Incremental Evaluation of Tabled Logic Programs[⋆]

Diptikalyan Saha and C. R. Ramakrishnan

Department of Computer Science,
State University of New York at Stony Brook
Stony Brook, New York, 11794-4400, U.S.A.
E-mail: {dsaha, cram}@cs.sunysb.edu

Abstract. Tabling has emerged as an important evaluation technique in logic programming. Currently, changes to a program (due to addition/deletion of rules/facts) after query evaluation compromise the completeness and soundness of the answers in the tables. This paper presents incremental algorithms for maintaining the freshness of tables upon addition or deletion of facts. Our algorithms improve on existing materialized view maintenance algorithms and can be easily extended to handle changes to rules as well. We describe an implementation of our algorithms in the XSB tabled logic programming system. Preliminary experimental results indicate that our incremental algorithms are efficient. Our implementation represents a first step towards building a practical system for incremental evaluation of tabled logic programs.

1 Introduction

Tabled resolution [19, 1, 3] removes some of the best-known shortcomings of Prolog's evaluation strategy, especially its susceptibility to infinite looping. The XSB system [20] has become a stable platform for evaluating tabled logic programs, and several alternative implementations are emerging [4, 22, 9]. The added power of tabling has been crucial to the construction of many applications— such as practical program analysis and verification systems (e.g., [5, 14]), object-oriented knowledge bases (e.g. Flora-2 [21]) and ontology management systems— by encoding them at a high level as logic programs.

Tabled resolution-based systems evaluate programs by memoizing subgoals (referred to as *calls*) and their provable instances (referred to as *answers*) in a set of tables. Traditionally, the systems keep all calls in a *call table*. For each subgoal in the call table, its provable instances are kept in an *answer table*. During resolution, if a subgoal is present in the call table, then it is resolved against the answers recorded in the corresponding answer table; otherwise the subgoal is entered in the call table, and its answers, computed by resolving the subgoal against program clauses, are also entered in the answer table.

⋆ This research was supported in part by NSF grants EIA-9705998, CCR-9876242, IIS-0072927, CCR-0205376, CCR-0311512, and ONR grant N000140110967. We would like to thank Luis Castro for his extensive help with the implementation of our techniques in XSB.

```
1:  reach(X,Y) :- edge(X,Y).
2:  reach(X,Y) :- reach(X,Z), edge(Z,Y).
```

(a)

```
edge(0,1).
edge(0,2).
edge(1,1).
edge(1,2).
```

(b)

```
edge(0,1).
edge(0,2).
edge(1,1).
edge(1,2).
edge(2,3).
```

(c)

Fig. 1. Example reachability program (a) and two `edge/2` relations (b & c).

The Problem: The answers in the tables represent conclusions that can be inferred from the set of facts and rules in the program. *When the program changes (either by addition or deletion of facts/rules), the tables become stale:* they may not have all the answers or the answers in the tables may be incorrect.

For instance, consider the evaluation of query `reach(0,X)` over the program in Figure 1(a) using the definition of `edge/2` relation in Figure 1(b). Tabled evaluation will create an answer table for `reach(0,X)` with {`X=1`, `X=2`} as the answers. Subsequent invocation of `reach(0,Y)` will simply resolve the subgoal against the answers in the table, returning {`Y=1`, `Y=2`} as answers.

Now let a new tuple `edge(2,3)` be added to the `edge/2` relation. Note that the answer table for call `reach(0,X)` contains only answers {`X=1`, `X=2`} and hence is stale. Invocation of, say `reach(0,Z)`, will return only answers `Z=1` and `Z=2`, and miss the answer `Z=3`. The problem becomes worse if tuples can be deleted. If the tuple `edge(0,1)` is deleted from the `edge/2` relation, the query `reach(0,Z)` will still return answers `Z=1` and `Z=2`, even though `reach(0,1)` is no longer true!

Tabling systems currently provide no mechanism to refresh the tables after a change to the program. In fact, in the applications we have built so far, we remove all affected tables after an update to the program and then reissue the query. This approach is clearly wasteful, especially if the changes to the program are small. For instance, in the above example, after the addition of `edge(2,3)`, the subgoal `reach(0,Z)` and its answer table must be removed and recomputed, deriving answers {`Z=1`, `Z=2`, `Z=3`}, in effect *rederiving* answers `Z=1` and `Z=2`.

