Incremental Maintenance of Multi-Source Views

Gianluca Moro
DEIS - University of Bologna
Via Rasi e Spinelli, 176
I-47023 Cesena (FC)
gmoro@deis.unibo.it

Claudio Sartori
DEIS - University of Bologna
CSITE-CNR, Viale Risorgimento, 2
I-40136 Bologna
csartori@deis.unibo.it

Abstract

In recent years, numerous algorithms have been proposed for incremental view maintenance of data warehouses. As a matter of fact, all of them follow almost the same general approach, namely they compute the change of a multi-source view in response to an update message from a data source following two steps: (i) issue a set of queries against the other data sources and (ii) compensate the query result due to concurrent updates interfering with the first step. Despite many recent improvements, the compensation approach needs precise detection of interfering updates occurring remotely in autonomous data sources, and the assumption that messages are never lost and are delivered in the order in which they are sent. However in real networks loss and misordering of messages are usual. In this paper we propose a maintenance algorithm that does not need the compensation step and applies to general view expressions of the bag algebra, without limit on the number of base relations per data source.

1 Introduction

Materialized views are a common way to improve performances in the execution of queries against views, the improvement depending on the frequency of the query execution. On the other hand, a materialized view has to be updated when the base tables on which it is defined are updated. Many works [2], [3], [4] argued that incremental recomputation of views can be used to significantly improve the performances of the maintaining activity.

At present there are many solutions, in particular, the major interest is when the data sources are independent, as is the most frequent case for data warehouses. In particular, a popular method is that of compensation, which goes through an iterative process to deal with updates, taking into account concurrent updates in the data sources. The correctness of compensation relies on the reliability of the network, but we prove that this is still not enough, and in section 4 we give an example where compensation fails.

Different maintenance methods have been proposed for centralized environments, based on a set of change propagation rules. The objective of this paper is to overcome the limitations of compensation algorithms and to define an effective set of change propagation rules in data warehouse environment.

The structure of the paper is the following. Section 2 explains the assumptions and the motivations for our data warehouse model. Section 3 discusses previous approaches to incremental view maintenance and provides a background on the problem. Section 4 points out the problems that can occur with compensation. Section 5 motivates and describes our algorithm and its approach to view maintenance and, finally, section 6 discusses some possible evolutions of this work.

2 Data Warehouse Model Assumptions

We consider the case of relational data sources containing one or more base relations. The materialized views are defined by query expressions of the bag algebra [11, 15], namely relational algebra with duplicates, and each views can involve in general one or more base relations belonging to one or more data sources.

We assume the data sources do not know the existence of other data sources, and transactions do not span across multiple data sources, in other words the data sources are autonomous. Such a choice is classified in the data warehouse model proposed by [25] as *source local transactions* performing multi-update operations atomically; this choice has been adopted in most of the works on data warehouse view maintenance [1, 8, 13, 14, 16, 20, 22, 23, 24, 26].

The updates to the base relations are assumed to be insertions and deletion of tuples while an update is modeled as a delete followed by an insert[1].

The updates at the data sources can be handled in different ways, conceptually we assume, as all previous works, that when a base table R_i is involved in a view definition, two delta tables $\blacktriangle R_i$ and $\blacktriangledown R_i$ storing respectively insertions and deletions on R_i, are available into the same source of R_i.

Depending on how the updates are incorporated into the view at the data warehouse, different notions of view consistency, with increasing effectiveness, have been identified in [24] as follows:

- *Convergence*, where the updates are eventually incorporated into the materialized view.

[1]At the best of our knowledge only the algorithm in [14] considers modification as one type of update other than deletion and insertion.

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

Commercially available data warehouse products, such as Redbrick [21], Oracle [18] and others ensure convergence. Several authors recognize that complete consistency is almost useless for practical situations and moreover its computation and communication costs are higher with respect to the two preceding levels. The algorithm described in this paper ensures strong consistency, but could be easily extended to obtain complete consistency too.

For what concerns the communications between the data warehouse and each data source, we assume to work with usual networks with capabilities to support traditional client-server systems. A study carried out by [6] shows that 1 percent of all messages are delivered and received in different orders when a local area network is heavily loaded; this means that misordering and loss of messages in a wide area network is surely even greater. The above mentioned algorithms, except [14], need reliable and FIFO communication channels in order to work correctly.

