIGMOD*, pages 469–480, June 1996.

[4] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *SIGMOD*, pages 328–339, May 1995.

[5] A. Gupta, I. Mumick, and V. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, pages 157–166, May 1993.

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

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

[8] X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. *IEEE Transactions on Knowledge and Data Engineering*, pages 337–341, Sept. 1991.

[9] D. Quass, A. Gupta, I. Mumick, and J. Widom. Making views self-maintainable for data warehousing. In *Proceedings of the Conference on Parallel and Distributed Information Systems*, Dec. 1996.

[10] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *SIGMOD*, pages 316–327, May 1995.

[11] Y. Zhuge, H. Garcia-Molina, and J. Wiener. Consistency algorithms for multi-source warehouse view maintenance. *Journal of Distributed and Parallel Databases*, pages 7–40, Jan. 1998.