Staggered Maintenance of Multiple Views

Latha S. Colby
Bell Laboratories
latha.colby@bell-labs.com

Inderpal Singh Mumick
Bell Laboratories
mumick@research.att.com

Abstract

A materialized view needs to be maintained in response to changes to the tables used to derive the view. Incremental maintenance is typically done in two steps – (1) compute the changes to the materialized view, and (2) refresh the materialized view by applying these changes.

In this paper, we show that it is possible to interleave change computation of one view with refresh of other views so that the changes can be computed more efficiently than would be possible if the two steps were to occur sequentially in either order. The algebraic equations to compute the changes to a view are simplest when written in terms of both the pre-update and post-update states of the deriving tables. By interleaving the refresh and change computation steps, we can guarantee that a table is always in the *correct* state during change computation. Such an interleaving corresponds to a staggered evaluation of the algebraic equations defining the changes – hence the technique is called the *staggered maintenance algorithm.*

We permit multiple materialized views to be defined from base tables and other materialized views using select, project, join, union, and the SQL Except operations. We use a model where views may be maintained in an immediate, deferred, or snapshot manner and where we may have no control on when the base tables are updated by user transactions.

1 Introduction

We are interested in efficient maintenance of materialized views in a system that supports multiple materialized views with varied maintenance policies.

A materialized view can be maintained with different maintenance policies. Immediate maintenance involves updating the view within the same transaction that updates its deriving tables. So, an immediate view is always consistent with its deriving tables. In contrast, a deferred view is allowed to become inconsistent with its deriving tables and is refreshed when the view is queried. A view may also be refreshed periodically; we call such a view a snapshot view.

Incremental maintenance of a materialized view V involves (1) computing the change set ΔV (sometimes called the differential table), and (2) applying the changes represented in ΔV to the materialized view. Step (1) will be called *propagate* in this paper, and step (2) will be called *refresh.*

Consider a materialized view V derived from tables R_1 and R_2. Algebraic equations are given in [BLT86, CGL$^+$96, GL95, GMS93, QW91] to compute the change set for V using the change sets of the tables R_1 and R_2, and values of the tables R_1 and R_2. An important issue is the state in which the values of R_1 and R_2 are available – a *pre-update* state, before the changes are applied to R_1 and R_2, or a *post-update* state, after the changes are applied to R_1 and R_2. [CGL$^+$96] uses the post-update state for deferred (and snapshot) maintenance while [BLT86, GL95, QW91] use the pre-update state in the context of immediate maintenance. [GMS93] uses simplified expressions based on both the pre-update and post-update states of R_1 and R_2.

Problem Statement

We consider views defined from (multiset-based) select, project, join, union and except operations. Several formats have been proposed to represent the change set of a table. For instance, [CGL$^+$96, GL95, QW91] use a set/bag of inserted tuples, and a separate set/bag of deleted tuples. We will use the representation of [GMS93], where we keep one bag of tuples along with counts. A negative count indicates that a tuple is deleted; a positive count indicates that a tuple is inserted.

Let V be a select-project-join view defined as $V \stackrel{def}{=} R_1 \bowtie R_2$. Suppose that ΔR_1 and ΔR_2 are the change sets that are to be applied to R_1 and R_2, respectively. The change sets will be assumed to be strongly minimal [GL95], that is, a tuple cannot be deleted from a relation unless it was present in the relation, and the same tuple cannot be inserted and deleted. The incremental change to V, i.e., ΔV, can be computed using the following equation from [BLT86, GMS93]. ΔV represents both the insertions and deletions to the view V, $\uplus$ represents bag union, and $\bowtie$ represents a bag join. R_1^{old} and R_2^{old} represent the pre-update states of R_1 and R_2.

$$\Delta V = \Delta R_1 \bowtie R_2^{old} \uplus \Delta R_1 \bowtie \Delta R_2 \uplus R_1^{old} \bowtie \Delta R_2.$$

The above algebraic equation can, in general, involve $2^{n+1} - 1$ join expressions, where n is the number of joins in the expression defining view V. However, if we are allowed to reference both the pre- and post-update states of R_1 and R_2, we can simplify the above expression to use only n join expressions as follows:

$$\Delta V = \Delta R_1 \bowtie R_2^{old} \quad \uplus \quad R_1^{new} \bowtie \Delta R_2$$

where R_1^{new} is the post-update state of R_1. If the second equation is evaluated before the updates are applied to R_1 and R_2, then R_1^{new} must be computed from the current (pre-update) state of R_1 and from ΔR_1. Similarly, if the equation is evaluated after the updates have been applied to R_1 and R_2, then R_2^{old} must be computed from the current (post-update) state of R_2 and ΔR_2. A third possibility is to evaluate $\Delta R_1 \bowtie R_2^{old}$ before applying the updates to R_1 and R_2, and to evaluate $\Delta R_1 \bowtie R_2^{new}$ after applying the updates. Since the updates to R_1 and R_2 have to be performed anyway, we can interleave the application of these updates and the ΔV computations so as to guarantee that R_1 and R_2 are always referenced in their current state, i.e., their pre-update or post-update values do not have to be recomputed. We call such a computation staggered since one can visualize it as going through the join expressions in a staggered fashion, first propagating the effect of ΔR_1, then applying ΔR_1 to refresh R_1, and then propagating the effect of ΔR_2, followed by a refresh of R_2.