3 Previous Approaches for Incremental View Maintenance

When one or more base tables, on which a materialized view is based, change, the materialized view has to be updated too. The incremental approach, differently from the complete recomputation, aims to reuse the previous contents by computing only the delta, namely new tuples to be added and to be deleted.

Many works on incremental view maintenance [5, 9, 12, 19] do not apply in a multi-source environment because they deal with views and tables in the same source. In fact, in a multi-source environment a view can be defined on remote base tables belonging to different data sources, possibly autonomous, and the view can be in a different site with respect to the same data sources.

Several algorithms have also been developed for incremental view maintenance in a multi-source scenario, which is typical for data warehouses [1, 8, 10, 13, 14, 16, 23, 24, 25, 26]. These works provide a spectrum of solutions ranging in a multidimensional space, depending on the different capabilities they offer and the different operating constraints they require. However, all of them follow almost the same general approach: when the data warehouse receives an update message from a data source 1) it generates incremental maintenance queries to be computed at the other data source sites and 2) since the results of such queries can contain *error terms*, namely anomalies due to concurrent updates interfering with the first step, it generates compensation queries. Compensation is in general expensive and requires additional complicated tasks, such as: precise detection of interfering updates and further round trip of queries.

Some algorithms issue a number of compensation queries which can be quadratic [24] or factorial [25] in the number of data sources. Some others reduce the number of compensation queries by performing part of the compensation *locally* at the data warehouse [1, 6, 14, 23], by imposing several restrictions or by requiring period of quiescence in the data warehouse [26].

The local compensation is not applicable in general. In particular, it does not work when a compensation query contains at least a join between two base tables. This implies that the local compensation is not applicable when a view is defined over at least two base tables belonging to the same data source.

In general such a limitation prevents the maintenance of views which are defined over n base tables belonging to m data sources where $m < n$. As shown in [16], further restrictions are needed, for instance several kind of self-join are not allowed, cyclic queries included, and each compensation query must be in the form of a conjunctive query containing only one predicate.

The maintenance processing algorithms presented in [1, 7, 14, 23, 25, 26] assume that each view at the data warehouse is defined on base tables which are all situated on different data sources. In other words if a view involves n base tables then n data sources are involved too. This means that if two or more base tables of a view are in the same data source, the above algorithms have to consider such base tables as belonging to different fictitious sources.

The consequence is the following: let us suppose to have a view V defined as a simple join over $n + 1$ base tables belonging to m data sources (where $m < n + 1$), the maintenance query of V is anyway broken down in n subqueries which give rise at least to n remote subquery executions and n distinct replies. On the contrary a better strategy could generate only m subqueries, saving therefore $n - m$ subqueries and $n - m$ replies via network.

[8] propose to circumvent the problem by adding a further software layer between the data warehouse and the data sources, which treats a data source composed of multiple base relations as one local view so that the data warehouse will be aware of this one view relation only. This introduces further delays because it requires to intercept all queries sent by the data warehouse and all updates coming from the data sources in order to opportunely map them on the data sources and on the data warehouse respectively.

[24] deals with only one data source, and [6] is limited to two base tables, so the above consideration is not applicable in such cases.

All the algorithms, including the most recent ones in [1, 7, 14, 23], are limited to relational select-project-join views and assume that base tables do not contain duplicates. [16] propose a maintenance framework, which operates similarly to [1] and [23] and, in addition, sketches out the maintenance of views with group by having and aggregate functions as a further processing at the end of the view maintenance.

Almost all of the algorithms, except [14], work correctly if they are able to identify each interfering update. To detect an interfering update they assume that the network delivers the messages in the same order they are sent, without any loss.

On the other hand, we are able to prove that such assumptions in general are not sufficient to detect interfering updates. The example in section 4 shows that two kinds of errors are anyway possible: interfering updates not detected and updates erroneously detected as interfering updates.

Time	Transaction 1	Transaction 2	Transaction 3
T0	*The DW forwards the query Q for retrieving tx and ty - initial value of the two tuples are tx(1) and ty(2)*		
T1	• Locks tx • Updates tx setting x = 10		
T2		Computation of the query Q • Reads ty(2) • Tries to read tx • Waiting for the unlock of tx	
T3		• Waiting	• locks ty • updates ty setting y = 20
T4		• Waiting	Commit (and unlock ty)
T5		• Waiting	Sends notification of ty(20) *Erroneous detection as interfering update*
T6	Commit (and unlock tx)	• Waiting	
T7		• reads tx(10)	
T8		Atomic answer to the DW query Q = {tx(10), ty(2)}	
T9		Commit	
T10	Sends notification of tx(10) *Interfering update not detected*		

Figure 1. An interfering update not detected and an erroneous detection as interfering update

4 Problems with Interfering Updates Detection

Compensation algorithms are supposed to work well if they are able to correctly detect interfering updates. They claim that interfering updates can always be detected if networks do not lose messages and messages are received in the same order in which they are sent (FIFO delivery).

