Incremental Maintenance of Views with Duplicates

Timothy Griffin **Leonid Libkin**

AT&T Bell Laboratories

600 Mountain Avenue, Murray Hill NJ 07974, USA

email: {|griffin, libkin|}@research.att.com

Abstract

We study the problem of efficient maintenance of materialized views that may contain duplicates. This problem is particularly important when queries against such views involve aggregate functions, which need duplicates to produce correct results. Unlike most work on the view maintenance problem that is based on an algorithmic approach, our approach is algebraic and based on equational reasoning. This approach has a number of advantages: it is robust and easily extendible to new language constructs, it produces output that can be used by query optimizers, and it simplifies correctness proofs.

We use a natural extension of the relational algebra operations to bags (multisets) as our basic language. We present an algorithm that propagates changes from base relations to materialized views. This algorithm is based on reasoning about equivalence of bag-valued expressions. We prove that it is correct and preserves a certain notion of minimality that ensures that no unnecessary tuples are computed. Although it is generally only a heuristic that computing changes to the view rather than recomputing the view from scratch is more efficient, we prove results saying that under normal circumstances one should expect the change propagation algorithm to be significantly faster and more space efficient than complete recomputing of the view. We also show that our approach interacts nicely with aggregate functions, allowing their correct evaluation on views that change.

1 Introduction

In database management systems base relations are often used to compute views. Views are derived data that can be materialized (stored in a database) and subsequently queried against. If some of the base relations are changed, materialized views must be recomputed to ensure correctness of answers to queries against them. However, recomputing the whole view from scratch may be very expensive. Instead one often tries to determine the changes that must be made to the view, given the changes to the base relations and the expression that defines the view.

The problem of finding such changes to the views based on changes to the base relations has come to be known as the *view maintenance problem*, and has been studied extensively [27, 4, 3, 6, 7, 14, 29, 12, 5, 18, 26]. The name is slightly misleading since, as a reading of this literature will indicate, any solution to the problem is applicable in a large number of practical problems, including integrity constraint maintenance, the implementation of active queries, triggers and monitors.

Most of the work on view maintenance has assumed that relations are set-valued, that is, duplicates are eliminated. However, most practical database systems use bags (multisets) as the underlying model. They do handle duplicates, which is particularly important for evaluation of aggregate functions. For instance, if the average salary of employees is to be computed, then one applies the aggregate AVG to Π_{Salary}(Employees). Duplicates cannot be removed from the projection since the result would be wrong when at least two employees had the same salary. Not eliminating duplicates also speeds up query evaluation, as duplicate elimination is generally a rather expensive operation.

Many theoretical results obtained for set-theoretic semantics do not carry over to bags. In trying to bridge the gap between theoretical database research and practical languages, one particularly active research topic has been the design of bag languages [17, 24, 16]. Bag primitives [2] formed the basis for the algebras suggested by [11, 19]. These algebras turned out to be equivalent and accepted as the basic bag algebra. In this paper we use the basic bag algebra from [11, 19]. It was also shown that the basic bag algebra essentially adds the correct evaluation of aggregate functions to the relational algebra, and this continues to hold when nested relations are present [20]. There are a number of deep results on the complexity and expressive power of bag languages [11, 2, 19, 20, 21, 22, 31].

The main goal of this paper is to lay the foundation for incremental maintenance of views defined in the bag algebra. We advocate an approach based on *equational reasoning.* That is, for each primitive in the bag algebra we derive an equation that shows how the result of applying this primitive changes if some changes are made to its arguments. We do it independently for each primitive in the language; then, if an expression is given, the change propagation algorithm calculates changes to the view by recursively applying those rules.

This differs from most approaches found in the literature. Most papers on view maintenance suggest an *algorithmic* approach. That is, given changes to the base relations, an algorithm — sometimes quite *ad hoc* — is produced that computes the changes to the view. This approach makes it harder to prove a correctness result. For example, in [6] a new primitive when is introduced: E when δ is the value of the expression E if all changes given in δ are applied to E's arguments. The semantics of when is explained in [6] by means of low-level algorithms that compute the value of E when δ when E is a relational algebra operation. The correctness result for those low-level algorithms has not been proved in [6].

The only paper that addresses the issue of incremental view maintenance when duplicates are present is [12], where a similar algorithmic approach is adopted, but this time in a Datalog setting. A correctness result is proved in [12], but it is not particularly robust as it is not clear how the algorithm would react to slight changes in the language. We shall explain the difference between [12] and our approach in more detail in section 7. An equational approach has been used in [9] for the relational algebra. That work grew out of an analysis of [29], which presented an iterative algorithm for propagating changes. This was improved in [9] with a recursive algorithm that is similar in style to ours. This allows correctness to be proved with a simple proof by induction.

We believe that the equational approach to the view maintenance problem has a number of advantages over the algorithmic approach. In particular:

- Unlike the algorithmic approach, it provides us with precise semantics of changes to the views. Consequently, using the equational approach makes it easier to prove correctness of the change propagation algorithm. Also, as we shall see in section 4, the recursive form of our algorithm allows us to use invariants concerning minimality to further simplify change expressions. Such assumptions would not be available to a later phase of query optimization.

- This approach is robust: if language changes (e.g. new primitives are added), one only has to derive new rules for the added primitives, leaving all other rules intact. As long as the new rules are correct, the correctness of the change propagation algorithm is not affected.

- The resulting changes to the view are obtained in form of expressions in the same language used to define the view. This makes additional optimizations possible. For example, the expressions for changes that are to be made (e.g. for sets/bags of tuples to be deleted/added) can be given as an input to any query optimizer that might find an efficient way of calculating them.

Example. Suppose we have a database with the relations S1(Pid, Cost, Date) and S2(Pid, Cost, Date) for recording shipments of parts received from two different suppliers. The attribute Pid is a unique identifier for parts, Cost is the associated cost of a part, and Date is the day the shipment arrived. In addition we have the relation Paid(Pid, Cost, S), which registers parts that have been paid for. The attribute S must have the value 1 or 2, indicating which supplier was paid (see Figure 1).

We would like to compute the total amount of money we owe — the cost of all parts received by not yet paid for. One way of doing this is to define a view Unpaid as

$$V_1 \stackrel{\text{def}}{=} (\Pi_{\text{Pid,Cost}}(\text{S1}) \uplus \Pi_{\text{Pid,Cost}}(\text{S2}))$$

$$V_2 \stackrel{\text{def}}{=} \Pi_{\text{Pid,Cost}}(\text{Paid})$$

$$\text{Unpaid} \stackrel{\text{def}}{=} V_1 \dot{-} V_2$$

Here $\uplus$ is the additive union that adds up multiplicities of elements in bags. In particular, it will produce two copies of the record $[\text{Pid} \Rightarrow P1,\ \text{Cost} \Rightarrow 1,200]$ in calculating V_1. The modified subtraction monus $\dot{-}$ subtracts multiplicities. If a record r occurs n times in S and m times in T, then the number of occurrences of r in $S \dot{-} T$ is $n - m$ if $n \geq m$ and 0 if $n < m$.

Assume that the derived data that we are interested in is the amount still owed:

$$\text{Owe} = \mathsf{TOTAL}(\Pi_{\text{Cost}}(\text{Unpaid}))$$

Note that multiset semantics gives the correct answer here, while set-theoretic semantics would not. For example, for relations shown in figure 1, the amount Owe is \$7,400. However, if we switch to set semantics and calculate Unpaid as $(\Pi_{\text{Pid,Cost}}(\text{S1}) \cup \Pi_{\text{Pid,Cost}}(\text{S2})) - \Pi_{\text{Pid,Cost}}(\text{Paid})$, then Owe calculated by the same formula equals \$4,800. Thus, we do need multiset semantics for maintaining the view Unpaid in a correct manner.

Suppose that a transaction changes Paid by deleting the bag $\bigtriangledown$Paid and inserting the bag $\triangle$Paid. That is,

$$\text{Paid}^{\text{new}} = (\text{Paid} \dot{-} \bigtriangledown\text{Paid}) \uplus \triangle\text{Paid}.$$

S1

Pid	Cost	Date
P1	1,200	09/12
P2	2,100	08/27
P3	1,300	09/11
P4	1,400	08/25

S2

Pid	Cost	Date
P1	1,200	09/05
P4	1,400	08/24
P5	4,000	09/03

Paid

Pid	Cost	S
P1	1,200	1
P5	4,000	2

Unpaid

Pid	Cost
P1	1,200
P2	2,100
P3	1,300
P4	1,400
P4	1,400

Figure 1: Relations S1, S2, Paid and view Unpaid

Rather than recomputing the entire view Unpaid from scratch, we would like to find expressions $\triangledown$Unpaid and $\triangle$Unpaid such that

$$\text{Unpaid}^{\text{new}} = (\text{Unpaid} \dot{-} \triangledown\text{Unpaid}) \uplus \triangle\text{Unpaid}.$$

This has the potential of greatly reducing the amount of computational resources needed to recompute this new value.