The above example illustrates the need to incrementally maintain the "freshness" of the tables. We address this problem in this paper. This problem, considered as the *materialized view maintenance problem*, has been extensively studied in the deductive database community (see, e.g. [8, 12] for extensive surveys in this area). Most existing solutions have been derived in the context of bottom-up (semi-naive) evaluation; we address this problem in the context of top-down tabled evaluation. See Section 5 for further discussion on the related work.

Our Solution: In this paper we first consider definite logic programs where all non-tabled[1] user-defined predicates are defined by facts. We consider recursive rules but permit only additions and deletions of facts and rules in a program. We subsequently describe straightforward extensions that remove these restrictions.

[1] Systems such as XSB permit a programmer to mark specific predicates as *tabled*; calls and answers involving other predicates (called *non-tabled* predicates) are not stored in the table.

Handling Addition: Top-down goal-oriented evaluation systems (such as those based on the SLGWAM) inherently process answers incrementally. A subgoal that causes answers to be added to the tables is called a producer, and a subgoal which is resolved against answers already in the tables is called a consumer. The evaluation engine maintains auxiliary data structures to ensure that no consumer sees an answer more than once: e.g. environments to produce and consume answers and control structures linking answer producers to answer consumers. These data structures are torn down when all answers to a call have been derived, an operation that is crucial to memory efficiency of top-down evaluators. Retaining these after query evaluation to support incremental additions imposes unacceptable overheads; e.g., the space usage for evaluating left-recursive reachability queries increases by 2-6 times.

An alternative, similar to the approach used in prior works such as [7], is to generate rules to capture the new answers due to addition of facts. For instance, the changes to the `reach/2` relation can be computed by evaluating the predicate `reach'/2` defined as follows (where the additions to `edge/2` are given by the `edge'/2` relation):

```
reach'(X,Y) :- edge'(X,Y).
reach'(X,Y) :- reach'(X,Z), edge(Z,Y).
reach'(X,Y) :- (reach(X,Z); reach'(X,Z)), edge'(Z,Y).
```

Direct tabled evaluation of the auxiliary rules will lead to two distinct tables for `reach` and `reach'`. Consequently, the same answer may be stored in both the tables, and the two tables must be merged after the incremental evaluation. In Section 2 we describe a data structure that enables the two predicates to share the same table, eliminating most of the overheads of incremental evaluation.

Handling Deletion: Deletion of facts in a program pose a more complicated problem, especially in the presence of recursive rules. Algorithms that incrementally maintain recursive views typically follow the two-phase delete-rederive approach best exemplified by the *DRed* algorithm [7]. The first phase deletes *all* answers which can be derived from the deleted facts. For instance, consider the answers to `reach(0,X)` after the deletion of `edge(0,1)` from Figure 1(c). Using `DRed`, we will delete the answers `reach(0,1)`, `reach(0,2)`, and `reach(0,3)` since all of them can be derived using `edge(0,1)`. The second phase rederives answers deleted in the first phase that have alternative derivations not involving the deleted facts. Continuing with our example, we will now rederive `reach(0,2)` (due to `edge(0,2)`) and consequently `reach(0,3)`. It must be noted that the MCI algorithm for incremental model checking [17] follows a strikingly similar approach (see Section 5).

The delete-rederive algorithms have relatively high overheads. As the above example illustrates, these algorithms hastily delete a number of answers in the first phase, only to rederive many of them (with considerable additional effort) in the second phase.

Our technique handles the deletion of facts as well as rules but achieves considerable savings compared to the delete/redeive algorithms by carefully controlling the deletion of answers in the first phase. Note that, in the above example, `reach(0,2)` need not even be considered for deletion since it has an alternative derivation (due to `edge(0,2)`) when `reach(0,1)` is deleted. We keep a succinct representation of all possible derivations of an answer by keeping a set of *supports* for the answer. Informally, a support for an answer is a set of atoms such that there is a rule $\alpha :- \beta_1, \ldots \beta_n$ where the answer is an instance of α and the support is an instance of $\{\beta_1, \ldots, \beta_n\}$. For instance, the supports for answer `reach(0,2)` are {`edge(0,2)`} and {`reach(0,1), edge(1,2)`}.