The way used to detect interfering updates is the following: let δR_j be an update happened after an update δR_i $(i \neq j)$, δR_j is considered an interfering update on the result of incremental maintenance query of δR_i if δR_j occurs before the view maintenance sub-query of δR_i is processed by the data source. In general all of the compensation algorithms above mentioned follow this principle to detect interfering updates.

Here we show a simple example which highlights that interfering updates sometimes cannot be detected although networks are reliable and support FIFO message delivery. In other words network reliability and FIFO message delivery are necessary but not sufficient conditions.

Let TR1, TR2 and TR3 be three concurrent transactions on the same data source S_i as illustrated in the table of figure 1. TR1 and TR3 are transactions which update a base table R_i belonging to S_i, while TR2 represents an incremental query or subquery Q issued by the data warehouse in response to a given update δR_n from any source S_n $(n \neq i)$. For simplicity we assume $R_i(X)$ as the schema of R_i where X is an integer attribute, and let tx(1), ty(2) be two tuples whose values are $X = 1$ and $X = 2$ respectively.

Algorithms above do not block update transactions on any data source and assume an atomic answer from a data source query. Therefore the isolation levels of TR2 must avoid to block the update transactions at the data source (superior bound), and at the same time it must guarantee that the data warehouse reads only committed values (inferior bound). This level corresponds exactly to read committed, it does not put any lock when reads tuples and is supported by commercial DBMSs. A different level, lower or higher, would not be suitable or could be not available in the data sources. In a real application the other two transactions can adopt the same isolation level of TR2 or a higher level one, anyway it does not affect the problem.

As we can see the data warehouse forwards the query Q at time T0. The notification update ty(20) is sent on the network at time T5, while the answer of the query Q is replied at time T8, therefore due to FIFO delivery update ty(20) is received at the data warehouse before the answer of Q. The data warehouse detects such update as an interfering update. But it is not an interfering update because the answer of query Q contains the old value of ty, namely ty(2). The data warehouse will erroneously detects such update as interfering update to be compensated.

On the contrary, the update tx at time T1, which is a real interfering update because it is included in the answer of Q, will not be detected. In fact the answer of query Q is replied to the data warehouse at time T8, while the interfering update tx is sent at time T10. Again, due to FIFO delivery the update tx reaches the data warehouse after the answer of Q, therefore the data warehouse will not perform the necessary compensation.

5 Our Approach for Multi-Source View Maintenance

The root of these problems is mainly due to the impossibility to execute atomically an incremental maintenance query, which is in general a distributed query over autonomous data sources continuously changing their state. The ideal scenario would be that where the data warehouse, when receives an update from a data source, performs the correspondent distributed maintenance query before the other data sources change their state. This is the same as saying that the distributed query is to be executed within a well precise *distributed* state made up by the set of such data sources' states. In previous approaches the answers to the subqueries of the maintenance query contain errors because they cannot be executed within the same state. The compensation is a way to achieve correct results by rebuilding the above-mentioned set of states a-posteriori, that is after the execution of the subqueries.

Conversely our approach makes available such a set of states a-priori, namely before the maintenance query execution, allowing the parts of the maintenance query to be computed within the same state. In this way the view maintenance occurs without interfering updates as such concurrent updates will belong to a subsequent state. This implies that query results do not contain errors and that the compensation is not required, including all additional tasks and unpractical requirements they need to work.

Obviously, in order to make available such a set of states, we cannot assume any kind of coordination among the data sources, as they are in general autonomous and heterogeneous. On the other hand we cannot think out solutions which block or interfere with update transactions of the data sources.

As said in section 2, almost all described algorithms, included our solution, assume to work with relational data sources supporting local transactions and delta tables, which keep trace of the differences by storing delta tuples, that is old values of the tuples modified. A data source changes its state each time a local transaction commits, therefore there is a one-to-one mapping between states and transactions. As we will show in the following, if delta tuples are labeled with the identifier of the transaction that generates them, it becomes possible to isolate data source states during the execution of concurrent transactions.

