Maintaining Views Incrementally

Extended Abstract

Ashish Gupta *
Stanford University
agupta@cs.stanford.edu

Inderpal Singh Mumick
AT&T Bell Laboratories
mumick@research.att.com

V.S. Subrahmanian†
University of Maryland
vs@cs.umd.edu

Abstract

We present incremental evaluation algorithms to compute changes to materialized views in relational and deductive database systems, in response to changes (insertions, deletions, and updates) to the relations. The view definitions can be in SQL or Datalog, and may use `UNION`, negation, aggregation (*e.g.* `SUM, MIN`), linear recursion, and general recursion.

We first present a *counting* algorithm that tracks the number of alternative derivations (counts) for each derived tuple in a view. The algorithm works with both set and duplicate semantics. We present the algorithm for *nonrecursive views* (with negation and aggregation), and show that the count for a tuple can be computed at little or no cost above the cost of deriving the tuple. The algorithm is optimal in that it computes exactly those view tuples that are inserted or deleted. Note that we store only the *number* of derivations, not the derivations themselves.

We then present the Delete and Rederive algorithm, DRed, for incremental maintenance of recursive views (negation and aggregation are permitted). The algorithm works by first deleting a superset of the tuples that need to be deleted, and then rederiving some of them. The algorithm can also be used when the view definition is itself altered.

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

†This work was supported by ARO grant DAAL-03-92-G-0225, and NSF grant IRI-9109755, and AFOSR grant F49620-93-1-0065.

1 Introduction

A view is a derived (idb) relation defined in terms of base (stored, or 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. Materialized views are especially useful in distributed databases. However, deletion, insertions, and updates to the base relations can change 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 base relations), 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 base relation is deleted, it may be cheaper to recompute a view that depends on the deleted relation (if the new view will quickly evaluate 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 base relations are called *incremental view maintenance* algorithms. Several such algorithms with different applicability domains have been proposed [BC79, NY83, SI84, BLT86, BT88, BCL89, CW91, Kuc91, QW91, WDSY91, CW92, DT92, HD92]. View maintenance has applications in integrity constraint maintenance, index maintenance in object-oriented databases (define the index between attributes of interest as a view), persistent queries, active database [SPAM91, RS93] (a rule may fire when a particular tuple is inserted into a view).

We present two algorithms, *counting* and DRed, for incremental maintenance of a large class of views. Both algorithms use the view definition to produce rules that compute the changes to the view using the changes made to the base relations and the old materialized views. Both algorithms can handle recursive and nonrecursive views (in SQL or Datalog extensions) that use negation, aggregation, and union, and can respond to insertions, deletions and updates to the base relations. However, we are proposing the *counting* algorithm for nonrecursive views, and the DRed algorithm for recursive views, as we believe each is better than the other on the specified domain.

Example 1.1 Consider the following view definition. $\mathtt{link}(S, D)$ is a base relation and $\mathtt{link}(a, b)$ is true if there is a link from source node a to destination b. $\mathtt{hop}(c, d)$ is true if c is connected to d via two links *i.e.* there is a `link` from node c to some node x and a `link` from x to d.

```
CREATE VIEW hop(S, D) AS
        (SELECT r1.S, r2.D
         FROM link r1, link r2
         WHERE r1.D = r2.S).
```

Given $\mathtt{link} = \{(a, b), (b, c), (b, e), (a, d), (d, c)\}$, the view `hop` evaluates to $\{(a, c), (a, e)\}$. The tuple $\mathtt{hop}(a, e)$ has a unique derivation. $\mathtt{hop}(a, c)$ on the other hand has two derivations. If the view had duplicate semantics then $\mathtt{hop}(a, e)$ would have a count of 1 and $\mathtt{hop}(a, c)$ would have a count of 2.

Suppose the tuple $\mathtt{link}(a, b)$ is deleted. Then we can re-evaluate `hop` to $\{(a, c)\}$.

The counting algorithm infers that one derivation of each of the tuples $\mathtt{hop}(a, c)$ and $\mathtt{hop}(a, e)$ is deleted. The algorithm uses the stored counts to infer that $\mathtt{hop}(a, c)$ has one remaining derivation and therefore only deletes $\mathtt{hop}(a, e)$, which has no remaining derivation.

The DRed algorithm first deletes tuples $\mathtt{hop}(a, c)$ and $\mathtt{hop}(a, e)$ since they both depend upon the deleted tuple. The DRed algorithm then looks for alternative derivations for each of the deleted tuples. $\mathtt{hop}(a, c)$ is rederived and reinserted into the materialized view in the second step. □

Counting The counting algorithm works by storing the number of alternative derivations of each tuple t in the materialized view. We call this number $\mathrm{count}(t)$. $\mathrm{count}(t)$ is derived from the multiplicity of tuple t under duplicate semantics, as defined in [Mum91] for positive programs and in [MS93a] for programs with stratified negation. Given a program T defining a set of views $V_1, \ldots, V_k$, the counting algorithm derives a program T_Δ at compile time. T_Δ uses the changes made to base relations and the old values of the base and view relations to produce as output the set of changes, $\Delta(V_1), \ldots, \Delta(V_k)$, that need to be made to the view relations. We assume that the count value for each tuple is stored in the materialized view. In the set of changes, inserted tuples are represented with positive counts and deleted tuples are represented with negative counts. The new materialized view is obtained by combining the changes $\Delta(V_1), \ldots, \Delta(V_k)$ with the stored views $V_1, \ldots, V_k$ (combining counts as defined in Section 3). The incremental view maintenance algorithm works for both set and duplicate semantics. On nonrecursive views we show that counts can be computed at little or no cost above the cost of evaluating the view (Section 5) for both set and duplicate semantics; hence it can be used for SQL. We propose to use the *counting* algorithm only for nonrecursive views, and describe the algorithm for nonrecursive views only.

