Counting solutions to the View Maintenance Problem

Ashish Gupta [*]
Dinesh Katiyar
Department of Computer Science
Stanford University
CA 94305, USA
{agupta,katiyar}@cs.stanford.edu

Inderpal Singh Mumick
AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974, USA
mumick@research.att.com

Abstract

We present an incremental evaluation algorithm for materialized views in relational and deductive database systems. The algorithm computes, in an incremental fashion, the changes to the materialized view in response to changes (insertions, deletions, and updates) to the base relations. The view may be defined in SQL or in Datalog, and may use `UNION` , negation, aggregation (*e.g.* `SUM, MIN`), linear recursion, and general recursion. The algorithm is optimal in that it computes exactly those view tuples that are inserted or deleted.

The algorithm works by tracking the number of derivation trees ([Mum91, MS92]) for each tuple in the view. The number of derivation trees for a tuple corresponds to the count of a tuple in the duplicate semantics used in relational systems such as those based on SQL. For deductive databases using set semantics, we show that the number of derivation trees for nonrecursive queries can be computed at little or no cost above the cost of evaluating the query. For recursive queries, keeping track of derivation trees may require some additional work. A restriction of our algorithm is that it is applicable to a recursive query only if the number of derivation trees is finite.

1 Introduction

A view is a derived (idb) relation defined in terms of base (edb) relations. A view can be materialized by storing its extent in the database. Index structures can be built on the materialized view. Consequently, database accesses to materialized view tuples is much faster than by recomputing the view. However, deletion, insertions, and updates to the edb relations can induce changes in the view. Recomputing the view from scratch is too wasteful in most cases. Using the heuristic of inertia (only a part of the view changes in response to changes in the edb), it is often cheaper to compute only the changes in the view. We stress that the above is only a heuristic. For example, if an entire edb relation is deleted, it will be cheaper to recompute a view (to an empty relation) than to compute the changes to the view.

Algorithms that compute changes to a view in response to changes to the edb are called *incremental view maintenance* algorithms. Several such algorithms with different applicability domains have been proposed [BC79, SI84, BLT86, BlTo88, BCL89, QW91, WDSY91, CW91, DT92]. View maintenance also has applications in integrity constraint maintenance and index maintenance.

We present a new algorithm for incremental maintenance of a large class of views, including nonrecursive and recursive views (in SQL or Datalog extensions) defined using negation, grouping with aggregation, and union, in response to insertions, deletions and updates to the edb. The algorithm works by counting the number of derivation trees for each tuple in the view. We illustrate the approach with the following example:

[*]Work was supported by NSF grants IRI-91-16646 and IRI-90-16358, and ARO DAAL-03-G-0177.

Example 1.1 Consider the transitive closure program T:

$(T1)$: $\texttt{path}(X, Y)$:- $\texttt{edge}(X, Y)$.
$(T2)$: $\texttt{path}(X, Y)$:- $\texttt{path}(X, Z)$ & $\texttt{edge}(Z, Y)$.

Given the edb $\texttt{edge} = \{12, 23, 13\}$, the tuple $\texttt{path}(1, 3)$ has two derivations: one from rule $T1$ using $\texttt{edge}(1, 3)$, and the other from rule $T2$ using $\texttt{path}(1, 2)$ and $\texttt{edge}(2, 3)$. Tuples $\texttt{path}(1, 2)$ and $\texttt{path}(2, 3)$ have only one derivation. We represent this by saying $\texttt{path} = \{12, 23, 13 * 2\}$, where $13 * 2$ means that the tuple $\texttt{path}(1, 3)$ has 2 derivations. In general $t * N$ indicates that the tuple t has N derivations. If $N = 1$, we simplify $f * 1$ to f.

Now if $\texttt{edge}(1, 2)$ is deleted, our algorithm computes that one derivation each for $\texttt{path}(1, 2)$ and $\texttt{path}(1, 3)$ is deleted, and derives the new relation $\texttt{path} = \{23, 13\}$. □

The number of derivation trees for a tuple t is equal to the multiplicity ($\texttt{count}(t)$) of the tuple t under duplicate semantics, and has been defined for positive programs [Mum91] and for programs with stratified negation [MS92]. Since our incremental view maintenance algorithm works by counting the number of derivation trees, recursive queries can be handled only if they have a finite number of derivation trees. An algorithm to detect finiteness is given in [MS92]. On nonrecursive queries we show that counts can be computed at little or no cost above the cost of evaluating the query (Section 5).