On the other hand, the availability of transaction identifiers is a common feature available since several years on most popular commercial DBMSs (i.e. Oracle7 [17] allows a trigger to get the transaction identifier that fires it).

The algorithm works without the need to receive update notifications from the data sources, indeed problems of loss or misordering of such notifications do not subsist. Moreover the solution allows dealing with both set of states, namely set of update transactions, and with an update transaction at a time. In other words it can ensure strong and complete consistency. As anticipated in section 2, complete consistency introduces inefficiency and it is not scalable, indeed the following subsections concentrate in describing the algorithm ensuring strong consistency.

As shown in section 2, most of the algorithms are limited to relational select-project-join views and in general assume only one base relation per data source without duplicates. The solution proposed in this paper allows bag algebra select-project-join view expressions, no limit on the number of base relations per data source and each base relation may contain duplicates thanks to semantics for duplicates of bag algebra operators. Subsection 5.2 shows details on this argument.

5.1 Basic Idea

Thanks to the capability to isolate data source states, the algorithm eliminates the problem of interfering updates by isolating, for each data source, the state in which the algorithm begins the maintenance, from the new ongoing states that each data source manifests due to concurrent updates. This allows to compute maintenance queries over a precise and immutable set of states.

Each maintenance query is derived by using change propagation rules that basically transform the query expression of the view in order to minimize costs of computing the tuples to be added to and deleted. Change propagation rules for centralized database has been identified by [9, 19]. These rules work correctly only in pre-update state and without concurrency, namely within the same transaction generating the changes. As it will be shown in subsection 5.3 these rules have been improved by [5] in order to work also in post-update state using a distinct transaction. However it will be shown that they are not suitable for a multi-source environment and in subsection 5.4 we will give change propagation rules for data warehouse.

The algorithm works executing two general phases: configuration and maintenance.

The configuration phase is executed only at view-definition-time or when the view definitions change. Basically it consists of applying change propagation rules to each view expression Q in order to achieve two expressions that ensure the incremental maintenance of Q.

Basically, the maintenance phase consists of four steps:

1. executes a query over each data source which retrieves from delta tables the identifiers of all committed transactions which have terminated since last maintenance execution; these queries can be executed in parallel;
2. performs the incremental maintenance of each view by evaluating its incremental expression derived in configuration phase; transaction identifiers allow to execute incremental queries over a state isolated from new ongoing states; this step can occur by using a single process or a process per each view;
3. installs the changes into each view concurrently with data warehouse readers preserving, at the same time, session consistency of readers;
4. deletes from delta tables the tuples used by the maintenance.

The maintenance phase can be executed periodically or continuously, depending on:

- the minimum time of view refreshment one needs
- the computational power of local and remote resources, including the network bandwidth
- how large the data warehouse is.

5.2 Bag Algebra Allows Data Source Tables with Duplicates

A bag (or multiset) X is like a set, except that multiple occurrences of elements are allowed. An element x is said to have multiplicity n in the bag X if X contains exactly n copies of x. The notation $x \in X$ means that x has multiplicity $n > 0$ in X, and $x \notin X$ means that x has multiplicity 0 in X. Our query language will be the bag algebra of [11, 15], restricted to flat bags (i.e. bags of tuples, no bag-valued attributes). SQL statements such as *select distinct* and *select average of column* can be better explained with the bag algebra rather than sets algebra. Let p range over quantifier-free predicates, and A range over sets of attribute names. The following grammar generates bag algebra expressions:

$Q ::= \emptyset$	Empty Bag
$\mid R$	Table name
$\mid \sigma_p(Q)$	Selection
$\mid \Pi_A$	Projection
$\mid Q_1 \uplus Q_2$	Additive union
$\mid Q_1 \dot{-} Q_2$	Monus
$\mid \epsilon(Q)$	Duplicate elimination
$\mid Q_1 \times Q_2$	Cartesian Product

We will use the symbol Q, Q_1, Q_2, E to denote bag algebra expressions, also called queries. Monus is the only operator which may require a short description. If x occurs n times in Q_1 and m times in Q_2, then the number of occurrences of x in $Q_1 \dot{-} Q_2$ is the maximum of 0 and n - m.