For example, let $\triangledown$Paid contain the single record $[\text{Pid} \Rightarrow \text{P5}, \text{Cost} \Rightarrow 4,000, \text{S} \Rightarrow 2]$ and $\triangle$Paid contain the single record $[\text{Pid} \Rightarrow \text{P3}, \text{Cost} \Rightarrow 1,300, \text{S} \Rightarrow 1]$. (That is, we discovered that a payment was made to the first supplier for P3 rather than the second for P5.) Then it is fairly easy to see that $\triangledown$Unpaid should evaluate to $[\text{Pid} \Rightarrow \text{P3}, \text{Cost} \Rightarrow 1,300]$ and that $\triangle$Unpaid should evaluate to $[\text{Pid} \Rightarrow \text{P5}, \text{Cost} \Rightarrow 4,000]$.

Our algorithm treats the changes to base relations as black boxes. For this example it produces the delete bag, $\triangledown$Unpaid,

$$[\Pi_{\text{Pid},\text{Cost}}(\triangle\text{Paid}) \dot{-} \Pi_{\text{Pid},\text{Cost}}(\triangledown\text{Paid})] \min \text{Unpaid},$$

and the insert bag, $\triangle$Unpaid,

$$(\Pi_{\text{Pid},\text{Cost}}(\triangledown\text{Paid}) \dot{-} \Pi_{\text{Pid},\text{Cost}}(\triangle\text{Paid})) \dot{-} (V_2 \dot{-} V_1).$$

Here $S \min T$ is a multiset such that the number of occurrences of a record r in it is $\min(n, m)$, where n and m are numbers of occurrences in S and T respectively. Notice that the evaluation of the expressions for $\triangledown$Unpaid and $\triangle$Unpaid can be made very efficient. First we assume that all relations S1, S2 and Unpaid have an index built on the them that uses Pid as a key. Then, in order to evaluate the expressions for $\triangledown$Unpaid and $\triangle$Unpaid we only have to find the numbers of occurrences of elements of $\Pi_{\text{Pid},\text{Cost}}(\triangledown\text{Paid})$ and $\Pi_{\text{Pid},\text{Cost}}(\triangle\text{Paid})$ in V_1, V_2 and $\Pi_{\text{Pid},\text{Cost}}(\text{Paid})$. For example, to find $\triangledown$Unpaid, for each $r \in \triangle$Paid we find x, y, z, v as the numbers of occurrences of r in $\triangle$Paid, $\triangledown$Paid, V_1 and V_2. Then R occurs $\min\{(x \dot{-} y), (z \dot{-} v)\}$ times in $\triangledown$Unpaid. Thus, the complexity of such evaluation depends on how fast we can access elements of the *base* relations. Access to the base relations is typically fast, compared to access to views.

Even if no index exists, the time complexity is still linear in the sizes of the base relations. The big win here is in space usage. Whereas recomputing the whole view Unpaid would require space linear in the size of base relations, the propagation algorithm only requires that we find the number of occurrences of certain records in base relations and then evaluate an arithmetic expression. Therefore, space needed for updating the view Unpaid is linear in the size of *changes* to the base relations. Typically, these changes are relatively small compared to the size of the relations. Thus, calculating changes to the view as opposed to reconstructing the view from scratch leads at least to substantial improvement in space usage.

Once changes to Unpaid are calculated, the new value of Owe is found as

$$\begin{aligned}\text{Owe}^{\text{new}} &= (\text{Owe} - \mathsf{TOTAL}(\Pi_{\text{Cost}}(\triangledown\text{Unpaid}))) \\ &+ \mathsf{TOTAL}(\Pi_{\text{Cost}}(\triangle\text{Unpaid})).\end{aligned}$$

The correctness of this is guaranteed for our solution. Indeed, Owe^{new} calculated above is \$10,100, and one can see that it is the correct amount still owed once changes to Paid have been made.

Organization. In Section 2 we introduce our notation, describe our basic bag algebra, and state the problem. We present some basic facts concerning equational reasoning for our bag algebra in Section 3. In Section 4 we present our change propagation algorithm for view maintenance. We then address aggregate functions in Section 5. In Section 6 we analyze the complexity of the results produced by our algorithm. We discuss related work in Section 7. Finally, we conclude in Section 8 with some remarks concerning future work. All proofs can be found in [10].

2 Notation and Problem Statement

2.1 The Bag Algebra, $\mathcal{BA}$

As we mentioned in the introduction, several equivalent approaches to bag-based database languages have been proposed [11, 19, 22]. As our basic language in which we formulate the change propagation algorithm we take a restriction of those languages to flat bags (that is, bag-valued attributes are not allowed).

In what follows, base relation names are denoted by the symbols R, R_1, R_2, ... Let p range over quantifier-free predicates, and A range over sets of attribute names. $\mathcal{BA}$ expressions are generated by the following grammar.

$$
\begin{array}{rcll}
S & ::= & \phi & \text{empty bag} \\
 & | & R & \text{name of stored bag} \\
 & | & \sigma_p(S) & \text{selection} \\
 & | & \Pi_A(S) & \text{projection} \\
 & | & S \uplus S & \text{additive union} \\
 & | & S \dot{-} S & \text{monus} \\
 & | & S \min S & \text{minimum intersection} \\
 & | & S \max S & \text{maximum union} \\
 & | & \epsilon(S) & \text{duplicate elimination} \\
 & | & S \times S & \text{cartesian product}
\end{array}
$$

To define the semantics of these operations, let $count(x, S)$ be the number of occurrences of x in a bag S. Then, for any operation e in the language, we define $count(x, e(S, T))$ or $count(x, e(S))$ as a function of $count(x, S)$ and $count(x, T)$ as follows:

$$
\begin{array}{rcl}
count(x, \sigma_p(S)) & = & \begin{cases} count(x, S) & p(x) \text{ is true} \\ 0 & p(x) \text{ is false} \end{cases} \\
count(x, \Pi_A(S)) & = & \displaystyle\sum_{y \in S, \Pi_A(y) = x} count(y, S) \\
count(x, S \uplus T) & = & count(x, S) + count(x, T) \\
count(x, S \dot{-} T) & = & \max(count(x, S) - count(x, T), 0) \\
count(x, S \min T) & = & \min(count(x, S), count(x, T)) \\
count(x, S \max T) & = & \max(count(x, S), count(x, T)) \\
count(x, \epsilon(S)) & = & \begin{cases} 1 & count(x, S) > 0 \\ 0 & count(x, S) = 0 \end{cases} \\
count((x, y), S \times T) & = & count(x, S) \times count(y, T)
\end{array}
$$

This language is not intended to be minimal. For example, min can be defined as $S \min T \stackrel{\text{def}}{=} S \dot{-} (S \dot{-} T)$. For the full characterization of interdefinability of the operations of $\mathcal{BA}$, consult [19].

We use the symbols S, T, W, and Z to denote arbitrary $\mathcal{BA}$ expressions, and s to denote a database state, that is, a partial map from relation names to multisets. If s is a database state and T is a $\mathcal{BA}$ expression such that s is defined on all relation names mentioned in T, then $s(T)$ denotes the multiset resulting from evaluating T in the state s. (Note that s is a function, so we consider evaluating T in s as the result of applying s to T.) The notation $T =_b S$ means that for all database states s, if s is defined on all relation names mentioned in S and T, then $s(T) = s(S)$.

2.2 Transactions

A transaction is a program that changes the state of a database in one atomic step. There are many approaches to languages for specifying transactions, (see for example [1, 28, 30]). In this paper we prefer to adopt an abstract view of transactions, in order to make the results independent of a particular language used, but at the same time readily applicable to any such language.

The abstract transactions to be considered are of the form

$$
\begin{array}{rcl}
t & = & \{R_1 \leftarrow (R_1 \dot{-} \nabla R_1) \uplus \triangle R_1, \\
 & & \quad \ldots, \\
 & & \quad R_n \leftarrow (R_n \dot{-} \nabla R_n) \uplus \triangle R_n\}.
\end{array}
$$

The expressions ∇R_i and $\triangle R_i$ represent the multisets deleted from and inserted into base relation R_i. More formally, when transaction t is executed in state s, then value of R_i in state $t(s)$ becomes $s((R_i \dot{-} \nabla R_i) \uplus \triangle R_i)$.