Deletion and Rederivation The DRed algorithm can incrementally maintain (general) recursive views, with negation and aggregation. Given the changes made to base relations, changes to the view relations are computed in three steps. First, the algorithm computes an overestimate of the deleted derived tuples: a tuple t is in this overestimate if the changes made to the base relations invalidate *any* derivation of t. Second, this overestimate is pruned by removing (from the overestimate) those tuples that have alternative derivations in the new database. Finally, the new tuples that need to be added are computed using the partially updated materialized view and the changes made to the base relations. Only set semantics can be used for this algorithm. The algorithm can also maintain materialized views incrementally when rules defining derived relations are inserted or deleted.

Paper Outline Section 2 compares the results in our paper with related work. Section 3 introduces the notation used in the paper. Section 4 describes the *counting* algorithm for maintaining nonrecursive views. Section 5 describes how the counting algorithm can be implemented efficiently. We show that a computation of counts imposes almost no overhead in execution time and storage. Section 6 explains how negation and aggregation are handled by the counting algorithm of Section 4. Section 7 discusses the DRed algorithm for maintaining general recursive views. The results are summarized in Section 8.

2 Related Work

Ceri and Widom [CW91] describe a strategy to efficiently update views defined in a subset of SQL without negation, aggregation, and duplicate semantics. Their algorithm depends on keys, and cannot be used if the view does not contain the key attributes of a base relation. Qian and Wiederhold [QW91] use algebraic operations to derive the minimal relational expression that computes the change to select-project-join views. The algorithms by Blakeley *et. al.* [BLT86] and Nicolas and Yazdanian (The BDGEN system [NY83]) are perhaps most closely related to our counting algorithm. Blakeley's algorithm is a special case of the counting algorithm applied to select-project-join expressions (no negation, aggregation, or recursion). In BDGEN, the counts reflect only certain types of derivations, are multiplied to guarantee an even count for derived tuples, and all recursive queries are given finite counts. Thus the BDGEN counts, unlike our counts, do not correspond to the number of derivations of a tuple, are more expensive to compute, and the BDGEN algorithm cannot be used with (SQL) duplicate semantics or with aggregation, while our algorithm can be.

Kuchenhoff [Kuc91] and Harrison and Dietrich (the PF algorithm [HD92]) discuss recursive view maintenance algorithms related to our *rederivation* algorithm. Both of these algorithms cannot handle aggregation (we can). Where applicable, the PF (Propagation/Filteration) algorithm computes changes in *one* derived predicate due to changes in *one* base predicate, iterating over all derived and base predicates to complete the view maintenance. An attempt to recompute the deleted tuples is made for each small change in each derived relation. In contrast, our *rederivation* algorithm propagates changes from all base predicates onto all derived predicates stratum by stratum, and recomputes deleted tuples only once. The PF algorithm thus fragments computation, can rederive changed and deleted tuples again and again, and can be worse that our *rederivation* algorithm by an order of magnitude (examples in full paper). Kuchenhoff's algorithm needs to store auxiliary relations, and fragments computation in a manner similar to the PF algorithm.

Dong and Topor [DT92] derive nonrecursive programs to update right-linear chain views in response to insertions only. Dong and Su [DS92] give nonrecursive transformations to update the transitive closure of specific kinds of graphs in response to insertions and deletions. The algorithm does not apply to all graphs or to general recursive programs. They also need auxiliary derived relations, and cannot handle negation and aggregation. Urpi and Olive [UO92] need to derive functional dependencies, a problem that is known to be undecidable. Wolfson *et. al.* [WDSY91] use a rule language with negation in the head and body of rules, along with auxiliary information about the number of certain derivations of each tuple. They do not discuss how to handle recursively defined relations that are derivable in infinitely many iterations, and do not handle aggregation.

3 Notation

We use Datalog, mostly as discussed in [Ull89], extended with stratified negation [VG86, ABW88], and stratified aggregation [Mum91]. Datalog extended with stratified negation and aggregation can be mapped to a class of recursive SQL queries, and vice versa [Mum91]. We chose Datalog syntax over SQL syntax for conciseness.

Definition 3.1 Stratum Numbers (SN) and Rule Stratum Number (RSN): Stratum numbers are assigned as follows: Construct a *reduced dependency graph (RDG)* of the given program by collapsing every strongly connected component (scc) of the dependency graph (as defined by [ABW88]) to a single node. A RDG is guaranteed to be acyclic. A topological sort of the RDG assigns a stratum number to each node. If a node represents a scc, all predicates in the scc are assigned the stratum number of the node. By convention, base predicates have stratum number $= 0$. The rule stratum number of a rule r, $\mathsf{RSN}(r)$, having predicate p in the head is equal to $\mathsf{SN}(p)$. $\Box$

P refers to the relation corresponding to predicate p. $P = \{ab, mn\}$ represents tuples $p(a, b)$ and $p(m, n)$.