The join operator is derived as in the relational algebra, the difference is that here it works with duplicates too.

We allow defining view using selection, projection and join (SPJ), however all operators, except for duplicate elimination and cartesian product, are used by the algorithm to carry out the incremental maintenance process. It could seem that the expressive power of our approach be equivalent to algorithms that allow to define SPJ views using relational algebra operators along with counting mechanisms for dealing with duplicates. This is not true, in fact algorithms based on relational operators deals only with duplicates of materialized views at the data warehouse and they are obliged to assume that none base table contains duplicate because relational operators could not deal with them. Our algorithm allow duplicates both in base tables and in materialized view at the data warehouse.

To simplify the notation we will use in the rest of the paper $\cup$ and $-$ intended as $\uplus$ and $\dot{-}$ respectively.

5.3 Change Propagation Rules for a Centralized Database

The incremental maintenance of a materialized view MV, defined by a query Q, aims to update MV with respect to changes that transactions perform on the base tables of MV and consists of computing two sets (or rather multiset) of tuples:

- ΔMV, which denotes the set of tuples to be added to MV
- ∇MV, which denotes the set of tuples to be deleted from MV

therefore, if we denote the past content of MV as MV^p, the new content of MV updated with respect to the current state, denoted as MV^c, will be given by:

$$MV^c = (MV^p - \nabla MV) \cup \Delta MV$$

Griffin et al. identified in [9] change propagation rules in order to derive the incremental expressions $\Delta(Q,\mathrm{T})$ and $\nabla(Q,\mathrm{T})$ that compute respectively ΔMV and ∇MV from a given view expression Q and from insertions and deletions that a single transaction T wants to perform.

These rules are suitable for centralized database because they derive expressions which produce correct results when the base tables of the view are all in the same database and when the expressions are evaluated *immediately* in *pre-update* state. Immediately means to evaluate each expression within the same transaction performing the updates, while pre-update state means that base tables are available in the state where changes have not yet been applied.

Griffin et al. generalized previous rules [5] in order to also derive expressions suitable for deferred evaluation which allows view maintenance using a distinct transaction with respect to those performing base table updates. However the approach cannot be employed for data warehouse view maintenance due to the following main reasons:

1. it requires that base tables and views are in the same database
2. for each view Q it needs to atomically execute both incremental query expressions $\Delta(Q,\mathrm{T})$ and $\nabla(Q,\mathrm{T})$
3. if the database state changes during the incremental maintenance of several views, the up-to-date views will not reflect the same database state.

On the other hand in a data warehouse environment (1) base tables reside in several data sources, (2) the incremental query expressions cannot be executed atomically because they are subdivided in subqueries to be evaluated at different data source sites, (3) the user session consistency needs to be preserved. Two words on this point. When two views, which are defined for instance over base tables of the same source, are not updated with respect to the same database state, they give rise to inconsistent data analysis due to the two different updating states they reflect. Moreover the problem is even more evident when different views share a set of base tables belonging to different data sources (or to a single source). In fact in this case the views can also reflect contradictory data.

In the following subsection we will give solutions for previous points focusing on views defined by select, project and join operators of the bag algebra.

5.4 Change Propagation Rules for Data Warehouse Environment

A common root causes the problems described in previous subsection. For what concerns points 2 and 3, previous approach assumes that the database state does not change during the whole incremental maintenance or alternatively it assume to lock everything for the whole duration of the task. In effect under this hypothesis the solution can be modeled by only referencing two states: *past* and *current* state, respectively the views' state before updating and the views' state after updating, where *current* correspond to the last database state.

However real life applications assume that a database may frequently change its state due to concurrent transactions and they assume a minimum lock usage in order to favor the parallelisms. Under this hypothesis two states are not more sufficient, in fact the *current* state cannot be referenced as the final state (or the updating state) because the database may continuously change producing ongoing current states *current*+1, *current*+2 ... that we denote

Q	$\text{ADD}(Q) \overset{\text{def}}{=}$
R	$\blacktriangle R^o$ (delta table in state *old* containing tuples to be added to base table R^p in state *past*)
$\sigma_p(E)$	$\sigma_p(\text{ADD}(E))$
$\Pi_A(E)$	$\Pi_A(\text{ADD}(E))$
$E \bowtie F$	$(\text{ADD}(E) \bowtie_{ef} \text{ADD}(F)) \cup_{dw} (\text{ADD}(E) \bowtie_f (F^o -_f \text{ADD}(F))) \cup_{dw} ((E^o -_e \text{ADD}(E)) \bowtie_e \text{ADD}(F))$