The expression T is a *pre-expression* of S w.r.t. t if for every database state s we have $s(T) =_b t(s)(S)$. It is easy to check that

$$
\begin{array}{rcl}
\text{pre}(t, S) & \stackrel{\text{def}}{=} & S((R_1 \dot{-} \nabla R_1) \uplus \triangle R_1, \\
 & & \quad \ldots, \\
 & & \quad (R_n \dot{-} \nabla R_n) \uplus \triangle R_n))
\end{array}
$$

is a pre-expression of S w.r.t. t. In other words, we can evaluate $\text{pre}(t, S)$ *before* we execute t in order to determine the value that S will have *afterwards.*

2.3 Problem Statement

Suppose $S(R_1, \cdots, R_n)$ is a $\mathcal{BA}$ expression and t is a transaction. We would like to determine how t's changes to the base relations propagate to changes in the value of S. In particular, we seek to construct expressions $\triangle S$ and ∇S, called a *solution* for $\text{pre}(t, S)$, such that

$$\text{pre}(t, S) =_b (S \dot{-} \nabla S) \uplus \triangle S.$$

Note that the expressions ∇S and $\triangle S$ are to be evaluated before t is executed (and committed). These solutions can be used in many applications involving the maintenance of derived data. For example, in the case of view maintenance this allows us to recompute the value of S in the new state from its value in the old state and the values of ∇S and $\triangle S$. For integrity maintenance it allows us to check data integrity *before* a transaction is committed, thus allowing for the transaction to be aborted without the expense of a roll-back operation.

Clearly, not all solutions are equally acceptable. For example, $\nabla S = S$ and $\triangle S = \text{pre}(t, S)$ is always a solution. How can we determine which are "good" solutions? First, if S is a materialized view, then it should be generally cheaper to evaluate $(S \dot{-} \nabla S) \uplus \triangle S$ than to evaluate $\text{pre}(t, S)$ in the current state (or to evaluate S after t has been executed). Second, we should impose some "minimality" conditions on ∇S and $\triangle S$ to make sure that no unnecessary tuples are produced. In particular,

1. $\nabla S \dot{-} S =_b \phi$: We only delete tuples that are in S.
2. $\triangle S \min \nabla S =_b \phi$: We do not delete a tuple and then reinsert it.

A solution meeting condition (1) will be called *weakly minimal*, while a solution meeting both conditions (1) and (2) will be called *strongly minimal*. Note that, in contrast to the relational case [29], it does not make sense to insist that S be disjoint from $\triangle S$ since a transaction may increase the multiplicities of elements in S.

We will argue that minimality (weak or strong) is especially desirable due to the way in which changes interact with aggregate functions. For example, we have

$$\begin{aligned} & \mathsf{TOTAL}((S \dot{-} \nabla S) \uplus \triangle S) \\ = \; & (\mathsf{TOTAL}(S) - \mathsf{TOTAL}(\nabla S)) + \mathsf{TOTAL}(\triangle S) \end{aligned}$$

assuming a (weakly or strongly) minimal solution.

Again, not all strongly minimal solutions are equally acceptable. For example, the pair

$$\nabla Q = Q \dot{-} \mathrm{pre}(t, Q)$$

and

$$\triangle Q = \mathrm{pre}(t, Q) \dot{-} Q$$

is a strongly minimal solution. However, one does not win by using it for maintaining the view given by Q.

The main goal of this paper is to present an algorithm for generating (at compile time) strongly minimal solutions to the view maintenance problem and demonstrate that they are computationally more efficient than recomputing the view (at run-time).

3 Preliminaries

This section presents the equational theory underlying our change propagation algorithm. A change propagation algorithm for the relational algebra was presented in [29], based on a collection of equations that are used to "bubble up" change sets to the top of an expression. For example, [29] uses the equation

$$(S \cup \triangle S) - T = (S - T) \cup (\triangle S - T)$$

to take the insertion $\triangle S$ into S and propagate it upward to the insertion $\triangle S - T$ into $S - T$.

Our first step is to define a collection of such propagation rules for bag expressions. The situation is more complicated for $\mathcal{BA}$ expressions since they do not obey the familiar laws of boolean algebra that we are accustomed to using with set-valued relational expressions. For bag expression, the above example now becomes

$$(S \uplus \triangle S) \dot{-} T = (S \dot{-} T) \uplus (\triangle S \dot{-} (T \dot{-} S)),$$

which is not immediately obvious.

Figure 2 contains our equations for change propagation in bag expressions. Some subexpressions are annotated with a ∇ (for a deletion bag) or a $\triangle$ (for an insertion bag). This annotation simply emphasizes the intended application of these equations : when read as left-to-right rewrite rules, they tell us how to propagate changes upward in an expression. Note that the correctness of these equations involves no assumptions concerning minimality of the change bags.

Theorem 1 *The equations of Figure 2 are correct.*

Example. By repeated applications of the rules in figure 2 we can propagate any number of changes upward. Consider the expression $U = S \uplus T$. Suppose that

$$\mathrm{pre}(t, U) =_b ((S \dot{-} \nabla S) \uplus \triangle S) \uplus ((T \dot{-} \nabla T) \uplus \triangle T).$$

The changes to S and T can be propagated upward and expressed as changes to U as follows:

$$\begin{aligned} & ((S \dot{-} \nabla S) \uplus \triangle S) \uplus ((T \dot{-} \nabla T) \uplus \triangle T) \\ \overset{\text{P6}}{=_b} \; & (((S \dot{-} \nabla S) \uplus \triangle S) \uplus (T \dot{-} \nabla T)) \uplus \triangle T \\ \overset{\text{P6}}{=_b} \; & (((S \dot{-} \nabla S) \uplus (T \dot{-} \nabla T)) \uplus \triangle S) \uplus \triangle T \\ \overset{\text{P5}}{=_b} \; & ((((S \dot{-} \nabla S) \uplus T) \dot{-} (\nabla T \min T)) \uplus \triangle S) \uplus \triangle T \\ \overset{\text{P5}}{=_b} \; & ((((S \uplus T) \dot{-} (\nabla S \min S)) \dot{-} (\nabla T \min T)) \uplus \triangle S) \uplus \triangle T \\ =_b \; & (U \dot{-} \nabla_1 U) \uplus \triangle_1 U \end{aligned}$$

where $\nabla_1 U = (\nabla S \min S) \uplus (\nabla T \min T)$ and $\triangle_1 U = \triangle S \uplus \triangle T$. The last step is simply an application of the general rules

$$\begin{aligned} &\text{G1.} \quad (S \uplus T) \uplus W =_b S \uplus (T \uplus W) \\ &\text{G2.} \quad (S \dot{-} T) \dot{-} W =_b S \dot{-} (T \uplus W) \end{aligned}$$

which are applied in order to collect all deletions into one delete bag and all insertions into one insert bag.

Repeated application of the rules of figure 2 guarantees a solution, but not necessarily a strongly minimal one. However, the following theorem tells us that any solution can be transformed into a strongly minimal one.

Theorem 2 *Suppose that* $W =_b (Q \dot{-} \nabla_1 Q) \uplus \triangle_1 Q$. *Let* $\nabla_2 Q = (Q \min \nabla_1 Q) \dot{-} \triangle_1 Q$ *and* $\triangle_2 Q = \triangle_1 Q \dot{-} (Q \min \nabla_1 Q)$. *Then*

a) $W =_b (Q \dot{-} \nabla_2 Q) \uplus \triangle_2 Q$

b) $\nabla_2 Q \dot{-} Q =_b \phi$

c) $\nabla_2 Q \min \triangle_2 Q =_b \phi$.