Now suppose that we have several views defined over R_1 and R_2, then, is it still possible to interleave the updates of R_1 and R_2 and the change computations of the views so as to guarantee that R_1 and R_2 will always be referenced in their current states? Furthermore, if these views had other views defined over them, can we still maintain this property? Can the interleaving work for the union and SQL except operations also?

If R_1 and R_2 are base tables updated by user transactions, we may not be able to control how these are updated. In general, if we have multiple views, with a mix of immediate, deferred, and snapshot maintenance, we may not be able to control the refresh of some view tables with the change computation of other view tables. When can we successfully interleave propagate and refresh?

Once the change sets are applied to the view (refreshed), we would like to remove the change sets. What is the earliest time that the change sets can be removed?

Summary of Results

We obtain a staggered view maintenance algorithm that interleaves refresh and propagate operations using the counting algorithm of [GMS93] for the select, project, join, union, and SQL except operations. The algorithm evaluates propagate expressions in an order such that a table is guaranteed to be in the (pre- or post- update) state required by the expression, provided the algorithm has control over refreshing the referenced table. The algorithm accounts for multiple materialized views with a mix of immediate, deferred, and snapshot maintenance.

2 Preliminaries

2.1 Notation

A table in a database is either a *base table* or a *view table*. A view table is derived from a query Q over other tables, denoted by $V \stackrel{def}{=} Q(R_1, ..., R_n)$. The deriving tables $R_1, ..., R_n$ are called V's *parents* and V is a *child* of these tables. A view-dependency graph $\mathcal{G}$ is the graph representing the parent-child relationships among tables in the database.

A state s is a mapping from table names to bags of tuples (of the appropriate types). The value of a table R in state s is denoted by $R(s)$. We assume that states have a temporal ordering and that s_{curr} denotes the current state. The result of a query Q evaluated in state s is denoted as $Q(s)$.

We say that a view V is *consistent* with some past state s_i of its deriving tables if $V(s_{curr}) = Q(s_i)$, where s_{curr} denotes the current state and $s_i \leq s_{curr}$. We say that a view V is *consistent* if it is consistent with the current state s_{curr} (i.e., $V(s_{curr}) = Q(s_{curr})$). A view can become inconsistent when any of its deriving tables are updated. For an immediate view, consistency must be restored immediately after the deriving tables are updated.

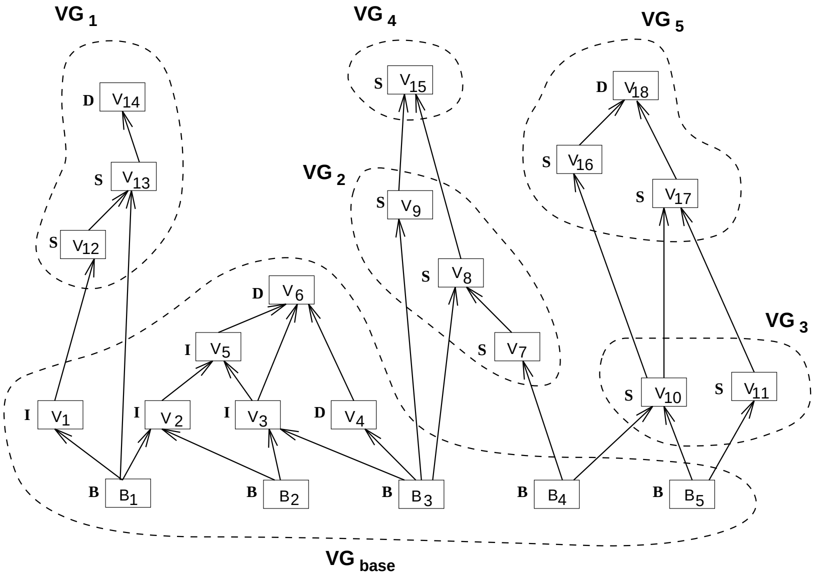


Figure 1: An example of a view-dependency graph.

If $V \stackrel{def}{=} Q(R_1, ..., R_n)$, let Q_B be a rewrite of the query Q in terms of base tables. Then, V is consistent with some state s_i ($s_i \leq s_{curr}$) of *base tables* if $V(s_{curr}) = Q_B(s_i)$.

2.2 Model

The following view model is taken from our implementation of views in the ODE system [CKL+96]. We permit multiple materialized views with multiple maintenance policies.