Q	$\text{DEL}(Q) \overset{\text{def}}{=}$
R	$\blacktriangledown R^o$ (delta table in state *old* containing tuples to be deleted from base table R^p in state *past*)
$\sigma_p(E)$	$\sigma_p(\text{DEL}(E))$
$\Pi_A(E)$	$\Pi_A(\text{DEL}(E))$
$E \bowtie F$	$(\text{DEL}(E) \bowtie_{ef} \text{DEL}(F)) \cup_{dw} (\text{DEL}(E) \bowtie_f (F^o -_f \text{ADD}(F))) \cup_{dw} ((E^o -_e \text{ADD}(E)) \bowtie_e \text{DEL}(F))$

Figure 2. Recursive Functions ADD(Q) and DEL(Q) for Data Warehouse

as c^*. Therefore we need to isolate and reference a stable intermediate state, that we call *old*, between *past* and *current* states, namely in short $p < o < c^*$.

Remodeling the problem using these states the incremental maintenance of a view MV defined by a query Q becomes as follows:

$$MV^o = (MV^p - \nabla MV^o) \cup \Delta MV^o$$

where

$$\nabla MV^o = DEL(Q)$$

and

$$\Delta MV^o = ADD(Q)$$

To update MV^o (at the state old) it is only needed to compute DEL(Q) and ADD(Q) as MV^p is already available at the data warehouse. DEL(Q) and ADD(Q) are recursive functions (figure 2) for deriving the incremental maintenance expressions of a query Q. In other words, they allow to derive two incremental expressions per each view which will be used to compute respectively the tuple to be added to and deleted from each view. DEL(Q) and ADD(Q) are defined by the change propagation rules given in the table of figure 2.

Each binary operator in figure 2, such as join, additive union and monus, is labeled with the data source name where the operation must be performed. In particular e means to execute the operation at the source of base table E, indeed f at the source of the base table F and dw at the data warehouse. When the label of an operation contains both e,f means that the operation can be executed alternatively at the first or second source. Unary operators are executed at the source of the base table they involve.

This labeling resolve the first problem described in previous subsection as it allows derived incremental expressions to be also evaluated in a distributed environment such as a data warehouse with several data source sites.

Now we focus on second and third problem also described in previous subsection; in particular we explain how above change propagation rules allow to derive incremental expressions which correctly update views, namely ensuring that they will reflect an unique state per each data source site.

First of all let us explain some symbols of figure 2. $\blacktriangle R^o$ and $\blacktriangledown R^o$ are delta tables in state old, namely containing all of the tuples that have been modified in table R between states past and old, in particular the first contains tuples that have been added to R, while the second contains tuples that have been deleted from R. Here we are mainly focusing on the model of change propagation rules, in subsection 5.6 it will be shown a technique based on transaction identifiers that allows to access $\blacktriangle R^o$ and $\blacktriangledown R^o$ in such a state.

E^o and F^o are base tables to be accessed in the state old , but base tables are only available in current state unless it is possible to alter their schema for obtaining a sort of versioning. In general we have to assume that base tables are untouchable in order to ensure the correct functioning of update transactions and applications that use the database.

Nevertheless we can obtain base tables in old state by resolving for R^p in the equation:

$$R^o = (R^p - \blacktriangledown R^o) \cup \blacktriangle R^o \qquad (1)$$

since $\blacktriangle R^{c*}$, $\blacktriangledown R^{c*}$ contain the tuples that have been modified in R between state p and c^*, R^p can always be obtained, independently from the last current state c^*, by executing atomically the following expression:

$$R^p = (R^{c*} - \blacktriangle R^{c*}) \cup \blacktriangledown R^{c*} \qquad (2)$$

finally by replacing 2 in 1 we obtain 3, namely R^o

$$R^o = (((R^{c*} - \blacktriangle R^{c*}) \cup \blacktriangledown R^{c*}) - \blacktriangledown R^o) \cup \blacktriangle R^o \qquad (3)$$

The functions defined in figure 2 obviously apply to complex expressions containing more than two tables. For instance consider the following query:

$$Q \overset{def}{=} E \bowtie F \bowtie G$$

the incremental maintenance expression that computes the tuples to be added to Q is recursively derived as follows:

$$ADD(Q) \overset{def}{=} ADD(E \bowtie F) \bowtie ADD(F \bowtie G)$$

Likewise we can derive DEL(Q).