The following key observation permits the use of support sets to control the first phase. An answer α is valid whenever there is a valid support s such that s does not depend on α, and a support is valid whenever all answers in it are valid. However, short of maintaining strongly connected components, there is no efficient mechanism to determine the interdependencies between supports and answers. But it turns out that the first support that adds an answer to the table will always be independent of the answer. We use this as a heuristic to control the propagation of deletions. In the above example, deletion of `reach(0,1)` does not delete `reach(0,2)` since the latter is supported by {`edge(0,2)`}. Consequently, `reach(0,3)` is also retained without additional work.

Support sets are also central to the efficiency of the rederivation phase where we attempt to rederive each answer deleted in the first phase. Using support sets, an answer can be rederived by simply checking the validity of supports (analogous to proof checking) instead of launching a full-fledged proof search.

We have implemented our incremental techniques in the XSB tabling engine. Preliminary experiments with our implementation indicate that our support-based technique is superior to standard delete-rederive techniques. Furthermore, the overheads for generating and maintaining supports are negligible compared to the benefits. Our implementation also indicates the ease of deployment, efficiency, and practicality of our techniques.

Summary: We present, to the best of our knowledge, the first techniques for incremental evaluation of tabled logic programs. We describe efficient data structures and algorithms for incremental maintenance of tables in the presence of addition (see Section 2) and deletion (see Section 3) of rules/facts. We also describe how to handle non-tabled predicates and stratified negation. We present preliminary experimental results which show the effectiveness of our techniques (see Section 4). We compare of our work with previous works in incremental view maintenance in Section 5. Further research problems are sketched in Section 6.

2 Addition

Preliminaries The XSB system uses trie-based data structures for storing terms in call and answer tables [15]. Tries permit efficient lookup and one-pass check-insert operations. However, tries do not maintain the terms in the order of insertion. When resolving answers against an incomplete table (where new answers may be added), XSB maintains and uses an *answer list*, which links leaf nodes in the trie in their order of insertion. When a table is complete, which means no new answers can be added to the table with respect to a given set of facts, answer resolution is done by backtracking through the trie top-down; the answer list is no longer needed and is deleted.

Incremental Evaluation after Additions: Let P be a definite logic program and γ be a query. We denote the answers to γ with respect to P by $ans_P(\gamma)$. Let δ_p be a set of facts and rules added to the program P. The problem of incremental evaluation of query γ then is one of computing the smallest set Δ of answers such that $ans_{P \cup \delta_p}(\gamma) = \Delta \cup ans_P(\gamma)$. That is, Δ is the set of *new* answers for γ.

Given a definite logic program P and an added program δ_p we derive a transformed program P' used for incremental evaluation as follows. For each predicate p/n defined in the program $P \cup \delta_p$, we introduce an incremental predicate p'/n. If γ is an atom with p at its root, we denote by γ' the atom obtained by replacing the p in γ by p'. The transformed program P' is such that $ans_P(\gamma) \cup ans_{P'}(\gamma') = ans_{P \cup \delta_p}(\gamma)$.

First of all, P' contains all the clauses in P. For each fact α in δ_p we add α' to P'. For every clause of the form $\gamma \text{ :- } \beta_1, \beta_2, \ldots \beta_n$ in the program $P \cup \delta_p$, we add the clause $\gamma' \text{ :- } (\beta_1; \beta_1'), \ldots, (\beta_{i-1}; \beta_{i-1}'), \beta_i', \beta_{i+1}, \ldots, \beta_n$ for each $i \in [1, n]$. The i-th clause computes new answers of γ due to new answers of β_i. The incremental predicate `reach'` defined in the introduction is derived from the original definition of `reach` by the above transformation. The transformation is a straightforward application of finite differencing [13], and its variants have been widely used for materialized view maintenance [7, 10].

Direct evaluation of the transformed program has two sources of inefficiency. Firstly, the new answers of a query γ are actually added as answers to the new query γ'; consequently, we must merge the two answer tables after the incremental evaluation is complete. Secondly, to ensure that γ' computes only the new answers, each derived answer must be first checked against answers to the original query γ (e.g. using the goal $\neg\gamma$ [18]), causing an extra table lookup.

We overcome these problems by sharing the call table entry and the answer tables between the incremental goal and the original goal, although the calls access the answer table in different ways. Let γ be an original goal and γ' be its incremental counterpart. The first call to γ' creates a new subgoal frame. Answers to γ' are computed by program clause resolution, are added to the answer table of γ, and also kept in a separate answer list. Subsequent calls to γ' consume from this answer list (even after completion of γ')— exactly the same way answers are currently consumed from incomplete tables.