Definition 3.2 $\Delta(P)$: For every relation P, relation $\Delta(P)$ contains the changes made to P. $\Box$

For each tuple $t \in P$, $\mathsf{count}(t)$ represents the number of distinct derivations of tuple t. Similarly every tuple in $\Delta(P)$ 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 tuple $p(a, b)$ are inserted into P and two derivations of tuple $p(m, n)$ are deleted.

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

1. If 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 $S1$ and $S2$ with counts of c_1 and c_2 respectively, and $c_1 + c_2 \neq 0$, then tuple t appears in $S1 \uplus S2$ with a count $c_1 + c_2$. If $c_1 + c_2 = 0$ then t does not appear in $S1 \uplus S2$.

P^{ν} refers to the relation P after incorporating the changes in $\Delta(P)$. Thus, $P^{\nu} = P \uplus \Delta(P)$. The correctness of our algorithm guarantees that a tuple in P^{ν} will not have a negative count; only tuples in relation $\Delta(P)$ will have negative counts. 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.

4 Incremental Maintenance of Nonrecursive Views using Counting

This section gives an algorithm that can be used to maintain nonrecursive views that use negation and aggregation. We first give the intuition using an example.

Example 4.1 Intuition. Consider the view hop defined in Example 1.1. We rewrite the view definition using Datalog for succinctness and ease of explanation. Recall that $\texttt{link} = \{ab, mn\}$ represents tuples $\texttt{link}(a,b)$ and $\texttt{link}(m,n)$.

$$(v1): \quad \texttt{hop}(X,Y) :\!-\ \texttt{link}(X,Z)\ \&\ \texttt{link}(Z,Y).$$

If the base relation link changes, the derived relation hop may also change. Let the change to the relation link be represented as $\Delta(\texttt{link})$. $\Delta(\texttt{link})$ contains both inserted and deleted tuples, represented by positive and negative counts respectively. The new relation $\texttt{link}^{\nu}$ can be written as: $\texttt{link} + \Delta(\texttt{link})$. The following rule computes the new value for the hop relation in terms of the relation $\texttt{link}^{\nu}$:

$$\texttt{hop}^{\nu}(X,Y) :\!-\ \texttt{link}^{\nu}(X,Z)\ \&\ \texttt{link}^{\nu}(Z,Y).$$

Using $\texttt{link}^{\nu} = \texttt{link} \uplus \Delta(\texttt{link})$ and distributing joins over unions, the above rule can alternatively be written as the following set of rules:

$$\begin{array}{l}
\texttt{hop}^{\nu}(X,Y) :\!-\ \texttt{link}(X,Z)\ \&\ \texttt{link}(Z,Y).\\
\texttt{hop}^{\nu}(X,Y) :\!-\ \Delta(\texttt{link})(X,Z)\ \&\ \texttt{link}(Z,Y).\\
\texttt{hop}^{\nu}(X,Y) :\!-\ \texttt{link}(X,Z)\ \&\ \Delta(\texttt{link})(Z,Y).\\
\texttt{hop}^{\nu}(X,Y) :\!-\ \Delta(\texttt{link})(X,Z)\ \&\ \Delta(\texttt{link})(Z,Y).
\end{array}$$

The first rule recomputes relation hop. The remaining three rules define $\Delta(\texttt{hop})$, the changes in relation hop. Of these three rules, the last two can be combined, using the fact that $\texttt{link}^{\nu} = \texttt{link} \uplus \Delta(\texttt{link})$. The set of rules that defines predicate $\Delta(\texttt{hop})$ is therefore:

$$\begin{array}{l}
(d1) : \Delta(\texttt{hop})(X,Y) :\!-\ \Delta(\texttt{link})(X,Z)\ \&\ \texttt{link}(Z,Y)\\
(d2) : \Delta(\texttt{hop})(X,Y) :\!-\ \texttt{link}^{\nu}(X,Z)\ \&\ \Delta(\texttt{link})(Z,Y)
\end{array}$$

□

Definition 4.1 Delta Rules: With every rule r of the form:

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

we associate n delta rules $\Delta_i(r)$, $1 \leq i \leq n$, defining predicate $\Delta(p)$ as follows:

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

□

The counting algorithm is listed as Algorithm 4.1. Ignore statement (2) (surrounded by a box) for now; it will be discussed in Section 5.