In order to guarantee consistency with respect to base tables, we need to impose certain restrictions on the specifications of materialized views and maintenance policies. For example, if a view is defined over two snapshot views with different refresh times or frequencies, consistency of the former cannot be guaranteed. We adopt the following model for specifying and maintaining views to allow consistency requirements to be satisfied.

We partition the set of tables into viewgroups $VG_1, ..., VG_n$ such that all non-deferred views in a viewgroup VG_i are consistent with each other and with some common past state of the base tables. A deferred view will become consistent (with this common past state) when it is queried. A query involving views from a single viewgroup will thus see a consistent past state of the base tables.

We assume a model in which the following conditions hold:

1. An immediate view is defined only over base tables or other immediate views.
2. All immediate views and base tables are placed in a single viewgroup that does not contain any snapshot views.
3. Views in a viewgroup can be derived from tables in at most one other viewgroup.
4. All tables in a viewgroup excluding deferred views are updated/refreshed synchronously.
5. Deferred views must be defined from views in the same viewgroup and cannot be used to define immediate or snapshot views.

These conditions are imposed to guarantee our consistency requirements. In addition, they ensure that refresh operations are localized in the sense that they do not trigger refresh operations in other viewgroups. Further details of these rules may be found in [CKL+96].

Figure 1 is an example of a view dependency graph and shows tables grouped into viewgroups. The label `B` denotes base tables, and labels `I`, `D`, and `S` denote immediate, deferred and snapshot views, respectively. When updates to base tables are performed all immediate views in the viewgroup (VG_{base}) are refreshed. All other viewgroups are refreshed periodically at intervals determined by each viewgroup's refresh frequency. Each periodic refresh of a viewgroup causes the snapshot views in the viewgroup to be refreshed. Deferred views are refreshed only when they are queried.

3 The Staggered Algorithm

Let VG be a viewgroup and let M be the staggered maintenance procedure that is invoked to refresh all the (non-deferred) views in viewgroup VG. Then, we would like that (1) for any materialized view V in VG, M does not have to reference its post-update value before V is actually refreshed, and (2) once V has been refreshed, M does not have to reference its pre-update value.

It is assumed that all base tables and external views (i.e., views outside viewgroup VG) are available in the post-update state, along with their change sets. Let $\mathcal{G}$ be the view-dependency graph of the set of view definitions. Let the tables in $\mathcal{G}$ be ordered by a topological sort [AHU83].

We now present the staggered algorithm to refresh a viewgroup VG. Let the set of all the base tables in VG and the set of base tables and views outside VG used to derive views in VG be $\{T_1, ..., T_m\}$. Further, let $V_1, \ldots, V_n$ be the immediate and snapshot views in the viewgroup VG, in order of the topological sort.

Each table T_i in $\{T_1, ..., T_m\}$ has a differential table ΔT_i associated with it. We assume that ΔT_i contains the portion of changes made to T_i since the last refresh of VG. In other words, we are using the assumption that for each table, we have a differential table for each viewgroup containing views derived from the table. This assumption is only for the sake of simplifying the presentation of the algorithm. Techniques for maintaining a single log for supporting multiple views and viewgroups, from which the ΔT_i relevant to a specific viewgroup can be extracted have been discussed in [CKL+96, CG96, SP89]. For each table V_i in $\{V_1, \ldots, V_n\}$, the staggered algorithm will compute a differential table ΔV_i, and then refresh V_i by applying the changes of ΔV_i to V_i. We will discard ΔV_i, unless V_i is used to derive a deferred view, or a view in another viewgroup.

Let Q be a select-project-join query without self-joins. (We will show later how the algorithm can be extended for self-joins, except, and union operators.) For any view table $V \stackrel{def}{=} Q(R_1, ..., R_k)$, we use $\eta(Q, R_i)$ to denote the expression

$$Q(R_1, ..., R_{i-1}, \Delta R_i, R'_{i+1}, ..., R'_k)$$

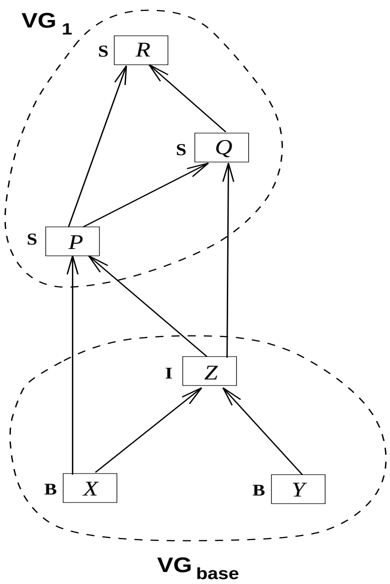
where each R'_r (for $i < r \leq k$) is R_r if R_r is in $\{V_1, \ldots, V_n\}$ and $R'_r = R_r^{old}$ if R_r is in $\{T_1, \ldots, T_m\}$. The idea is that if R_r is one of $T_1, \ldots, T_m$, it was refreshed before the current refresh of viewgroup VG is started and so we have to compute its old value from the post-update state. If R_r is one of $V_1, \ldots, V_n$, then it will not have been refreshed when $\eta(Q, R_i)$ is evaluated, so R_r will refer to the pre-update state. We use the (C-like) notation $A \mathrel{+}= B$ to denote the assignment $A := A \uplus B$.