Our incremental view maintenance algorithm works as follows: Given a program T defining a set of views $V_1, \ldots, V_k$, we derive, at compile time, a program T_V that takes as input the changes to edb relations and the old values of edb and view relations (with some representation of multiplicities), and produces as output the new values of the view relations, along with the new multiplicities. The program T_V does differential computation on rules of program T, in a manner similar to the way semi-naive evaluation [Ull89] does differential computation across iterations so as to derive only those tuples that could not be derived without using the newly deduced tuples. However, semi-naive evaluation cannot handle deletions, and does not track the counts of tuples.

The rest of this paper is organized as follows. Section 2 compares our work with some of the other proposals for incremental view maintenance. The notation used to describe our algorithms is given in Section 3, and in Section 4 we introduce a version of our algorithm for nonrecursive queries without negation and aggregation. Section 5 discusses implementation issues relating to maintenance of counts. We extend our algorithm to handle negation and aggregation in Section 6. General recursion is considered in Section 7. We omit a discussion of the special case of linear recursion, for which the algorithm can be made much simpler and more efficient than for general recursion. We work out the general recursion algorithm on an example in the Appendix.

2 Related Work

Our algorithms are most closely related to the count-based algorithm of Blakeley *et. al.* [BLT86] for updating select-project-join expressions (no negation aggregation, or recursion). In fact their algorithm would be a special case of our algorithm of Section 4 applied to single rule programs.

Ceri and Widom [CW91] describe a strategy to update views defined in SQL without negation, aggregation, and duplicate semantics. They use the idea that if the attributes projected out into the view functionally determine the key of some base relation used in the definition, then updates to the base relation can be translated to updates in the view definition. If the view attributes do not functionally determine keys of base relations, their algorithm does not apply. For instance consider the following simple join:

$p(X, Y)$:- $q(X, Z)$ & $r(Z, Y)$.

where the attribute Z is the key for both relations q and r. Attribute Z is not a *bound* attribute by the definition given in [CW91] and therefore updates to neither relation q nor r can be translated to updates to p. Similarly a view defined using the SQL `UNION` operator cannot be handled. Also, their approach is not likely to extend to recursive queries since deriving functional dependencies in recursive queries is undecidable [AH88].

Dong and Topor [DT92] derive a nonrecursive program to compute changes to a right-linear chain query in response to insertions into the base relation. Their algorithm is better than our general recursion algorithm for handling insertions into right-linear chain programs. However, they cannot handle deletions, negation, and/or aggregation. They do discuss a semi-naive incremental evaluation for general recursive queries in response to insertions into the edb.

3 Notation

We use Datalog, mostly as discussed in [Ull89], with stratified aggregation as described in [Mum91]. Datalog extended with stratified negation and negation can be mapped to a class of recursive SQL queries, and vice versa [Mum91]. We chose Datalog syntax over SQL syntax for ease of exposition.

Definition 3.1 Stratum Numbers (SN) and Rule Stratum Number (RSN): Stratum numbers are assigned as follows: Construct a *reduced dependency graph* of the given program by collapsing every strongly connected component of the dependency graph to a single node. A reduced graph is guaranteed to be acyclic. A topological sort [AHU83] of the reduced dependency graph assigns a stratum number to each node. If a node represents a strongly connected component, all predicates in the strongly connected component are assigned the stratum number of the node. By convention, the base predicates are assigned stratum number $= 0$. The rule stratum number of a rule r for predicate p is equal to $\mathsf{SN}(p)$. $\Box$

The extension of a relation p will be referred to by P. $P = \{ab, mn\}$ represents facts $p(a, b)$ and $p(m, n)$. Changes to predicate p are represented by the predicate $\Delta(p)$. P^ν represent the relation for predicates p after the changes.

With each fact t for p, we also maintain a $\mathsf{count}(t)$ of the number of derivations for the fact. Similarly every $\Delta(p)$ fact has a count associated with it. Negative and positive counts correspond to deletions and insertions respectively. For instance, $\Delta(p) = \{ab * 4, mn * -2\}$ says that four derivations of fact $p(a, b)$ are inserted into P and two derivations of fact $p(m, n)$ are deleted.