Returning to the example from above, $\nabla_1 U$ and $\triangle_1 U$ can be transformed to a strongly minimal solution by taking $\nabla_2 U$ to be

$$(U \min ((\nabla S \min S) \uplus (\nabla T \min T))) \dot{-} (\triangle S \uplus \triangle T)$$

and $\triangle_2 U$ to be

$$(\triangle S \uplus \triangle T) \dot{-} (U \min ((\nabla S \min S) \uplus (\nabla T \min T)))$$

$$
\begin{array}{ll}
\text{P1.} & \sigma_p(S \dot- \nabla S) =_b \sigma_p(S) \dot- \sigma_p(\nabla S) \\
\text{P2.} & \sigma_p(S \uplus \triangle S) =_b \sigma_p(S) \uplus \sigma_p(\triangle S) \\
\text{P3.} & \Pi_A(S \dot- \nabla S) =_b \Pi_A(S) \dot- \Pi_A(\nabla S \min S) \\
\text{P4.} & \Pi_A(S \uplus \triangle S) =_b \Pi_A(S) \uplus \Pi_A(\triangle S) \\
\text{P5.} & (S \dot- \nabla S) \uplus T =_b (S \uplus T) \dot- (\nabla S \min S) \\
\text{P6.} & (S \uplus \triangle S) \uplus T =_b (S \uplus T) \uplus \triangle S \\
\text{P7.} & (S \dot- \nabla S) \dot- T =_b (S \dot- T) \dot- \nabla S \\
\text{P8.} & S \dot- (T \dot- \nabla T) =_b (S \dot- T) \uplus ((\nabla T \min T) \dot- (T \dot- S)) \\
\text{P9.} & (S \uplus \triangle S) \dot- T =_b (S \dot- T) \uplus (\triangle S \dot- (T \dot- S)) \\
\text{P10.} & S \dot- (T \uplus \triangle T) =_b (S \dot- T) \dot- \triangle T \\
\text{P11.} & (S \dot- \nabla S) \min T =_b (S \min T) \dot- (\nabla S \dot- (S \dot- T)) \\
\text{P12.} & (S \uplus \triangle S) \min T =_b (S \min T) \uplus (\triangle S \min (T \dot- S)) \\
\text{P13.} & (S \dot- \nabla S) \max T =_b (S \max T) \dot- (\nabla S \min (S \dot- T)) \\
\text{P14.} & (S \uplus \triangle S) \max T =_b (S \max T) \uplus (\triangle S \dot- (T \dot- S)) \\
\text{P15.} & \epsilon(S \dot- \nabla S) =_b \epsilon(S) \dot- (\epsilon(\nabla S \min S) \dot- (S \dot- \nabla S)) \\
\text{P16.} & \epsilon(S \uplus \triangle S) =_b \epsilon(S) \uplus (\epsilon(\triangle S) \dot- S) \\
\text{P17.} & (S \dot- \nabla S) \times T =_b (S \times T) \dot- (\nabla S \times T) \\
\text{P18.} & (S \uplus \triangle S) \times T =_b (S \times T) \uplus (\triangle S \times T)
\end{array}
$$

Figure 2: Change propagation equations for bag expressions

Although these expressions are rather complex, they can be greatly simplified to

$$\nabla_3 U \stackrel{\text{def}}{=} (\nabla S \dot- \triangle T) \uplus (\nabla T \dot- \triangle S)$$

$$\triangle_3 U \stackrel{\text{def}}{=} (\triangle S \dot- \nabla T) \uplus (\triangle T \dot- \nabla S)$$

under the assumption that the solutions $(\nabla S, \triangle S)$ and $(\nabla T, \triangle T)$ are strongly minimal (details omitted).

This example illustrates the three-step process that was used to derive the recursive algorithm presented in the next section. First, a general solution is derived by repeated application of the propagation rules of figure 2. Second, a strongly minimal solution is obtained by application of theorem 2. Third, the results are simplified under the assumption that all solutions for subexpressions are strongly minimal.

Note that if we are *only* concerned with correctness, then there is considerable freedom in the design of the propagation rules presented figure 2. For example, we could replace rule P8 with

$$S \dot- (T \dot- \nabla T) =_b (S \dot- T) \uplus ((S \dot- (T \dot- \nabla T)) \dot- (S \dot- T))$$

However, we have designed our rules from a *computational* point of view. Note that the structure of each equation in figure 2 follows the same pattern. For any operation e and its value $V = e(R_1, \ldots, R_n)$, $n = 1$ or $n = 2$, if one of its arguments changes, then its value V' on changed arguments is obtained as either $V \dot- \nabla$ or $V \uplus \triangle$. The expressions for ∇ and $\triangle$ are always of special form. Intuitively, they are "controlled" by ∇R_is and $\triangle R_i$s, that is, could be computed by iterating over them and fetching corresponding elements from base relations, rather than by iterating over base relations. Furthermore, this special form is preserved in the transformations defined in theorem 2.

For example, to compute $Z = ((\nabla T \min T) \dot- (T \dot- S))$ (rule P8 in figure 2), for each element $x \in \nabla T$, let n, m and k be numbers of occurrences of x in ∇T, T and S respectively. Then x occurs $\min(n, m) \dot- (m \dot- k)$ times in Z. Thus, to compute Z, we only fetch elements in ∇T from T and S. Since ∇R_is and $\triangle R_i$s are generally small compared to the size of base relations R_is, this special form of expressions for ∇ and $\triangle$ will make the change propagation algorithm suitable for maintaining large views. This intuition will be made more precise in the analysis of section 6.

4 Change Propagation Algorithm

This section presents our algorithm for computing a strongly minimal solution to a given view maintenance problem. That is, given a transaction t and a $\mathcal{BA}$ expression Q, we will compute expressions ∇Q and $\triangle Q$ such that $\text{pre}(t, Q) =_b (Q \dot- \nabla Q) \uplus \triangle Q$.

We first define two mutually recursive functions $\nabla(t, Q)$ and $\triangle(t, Q)$ such that for any transaction t $\text{pre}(t, Q) =_b (Q \dot- \nabla(t, Q)) \uplus \triangle(t, Q)$. These functions are presented in figure 3. For readability, we use the abbreviations $\text{add}(t, S)$ for $S \uplus \triangle(t, S)$ and $\text{del}(t, S)$ for $S \dot- \nabla(t, S)$.

We derived the clauses of these recursive functions in three steps: a) a general solution was obtained by repeated applications of the propagation rules of figure 2, b) theorem 2 was applied to obtain a strongly minimal solution, c) the results were further simplified

Q	$\triangledown(t, Q)$	#
R	$\triangledown R$, if $R \leftarrow (R \dot{-} \triangledown R) \uplus \triangle R$ is in t, and ϕ otherwise	$\triangledown 1$
$\sigma_p(S)$	$\sigma_p(\triangledown(t, S))$	$\triangledown 2$
$\Pi_A(S)$	$\Pi_A(\triangledown(t, S)) \dot{-} \Pi_A(\triangle(t, S))$	$\triangledown 3$
$S \uplus T$	$(\triangledown(t, S) \dot{-} \triangle(t, T)) \uplus (\triangledown(t, T) \dot{-} \triangle(t, S))$	$\triangledown 4$
$S \dot{-} T$	$((\triangledown(t, S) \dot{-} \triangledown(t, T)) \uplus (\triangle(t, T) \dot{-} \triangle(t, S))) \min Q$	$\triangledown 5$
$S \min T$	$(\triangledown S \dot{-} (S \dot{-} T)) \max (\triangledown T \dot{-} (T \dot{-} S))$	$\triangledown 6$
$S \max T$	$(\triangledown(t, S) \uplus (\triangledown(t, T) \min (T \dot{-} \mathrm{add}(t, S)))) \min (\triangledown(t, T) \uplus (\triangledown(t, S) \min (S \dot{-} \mathrm{add}(t, T))))$	$\triangledown 7$
$\epsilon(S)$	$\epsilon(\triangledown(t, S)) \dot{-} \mathrm{del}(t, S)$	$\triangledown 8$
$S \times T$	$(\triangledown(t, S) \times \triangledown(t, T)) \uplus ((\mathrm{del}(t, S) \times \triangledown(t, T)) \dot{-} (\triangle(t, S) \times \mathrm{del}(t, T)))\uplus$ $((\triangledown(t, S) \times \mathrm{del}(t, T)) \dot{-} (\mathrm{del}(t, S) \times \triangle(t, T)))$	$\triangledown 9$

Q	$\triangle(t, Q)$	#
R	$\triangle R$, if $R \leftarrow (R \dot{-} \triangledown R) \uplus \triangle R$ is in t, and ϕ otherwise	$\triangle 1$
$\sigma_p(S)$	$\sigma_p(\triangle(t, S))$	$\triangle 2$
$\Pi_A(S)$	$\Pi_A(\triangle(t, S)) \dot{-} \Pi_A(\triangledown(t, S))$	$\triangle 3$
$S \uplus T$	$(\triangle(t, S) \dot{-} \triangledown(t, T)) \uplus (\triangle(t, T) \dot{-} \triangledown(t, S))$	$\triangle 4$
$S \dot{-} T$	$((\triangle(t, S) \dot{-} \triangle(t, T)) \uplus (\triangledown(t, T) \dot{-} \triangledown(t, S))) \dot{-} (T \dot{-} S)$	$\triangle 5$
$S \min T$	$(\triangle(t, S) \uplus (\triangle(t, T) \min (\mathrm{del}(t, S) \dot{-} T))) \min (\triangle(t, T) \uplus (\triangle(t, S) \min (\mathrm{del}(t, T) \dot{-} S)))$	$\triangle 6$
$S \max T$	$(\triangle S \dot{-} (T \dot{-} S)) \max (\triangle T \dot{-} (S \dot{-} T))$	$\triangle 7$
$\epsilon(S)$	$\epsilon(\triangle(t, S)) \dot{-} S$	$\triangle 8$
$S \times T$	$(\triangle(t, S) \times \triangle(t, T)) \uplus ((\mathrm{del}(t, S) \times \triangle(t, T)) \dot{-} (\triangledown(t, S) \times T))\uplus$ $((\triangle(t, S) \times \mathrm{del}(t, T)) \dot{-} S \times \triangledown(t, T))$	$\triangle 9$

Figure 3: Mutually recursive functions $\triangledown$ and $\triangle$.

by assuming that all recursively derived solutions are strongly minimal.