Example 4.2 Consider the view tri_hop defined using the view hop (rule $v1$, Example 4.1).

$$(v2) : \texttt{tri\_hop}(X,Y) :\!-\ \texttt{hop}(X,Z)\ \&\ \texttt{link}(Z,Y)$$

The stratum numbers of predicates hop and tri_hop are 1 and 2 respectively. Consider the following base relation link and the associated derived relations hop and tri_hop.

$$\begin{array}{l}
\texttt{link} = \{ab,\ ad,\ dc,\ bc,\ ch,\ fg\}.\\
\texttt{hop} = \{ac * 2,\ dh,\ bh\}.\\
\texttt{tri_hop} = \{ah * 2\}.
\end{array}$$

Let the base relation link be altered as follows:

$$\begin{array}{l}
\Delta(\texttt{link}) = \{ab * -1,\ df,\ af\}.\\
\texttt{link}^{\nu} = \{ad,\ af,\ bc,\ dc,\ ch,\ df,\ fg\}.
\end{array}$$

In order to compute the changes, first consider the rule with the least RSN, namely $v1$.

$$\begin{array}{l}
(\Delta_1(v1)) : \Delta\texttt{hop}(X,Y) :\!-\ \Delta\texttt{link}(X,Z)\ \&\ \texttt{link}(Z,Y)\\
(\Delta_2(v1)) : \Delta\texttt{hop}(X,Y) :\!-\ \texttt{link}^{\nu}(X,Z)\ \&\ \Delta\texttt{link}(Z,Y)
\end{array}$$

Apply rule $\Delta_1(v1)$: $\Delta(\texttt{hop}) = \{ac * -1,\ ag,\ dg\}$
Apply rule $\Delta_2(v1)$: $\Delta(\texttt{hop}) = \{af\}$
Combining the above changes, we get:

$$\texttt{hop}^{\nu} = \{ac,\ af,\ ag,\ dg,\ dh,\ bh\}.$$

Now consider the rules with RSN 2, namely rule $v2$ that defines predicate tri_hop.

$$(\Delta_1(v2)): \quad \Delta\texttt{tri\_hop}(X,Y) :\!-\ \Delta\texttt{hop}(X,Z)\ \&\ \texttt{link}(Z,Y).$$

$$(\Delta_2(v2)): \quad \Delta\texttt{tri\_hop}(X,Y) :\!-\ \texttt{hop}^{\nu}(X,Z)\ \&\ \Delta\texttt{link}(Z,Y).$$

Algorithm 4.1
Input: A nonrecursive program $\mathcal{P}$.
Stored materializations of derived relations in $\mathcal{P}$.
Changes (deletions/insertions) to the base relations occurring in program $\mathcal{P}$.
Output: Changes (deletions/insertions) to the derived relations occurring in $\mathcal{P}$.
Method:

Mark all rules unprocessed.
For all derived predicates p in program $\mathcal{P}$, **do**
  initialize P^{ν} to the materialized relation P.
  initialize $\Delta(P) = \{\}$
While there is an unprocessed rule
{ $\mathcal{Q} = \{r \mid$ rule r has the least RSN among all unprocessed rules$\}$
  For every rule $r \in \mathcal{Q}$ **do**
  { Compute $\Delta(P)$ using the delta rules $\Delta_i(r)$, $1 \leq i \leq n$ derived from rule r (Definition 4.1)
    $(\Delta_i(r))$: $\Delta(p) :- s_1^{\nu}$ & ... & s_{i-1}^{ν} & $\Delta(s_i)$ & s_{i+1} & ... & s_n ....(1).
    $P^{\nu} = P \uplus \Delta(P)$. *% Update the predicate that is defined by rule r.*
    $\boxed{\Delta(P) = \mathsf{set}(P^{\nu}) - \mathsf{set}(P)}$ ....(2)
    % For optimization only, discussed in Section 5
    Mark rule r as processed.
} } $\diamond$

Apply rule $\Delta_1(v2)$: $\Delta(\texttt{tri\_hop}) = \{ah * -1, ag\}$
Apply rule $\Delta_2(v2)$: $\Delta(\texttt{tri\_hop}) = \{\}$
Combining the above changes, we get:

$\texttt{tri\_hop}^{\nu} = \{ah, ag\}$.

$\Box$

Lemma 4.1 *Let Δ^- be the set of base tuples deleted from E, and t be any ground atom that has* $\mathsf{count}(t)$ *derivations w.r.t. program $\mathcal{P}$ and database state (edb) E. If $\Delta^- \subseteq E$ then Algorithm 4.1 derives tuple $\Delta(t)$ with a* count *of at least* $-1 * \mathsf{count}(t)$. $\Box$

That is, given that we insist that the deleted base tuples be a subset of the original database, no more than the original number of derived tuples are deleted from any derived relation during evaluation of Algorithm 4.1. Therefore all non-Δ subgoals have positive counts.

Theorem 4.1 *Let t be any ground atom that has* $\mathsf{count}(t)$ *derivations w.r.t. program $\mathcal{P}$ and database state (edb) E. Suppose tuple t has* $\mathsf{count}(t^{\nu})$ *derivations when edb E is altered to a new edb E^{ν} (by insertions/deletions). Then: Algorithm 4.1 derives tuple $\Delta(t)$ with a* count *of* $\mathsf{count}(t^{\nu}) - \mathsf{count}(t)$. $\Box$

If the program needed set semantics, then Algorithm 4.1 is optimized by propagating changes to predicates in higher strata only if the relation P^{ν} considered as a set changes from relation P considered as a set. This optimization is done by Statement (2) in Algorithm 4.1 and illustrated in Example 5.1.