We define the union operator, $\uplus$, over sets of tuples with counts. Given two sets of tuples $S1$ and $S2$, $S1 \uplus S2$ is defined as follows:

1. If a tuple t appears in only one of $S1$ or $S2$ with a count c, then tuple t appears in $S1 \uplus S2$ with a count c.
2. If a tuple t appears in both $S1$ and $S2$ with counts of c_1 and c_2, where at least one of the counts must be positive, and $c_1 + c_2$ must be non-negative, then
   (a) If $c_1 + c_2 > 0$, .tuple t appear in $S1 \uplus S2$ with count $c_1 + c_2$.
   (b) If $c_1 + c_2 = 0$, .tuple t does not appear in $S1 \uplus S2$.

Thus, $P^\nu = P \uplus \Delta(p)$. The join operator is also redefined for relations with counts: when two or more tuples join, the count of the resulting tuple is a product of the counts of the tuples joined. The correctness of the algorithms in the paper guarantees that no tuple in either relation P or P^ν has a negative count. Tuples in the change relations $\Delta(p)$ can have negative counts.

4 Incremental Maintenance of Nonrecursive Views

In this section we present a basic version of our incremental view maintenance algorithm for the case of nonrecursive views without negation or aggregation. The algorithm handles both insertions and deletions to edb relations, and models updates as deletions followed by insertions. The algorithm computes changes stratum by stratum, working from the lowest stratum to the highest stratum.

We consider a simple example to give the intuition behind our algorithm.

Example 4.1 Consider the following simple view definition for derived relation p in terms of base relations a and b. Using the tuples in A and B, the relations for p are also computed.

(v): $\quad p(X,Y) :- a(X,Z) \ \& \ b(Z,Y).$

If the base relations a and b are changed, the tuples in the derived relation p will also change. The new sets, A^ν and B^ν are written as $A + \Delta(a)$ and $B + \Delta(b)$ respectively. Note that $\Delta(a)$ and $\Delta(b)$ can contain both inserted and deleted tuples represented by positive and negative counts respectively. The new set of facts for p can be computed by the following program:

$p^\nu(X,Y) :- a^\nu(X,Z) \ \& \ b^\nu(Z,Y).$

Replacing a by $(a + \Delta(a))$ and b by $(b + \Delta(b))$ and distributing joins over unions, this rule can be written out as the following set of rules:

$p^\nu(X,Y) :- a(X,Z) \ \& \ b(Z,Y).$
$p^\nu(X,Y) :- \Delta(a)(X,Z) \ \& \ b(Z,Y).$
$p^\nu(X,Y) :- a(X,Z) \ \& \ \Delta(b)(Z,Y).$
$p^\nu(X,Y) :- \Delta(a)(X,Z) \ \& \ \Delta(b)(Z,Y).$

The first rule recomputes the old value of p. The remaining three rules then compute $\Delta(p)$, the changes to p. We therefore have:

$\Delta(p)(X,Y) :- \Delta(a)(X,Z) \ \& \ b(Z,Y).$
$\Delta(p)(X,Y) :- a(X,Z) \ \& \ \Delta(b)(Z,Y).$
$\Delta(p)(X,Y) :- \Delta(a)(X,Z) \ \& \ \Delta(b)(Z,Y).$

The last two rules can be combined, using the fact that $a^\nu = a \uplus \Delta(a)$ to give the simplified rules:

$(v1)$: $\quad \Delta(p)(X,Y) :- \Delta(a)(X,Z) \ \& \ b(Z,Y).$
$(v2)$: $\quad \Delta(p)(X,Y) :- a^\nu(X,Z) \ \& \ \Delta(b)(Z,Y).$

We apply the previous rules to an example as follows:

$A = \{12, 16, 23, 24\}, \quad B = \{28, 68, 39\}.$
$P = \{18 * 2, 29\}.$

$\Delta(a) = \{16 * -1\} \ , \ \Delta(b) = \{68 * -1, 49, 47\}.$
$A^\nu = \{12, 23, 24\} \ , \ B^\nu = \{28, 39, 49, 47\}.$

Substituting the values of $\Delta(a)$, $\Delta(b)$, and A^ν in rules $v1$ and $v2$ we get the value of $\Delta(p)$ and P^ν as follows:

rule $v1$ gives $\Delta(p) = \{18 * -1\}$.
rule $v2$ gives $\Delta(p) = \{29, 27\}$.
$P^\nu = P \uplus \Delta(p) = \{18 * 2, 29\} \uplus \{18 * -1, 29, 27\} = \{18, 29 * 2, 27\}.$

This is exactly the answer that joining A^ν and B^ν would give. Generalizing the algebraic manipulation done in this example, a rule $p :- a \ \& \ b \ \& \ c$ results in rules:

$\Delta(p) :- \Delta(a) \ \& \ b \ \& \ c.$
$\Delta(p) :- a^\nu \ \& \ \Delta(b) \ \& \ c.$
$\Delta(p) :- a^\nu \ \& \ b^\nu \ \& \ \Delta(c).$

This pattern is generalized and exploited in the algorithm for nonrecursive programs. However for recursive programs, this simple transformation is not correct. □

Note that the correctness of the algorithm should guarantee that all facts in non-Δ subgoals have positive count because the number of facts deleted from a base predicate are never more than the facts present.