This last step is quite important since the assumptions of strong minimality would not be available to a query optimizer at a later stage. It is also why we want to apply theorem 2 at every stage, rather than just once at the end. The three steps were outlined in the previous section for the $S \uplus T$ case.

Algorithm. Our algorithm is simply this: given inputs t and Q, use the functions $\triangledown(t, Q)$ and $\triangle(t, Q)$ to compute a solution for $\mathrm{pre}(t, Q)$. Note that in an actual implementation $\triangledown(t, Q)$ and $\triangle(t, Q)$ could be combined into one recursive function. Thus the algorithm requires only one pass over the expression Q.

The following theorem shows that the functions $\triangledown$ and $\triangle$ correctly compute a solution to the view maintenance problem and that they preserve strong minimality.

Theorem 3 *Let t be a strongly minimal transaction. That is, $R \dot{-} \triangledown R =_b \phi$ and $\triangledown R \min \triangle R =_b \phi$ for any $R \leftarrow (R \dot{-} \triangledown R) \uplus \triangle R$ in t. Let Q be a $\mathcal{BA}$ expression. Then*

1. $\mathrm{pre}(t, Q) =_b (Q \dot{-} \triangledown(t, Q)) \uplus \triangle(t, Q)$
2. $\triangledown(t, Q) \dot{-} Q =_b \phi$
3. $\triangle(t, Q) \min \triangledown(t, Q) =_b \phi$

Although some of the clauses in the definition of functions $\triangledown$ and $\triangle$ are rather complex, we believe that in practice many of the subexpressions will be ϕ or will easily simplify to ϕ. To illustrate this, recall the example from section 1:

$$V_1 \stackrel{\mathrm{def}}{=} (\Pi_{\mathrm{Pid,Cost}}(\mathrm{S1}) \uplus \Pi_{\mathrm{Pid,Cost}}(\mathrm{S2}))$$

$$V_2 \stackrel{\mathrm{def}}{=} \Pi_{\mathrm{Pid,Cost}}(\mathrm{Paid})$$

$$\mathrm{Unpaid} \stackrel{\mathrm{def}}{=} V_1 \dot{-} V_2$$

where the t is a transaction that changes Paid to $(\mathrm{Paid} \dot{-} \triangledown\mathrm{Paid}) \uplus \triangle\mathrm{Paid}$. Using our change propagation functions, the delete bag can be calculated as follows.

$$\begin{aligned}
& \triangledown(t, \mathrm{Unpaid}) \\
= \; & \triangledown(t, V_1 \dot{-} V_2) \\
\stackrel{\triangledown 5}{=} \; & ((\triangledown(t, V_1) \dot{-} \triangledown(t, V_2)) \uplus (\triangle(t, V_2) \dot{-} \triangle(t, V_1))) \\
& \min \mathrm{Unpaid} \\
= \; & ((\phi \dot{-} \triangledown(t, V_2)) \uplus (\triangle(t, V_2) \dot{-} \phi)) \min \mathrm{Unpaid} \\
= \; & \triangle(t, V_2) \min \mathrm{Unpaid} \\
= \; & \triangle(t, \Pi_{\mathrm{Pid,Cost}}(\mathrm{Paid})) \min \mathrm{Unpaid} \\
\stackrel{\triangle 3}{=} \; & [\Pi_{\mathrm{Pid,Cost}}(\triangle(t, \mathrm{Paid})) \dot{-} \Pi_{\mathrm{Pid,Cost}}(\triangledown(t, \mathrm{Paid}))] \\
& \min \mathrm{Unpaid} \\
\stackrel{\triangledown 1, \triangle 10}{=} \; & [\Pi_{\mathrm{Pid,Cost}}(\triangle\mathrm{Paid}) \dot{-} \Pi_{\mathrm{Pid,Cost}}(\triangledown\mathrm{Paid})] \min \mathrm{Unpaid}
\end{aligned}$$

In a similar way we can compute the change bag for insertions, $\triangle(t, \text{Unpaid})$, to be

$$[\Pi_{\text{Pid,Cost}}(\triangledown\text{Paid}) \dot{-} \Pi_{\text{Pid,Cost}}(\triangle\text{Paid})] \dot{-} (V_2 \dot{-} V_1).$$

One advantage of our approach is that it produces queries that can be further optimized by a query optimizer. Consider the following example. Suppose that we have a view WellPaid defined as

$$\text{WellPaid} = \Pi_{\text{Name}}(\sigma_{\text{Salary}>50,000}(\text{Employees}))$$

Now if a deletion has been made to Employees, then we compute

$$\triangledown\text{WellPaid} = \Pi_{\text{Name}}(\sigma_{\text{Salary}>50,000}(\triangledown\text{Employees}))$$

We have treated deletions and insertions as black boxes, but often they are specified in some transaction language or as queries. For example, if $\triangledown\text{Employees} = \sigma_{\text{Salary}<5,000}(\text{Employees})$, then we can substitute this value for $\triangledown$Employees in the equation for $\triangledown$WellPaid, obtaining

$$\Pi_{\text{Name}}(\sigma_{\text{Salary}>50,000}(\sigma_{\text{Salary}<5,000}(\text{Employees})))$$

for $\triangledown$WellPaid. Any query optimizer that "knows" that $\sigma_{p_1}(\sigma_{p_2}(S)) = \sigma_{p_1\&p_2}(S)$ and that $5 < 50$ will figure out that $\triangledown\text{WellPaid} = \emptyset$ and no computation needs to be done.

5 Aggregate Functions

Most database query languages provide a number of aggregate functions such as COUNT, TOTAL, AVG, STDEV, MIN, MAX [23, 12, 30]. It was noticed in [20, 22] that a number of aggregates (in fact, all of the above except MIN and MAX) can be expressed if the query language is endowed with arithmetic operations and the following summation operator:

$$\Sigma_f \ \{\!|x_1, \ldots, x_n|\!\} \ = \ f(x_1) + \ldots + f(x_n)$$

For example, COUNT is Σ_1 where the function 1 always returns 1; TOTAL is Σ_{id}, AVG is TOTAL/COUNT. For more complex examples, see [20, 21].

Any strongly minimal solution for the view maintenance problem allows us to handle duplicates correctly because the following will hold:

$$\Sigma_f \ ((S \dot{-} \triangledown S) \uplus \triangle S) \ = \ (\Sigma_f(S) - \Sigma_f(\triangledown S)) + \Sigma_f(\triangle S)$$

Now if an aggregate function is defined as $\mathsf{AGR}(S) = \varphi(\Sigma_{f_1}(S), \ldots, \Sigma_{f_k}(S))$ where φ is an arithmetic expression in k arguments, to be able to maintain the value of AGR when the view S changes, one has to keep k numbers, $\Sigma_{f_i}(S)$, $i = 1, \ldots, k$. Once changes to the view ($\triangledown S$ and $\triangle S$) become known, the values of Σ_{f_i} are recomputed by the formula above and then φ is applied to obtain the value of AGR.

For example, $\mathsf{AVG}(S) = \mathsf{TOTAL}(S)/\mathsf{COUNT}(S) = \Sigma_{id}(S)/\Sigma_1(S)$. Assume that $n = \mathsf{TOTAL}(S)$ and $m = \mathsf{COUNT}(S)$. If S changes and a strongly minimal solution $S^{\text{n}} = (S \dot{-} \triangledown S) \uplus \triangle S$ is computed, let $n_1 = \Sigma_{id}(\triangledown S)$, $n_2 = \Sigma_{id}(\triangle S)$, $m_1 = \Sigma_1(\triangledown S)$, $m_2 = \Sigma_1(\triangle S)$. Then $\mathsf{AVG}(S^{\text{n}})$ can be computed as $(n - n_1 + n_2)/(m - m_1 + m_2)$. Notice that all additional computation of aggregates is performed on changes to the views, so one may expect it to be fast.