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 in the above systems requires duplicates to be retained for semantic correctness [ISO90]. In an implementation, duplicates may be represented either by keeping multiple copies of a tuple, or by keeping a count with each tuple. In both cases, 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 *Glue-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 is likely to be useful only for nonrecursive queries because recursive queries may have infinite counts associated with them. The first implementation 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 for nonrecursive queries.

5.1 Optimization

The boxed statement 2 in Algorithm 4.1 optimizes the counting algorithm for views where duplicate semantics is not desired, and for implementations that eliminate duplicates.

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. 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 stratum (by extending the evaluation method in 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 of tuples in stratum i or lower while evaluating tuples in stratum $i+1$. We assume that each tuple of stratum i or less has a count of one, 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 count 0 or 1. Maintaining counts as above influences the propagation of changes in a positive manner. For instance, let predicate p in stratum 1 have 20 derivations for a tuple $p(a)$, and let changes to the base tuples 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. The boxed statement 2 in Algorithm 4.1 causes us to maintain counts as above. Consider Example 4.2 if the views had set semantics.

Example 5.1 Consider relations $\mathtt{hop}^{\nu}$ and $\mathtt{hop}$ after the rules $\Delta_1(v1)$ and $\Delta_2(v1)$ have been applied. In order to compute $\Delta(\mathtt{hop})$ we apply the optimization of Statement 2 from Algorithm 4.1.

$$\begin{aligned}\Delta(\mathtt{hop}) &= set(\mathtt{hop}^{\nu}) - set(\mathtt{hop})\\ &= \{ac, af, ag, dg, dh, bh\} - \{ac, dh, bh\}\\ &= \{af,\ ag,\ dg\}.\end{aligned}$$

Note that unlike Example 4.2, the tuple $\mathtt{hop}(ac * -1)$ does not appear in $\Delta(\mathtt{hop})$ and is not cascaded to relation $\mathtt{tri\_hop}$. Consequently the tuple $(ah * -1)$ will not be derived for $\Delta(\mathtt{tri\_hop})$. □

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.

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

6 Negation and Aggregation

Algorithm 4.1 can be used to incrementally maintain views defined using negation and aggregation. However, we need to describe how statement 1 in Algorithm 4.1 is executed for rules with negated and aggregated subgoals, specifically how $\Delta(S)$ is evaluated for a negated or GROUPBY subgoal s in rule r.

6.1 Negation

We consider safe stratified negation. Negation is safe as long as the variables that occur in a negated subgoal also occur in some positive subgoal of the same rule. Negation is stratified if whenever a 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 always stratified.

The following Example 6.1 gives the intuition for computing counts with negated subgoals. A negated subgoal is computed in the same way for both set and duplicate semantics.[1]

[1] Formal semantics of Duplicate Datalog with negation is given in [MS93a]

Example 6.1 Consider view `only_tri_hop` that uses views `tri_hop` and `hop` as defined in Example 4.2. `only_tri_hop` contains all pairs of nodes that are connected using three links but not using just two.

$(v3)$: $\texttt{only\_tri\_hop}(X,Y)$:- $\texttt{tri\_hop}(X,Y)$ & $\neg\texttt{hop}(X,Y)$.

Consider the relation $\texttt{link} = \{ab, ae, af, ag, bc, cd, ck, ed, fd, gh, hk\}$. The relations `hop` and `tri_hop` are $\{ac, ad*2, ah, bd, bk, gk\}$ and $\{ad, ak*2\}$ respectively. The relation $\texttt{only\_tri\_hop} = \{ak*2\}$. Tuple (a,d) does not appear in `only_tri_hop` because $\texttt{hop}(a,d)$ is true. Note that $\texttt{hop}(a,d)$ is true as long as $\texttt{count}(\texttt{hop}(a,d)) > 0$. Therefore even if $\texttt{count}(\texttt{hop}(a,d))$ was 1 or 5 (as against the indicated value of 2), relation `only_tri_hop` would not have tuple (a,d). □

Consider a negated subgoal $\neg q(X,Y)$ in rule r defining a view. Because negation is safe, the variables X and Y also occur in positive subgoals in rule r. We represent the relation corresponding to the subgoal $\neg q$ as $\bar{Q}$. The relation $\bar{Q}$ is computed using relation Q and the particular bindings for variables X and Y provided by the positive subgoals in rule r. A tuple (a,b) is in $\bar{Q}$ with a `count` of 1 if, and only if, (i) the positive subgoals in rule r assign the values a and b to the variables X and Y, and (ii) the tuple $q(a,b)$ is false ($(a,b) \notin Q$).

Recall that Algorithm 4.1 creates and evaluates Delta rules of the form $\Delta_i(r)$:

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

In order to define how rule $\Delta_i(r)$ is evaluated, we exhaustively consider all the positions where a negated subgoal can occur in rule $\Delta_i(r)$ and define how the subgoal will be computed:

<u>Case 1:</u> **Subgoal** $s_j = \neg q$, *j between* $i+1$ *and* n**:** The relation $\bar{Q}$ is computed as described above.