Definition 4.1 $\Delta(p)$**:** $\Delta(p)$ is a new derived predicate that contains the changes made to a derived predicate p when some underlying base predicate changes. The next definition discusses rules that define this new predicate given rules defining the derived predicate p. □

Definition 4.2 r_i**:** Consider a rule r in program P:

(r): $\quad p :- s_1 \ \& \ \ldots \ \& \ s_n$.

Rule r_i is defined for $1 = 1 \ldots n$, as follows:

(r_i): $\quad \Delta(p) :- s_1^\nu \ \& \ \ldots \ \& \ s_{i-1}^\nu \ \& \ \Delta(s_i) \ \& \ s_{i+1} \ \& \ \ldots \ \& \ s_n$.

r_i is not defined if $i > n$. □

Algorithm 4.1

Input: A nonrecursive program $\mathcal{P}$.

Output: A program that computes the facts to be deleted/inserted in response to deletions/insertions in base predicates.

Method:

Compute the RSNs of all the rules in $\mathcal{P}$.
Mark all rules unprocessed.
For all derived predicates p, initialize $\Delta(p) = \{\}$
While there is an unprocessed rule
{ $\quad \mathcal{Q} = \{r \mid \mathsf{RSN}(r)$ is minimum among all unprocessed rules$\}$
  For every rule $r \in \mathcal{Q}$ **do**
    For $i = 1 \ldots \mathsf{length}(r)$ **do**
      Compute $\Delta(p)$ using rule r_i
        (r_i): $\quad \Delta(p) :- s_1^\nu \ \& \ \ldots \ \& \ s_{i-1}^\nu \ \& \ \Delta(s_i) \ \& \ s_{i+1} \ \& \ \ldots \ \& \ s_n$.
  $P^\nu = P \uplus \Delta(p)$
  Mark every rule in $\mathcal{Q}$ as processed.
}

◇

Theorem 4.1 *Given program P and database E say fact f had* $\mathsf{count}(f)$ *derivations. When database E is altered to $E^\nu = E \uplus \Delta(E)$, say fact f has* $\mathsf{count}(f^\nu)$ *derivations. Then Algorithm 4.1 derives fact $\Delta(f)$ with a count of* $\mathsf{count}(f^\nu) - \mathsf{count}(f)$*, for all facts f derived by program P using either database E or E^ν.* □

5 Implementation Issues and Optimizations

Algorithm 4.1 needs a count of the number of derivations for each tuple. Let us see how counts might be computed during bottom-up evaluation in a database system.

We first consider database systems that implement duplicate semantics, such as DB2 and SQL/DS from IBM, and Nonstop SQL from Tandem. The query language, SQL [ISO90] in the above language requires the duplicates to be retained for semantic correctness. In an implementation, the duplicates may be represented in one of two ways: by keeping multiple copies of a tuple, or by keeping a count with each tuple. In either case, our algorithm works without incurring any overhead due to duplicate computation. The $\uplus$ operator in our algorithm is mapped to the union operator of the system when the operands have positive counts. When an operand has negative counts, the $\uplus$ operator is equivalent to multiset difference. Though multiset difference is not provided in any of the above example SQL systems, it can be executed in time $O(n)log(n)$ or $O(n)$ (where n is the size of the operands) depending on the representation of duplicates.

Second, consider systems that have set semantics, such as *NAIL!* , and LDL. Such systems can treat duplicates in one of two possible ways during query evaluation: (1) Do not eliminate duplicates since duplicate elimination is expensive, and may not have enough payoff, and (2) Eliminate duplicates after each iteration of the semi-naive evaluation. The first implementation, likely to be useful only for nonrecursive queries, is similar to computing duplicate semantics since all derivation trees will be derived during evaluation. The second implementation removes duplicates, and so it may seem that our incremental view maintenance algorithm must do extra work to derive all the remaining derivation trees. But it is not so, at least not for nonrecursive queries.