Two aggregates that require special treatment are MIN and MAX. Assume that $\mathsf{MIN}(S) = n$, and we want to compute $\mathsf{MIN}(S^{\text{n}})$ where $S^{\text{n}} = (S \dot{-} \triangledown S) \uplus \triangle S$ is strongly minimal. If we compute $m = \mathsf{MIN}(\triangledown S)$ and $k = \mathsf{MIN}(\triangle S)$, then $k \leq n$ implies $\mathsf{MIN}(S^{\text{n}}) = k$ and $m > n$ implies $\mathsf{MIN}(S^{\text{n}}) = \min(n, k)$. However, if $n = m$ and $k \geq n$, then there is no way to say what the value of $\mathsf{MIN}(S^{\text{n}})$ is for the minimal value n can be reached at several elements of S and we do not know if *all* of them were deleted in $\triangledown S$. Thus, in only this case one has to recompute S^{n} in order to evaluate MIN correctly.

6 Complexity Analysis

While it is generally faster to compute changes to the view from changes to base relations rather than recompute the whole view from scratch, this is only a heuristic and need not be true in all cases. Changes to base relations are also typically small, but it is conceivable that in some situations a base relation R can be replaced by another relation R'. In this case $\triangledown R = R$ and $\triangle R = R'$, so changes to R are not small compared to R itself. If these changes "dominate" computing $\triangledown(t, Q)$ and $\triangle(t, Q)$, then one should not expect a significant improvement in time and space efficiency from using the change propagation algorithm.

All this tells us that it is impossible to prove a general statement saying that it is better to use the change propagation algorithm rather than recompute the view. This is also one of the reasons why so little effort has been devoted to the complexity analysis of the view maintenance problem. But intuitively, if changes are small, computing solutions for pre-expressions should be easier than computing pre-expressions themselves. In particular, one may expect that in most cases the sizes of $\triangle S$ and $\triangledown S$ are small compared to the size of S, and these are relatively easy to compute. In this section we present an attempt to formalize this statement.

Our approach is the following. We define two functions on $\mathcal{BA}$ expressions. These functions give a reasonable time (or space) estimate for computing the delta-expressions for the change propagation algorithm (the function $\mathbf{t}_\triangle$) and for recomputing the view from scratch (the function $\mathbf{t}_{view}$). Then we shall prove that if changes to base relations are small, the expected complexity of evaluating $\triangle(t, Q)$ and $\triangledown(t, Q)$ is small compared to the expected complexity of re-evaluating Q on changed arguments. In other words, $\mathsf{t}_\triangle(\triangle(t,Q)) + \mathsf{t}_\triangle(\bigtriangledown(t,Q))$ is small compared to $\mathsf{t}_{view}(\mathrm{pre}(t,Q))$. The special form of $\bigtriangledown(t,Q)$ and $\triangle(t,Q)$ where all expressions that are hard to evaluate occur inside the scope of a simpler $\bigtriangledown$ or $\triangle$ will play the crucial role.

Our first step is to define t_{view}. We give an *optimistic* estimate for t_{view}, because our goal is to prove that generally recomputing the view is more expensive than maintaining it. We first define $\mathsf{t}_{view}(R) = \mathsf{size}(R)$ for any base relation R. For binary operation define

$$\begin{array}{rl} & \mathsf{t}_{view}(S \min T) = \mathsf{t}_{view}(S \uplus T) = \mathsf{t}_{view}(S \min T) \\ = & \mathsf{t}_{view}(S \max T) \;=\; \mathsf{t}_{view}(S) + \mathsf{t}_{view}(T) \end{array}$$

The idea is that to compute the new view, we have to compute S and T, and then, being optimistic, we disregard the time needed to compute min, max, $\uplus$ or $\dot{-}$. For cartesian product, define $\mathsf{t}_{view}(S \times T) = \mathsf{t}_{view}(S) \cdot \mathsf{t}_{view}(T)$. Finally, for unary operations we use the optimistic estimate again, and disregard overhead for doing computation on the argument. That is,

$$\mathsf{t}_{view}(\sigma_p(S)) \;=\; \mathsf{t}_{view}(\Pi_A(S)) \;=\; \mathsf{t}_{view}(\epsilon(S)) \;=\; \mathsf{t}_{view}(S)$$

To define the function $\mathsf{t}_\triangle$ that estimates a reasonable evaluation time for expressions used in the change propagation algorithm, we use the special form of the expressions in figure 3 that allow us to iterate over subexpressions in scope of $\bigtriangledown$ or $\triangle$, as was explain before. To do this, as the first step, we define a new function $\mathsf{fetch}(S)$ that estimates the complexity of *retrieving* a given element from the value of $S(R_1,\ldots,R_n)$. We assume that $\mathsf{fetch}(R_i)$s are given and bounded above by some number F. Then for any binary operation $* \in \mathcal{BA}$ we define $\mathsf{fetch}(S * T) = \mathsf{fetch}(S) + \mathsf{fetch}(T)$. For example, to retrieve x from $S \times T$, we first retrieve x's projection onto attributes of S from S, and then x's projection onto T's attributes from T, and use the result to obtain the right number of x's duplicates in $S \times T$. For $\sigma_p(\cdot)$ and $\epsilon(\cdot)$ we assume $\mathsf{fetch}(\sigma_p(S)) = \mathsf{fetch}(\epsilon(S)) = \mathsf{fetch}(S)$ as an upper bound. Finally, we make an assumption that $\mathsf{fetch}(\Pi_A(S)) = \mathsf{fetch}(S)$ which need not be true in general but holds if the index on S is not projected out. As we explained in the introduction, if the index does get projected out, there is no guarantee of winning in terms of time, but we still win in terms of space. Indeed, the space occupied by $\Pi_A(S)$ is bounded by the space needed for S itself, and then the following theorem can be seen as a confirmation of the fact that one should expect to reduce the *space* complexity.

Now we define inductively the estimated time complexity of evaluation of $\triangle S$ and $\bigtriangledown S$. First, we assume that for any base relation R, $\mathsf{t}_\triangle(\triangle R) = \mathsf{size}(\triangle R)$ and $\mathsf{t}_\triangle(\bigtriangledown R) = \mathsf{size}(\bigtriangledown R)$. In the definitions for $\mathcal{BA}$ operations we disregard time needed for projecting out some fields or checking the selection conditions, assuming that it is constant. We also assume that the number of duplicates is known for all elements, and disregard the computational overhead of duplicate elimination. That is,

$$\mathsf{t}_\triangle(\sigma_p(S)) \;=\; \mathsf{t}_\triangle(\Pi_A(S)) \;=\; \mathsf{t}_\triangle(\epsilon(S)) \;=\; \mathsf{t}_\triangle(S)$$

For operations $\uplus$, max and $\times$ we define $\mathsf{t}_\triangle(S \uplus T) = \mathsf{t}_\triangle(S \max T) = \mathsf{t}_\triangle(S) + \mathsf{t}_\triangle(T)$ and $\mathsf{t}_\triangle(S \times T) = \mathsf{t}_\triangle(S) \cdot \mathsf{t}_\triangle(T)$. The only thing out of ordinary in the definition of $\mathsf{t}_\triangle$ is the clauses for min and $\dot{-}$:

$$\begin{array}{rcl} \mathsf{t}_\triangle(S \dot{-} T) & = & \mathsf{t}_\triangle(S) \cdot \mathsf{fetch}(T) \\ \mathsf{t}_\triangle(S \min T) & = & \min(\mathsf{t}_\triangle(S) \cdot \mathsf{fetch}(T), \mathsf{t}_\triangle(T) \cdot \mathsf{fetch}(S)) \end{array}$$

Unlike in the case of $\uplus$, max and $\times$, elements of T need not be stored as they are only used to reduce the size of S. Hence, to compute $S \dot{-} T$ or $S \min T$, one only has to fetch elements of the computed value S from T, and that requires $\mathsf{fetch}(T)$ rather than $\mathsf{t}_\triangle(T)$ time for each element in S. In the case of min, which is a symmetric operation, we can alternatively iterate over T; the estimated time complexity is obtained by taking the minimum of the two possible iterations.

Let $\mathcal{D} = \{R_1,\ldots,R_n\}$ be a family of base relations stored in a database. We assume that a transaction t is fixed for the remainder of the section, and omit it in all definitions. Define

$$c(\mathcal{D}) \;=\; \max_{i=1,\ldots,n} \frac{\mathsf{size}(\bigtriangledown R_i) + \mathsf{size}(\triangle R_i)}{\mathsf{size}(R_i)}$$

That is, $c(\mathcal{D})$ gives the upper bound on the relative size of the changes to base relations. The following result shows that if $c(\mathcal{D})$ is small, then one should expect to win in terms of time (or space) by using the change propagation algorithm.