Algorithm 3.1 (Staggered Algorithm)
Input: Viewgroup VG to be refreshed.
Result: All immediate and snapshot views in VG are refreshed.
Method:

Initialize all ΔV_i's to be empty;
for $i = 1$ to m **do** /* propagate $\Delta T_1 \ldots \Delta T_m$ */
    for $j = 1$ to n **do** /* compute contributions to $\Delta V_1 \ldots \Delta V_n$ */
        if V_j has T_i as a parent
          /* T_i is used to define V_j */
            $\Delta V_j \mathrel{+}= \eta(Q_j, T_i)$;
              /* where Q_j is V_j's view-definition query */
    endfor
    delete ΔT_i if T_i is not used to define any deferred views;
endfor
for $i = 1$ to n **do** /* propagate $\Delta V_1 \ldots \Delta V_n$ */
    for $j =$ i+1 to n **do** /* compute contributions to $\Delta V_{i+1} \ldots \Delta V_n$ */
        if V_j has V_i as a parent
          /* V_i is used to define V_j */
            $\Delta V_j \mathrel{+}= \eta(Q_j, V_i)$;
              /* where Q_j is V_j's view-definition query */
    endfor
    $V_i \mathrel{+}= \Delta V_i$; /* refresh V_i */
    delete ΔV_i if V_i is not used to define deferred views or views in other viewgroups;
endfor
$\diamond$

Algorithm Explanation We first propagate changes from all external tables (parent tables in another viewgroup) or base tables, building the change set for a view piecemeal. We then start propagating change sets of the views in the viewgroup. Topological order guarantees that the change set of V is fully computed before it is propagated up.

In the case of base tables or tables that are parents of views in other viewgroups, the propagate operations for the children cannot be intermixed with the refresh of parent tables. Hence, the propagate operations based on these parent tables are done separately. We can show that the



$\Delta P, \Delta Q, \Delta R$	$:=$	ϕ
ΔP	$+=$	$\Delta X \bowtie Z^{old}$
ΔX	$:=$	ϕ
ΔP	$+=$	$X \bowtie \Delta Z$
ΔQ	$+=$	$\Delta Z \bowtie P$
ΔZ	$:=$	ϕ
ΔQ	$+=$	$Z \bowtie \Delta P$
ΔR	$+=$	$\Delta P \bowtie Q$
P	$+=$	ΔP
ΔP	$:=$	ϕ
ΔR	$+=$	$P \bowtie \Delta Q$
Q	$+=$	ΔQ
ΔQ	$:=$	ϕ
R	$+=$	ΔR
ΔR	$:=$	ϕ

(a) (b)

Figure 2: Example view dependency graph and staggered operations.

staggered algorithm has the following properties:

- All the join expressions in the change set equations derived by the counting algorithm of [GMS93] for ΔV are eventually evaluated.
- All the equations defining ΔV are evaluated before any equation using ΔV is evaluated.
- The old (pre-update) state has to be computed only for base tables or for views in the parent viewgroup.
- A join expression referencing V^{old} in the equations of [GMS93] is evaluated before V is refreshed.
- A join expression referencing V^{new} in the equations of [GMS93] is evaluated after V is refreshed.
- All equations referencing ΔV are evaluated consecutively.

EXAMPLE 3.1 Let the viewgroup VG_{base} contain two base tables X and Y and an immediate view Z defined from X and Y. Suppose that viewgroup VG_1 has snapshot views P, Q and R where P is defined from tables X and Z in VG_{base}, Q is defined from Z and P, and R is defined from P and Q. Figure 2(a) shows the view dependency graph representing these relationships.

A topological ordering of the view dependency graph is $X < Y < Z < P < Q < R$. Let us apply the staggered maintenance algorithm to the viewgroup VG_1. The set $\{T_1, ..., T_m\}$ of base and external tables in the staggered algorithm is $\{X, Z\}$ and the set of views to be refreshed $\{V_1, ..., V_n\}$ is $\{P, Q, R\}$. The sequence of operations performed using the staggered algorithm when a refresh of the viewgroup VG_1 is invoked is shown in Figure 2(b). The operations within each pair of lines correspond to an outermost for loop in the staggered algorithm. The computations between the first and third lines correspond to the processing of T_i's in the staggered algorithm. Since these tables were updated before the refresh of VG_1 was invoked, we have to compute the pre-update state (for Z). ∎

We now sketch how the algorithm can be extended to handle views with self-joins. We illustrate this through an example.