First note that duplicate elimination is an expensive operation, and we can augment the operation to count the number of duplicates eliminated without increasing the cost of the operation. These counts can then be associated with each tuple in the relation obtained after duplicate elimination. Let us assume that we do full duplicate computation within a recursive stratum (by extending the evaluation method is some way), and then do duplicate elimination and obtain counts for each tuple computed by the stratum. When computing the next higher stratum $i+1$, we do not need to make derivations once for each count of a tuple in stratum i or less. We do not even need to carry the counts from stratum i or lower through into the evaluation of stratum $i+1$. We assume that each tuple of stratum i or less has only one derivation, and compute the duplicate semantics of stratum $i+1$. Consequently, the count value for each tuple t corresponds to the number of derivations for tuple t assuming that all tuples of lower strata have 0 or 1 derivation.

Maintaining counts as above influences the propagation of changes in a positive manner. Let predicate p in stratum 1 have 20 derivations for a fact $p(a)$, and let changes to the base facts delete 10 of them. However the changes need not be cascaded to a predicate q in stratum 2 because as far as derivations of q are concerned, $p(a)$ has a count of one as long as its actual count is positive. The incremental computation therefore stops at stratum 1.

Using the above optimizations, the extra evaluation cost incurred by our incremental view maintenance algorithm is in computing the duplicate semantics of each stratum. For a nonrecursive stratum there is usually no extra cost in computing the duplicate semantics. A nonrecursive stratum consists of a single predicate defined using one or more rules, evaluated by a sequence of select,join,project, and union operators. Each of these operators derives one tuple for each derivation. Thus, tracking counts for a nonrecursive view is almost as efficient as evaluating the nonrecursive view. For a recursive stratum duplicate semantics can be computed using a method such as duplicate semi-naive [Mum91] which is polynomial in the size of the database. The presence of an infinite number of derivation trees in a recursive stratum can also be checked in polynomial time [MS92].

Even in SQL systems implementing duplicate semantics, it is possible for a query to require set semantics (by using `DISTINCT` operator). The implementation issues for such queries are similar to the case of systems implementing set semantics.

6 Negation and Aggregation

The class of view definitions is now extended to include the difference operator. We consider only *safe* datalog programs. In addition the negation is stratified i.e. if derived predicate q depends negatively on predicate p then $\mathsf{SN}(p) < \mathsf{SN}(q)$ where $\mathsf{SN}(p)$ is the stratum number of predicate p. Nonrecursive programs are stratified by default.

A negated subgoal $\neg p(X, Y, \ldots)$ in a stratified program, is computed as follows. A negated fact $\neg p(a, b, \ldots)$ has a count of either zero or one: zero if the fact $p(a, b, \ldots)$ is true and therefore $\mathsf{count}(\mathsf{p}(\mathsf{a}, \mathsf{b}, \ldots)) > 0$ and one if the fact $p(a, b, \ldots)$ is false. This is because a negated instantiated subgoal, $\neg p(a, b, \ldots)$, is an existential check and $\neg p(a, b, \ldots)$ is false if $p(a, b, \ldots)$ can be derived even once.

As discussed in Section 4, we rewrite rules to define changes to a derived predicate in terms of changes to subgoals of the rule. In a rule with negation, the negated subgoal can occur in one of three positions in the rewritten rule r_i:

(r_i): $\Delta(p)$:- s_1^ν & $\ldots$ & s_{i-1}^ν & $\Delta(s_i)$ & $s_{i+1} \ldots$ & s_n.

- *between* 1 *and* $i-1$: s_j is a negated subgoal $\neg q$. $(\neg q)^\nu$ is the same as $\neg(q^\nu)$. This can be computed because q is in a lower stratum than p and has already been computed (Lemma 6.1).
- *between* $i+1$ *and* n: s_j is evaluated as a usual negated subgoal.

- *The i^{th} subgoal*: s_i is a negated subgoal, $\neg q$. $\Delta(\neg q)$ is computed in terms of $\Delta(q)$ and the original facts for the predicate q. Both $\Delta(q)$ and q have already been computed and therefore $\Delta(\neg q)$ can be computed without having to compute the other subgoals in rule r_i. This is very useful because the size of the Δ-subgoal is usually the most restrictive component of the join and best used first.