<u>Case 2:</u> **Subgoal** $s_j^\nu = (\neg q)^\nu$, *j between* 1 *and* $i-1$**:** The relation $\bar{Q}^\nu$ is equal to the relation $\bar{Q^\nu}$ by the following Lemma:

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

Because negation is stratified, relation Q^ν is computed before rule $\Delta_i(r)$ is used. Relation $\bar{Q^\nu}$ is computed from Q^ν in the same way that $\bar{Q}$ is computed from Q.

<u>Case 3:</u> **Subgoal** $\Delta(s_i) = \Delta(\neg q)$**:** The relation $\Delta(\bar{Q})$ is computed from relations $\Delta(Q)$ and Q according to Definition 6.1.

Definition 6.1 $\Delta(\bar{Q})$**:** Say Q represents the relation for predicate q and $\Delta(Q)$ represents the changes made to Q. A tuple t is in $\Delta(\bar{Q})$ with $\texttt{count}(t) = 1$ if

$$t \in \Delta(Q) \text{ and } t \notin Q \uplus \Delta(Q),$$

and with $\texttt{count}(t) = -1$ if

$$t \in \Delta(Q) \text{ and } t \notin Q.$$

Note that $t \in \Delta(\bar{Q})$ only if $t \in \Delta(Q)$. □

Note that Definition 6.1 allows $\Delta(Q)$ to be computed without having to evaluate the positive subgoals in rule $\Delta_i(r)$. This is important for efficiency, since the Δ-subgoal is usually the most restrictive subgoal in the rule and would be used first in the join order.

Theorem 6.1 *Algorithm 4.1 works correctly in the presence of negated subgoals.* □

6.2 Aggregation

Aggregation is often used to reduce large amounts of data to more usable form. In this section we use the semantics for aggregation as discussed in [Mum91]. The following example illustrates the notation and semantics.

Example 6.2 Consider the relation `link` from Example 1.1 and let tuples in `link` also have a cost associated with them, *i.e.* $\texttt{link}(s,d,c)$ represents a link from source s to destination d of cost c. We now redefine the relation `hop` as follows:

$\texttt{hop}(S,\ D,\ C_1{+}C_2)$:- $\texttt{link}(S,\ I,\ C_1)$ & $\texttt{link}(I,\ D,\ C_2)$

Using `hop` we now define the relation `min_cost_hop` as follows:

$(v4)$: $\texttt{min\_cost\_hop}(S,D,M)$:-
$\texttt{GROUPBY}\ (\texttt{hop}(S,D,C),\ [S,D],\ M = \texttt{MIN}(C))$.

Relation `min_cost_hop` contains pairs of nodes along with the minimum cost of a hop between them. □

Consider a rule r that contains a `GROUPBY` subgoal defined over relation U. The `GROUPBY` subgoal represents a relation T whose attributes are the variables defined by the aggregation, and the variables $\bar{Y}$ over which the groupby occurs. In Example 6.2, the variable defined by the aggregation is M, and the groupby occurs over the variables $\{S,D\}$. The `GROUPBY` subgoal $\texttt{GROUPBY}\ (\texttt{hop}(S,D,C),\ [S,D],\ M = \texttt{MIN}(C))$ thus defines a relation over variables $\{S,\ D,\ M\}$. The relation T contains one tuple for every distinct value of the groupby attributes. All tuples in the grouped relation U that have the same values for the grouping attributes in set $\bar{Y}$, say $\bar{y}$, contribute one tuple to relation T, a tuple we denote by $T_{\bar{y}}$. In Example 6.2, the relation for the `GROUPBY` subgoal has one tuple for every distinct pair of nodes $[S, D]$.

Like negation, aggregation subgoals are non-monotonic. Consider inserting tuple μ into the relation U where the values of the $\bar{Y}$ attributes in tuple μ are $= \bar{c}$. Inserting μ can possibly change the value of the aggregate tuple in relation T that corresponds to $\bar{c}$, *i.e.* tuple $T_{\bar{c}}$. For instance, in Example 6.2, inserting the tuple $\mathtt{hop}(a, b, 10)$ can only change the `min_cost_hop` tuple from a to b. The change actually occurs if the previous minimum cost from a to b had a cost more than 10. A similar potential change can occur to tuple $T_{\bar{c}}$ if an existing tuple μ is deleted from relation U. Using the old tuple $T_{\bar{c}}$ and the tuples in $\Delta(U)$, the new tuple corresponding to the groupby attribute value $\bar{c}$ can be computed incrementally for each of the aggregate functions `MIN`, `MAX`, `COUNT`, `SUM`, and for any other incrementally computable function [DAJ91]. For instance consider the aggregation operation `SUM`. The sum of the attribute A of the tuples in a group can be computed using the old sum when a new tuple is added to the group by adding $\mu.A$ to the old sum. Functions like `AVERAGE` and `VARIANCE` that can be decomposed into incrementally computable functions can also be incrementally computed.