Theorem 4 *Let $Q(R_1,\ldots,R_n)$ be a $\mathcal{BA}$ expression. Let $\triangle Q$ and $\bigtriangledown Q$ be calculated according to the change propagation algorithm. Then*

$$\lim_{c(\mathcal{D})\to 0} \frac{\mathsf{t}_\triangle(\bigtriangledown Q) + \mathsf{t}_\triangle(\triangle Q)}{\mathsf{t}_{view}(\mathrm{pre}(Q))} = 0$$

Let us apply this theorem to our working example. Recall that the positive change to the view Unpaid was calculated as

$$\begin{array}{rl} & \triangle\mathrm{Unpaid} \\ = & (\Pi_{\mathrm{Pid,Cost}}(\bigtriangledown\mathrm{Paid}) \dot{-} \Pi_{\mathrm{Pid,Cost}}(\triangle\mathrm{Paid})) \dot{-} (V_2 \dot{-} V_1) \end{array}$$

Assuming that for base relations the value of the fetch function equals F, we obtain $\mathsf{t}_\triangle(\triangle\mathrm{Unpaid}) = \mathsf{size}(\bigtriangledown\mathrm{Paid}) \cdot 2F^2 = O(\mathsf{size}(\bigtriangledown\mathrm{Paid}))$. Similarly, $\mathsf{t}_\triangle(\bigtriangledown\mathrm{Unpaid}) = O(\mathsf{size}(\triangle\mathrm{Paid}))$. Therefore, changes to Unpaid can be expected to be calculated in $O(\mathsf{size}(\bigtriangledown\mathrm{Paid}) + \mathsf{size}(\triangle\mathrm{Paid}))$ time. One can derive the same result just by looking at the expressions for

$\bigtriangledown$Unpaid and $\bigtriangleup$Unpaid. Indeed, to calculate $\bigtriangleup$Unpaid, we iterate over $\bigtriangledown$Paid and fetch its elements from $\bigtriangleup$Paid, V_1 and V_2 and then compute the value of an arithmetic expression. The time needed for that is linear in the size of $\bigtriangledown$Paid, assuming F is constant.

On the other hand, to recompute the view Unpaid, one should expect to spend time $O(\mathsf{size}(S_1) + \mathsf{size}(S_2))$, and this is exactly what $\mathsf{t}_{view}(\text{pre}(\text{Unpaid}))$ is. If sizes of $\bigtriangledown$Paid and $\bigtriangleup$Paid are small, this tells us that it is better to compute $\bigtriangledown$Unpaid and $\bigtriangleup$Unpaid than to recompute Unpaid.

One may ask what happens if one tries to use the same evaluation strategy for both change propagation and recomputing the view. It should not be surprising that in several cases the complexity of both is the same, as we should not always expect to win by propagating changes. To give an example, let R_1, R_2 and R_3 be base relations, where R_1's attributes are a_1, a_2, R_2's sole attribute is a_1 and R_3's attribute is a_2. Define our view as $V := R_1 \min (R_2 \times R_3)$. Now assume that $\mathsf{size}(R_i) = n$, $i = 1, 2, 3$. Assume that $\mathsf{fetch}(R_i) = F$ is constant. Then it is easy to see that $\mathsf{t}_\Delta(\bigtriangledown V) = O(n)$ and $\mathsf{t}_\Delta(\bigtriangleup V) = O(n)$.

Now assume that changes to base relations R_is are small. Then one can use the evaluation strategy that gave us the function t_Δ and calculate that $\mathsf{t}_\Delta(\text{pre}(V)) = O(n)$, where $\text{pre}(V) = (R_1 \dot{-} \bigtriangledown R_1) \uplus R_1) \min(((R_2 \dot{-} \bigtriangledown R_2) \uplus R_2) \times ((R_2 \dot{-} \bigtriangledown R_2) \uplus R_2))$. The reason for this is that it is not necessary to calculate the second argument of min as we only have to retrieve certain elements from it.

This example shows that even for a simple view definition it may be the case that using the change propagation algorithm is as complex as recomputing the view from scratch, provided that we do not use a straightforward evaluation strategy (corresponding to t_{view}).

In some special cases of sublanguages of $\mathcal{BA}$ precise statements about the complexity of evaluation of changes to the views can be proved. As a consequence, we shall see that for one special class of views the ratio of $\mathsf{t}_\Delta(\bigtriangleup Q) + \mathsf{t}_\Delta(\bigtriangledown Q)$ and $\mathsf{t}_\Delta(\text{pre}(Q))$ is guaranteed to be small if so are changes to the base relations.

Proposition 1 *Let $Q(R_1, \ldots, R_n)$ be a projection- and product-free $\mathcal{BA}$ expression such that all conditions for selections can be calculated in $O(1)$ time. Let $n = \sum_{i=1}^n (\mathsf{size}(\bigtriangledown R_i) + \mathsf{size}(\bigtriangleup R_i))$. Let the upper bound for the time needed for retrieving an element from a base relation be a constant. Then the complexity of evaluation of $\bigtriangledown Q$ and $\bigtriangleup Q$ is bounded above by $O(n)$.*

This proposition can be seen as a corollary of a more general result. We define Δ-*controlled expressions* by means of the following grammar:

$$\begin{array}{rcl} S_\Delta & := & \bigtriangleup R \mid \bigtriangledown R \mid S_\Delta \min T \mid S_\Delta \dot{-} T \mid \\ & & S_\Delta \uplus S_\Delta \mid S_\Delta \max S_\Delta \mid \epsilon(S_\Delta) \mid \sigma_p(S_\Delta) \end{array}$$

where T is an arbitrary projection- and product-free $\mathcal{BA}$ expression.

Proposition 2 *Under the assumptions of proposition 1, any Δ-controlled expression can be evaluated in time $O(n)$.*

Corollary 5 *Let $\mathcal{D}$ be defined as above and*

$$c'(\mathcal{D}) = \max_i \frac{\sum_j \mathsf{size}(\bigtriangledown R_j) + \mathsf{size}(\bigtriangleup R_j)}{\mathsf{size}(R_i)}$$

Let $Q(R_1, \ldots, R_n)$ be a projection- and product-free $\mathcal{BA}$ expression such that all conditions for selections can be calculated in $O(1)$ time. Then

$$\lim_{c'(\mathcal{D}) \to 0} \frac{\mathsf{t}_\Delta(\bigtriangleup Q) + \mathsf{t}_\Delta(\bigtriangledown Q)}{\mathsf{t}_\Delta(\text{pre}(Q))} = 0$$

Summing up the results of this section, theorem 4 says that it is generally easier to compute changes to a materialized view than to recompute the view from scratch, although we demonstrated that it is not always the case. For a restricted class of view, we proved that computing changes is always more efficient.

7 Related Work

Our approach is closest to that of [9], which treats the standard relational algebra. That work grew out of an analysis of [29], which in turn was influenced by the notion of 'finite differencing' of [25]. The algorithm for change propagation in [29] is an iterative one that propagates changes, one-by-one, to the top of an expression. It was shown in [9] that this is not enough to guarantee strong minimality. Instead [9] defines recursive functions to compute change sets, as we have done here, and proves correctness by induction.

The only work on change propagation for multisets is [12], which is done in the context of a modified Datalog where programs produce multisets. Informally, a tuple's multiplicity in the multiset resulting from the evaluation of a program P indicates the number of different possible *derivations* showing that it was produced by P using Datalog semantics (see [12]).

Given a program P and a transaction t, the change propagation algorithm of [12] produces a program P^{n} by concatenating the clauses of program P with the clauses of a new program $\Delta^{\pm} P$. Concatenation corresponds to the additive union operation. The program $\Delta^{\pm} P$ is defined so that for any database state s, the evaluation of P^{n} in state s will result in the same multiset as the evaluation of program P in the new state $t(s)$. If P is a materialized query, then in order to evaluate P in the new state we need only evaluate the clauses of $\triangle^{\pm} P$ in the old state and form this union with the old (stored) value of P. In order to make this work with deletions, the semantics of [12] allows for *negative* multiplicities in the change sets $\triangle^{\pm} P(s)$.

For example, consider the program

$$\text{minus}(X) \quad :- \quad S(X)\ \&\ \neg T(X).$$

If we have a database transaction that induces changes to both S and T, then the algorithm of [12] produces the program $\triangle^{\pm}$minus with clauses

$$\begin{array}{lll} \text{minus}(X) & :- & \triangle^{\pm} S(X)\ \&\ \neg T(X). \\ \text{minus}(X) & :- & S^{\text{n}}(X)\ \&\ \overline{\triangle^{\pm} T}(X). \end{array}$$

where $\overline{\triangle^{\pm} T}$ computes a set W such that

$$\mathit{count}(x, W) = \begin{cases} -1 & \text{if } x \in \triangle^{\pm} T \text{ and } x \notin T \uplus \triangle^{\pm} T \\ 1 & \text{if } x \in \triangle^{\pm} T \text{ and } x \notin T \\ 0 & \text{otherwise} \end{cases}$$

There are many differences between our approach and that of [12]. First, we are treating different query languages. The nonrecursive fragment of the language of [12] cannot represent our operations of duplicate elimination, monus, min, and max. This follows from general results on the expressive power of bag languages [19]. On the other hand, our language does not handle GROUPBY or recursive queries, as does [12].