5.5 The Algorithm: Configuration Phase

The algorithm is quite simply thanks to the change propagation rules modeled in previous subsection. This phase is only performed at views definition-time or when view definitions change.

It consists of deriving for each view definition Q its incremental maintenance expressions ΔQ and ∇Q by applying respectively the recursive functions ADD(Q) and DEL(Q) (see section 5.4).

In this phase are created at data sources delta tables $\blacktriangledown R_i$, $\blacktriangle R_i$ per each base table R_i involved by Q, including triggers that insert into delta tables the base table tuples modified by update transactions. Obviously we only have two delta tables per each base table independently from the number of views at the data warehouse.

Delta tables $\blacktriangledown R_i$, $\blacktriangle R_i$ have the same schema of the correspondent base table R_i, plus an attribute for storing the transaction identifier (TID) that generates the delta tuple. A trigger is a very common mechanism available on all commercial DBMSs and typically it access the TID of the transaction that fires it as an environment variable or by invoking a function.

5.6 The Algorithm: Maintenance Phase

The maintenance phase is executed periodically or continuously as explained at the end of subsection 5.1 and each execution is called *session*. It consists of four steps.

Step 1. Querying TIDs from delta tables of each data source; these queries are executed using *read committed*[2] isolation level allowing to read only committed data and this implies only TIDs of committed transactions. This step allow to isolate and to fix the state *old* (see subsection 5.4).

The couple (TID_i, DS_j) denotes the identifier of an update transaction i performed at the data source j and TID(*old*) = $\{...,(TID_i, DS_j),...\}$ is the set containing all the couple (TID, DS) belonging to the state *old*.

Step 2. Evaluating ΔQ and ∇Q of each view Q and storing results at the data warehouse. This step can be executed using different level of parallelism ranging from a single process up to a process per each ΔQ and ∇Q of each view.

Step 3. Installing the maintenance results of previous step in data warehouse views by using a single transaction; this ensures user session consistency as all views will be made available to user in the new up-to-date state at the end of the whole installing. The installing of tuples will be executed by using an isolation level that does not require quiescence in the system and without blocking reading transactions of the data warehouse users. Moreover this isolation level should ensure to each user to access the data warehouse always in the same state within a user session although in the meantime the data warehouse completes one or more maintenance sessions. This isolation level is already available since some years in most popular commercial DBMSs (e.g. [18]) and it is referenced with the term *versioning*. The term probably derives from some similarity with versioning in temporal databases. As the above technology already resolve the problem effectively, we believe that proposing here other mechanisms would go beyond the scope of this paper.

Step 4. Deleting from delta tables all the delta tuples that contain a TID $\in$ TID(*old*), namely all the delta tuples handled by the current maintenance session.

TID(*old*) and each ΔQ and ∇Q are stored in the data warehouse until the end of a session in order to ensure a consistent recovery of the data warehouse in case of failing of the maintenance phase, due for instance to software, hardware or network failures.

[2]read committed is the minimal isolation level available on all commercial DBMSs

6 Conclusions

We presented an algorithm for incremental view maintenance in data warehouse environment with multiple independent data sources. The data model is quite powerful and includes SPJ relational views with bag algebra semantics, strong consistency is guaranteed. The algorithm is based on a set of change propagation rules and avoids the problems of compensation-based algorithms.

This work can evolve in several directions. One possibility is to deal with general bag algebra views and provide complete consistency. On the implementation side, it could be possible to find out maintenance sub-expressions which are common to different materialized views and evaluate them only once, obtaining cost reduction on global maintenance. Another possibility for increasing the efficiency is to perform a pipeline execution of several maintenance sessions working in parallel.