To apply Algorithm 4.1 we need to specify how a `GROUPBY` subgoal is evaluated in a Delta rule $\Delta_i(r)$:

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

$\Delta_i(r)$ could have one or more aggregate subgoals. If an aggregate subgoal t occurs between positions $i+1$ and n, then the relation T for the subgoal is computed as in the case of a normal aggregate subgoal. If an aggregate subgoal occurs between positions $1, \ldots, i-1$ then the relation T^{ν} for the subgoal can be computed as before using relation U^{ν}. If the aggregate subgoal occurs in position j, then the following algorithm is used to compute the relation $\Delta(T)$:

Algorithm 6.1
Input: An aggregate subgoal:
$t = \mathtt{GROUPBY}\ (U, \bar{Y}, \bar{Z} = \ldots)$.
Changes to the relation for the grouped predicate u: $\Delta(U)$.
Output: $\Delta(T)$.
Method:
For every grouping value $\bar{y} \in \pi_{\bar{Y}}\Delta(U)$
Incrementally compute ${T_{\bar{y}}}^{\nu}$ from $T_{\bar{y}}$ (old) and $\Delta(U)$.
If ${T_{\bar{y}}}^{\nu}$ and $T_{\bar{y}}$ are different **then**
$\Delta(T) = \Delta(T) \uplus \{(T_{\bar{y}}, -1)\}$ *% Insert old % tuple $T_{\bar{y}}$ into $\Delta(T)$ with a* $\mathsf{count} = -1$.
$\Delta(T) = \Delta(T) \uplus \{({T_{\bar{y}}}^{\nu}, 1)\}$ *% Insert new % ${T_{\bar{y}}}^{\nu}$ into $\Delta(T)$ with a* $\mathsf{count} = 1$.
% **Else** *the aggregate tuple is unchanged % and nothing needs to be done.*

$\diamond$

If the aggregation function is not incrementally computable [DAJ91], and is not even decomposable into incrementally computable functions, then the computation of ${T_{\bar{y}}}^{\nu}$ may be more expensive. For non incrementally computable aggregate functions, the tuple ${T_{\bar{y}}}^{\nu}$ has to be computed using the tuples of relation U that have the value $\bar{y}$ for the variables $\bar{Y}$.

Lemma 6.2 *For an aggregate subgoal t, relation T^{ν} is equivalent to $T \uplus \Delta(T)$.* $\square$

Theorem 6.2 *Algorithm 4.1 works correctly in the presence of aggregate subgoals.* $\square$

7 Incremental Maintenance of Recursive Views

We present the DRed algorithm to incrementally maintain recursive views that use negation and aggregation and have set semantics.[2] The DRed algorithm can also be used to maintain nonrecursive views; however the counting algorithm is expected to be more efficient for nonrecursive views. Conversely, we note that the counting algorithm can also be used to incrementally maintain certain recursive views [GKM92].

A semi-naive [Ull89] computation is sufficient to compute new inserted tuples for a recursively defined view when insertions are made to base relations. In the case of deletions however, simple semi-naive computation would delete a derived tuple that depends upon a deleted base tuple *i.e.* if tuple t has even one derivation tree that contains a deleted tuple, then t is deleted. Alternative derivations of t are not considered. Semi-naive therefore computes an overestimate of the tuples that actually need to be deleted. The DRed algorithm refines this

[2] The DRed algorithm is similar to an algorithm developed independently, and at the same time as our work, by Ceri and Widom [CW92], though their algorithm is presented in a production rule framework, and they don't handle aggregation and insertions/deletions of rules.

overestimate by considering alternative derivations of the deleted tuples (in the overestimate) as follows:

1. Delete a superset of the derived tuples that need to be deleted:
2. Put back those deleted tuples that have alternative derivations:

If base tuples are inserted into the database, then a third step is used to compute new derived tuples. This three step process is formalized as an algorithm, and proved correct, in [GMS92].

A recursive program $\mathcal{P}$ can be fragmented into programs $\mathcal{P}_1, \ldots, \mathcal{P}_n$, where $\mathcal{P}_i = \{r \mid \mathsf{RSN}(r) = i\}$ constitutes stratum i. The DRed algorithm computes change to a view defined by a recursive program $\mathcal{P}$ by applying the above three steps successively to every stratum of $\mathcal{P}$. Every derived predicate in program $\mathcal{P}_i$ depends only on predicates that are defined in $\mathcal{P}_1, \ldots, \mathcal{P}_{i-1}$. All base tuples are in stratum 0 *i.e.* in $\mathcal{P}_0$. Changes made to stratum i affect only those strata whose SN is $\geq i$. Propagating the changes stratum by stratum avoids propagating spurious changes across strata. Let Del_{i-1} (Add_{i-1}) be the set of tuples that have been deleted (inserted) from strata $1, \ldots, i-1$ respectively. Consider stratum i after strata $1, \ldots, i-1$ have been updated. Tuples are deleted from stratum i based on the set of deleted tuples Del_{i-1}. New tuples are inserted into stratum i based on the set of inserted tuples Add_{i-1}.

Theorem 7.1 *Let Δ^- and Δ^+ be the set of base tuples deleted and inserted respectively, from the original set of base tuples E. The new derived view computed by the* DRed *algorithm contains tuple t if and only if t has a derivation in the database $E^\nu = (E - \Delta^-) \cup \Delta^+$.* □

The DRed algorithm can be applied to recursive views with stratified negation and aggregation also. The details of the algorithm are given in [GMS92].