Our approach does not require negative multiplicities. If a program P can be represented as a $\mathcal{BA}$ expression $\hat{P}$, then an incremental change program $\triangle^{\pm} P$ can be represented in $\mathcal{BA}$ as a pair of queries $(\bigtriangledown \hat{P}, \triangle \hat{P})$ where $\bigtriangledown \hat{P}$ $(\triangle \hat{P})$ represents those tuples of $\triangle^{\pm} P$ with a negative (positive) multiplicity. Then program P^{n} corresponds to $(\hat{P} \dot{-} \bigtriangledown \hat{P}) \uplus \triangle \hat{P}$.

This highlights the fact that our approach is *linguistically closed.* That is, we give explicit algebraic representations to *all* expressions generated in change propagation, and these are represented in the language $\mathcal{BA}$. For example, while [12] must extend their language with a new operation in order to evaluate the program $\overline{\triangle^{\pm} T}$, we would represented this operation explicitly as the pair of queries

$$(\epsilon(\triangle(t, T)) \dot{-} T, \quad \epsilon(\bigtriangledown T) \dot{-} (T \dot{-} \bigtriangledown T)).$$

This makes additional optimizations possible, both in the process of generating change expressions and in any later optimization stages.

Next, our approach gives a declarative semantics to change propagation that is not tightly bound to one computational model. That is, we have an *algebraic* approach rather than an *algorithmic* one. This makes correctness proofs much easier, and also simplifies the process of extending the algorithm to new constructs. It also allows us to apply our results to problems other than view maintenance. For example, suppose that we are given the integrity constraint

$$\sigma \stackrel{\text{def}}{=} (\forall x \in R_1)\ x.a = \text{count}\ \underbrace{\{\!| z \in R_2 : z.b = x.b |\!\}}_{\text{multiset}}$$

and a strongly minimal transaction $t = \{R_2 \leftarrow (R_2 \dot{-} \bigtriangledown R_2) \uplus \triangle R_2\}$. Furthermore, suppose that we would like to transform t to a *safe* transaction,

$$t' = \ \textsf{if}\ \alpha\ \textsf{then}\ t\ \textsf{else}\ \mathit{abort},$$

that can never leave the database in a state violating σ. If we assume that σ will always hold before t' is executed, then we can use our algorithm, together with some logical manipulations, to produce

$$\begin{array}{l} (\forall x \in R_1) \\ \quad \text{count}\{\!| z \in \bigtriangledown R_2 : z.b = x.b |\!\} \\ \quad\ = \text{count}\{\!| z \in \triangle R_2 : z.b = x.b |\!\} \end{array}$$

as the formula α. Indeed, this type of problem provided the original motivation for our work (see [8]).

Finally, we are able to use the inductive assumptions of strong minimality to further simplify our solutions. Since this information is not available to a general purpose query optimizer, it may fail to produce an efficient solution that can be found with our approach.

A comparison of performance must wait for implementations of the two approaches.

8 Further Work

Our use of strong minimality in the simplification of queries suggests that this information should be available to a *specialized* query optimizer. We are currently working on the design of such an optimizer based on a collection of inference rules for deriving disjointness (for example, if S is disjoint form T, then $S \dot{-} W$ is disjoint from $T \dot{-} Z$) and simplification rules that exploit disjointness (for example, if S is disjoint from T, then $S \dot{-} T$ simplifies to S). The optimization process is initiated by recognizing that all pairs produced by our algorithm, $(\bigtriangledown S, \triangle S)$, are disjoint.

The work of [12] does handle recursive Datalog programs. One current drawback to our approach is that, as with the relational algebra, bag languages such as $\mathcal{BA}$ cannot express recursive queries [22]. We hope to address this issue in the future by extending $\mathcal{BA}$ with loops or a fixed-point operator, as in [13, 19].

The other extension of our approach deals with complex objects. Our bag algebra $\mathcal{BA}$ is the flat fragment of what was originally designed as an algebra for nested bags. We are currently working on an approach that allows us to extend the equations of the change propagation algorithm to complex objects.

Acknowledgements. We would like to thank Rick Hull for directing our attention to some of the relevant literature, Inderpal Mumick for his very helpful discussions, and Doug McIlroy and Jon Riecke for their careful reading of our working drafts.

References

[1] S. Abiteboul and V. Vianu. Equivalence and optimization of relational transactions. *J. ACM* 35 (1988), 70–120.

[2] J. Albert. Algebraic properties of bag data types. In *Proceedings of Very Large Databases–91*, pages 211–219.

[3] J. Blakeley, P.-Å. Larson, and F. W. Tompa. Efficiently udpdating materialized views. In *SIGMOD 86*, pages 61–71, 1986.

[4] P. Buneman and E. Clemons. Efficiently monitoring relational databases. *ACM TODS* 4(1979), 368–382.

[5] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proc. VLDB-91.*

[6] S. Ghandeharizadeh, R. Hull, D. Jacobs et al. On implementing a language for specifying active database execution models. In *Proc. VLDB-93.*

[7] S. Ghandeharizadeh, R. Hull and D. Jacobs. Implementation of delayed updates in Heraclitus. In *EDBT-92.*

[8] T. Griffin and H. Trickey. Integrity Maintenance in a Telecommunications Switch. IEEE Data Engineering Bulletin, Special Issue on Database Constraint Management. 17(2) : 43–46, 1994.

[9] T. Griffin, L. Libkin, and H. Trickey. A correction to "Incremental recomputation of active relational expressions" by Qian and Wiederhold. Technical Report, AT&T Bell Laboratories, Murray Hill NJ, 1994.

[10] T. Griffin and L. Libkin. Incremental Maintenance of Views with Duplicates. Technical Report, AT&T Bell Laboratories, Murray Hill NJ, 1995.

[11] S. Grumbach and T. Milo. Towards tractable algebras for bags. In *PODS-93*, pages 49–58.

[12] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD-93*, pages 157–166.

[13] M. Gyssens and D. Van Gucht. The powerset algebra as a natural tool to handle nested database relations. *JCSS* 45 (1992), 76–103.

[14] R. Hull and D. Jacobs. Language constructs for programming active databases. In *Proc. VLDB-91*, pages 455-468.

[15] P. Kanellakis. Elements of relational database theory. In *Handbook of Theoretical Computer Science*, vol. B. North Holland, 1989, pages 1075-1176.

[16] A. Klausner and N. Goodman. Multirelations: semantics and languages. In *Proc. VLDB-85*, pages 251–258.

[17] A. Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *Journal of the ACM* 29 (1982), 699–717.

[18] V. Kuchenhoff. On the efficient computation of the difference between consecutive database states. In *DOOD*, LNCS 566, 1991, pp. 478–502.

[19] L. Libkin and L. Wong. Some properties of query languages for bags. In *Proc. Database Programming Languages*, Springer Verlag, 1994, pages 97–114.

[20] L. Libkin and L. Wong. Aggregate functions, conservative extension and linear order. In *Proc. Database Programming Languages*, Springer, 1994, pages 282–294.

[21] L. Libkin and L. Wong. Conservativity of nested relational calculi with internal generic functions. *Information Processing Letters* 49 (1994), 273–280.

[22] L. Libkin and L. Wong. New techniques for studying set languages, bag languages and aggregate functions. In *PODS-94*, pages 155–166.

[23] I.S. Mumick, H. Pirahesh, R. Ramakrishnan. The magic of duplicates and aggregates. In *VLDB-90.*

[24] G. Ozsoyoglu, Z. M. Ozsoyoglu, and V. Matos. Extending relational algebra and relational calculus with set-valued attributes and aggregate functions. *ACM Transactions on Database Systems*, 12 (1987), 566–592.

[25] R. Paige. Applications of finite differencing to database integrity control and query/transaction optimization. In H. Gallaire, et al., eds., *Advances in Database Theory*, pages 170–209. Plenum Press, New York, 1984.

[26] O. Shmueli and A. Itai. Maintenance of views. *SIGMOD Record* 14 (1984), 240–255.

[27] M. Stonebraker. Implementation of integrity constraints and views by query modification. In *SIGMOD-75*, pages 65–78.

[28] X. Qian. An axiom system for database transactions. *Information Processing Letters* 36 (1990), 183–189.

[29] X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. *IEEE Trans. on Knowledge and Data Engineering*, 3(3):337–341, 1991.

[30] J. Ullman. *Principles of Database and Knowledge-base Systems.* Computer Science Press, 1988.

[31] J. Van den Bussche and J. Paredaens. The expressive power of structured values in pure OODB. In *PODS-91.*