References

[1] D. Agrawal, A. El Abbadi, A. Singh and T. Yurek. Efficient View Maintenance at Data Warehouses. *In proceedings of the 1997 ACM international conference on Management of Data*, pages 417-427, May 1997

[2] J. A. Blakeley, P. A. Larson, F.W. Tompa. Efficiently updating Materialized Views. *In proceedings of SIGMOD*, pages, 61-71, 1986

[3] L. Baekgaard, L. Mark. Incremental Computation of Nested Relational Query Expression. *ACM Transaction on Database Systems*, vol. 20, no. 2, 1995

[4] L. Baekgaard, L. Mark. Incremental Computation of Set Difference Views. *IEEE Transaction on Knowledge and Data Engineering*, vol. 9, No. 2, March-April 1997

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

[6] R. Chen and W. Meng. Efficient View maintenance in a multidatabase environment. *In proceedings of the Fifth International Conference on Database Systems for Advanced Applications*, pages 391-400, April 1997

[7] 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

[8] L. Ding, X. Zhang, E. A. Rudensteiner, The MRE Wrapper approach: Enabling Incremental View Maintenance of Data Warehouses defined on Multi-Relation Information Sources. *In proceedings of DOLAP*, pages 292-299, 1999

[9] T. Griffin, L. Libkin. Incremental maintenance of views with duplicates. *In proceedings of SIGMOD*, pages 328-339, May 1995

[10] H. Garcia-Molina, W. J. Labio, J. L. Wiener, Y. Zhuge. Distributed and Parallel Computing Issues in Data Warehousing. *In proceedings of the seventeenth annual ACM symposium on Principle of distributed computing*. Invited Talk. June-July 1999. http://www-db.stanford.edu/pub/papers/distrib.ps

[11] S. Grumbach and T. Milo. Towards tractable algebras for bags. *In proceedings of the Twelfth ACM Symposium on Principles of Database Systems*, pages 49-58, May 1993

[12] A. Gupta, I. Mumick, V. Subrahmanian. Maintaining views incrementally. *In proceedings of SIGMOD*, pages 157-166, May 1993

[13] Sachin Kulkarni, Mukesh K. Mohania. Concurrent Maintenance of Views Using Multiple Versions. *In proceedings of International Database Engineering and Applications Symposium*, pages 254-259, August 1999

[14] T. W. Ling and K. Sze. Materialized view Maintenance using version numbers. *In proceedings of the Sixth International Conference on Database Systems for Advanced Applications*, pages 263-270, April 1999

[15] L. Libkin and L. Wong. Some properties of query languages for bags. *In proceedings of the Fourth International Workshop on Database Programming Languages - Object Models and Languages*, August-September 1993

[16] K. O'Gorman, D. Agrawal, A. E. Abbadi. Posse: A Framework for Optimizing Incremental View Maintenance at Data Warehouse. *Technical Report TRCS99-18, University of California at Santa Barbara, Department of Computer Science, UCSB*, Santa Barbara, CA, 93106, June 1999

[17] ORACLE corp., *Oracle 7 server application developer's guide*, No. 6695-70, 1992;

[18] ORACLE corp., *Oracle 7.3 concept server guide*, No. A32535-1, 1996;

[19] X. Qian, G. Wiederhold. Incremental recomputation of active relational expressions. *In IEEE Transaction on Knowledge and Data Engineering*, pages 337-341, September 1991.

[20] Dallan Quass, Jennifer Widom. On-Line Warehouse View Maintenance. *In proceedings of SIGMOD Conference*, pages 393-404, May 1997

[21] RBS. *Data Warehouse applications*. Red Brick Systems, 1998

[22] Sunil Samtani, Vijay Kumar, Mukesh K. Mohania. Self Maintenance of Multiple Views in Data Warehousing. *In proceedings of ACM International Conference on Information and Knowledge Management*, CIKM, pages 292-299, November 1999

[23] X. Zhang, L. Ding and E. A. Rundensteiner. PSWEEP: Parallel View Maintenance Under Concurrent Data Updates of Distributed Sources. *Technical Report WPI-CS-TR-99-14, Worcester Polytechnic Institute, Dept. of Computer Science*, 1999.

[24] Yue Zhuge, Hector Garcia-Molina, Joachim Hammer, Jennifer Widom. View Maintenance in a Warehousing Environment. *In proceedings of SIGMOD Conference*, pages 316-327, May 1995

[25] Yue Zhuge, Hector Garcia-Molina and Janet L. Wiener. The Strobe Algorithms for Multi-sources Warehouse Consistency. *In proceedings of the Fourth International Conference on Parallel and Distributed Information Systems*, IEEE Computer Society, December 1996.

[26] Yue Zhuge, Hector Garcia-Molina and Janet L. Wiener. Consistency Algorithms for Multi-source Warehouse View Maintenance. *In Distributed and Parallel Databases* 6(1): 7-40 (1998)