Definition 6.1 $\Delta(\neg q)$**:** Note that $\Delta(\neg q) \subseteq \Delta(q)$. $\Delta(\neg q)$ is made equal to $\Delta(q)$ and then for every tuple $t \in \Delta(\neg q)$, $\mathsf{count}(t)$ is defined as follows:

$$\mathsf{count}(t) = \begin{cases} 1 & \textit{if } \mathsf{count}(t) = n \textit{ in } Q \textit{ and} \\ & \mathsf{count}(t) = -n \textit{ in } \Delta(q) \textit{ andn} > 0 \\ -1 & \textit{if } \mathsf{count}(t) = 0 \textit{ in } Q \textit{ and} \\ & \mathsf{count}(t) > 0 \textit{ in } \Delta(q) \\ 0 & \textit{otherwise} \end{cases}$$

Tuples with $\mathsf{count} = 0$ are deleted from $\Delta(\neg q)$. □

Lemma 6.1 *For a negated subgoal, $\neg q$, $(\neg q)^\nu$ is equivalent to $\neg q^\nu$.* □

Theorem 6.1 *Given program P with negated subgoals, and database E say fact f had* $\mathsf{count}(f)$ *derivations. When database E is altered to $E^\nu = E \uplus \Delta(E)$, say fact f has* $\mathsf{count}(f^\nu)$ *derivations. Then Algorithm 4.1 derives fact $\Delta(f)$ with a count of* $\mathsf{count}(f^\nu) - \mathsf{count}(f)$*, for all facts f derived by program P using either database E or E^ν.* □

Aggregation can also be handled by our algorithms. We do not give details in this paper because of lack of space. The semantics of aggregation is as described in [Mum91]. The basic idea is to use the fact that an aggregate subgoal generates one tuple for every value of the set of attributes $\bar{Y}$, where $\bar{Y}$ is the attributes over which the groupby takes place. When a tuple t is inserted or deleted from the relation in the aggregate subgoal, the aggregate tuple corresponding to $\pi_{\bar{Y}}(t)$ also changes. Often the aggregate tuple does not change despite changes being made to the underlying relation. For instance the maximum salary of a group of people does not change on removing a person who does not earn the maximum. This observation is very useful because it prevents irrelevant changes from being cascaded through derived relations. In the full version of the paper we give an algorithm to evaluate the change in aggregate subgoals in response to changes in base predicates and prove that Algorithm 4.1 works correctly on programs with aggregate subgoals.

7 Recursive View Definitions

This section discusses an incremental view maintenance algorithm for recursive view definitions. To simplify the presentation of this section, we first consider deleting base facts from positive recursive programs. We develop an algorithm to compute the derived facts that depend on these deleted EDB facts. In general the algorithm can handle concurrent insertions and deletions to base predicates. Stratified negation and aggregation are handled exactly as discussed in Section 6.

We follow the following strategy. Given the deleted base facts, delete all derived facts whose first subgoal is deleted. These new deleted facts could themselves occur as the first subgoals of other derived facts that are also then deleted. This cascading stops when no deleted fact appears as the first subgoal of a fact not yet deleted. The deletion process then considers the second subgoal position. We then derive all facts that can be deleted because of either their first or second subgoals being deleted. Note that the deletion front advances along the subgoal positions and reconsiders all positions that have already been considered. The trick is to avoid recounting facts that have already been counted and consider only new deleted facts. The rewrite ensures that this is indeed the case.

Changes to derived predicates are computed in two nested loops. The outer loop is called the **mega-iteration** and corresponds to advancing the deletion front along increasing subgoal positions. The k^{th}

mega-iteration deletes either (a) those facts whose k^{th} supporting subgoal has been deleted, or (b) those that have a supporting subgoal sg in a position $< k$ such that sg is itself deleted in the k^{th} mega-iteration. The inner loop computes deletions of type (b) till no more new facts are deleted. Subsequent mega-iterations advance the frontier of deletion to a larger subgoal position until the longest rule in the program has been considered.

Definition 7.1 $r_i, r_{i,j}$**:** For every rule r in program P, rules r_i, $r_{i,j}$ are defined as follows:

(r): p :- s_1 & ... & s_n.
(r_i): $\delta(p)$:- s_1^ν & ... & s_{i-1}^ν & $\Delta(s_i)$ & s_{i+1} & ... & s_n.
$(r_{i,j})$: $\delta(p)$:- s_1^ν & ... & s_{j-1}^ν & $\delta_l(s_j)$ & s''_{j+1} & ... & s''_i & s_{i+1} & ... & s_n.