Let $V \stackrel{def}{=} R \bowtie_{p_1} R \bowtie_{p_2} R$. The staggered algorithm given earlier would require simultaneous access to the pre- and post-update values of R in a propagate equation such as $R^{new} \bowtie_{p_1} \Delta R \bowtie_{p_2} R^{old}$, which we would like to avoid. Therefore, the propagate and refresh steps have to be handled differently for self-join views. For this example, we first compute $\Delta^1 R, \Delta^2 R, \Delta^3 R$ (when R is still R^{old}) as follows:

$$\begin{array}{lcl} \Delta^1 R & := & \Delta R \bowtie_{p_1} R \bowtie_{p_2} R \\ \Delta^2 R & := & \Delta R \bowtie_{p_2} R \\ \Delta^3 R & := & \Delta R \end{array}$$

We then refresh R (R now becomes R^{new}) and then compute ΔV as

$$\begin{array}{lcl} \Delta V & := & \Delta^1 R \\ & & \uplus\ R \bowtie_{p_1} \Delta^2 R \\ & & \uplus\ R \bowtie_{p_1} R \bowtie_{p_2} \Delta^3 R \end{array}$$

ΔV correctly evaluates to:

$$\begin{array}{ll} & \Delta R \bowtie_{p_1} R^{old} \bowtie_{p_2} R^{old} \\ \uplus & R^{new} \bowtie_{p_1} \Delta R \bowtie_{p_2} R^{old} \\ \uplus & R^{new} \bowtie_{p_1} R^{new} \bowtie_{p_2} \Delta R \end{array}$$

4 The Except and Union Operators

In this section, we show how the staggered algorithm can be extended to handle views defined using the SQL **EXCEPT** and **UNION** operators.

The EXCEPT operator of SQL has two different semantics depending on whether the keyword ALL is specified with it. The SQL query

$$V_1 \texttt{ EXCEPT } V_2$$

returns a single occurrence of each tuple t in V_1 if t does not occur in V_2. If t occurs in V_2, then it is not returned in the answer. The query

$$V_1 \texttt{ EXCEPT ALL } V_2$$

returns the bag difference (monus) of V_1 and V_2. In other words, if a tuple t has $m \geq 0$ and $n \geq 0$ occurrences in V_1 and V_2, respectively, then the result of the EXCEPT ALL operation will contain p occurrences of t where p is the maximum of 0 and $m - n$. Figure 3 shows examples of EXCEPT and EXCEPT ALL operations.

V_1
a
a
b
b
b
b
c

V_2
b
b
d

V_1 EXCEPT V_2
a
c

V_1 EXCEPT ALL V_2
a
a
b
b
c

Figure 3: An example of EXCEPT operations.

We first consider the case without the ALL operator. Although, the result of the EXCEPT operation contains distinct tuples, we maintain duplicate tuples in the view (using counts) for the purpose of view maintenance (as in [GMS93]) with the understanding that the duplicates are relevant only for view maintenance. For example, V defined as V_1 EXCEPT V_2 will contain $\{a, a, c\}$ or $\{[a, +2], [c, +1]\}$ where the multiplicities associated with the tuples are only used for maintaining V and ignored (treated as $+1$) for all querying operations.

Using the notation of [GMS93], we can rewrite V_1 EXCEPT V_2 as

$$V_1 \bowtie \overline{V_2}$$

for the purpose of change computation. $\overline{V_2}$ has a tuple x with multiplicity 1 if x appears in V_1, but does not appear in V_2. Note that $\overline{V_2}$ is defined with respect to the other argument of the join operation. We assume here that the joins denote natural join operations, i.e., only one set of common attributes are preserved. The change expressions are then derived as for joins, as follows (assuming $V \stackrel{def}{=} V_1$ EXCEPT V_2):

$$\begin{array}{lcl} \Delta V & = & \Delta V_1 \bowtie \overline{V_2}^{old} \\ & & \uplus\ V_1^{new} \bowtie \Delta \overline{V_2} \end{array}$$

Note that in the first term in the above equation, $\overline{V_2}^{old}$ is computed *with respect to* ΔV_1. In other words, the term

$$\Delta V_1 \bowtie \overline{V_2}^{old}$$

is equivalent to

$$\Delta V_1 \texttt{ EXCEPT } V_2^{old}$$

where (for the purpose of view maintenance) we assume that EXCEPT preserves duplicates from its left argument. In the second term,

$$V_1^{new} \bowtie \Delta \overline{V_2},$$

$\Delta\overline{V_2}$ is computed with respect to V_1^{new}, that is, $\Delta\overline{V_2}$ denotes the change to $\overline{V_2}$ in the expression $V_1^{new} \bowtie \overline{V_2}$ when changes in ΔV_2 are applied to V_2. We can rewrite $V_1^{new} \bowtie \Delta\overline{V_2}$ as $V_1^{new} \bowtie \widehat{\Delta} V_2$ where $\widehat{\Delta} V_2$ is the minimal set defined as follows:

- For every inserted (positive count) tuple t in ΔV_2, $(t, -1)$ is in $\widehat{\Delta} V_2$ if $t \notin V_2^{old}$. (This can be expressed algebraically as $-(\epsilon(\Delta^+ V_2) \bowtie \overline{V_2^{old}})$ where $\Delta^+ V_2$ denotes the tuples in ΔV_2 with a positive count, '$-$' denotes the unary minus operator that is used to reverse the sign of tuples in deltas, and ϵ is duplicate elimination.)
- For every deleted (negative count) tuple t in ΔV_2, $(t, +1)$ is in $\widehat{\Delta} V_2$ if $t \notin V_2^{new}$. (This can be expressed as $-(\epsilon(\Delta^- V_2) \bowtie \overline{V_2^{new}})$ where $\Delta^- V_2$ denotes the tuples in ΔV_2 with a negative count.)

Note that $\widehat{\Delta} V_2$ is defined above without reference to the table V_1^{new} being joined with. This makes it easy to compute $\widehat{\Delta} V_2$. The algebraic equivalence

$$V_1^{new} \bowtie \Delta\overline{V_2} \equiv V_1^{new} \bowtie \widehat{\Delta} V_2$$

is useful here since $\widehat{\Delta} V_2$ is not equal to $\Delta\overline{V_2}$. In general, $\widehat{\Delta} V_2$ can contain tuples that are not in V_1^{new}. As an example, consider the tables V_1 and V_2 in Figure 3. Suppose that ΔV_1 is empty and ΔV_2 is $\{(b, -1), (d, -1), (a, +1)\}$. V_1^{new} is the same as V_1^{old}. In the expression, $V_1 \bowtie \overline{V_2}$, the value of $\overline{V_2}$ before the update to V_2 is $\{(a, +1), (c, +1)\}$. The value of $\overline{V_2}$ after the update is $\{(c, +1)\}$. So the change in $\overline{V_2}$ (i.e., $\Delta\overline{V_2}$) is $\{(a, -1)\}$. But the value of $\widehat{\Delta} V_2$ is $\{(a, -1), (d, +1)\}$. However, the join with V_1^{new} will eliminate the extra tuples in $\widehat{\Delta} V$ ($(d, +1)$ in our example). The expression computing ΔV is now as follows:

$$\begin{aligned} \Delta V \quad = \quad & \Delta V_1 \bowtie \overline{V_2{}^{old}} \\ & \uplus \; V_1^{new} \bowtie \widehat{\Delta} V_2 \end{aligned}$$

The join subexpressions for ΔV are evaluated at the steps for propagating up ΔV_1 and ΔV_2, as indicated by the staggered algorithm in Section 3. The staggered algorithm guarantees that references to $V_2{}^{old}$ need to be evaluated before refreshing V_2, and that references to $V_2{}^{new}$ need to be evaluated after refreshing V_2. However, the computation of $\widehat{\Delta} V_2$ needs access to both V_2^{old} and V_2^{new}. We can interleave the computation of $\widehat{\Delta} V_2$ with the refresh of V_2 as follows:

1. For every inserted (positive count) tuple t in ΔV_2, look up V_2^{old}. If $t \notin V_2^{old}$, then insert $(t, -1)$ in $\widehat{\Delta} V_2$.
2. Refresh V_2^{old} to obtain V_2^{new}.
3. For every deleted (negative count) tuple t in ΔV_2, look up V_2^{new}. If $t \notin V_2^{new}$, then insert $(t, +1)$ in $\widehat{\Delta} V_2$.

Putting this all together, the computation of ΔV can be performed by the following sequence of steps.

$$\begin{array}{rcl} \Delta V, \widehat{\Delta} V_2 & := & \phi \\ \Delta V & += & \Delta V_1 \bowtie \overline{V_2} \\ V_1 & += & \Delta V_1 \text{ /* update } V_1 \text{ */} \\ \widehat{\Delta} V_2 & += & -(\epsilon(\Delta^+ V_2) \bowtie \overline{V_2}) \\ V_2 & += & \Delta V_2 \text{ /* update } V_2 \text{ */} \\ \widehat{\Delta} V_2 & += & -(\epsilon(\Delta^- V_2) \bowtie \overline{V_2}) \\ \Delta V & += & V_1 \bowtie \widehat{\Delta} V_2 \end{array}$$

The above interleaving continues to give us the desirable property that a materialized view is always in the state in which it is referenced in an expression to be evaluated.

We have shown how an interleaving strategy can be used for maintaining a single view defined via the `EXCEPT` operation. We now show that the staggered algorithm of Section 3 can be extended to handle multiple `EXCEPT` views (in addition to SPJ views).