8 Conclusions and Future Work

We have presented general techniques for maintaining views in relational and deductive databases, including SQL with duplicate semantics, when view definitions include negation, aggregation and general recursion. The algorithms compute changes to a materialized view in response to insertions, deletions and updates to the base relations.

The counting algorithm is presented for nonrecursive views. We show how this incremental view maintenance algorithm fits nicely into existing systems with both set and multiset semantics. The counting algorithm is a general-purpose algorithm that uniformly applies to all nonrecursive views, and is the first to handle aggregation. The DRed algorithm is presented for maintaining recursive views. DRed is the first algorithm to handle aggregation in recursive views. The algorithm first computes an over estimate of tuples that need to be deleted in response to changes to the underlying database. This estimate is refined to obtain an exact answer. New derived tuples are computed subsequently.

Counting can be used to maintain recursive views also. However computing counts for recursive views is expensive and furthermore counting may not terminate on some views. Techniques to detect finiteness [MS93a] and to use partial derivations for counting are being explored. Similarly DRed can be used for nonrecursive views also but it is less efficient than counting.

The techniques to handle negation and aggregation as described in this paper can be used to extend many other existing view maintenance techniques.

References

[ABW88] Krzysztof R. Apt, 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.

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

[BCL89] J. A. Blakeley, N. Coburn, and P. Larson. *Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates.* In *ACM TODS* Vol 14, No. 3, 369-400, 1989.

[BLR91] Veronique Benzaken, Christophe Lecluse, and Philippe Richard. Enforcing Integrity Constraints in Database Programming Languages. TR Altair 68-91, Altair, France, 1991.

[BLT86] J. A. Blakeley, P. Larson, and F. W. Tompa. *Efficiently Updating Materialized Views.* In *SIGMOD 1986*, pages 61–71.

[BMM92] F. Bry, R. Manthey, and B. Martens. Integrity Verification in Knowledge Bases. In *Logic Programming, LNAI 592*, pages 114–139, 1992.

[BT88] J. A. Blakeley and F. W. Tompa. Maintaining Materialized Views without Accessing Base Data. *Information Systems*, 13(4):393–406, 1988.

[CW91] Stefano Ceri and Jennifer Widom. Deriving Production Rules for Incremental View Maintenance. In *17th VLDB*, 1991.

[CW92] Stefano Ceri and Jennifer Widom. Deriving Incremental Production Rules for Deductive Data. IBM RJ 9071, IBM Almaden, 1992.

[DAJ91] S. Dar, R. Agrawal, and H. V. Jagadish. Optimization of generalized transitive closure. In *Seventh IEEE International Conference on Data Engineering*, Kobe, Japan, 1991.

[DS92] Guozhu Dong and Jianwen Su. Incremental and Decremental Evaluation of Transitive Closure by First-Order Queries. TRCS 92-18, University of California, Santa Barbara, 1992.

[DT92] Guozhu Dong and Rodney Topor. *Incremental Evaluation of Datalog Queries.* In *ICDT*, 1992.

[GKM92] Ashish Gupta, Dinesh Katiyar, and Inderpal Singh Mumick. Counting Solutions to the View Maintenance Problem. In *Workshop on Deductive Databases, JICSLP*, 1992.

[GMS92] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. TR 921214-19-TM, AT&T Bell Labs, 1992.

[HD92] John V. Harrison and Suzanne Dietrich. Maintenance of Materialized Views in a Deductive Database: An Update Propagation Approach. In *Workshop on Deductive Databases, JICSLP*, 1992.

[ISO90] ISO_ANSI. ISO-ANSI Working Draft: Database Language SQL2 and SQL3; X3H2; ISO/IEC JTC1/SC21/WG3, 1990.

[Kuc91] V. Kuchenhoff. On the Efficient Computation of the Difference Between Consecutive Database States. In *DOOD, LNCS 566*, 1991.

[MS93a] Inderpal Singh Mumick and Oded Shmueli. Finiteness Properties of Database Queries. In *Fourth Australian Database Conference*, 1993.

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

[NY83] J. M. Nicolas and Yazdanian. An Outline of BDGEN: A Deductive DBMS. In *Information Processing*, pages 705–717, 1983.

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

[RS93] Torre Risch and Martin Sköld. Active rules based on object-oriented queries. To Appear, *ACM TKDE*, 1993.

[SI84] Oded Shmueli and Alon Itai. *Maintenance of Views.* In *Sigmod Record*, 14(2):240-255, 1984.

[SPAM91] Ulf Schreier, Hamid Pirahesh, Rakesh Agrawal, and C. Mohan. *A*lert: An Architecture for Transforming a Passive DBMS Into an Active DBMS. In *17th VLDB*, pages 469–478, 1991.

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

[UO92] Toni Urpi and Antoni Olive. A Method for Change Computation in Deductive Databases. In *18th VLDB*, pages 225–237, 1992.

[VG86] Allen Van Gelder. Negation as failure using tight derivations for general logic programs. In *Third IEEE Symposium on Logic Programming*, 1986. Springer-Verlag.

[WDSY91] Ouri Wolfson, Hasanat M. Dewan, Salvatore J. Stolfo, and Yechiam Yemini. *Incremental Evaluation of Rules and its Relationship to Parallelism.* In *SIGMOD 1991*, pages 78–87.