$r_{i,j}$ is not defined for $i < j$. Relation s'' contains the value of relation s^ν as in the last iteration. An iteration consists of applying a given set of rules once, in parallel. $\delta(p)$ is the set of facts computed by rules of the type r_i and $r_{i,j}$ in one iteration. At the beginning of every iteration $\delta(p)$ is reinitialized to be empty. $\delta_l(p)$ is the set of changes made to p in the last iteration of the program. $\Delta(p)$ is the cumulative set of changes made to predicate p i.e. $\Sigma(\delta(p))$. □

Algorithm 7.1
Input: A program $\mathcal{P}$, and $\Delta(e)$ for each deleted base predicate e in $\mathcal{P}$.
Output: $\Delta(p)$ for each derived predicate p.
Method:

```
For each derived predicate p in program 𝒫, initialize
    p^ν = facts for p before changes are made to the database.
    p'' = {}                          % p'' contains the facts in p^ν from the last iteration
    Δ(p) = {}                         % p facts deleted so far
    δ_l(p) = {}                       % p facts deleted in the last iteration
Compute {P_1 ... P_n} where P_i is a strongly connected component of program 𝒫 and all
        predicates p ∈ P_i depend only on a subset of the set of predicates ∪_{j=1}^{i} P_j
        % Divide the program 𝒫 into its strongly connected components.
For l = 1...n do                      % Process the strongly connected components(SCC) in order
{   ℛ_l = {r | r is a rule in program 𝒫 and head(r) ∈ SCC P_l}.
    max_sg = max(length(r)), r ∈ ℛ_l.
    For i = 1...max_sg do                                  % start of a mega-iteration
    {   δ(p) = {}                     % Initialize δ(p) for every derived predicate
        δ_l(p) = {}                   % Initialize δ(p) for every derived predicate
        For rule r ∈ ℛ_l do
            Apply rule r_i to compute δ(p) where p = head(r)andp ∈ P_l.
        For each derived predicate p ∈ P_l do
            p^ν = p^ν ⊎ δ(p).
            δ_l(p) = δ(p)
        𝒬 = {r_{i,j} | r ∈ ℛ_l, j = 1...i}.
        Repeat                                             % iterate till fix point
        { Apply rules in 𝒬 to compute new δ(p) facts for predicate p ∈ P_l.
          p'' = p^ν.
          p^ν = p^ν ⊎ δ(p).
          Δ(p) = Δ(p) ⊎ δ(p).
          δ_l(p) = δ(p)
          δ(p) = {}.
        }
        Until no more changes are made to any predicate p ∈ P_l.
            % fix point computed ∀p in the l^th strongly connected component
```

} *% end of mega-iteration*

} *% All rules in the strongly connected component are processed*

$\diamond$

Theorem 7.1 *Given recursive program P and database E say fact f had* $\mathsf{count}(f)$ *derivations. When database E is altered to $E^\nu = E \uplus \Delta(E)$, say fact f has* $\mathsf{count}(f^\nu)$ *derivations. Then Algorithm 4.1 derives fact $\Delta(f)$ with a count of* $\mathsf{count}(f^\nu) - \mathsf{count}(f)$, *for all facts f derived by program P using either database E or E^ν.* $\square$

We omit the proof for the theorem.

Additions to base predicates are handled like deletions. Whereas for deletions, we needed to identify the *first* subgoal that was deleted, for insertions we need to identify the *last* newly inserted subgoal because only then can we conclude that the insertion is new.

In case the program has negation and/or aggregation, stratification ensures that a predicate in one strongly connected component uses negated and aggregated subgoals that are in lower strongly connected components. The facts in a strongly connected component in a lower stratum do not change while computing the results of a higher stratum component. Consequently a negative or aggregate predicate from a lower stratum can be used in a higher stratum exactly as outlined in Section 6. Proofs of correctness are again omitted.

8 Conclusions and Future Work

We present a general approach for maintaining views in relational databases when view definitions include negation, aggregation and general recursion. The algorithm computes changes to a materialized view in response to insertions, deletions and updates to the base relations. The algorithms proposed use a count of the number of derivations of each fact. We also show how the view maintenance algorithms fit nicely into existing systems with both set and multiset semantics.

The algorithm is limited to programs and databases where each derived fact has a finite number of proof trees. Mumick and Shmueli [MS92] give insight into how this problem could possibly be solved and thereby enhance the utility of our algorithms.