In the case of SPJ views, the topological ordering determines the order in which computations and refreshes for a table are performed. If $V_1 < V_2$, then V_1 is refreshed before V_2. This is necessary when V_1 is V_2's ancestor. Suppose that V_1 and V_2 are two views and V_3 is an SPJ view defined over V_1 and V_2. Then, the topological ordering will guarantee that the refresh operations for V_1 and V_2 are performed before the refresh for V_3. However, for the purpose of computing ΔV_3, it is immaterial whether $V_1 < V_2$ or vice-versa in the topological ordering because of commutativity of the join operator.

In the case of the `EXCEPT` operator commutativity does not hold. So, will any topological ordering still work? It may appear at first that this will not work and that we have to constrain the ordering to order the left-hand side argument of the `EXCEPT` operation before the right-hand side. We show that this is not necessary and that any topological ordering will still work.

We will illustrate this through an example. Suppose that R and S are two tables and that V_1 and V_2 are two views defined as $V_1 \stackrel{def}{=} R$ `EXCEPT` S

and $V_2 \overset{def}{=} S$ `EXCEPT` R. The delta equations for V_1 and V_2 are

$$\begin{array}{rcl} \Delta V_1 & = & \quad \Delta R \bowtie \overline{S^{old}} \\ & & \uplus \; R^{new} \bowtie \widehat{\Delta} S \end{array}$$

and

$$\begin{array}{rcl} \Delta V_2 & = & \quad \Delta S \bowtie \overline{R^{old}} \\ & & \uplus \; S^{new} \bowtie \widehat{\Delta} R. \end{array}$$

Suppose that we have the following topological ordering: $R < S < V_1 < V_2$. Can we do the update propagation while still preserving the property that old and new states do not have to be computed? We show that this can indeed be done by rewriting the equations for ΔV_2 as follows:

$$\begin{array}{rcl} \Delta V_2 & = & \quad \Delta S \bowtie \overline{R^{old}} \\ & & \uplus \; \Delta S \bowtie \widehat{\Delta} R \\ & & \uplus \; S^{old} \bowtie \widehat{\Delta} R. \end{array}$$

This may be further rewritten as

$$\begin{array}{rcl} \Delta V_2 & = & \quad \Delta S \bowtie \overline{R^{new}} \\ & & \uplus \; S^{old} \bowtie \widehat{\Delta} R \end{array}$$

using the property that, under strong minimality,

$$\Delta S \bowtie \overline{R^{new}} = \Delta S \bowtie \overline{R^{old}} \quad \uplus \quad \Delta S \bowtie \widehat{\Delta} R.$$

For the sake of convenience one could switch the order of the operands in the join operations for ΔV_2 with the understanding that $\overline{A} \bowtie B$ is considered semantically equivalent to $B \bowtie \overline{A}$, i.e, B `EXCEPT` A (modulo counts). Thus R appears before S in the equations for both ΔV_1 and ΔV_2. The staggered algorithm can proceed as before, using any topological ordering, with the modification noted below:

In the case of SPJ views, the processing of the tables is of the form

for each R_i **do**,
  Perform the portion of ΔV_j computations involving ΔR_i;
  Refresh R_i;

When `EXCEPT` views are also allowed in the system, we need to make the following minor modification:

for each R_i **do**,
  Perform the portion of ΔV_j computations involving ΔR_i and $\widehat{\Delta} R_i$ computations involving $\Delta^+ R_i$;
  Refresh R_i;
  Perform the portion of $\widehat{\Delta} R_i$ computations involving $\Delta^- R_i$
  Perform the portion of ΔV_j computations involving $\widehat{\Delta} R_i$

In the case of `EXCEPT` views incremental changes are computed from equations that reference V_1^{new}, V_2^{old} and V_2^{new}. The algorithm can be extended to handle `EXCEPT ALL` views but with references to V_1^{new}, V_2^{new} and V^{old} (the old value of view V). This can be done as follows:

Let $V \overset{def}{=} V_1$ `EXCEPT ALL` V_2. We first compute ΔV_{12} as

$$\Delta V_{12} = -(\Delta^- V_1) \cup \Delta^+ V_1 \cup -(\Delta^- V_2) \cup \Delta^+ V_2.$$

ΔV_{12} contains a single tuple (with a count of $+1$) for each tuple occurring in either ΔV_1 or ΔV_2. The $-$ operations are used to reverse the sign of tuples.