We are also working on the problem of how to avoid inserting and deleting the same facts when both insertions and deletions are made at the same time to base relations. For instance consider the transitive closure program and a database containing just the edge $e(1,2)$. If this edge is deleted and another edge $e(2,3)$ added at the same time, then the path $p(1,3)$ should ideally not be derived.

8.1 Acknowledgements

Discussions with Alberto Torres at the *NAIL!* group at Stanford led to the algorithm for updating recursive view definitions. Stefano Ceri and Jennifer Widom gave useful feedback on presentation issues.

References

[ABW88] Krzysztof R. Apt and Howard A. Blair and Adrian Walker. *Towards a Theory of Declarative Knowledge.* In *Foundations of Deductive Databases and Logic Programming.* Editor J. Minker, 1988 Morgan Kaufmann.

[AH88] Serge Abiteboul and Richard Hull. Data functions, datalog, and negation. In *Proceedings of ACM SIGMOD 1988 International Conference on Management of Data*, Chicago, IL, June 1988.

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

[BC79] Peter O. Buneman and Eric K. Clemons. *Efficiently Monitoring Relational Databases.* In *ACM Transactions on Database Systems*, Vol 4, No. 3, 1979, 368-382.

[BCL89] Jose A. Blakeley and N. Coburn and P. Larson. *Updating Derived Relations: Detecting irrelevant and Autonomously Computable Updates.* In *ACM Transactions on Database Systems* Vol 14, No. 3, 369-400, 1989.

[BLT86] Jose A. Blakeley and P. Larson and Frank Wm. Tompa. *Efficiently Updating Materialized Views.* In *Proceedings of ACM SIGMOD 1986 International Conference on Management of Data*, Washington D.C., 61-71.

[BlTo88] Jose A. Blakeley and F. W. Tompa. *Maintaining Materialized Views without Accessing Base Data.* In *Information Systems* Vol 13, No 4, 393-406, 1988.

[CW91] Stefano Ceri and Jennifer Widom. Deriving Production Rules for Incremental View Maintenance. In *Proceedings of the 17th VLDB Conference, Barcelona, Spain*, 1991.

[DT92] Guozhu Dong and Rodney Topor. *Incremental Evaluation of Datalog Queries.* In *Proceedings of the International Conference on Database Theory (ICDT)*, 1992.

[Doyle] Jon Doyle. *A Truth Maintenance System.* In *Readings In Artificial Intelligence*, Morgan Kaufmann, 1981.

[GM92] Ashish Gupta and Inderpal S. Mumick. Magic-Sets Transformation in Non-Recursive Systems . In *Proceedings of the Eleventh Symposium on Principles of Database Systems (PODS)*, 1992.

[ISO90] ISO_ANSI. Iso-ansi working draft: Database language sql2 and sql3; x3h2; iso/iec jtc1/sc21/wg3, 1990.

[Lloyd84] John W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1984.

[MPR90] Inderpal Singh Mumick, Hamid Pirahesh, and Raghu Ramakrishnan. The magic of duplicates and aggregates. In *Proceedings of the Sixteenth International Conference on Very Large Databases (VLDB)*, pages 264–277, Brisbane, Australia, August 13-16 1990.

[MS92] Inderpal Singh Mumick and Oded Shmueli. Finiteness properties of database queries. *Submitted for Publication*, 1992.

[Mum91] Inderpal Singh Mumick. *Query Optimization in Deductive and Relational Databases.* Ph.D. Thesis, Stanford University,Stanford, CA 94305, USA, 1991.

[Prz90] T. C. Przymusinski. *On the declarative semantics of deductive databases and logic programs.* In *Foundations of Deductive Databases and Logic Programming.* Editor J. Minker, 1988 Morgan Kaufmann.

[QW91] Xiaolei Qian and Gio Wiederhold. *Incremental Recomputation of Active Relational Expressions.* In *TKDE*, 1991.

[SI84] Oded Shmueli and Alon Itai. *Maintenance of Views.* In *Proceedings of Annual Meeting, Sigmod Record*, Vol 14, No. 2, 1984, 240-255.

[Ull89] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volumes 1 and 2.* Computer Science Press, 1989.

[WDSY91] Ouri Wolfson and Hasanat M. Dewan and Salvatore J. Stolfo and Yechiam Yemini. *Incremental Evaluation of Rules and its Relationship to Parallelism.* In *Proceedings of ACM SIGMOD 1991 International Conference on Management of Data*, Denver, CO.