Next we compute the subset of tuples in V_1^{new}, V_2^{new}, and V^{old} that are changed or can cause a change to V as a result of the changes ΔV_1 and ΔV_2. Let δV_1 be the subset of V_1^{new} that can likely cause a change in V, let δV_2 be the subset of V_2^{new} that can likely cause a change in V, and let $\delta^o V$ be the subset of V^{old} that can possibly be changed. These subsets, with their counts, can be computed as:

$$\begin{array}{rcl} \delta V_1 & = & V_1^{new} \bowtie \Delta V_{12} \\ \delta V_2 & = & V_2^{new} \bowtie \Delta V_{12} \\ \delta^o V & = & V^{old} \bowtie \Delta V_{12} \end{array}$$

If we remove from V^{old} the tuples that are identified as likely to be changed ($\delta^o V$), and add the tuples

δV_1 `EXCEPT ALL` δV_2,

we will get the new value of V^{new}. ΔV is thus given by:

$$\Delta V \;\; = \;\; (\delta V_1 \texttt{ EXCEPT ALL } \delta V_2) \uplus -(\delta^o V)$$

A simpler way of handling `EXCEPT ALL` views is by maintaining counts from both V_1 and V_2 in the view for each tuple that occurs in V_1. This, of course, incurs additional space overhead since counts from both parent views have to be maintained, and tuples that do not occur in the result may also be materialized. In other words, for each tuple that occurs in V_1, we maintain the number of occurrences of the tuple in V_1 and the number of occurrences in V_2. Thus the size of the materialization of V with counts is bounded by the size of V_1. The staggered algorithm can then be extended in a straightforward fashion. A detailed description of this is beyond the scope of this paper.

The SQL `UNION` operation can be a bag or a set union (by specifying `ALL` in the first case). In the case of a view defined using `UNION ALL`, the change set for the view is simply a bag union of the change sets of the operand expressions. Therefore, the propagation can be done without referring to the deriving tables. In the case of the set union, we treat the view as a `UNION ALL` view for the purpose of view maintenance. In other words, we maintain multiplicities in the view and perform change propagation as in the `UNION ALL` case, but ignore the multiplicities for all querying to the view. Alternatively, the view can be maintained without counts using an approach similar to the one for `EXCEPT ALL` views described above.

Other operations that return sets, such as project-distinct can be handled by always maintaining counts in the view and by carefully distinguishing the cases when multiplicities must be retained and when they should be ignored.

5 Related Work and Summary

An algorithm similar to the staggered algorithm was developed independently by Hull and Zhou [HZ96]. They developed the algorithm for maintaining views in a warehousing environment, where all views are maintained immediately upon receiving the updates. They considered fully materialized, partially materialized and virtual views and did not consider deferred and snapshot views. Unlike in our model, all the updates to all the tables in the warehouse were under the control of the maintenance process. The views in their system are derived from set-based relational operations as opposed to bag-based operations like Except. They did not consider the problem of self-joins.

In this paper, we presented a staggered view maintenance algorithm that interleaves refresh and propagate operations. Each invocation of the algorithm causes all (non-deferred) views in a view-group to be refreshed. The interleaving guarantees that a table is in the correct state when a propagate equation referencing the table is evaluated provided that the referenced table is not a base table or a table outside the viewgroup that is being refreshed. The algorithm handles views defined using select, project, join (including self-join), union and except operations. The staggered algorithm is being implemented in the ODE system [CKL+96]. We are also developing techniques for choosing optimal strategies (incremental vs. full recomputation) for view maintenance.

Acknowledgements We thank the referees for their comments and suggestions.

References

[AHU83] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1983.

[BLT86] J. A. Blakeley, P. Larson, and F. W. Tompa. Efficiently Updating Materialized Views. In Carlo Zaniolo, editor, *Proceedings of ACM SIGMOD 1986 International Conference on Management of Data*, pages 61–71, Washington, D.C., May 28-30 1986.

[CG96] Latha Colby and Timothy Griffin. An algebraic approach to supporting multiple deferred views. In Ashish Gupta and Inderpal Singh Mumick, editors, *Proceedings of the International Workshop on Materialized Views: Techniques and Applications*, Montreal, Canada, June 7 1996.

[CGL+96] Latha Colby, Timothy Griffin, Leonid Libkin, Inderpal Singh Mumick, and Howard Trickey. Algorithms for deferred view maintenance. In Jagadish and Mumick [JM96].

[CKL+96] Latha Colby, Akira Kawaguchi, Daniel Lieuwen, Inderpal Singh Mumick, and Kenneth A. Ross. Implementing materialized views. Unpublished Manuscript, 1996.

[GL95] Timothy Griffin and Leonid Libkin. Incremental maintenance of views with duplicates. In Michael Carey and Donovan Schneider, editors, *Proceedings of ACM SIGMOD 1995 International Conference on Management of Data*, pages 328–339, San Jose, CA, May 23-25 1995.

[GMS93] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings* *of ACM SIGMOD 1993 International Conference on Management of Data*, Washington, DC, May 26-28 1993.

[HZ96] Richard Hull and Gang Zhou. A framework for supporting data integration using the materialized and virtual approaches. In Jagadish and Mumick [JM96].

[JM96] H. V. Jagadish and Inderpal Singh Mumick, editors. *Proceedings of ACM SIGMOD 1996 International Conference on Management of Data*, Montreal, Canada, 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, 1991.

[SP89] Arie Segev and Jooseok Park. Updating distributed materialized views. *IEEE Transactions on Knowledge and Data Engineering*, 1(2):173–184, June 1989.