In order to prevent answers to γ' from being accessed when backtracking through the answers of γ, we mark all the newly added answers as "deleted". This exploits the current implementation of tries in XSB which provides a flag to mark terms as deleted without physically removing them. This flag is used in XSB for maintaining dynamic asserted data using tries. Finally, when the incremental evaluation is complete, we reset the deleted flag of all answers in the answer list of γ', thereby adding these answers to γ. Figure 2 shows the states

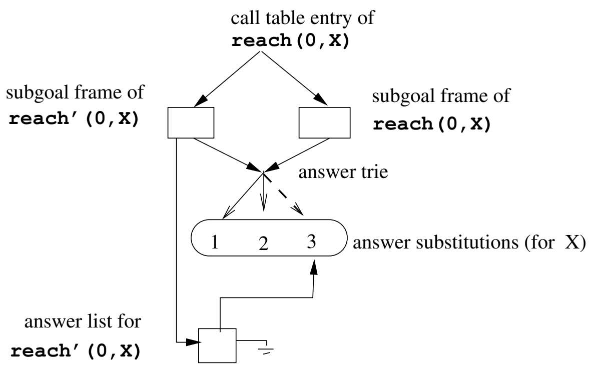

call table entry of reach(0,X)
subgoal frame of reach'(0,X)
subgoal frame of reach(0,X)
answer trie
1 2 3
answer substitutions (for X)
answer list for reach'(0,X)

Fig. 2. Example of the data structure to maintain tables for incremental predicates.

of the answer tables of `reach(0,X)` and its incremental version `reach'(0,X)`, just before the completion of the incremental evaluation.

Discussion: The data structure described above enables us to incrementally evaluate queries without changing any other part of the tabling engine. Moreover, in contrast to bottom-up techniques, we refresh a table only on demand when a query is made. Note, though, that we maintain only two versions of answers, and hence cannot maintain tables with varying "staleness". For instance, if γ_1 and γ_2 are two queries, when γ_1 is incrementally evaluated after changes to facts, observe that γ_1 has consumed all the new facts while γ_2 has not. However, since after this evaluation we merge all new answers with the old, γ_2 will remain stale. A promising approach for solving this problem is to associate timestamps with the added facts and answers, and drive the incremental computation based on the timestamps. With the time-stamp based solution, we can maintain multiple versions of tables within the same data structure and hence handle tables with different degrees of staleness. This is a topic of future research.

3 Deletion

Let P be a definite logic program, γ be a query and δ_p be a set of facts F and set of rules R_d to be deleted from the program. For notational purposes, we assume that every rule is associated with an unique identifier (e.g see Figure 1(a)). Following the notation used in Section 2, the problem of incremental evaluation of query γ after the deletion is that of computing a set Δ of answers such that $ans_{P-\delta_p}(\gamma) = ans_P(\gamma) - \Delta$. We develop an algorithm to efficiently compute the set Δ in this section and describe its implementation in terms of tabling data structures.

Formulation: Clearly, the only answers that can be in Δ are those that depend on δ_p. The algorithms based on the delete-rederive approach [7, 17] first over-

```
edge(0,1)
edge(0,2)
edge(1,1)
edge(1,2)
```
(a)

Answer	Supports
`reach(0,1)`	⟨`1`,{`edge(0,1)`}⟩, ⟨`2`,{`reach(0,1)`,`edge(1,1)`}⟩
`reach(0,2)`	⟨`1`,{`edge(0,2)`}⟩, ⟨`2`,{`reach(0,1)`,`edge(1,2)`}⟩

(b)

Fig. 3. Example edge relation (a); and supports for answers to query `reach(0,X)` over that relation (b).

approximate Δ by the set of <u>*all*</u> answers that depend on δ_p. We use a better approximation based on the notion of *support* for an answer defined below.

Definition 1 (Support) *Let P be a program, and let T be a set of answer tables obtained when evaluating a query γ over P. A tuple $s = \langle k, \{\beta_1, \beta_2, \ldots, \beta_n\}\rangle$ is called a support of an answer α of γ if there exists a clause k of the form $\alpha' \text{ :- } \beta'_1, \beta'_2, \ldots, \beta'_n$ and a substitution θ, such that $\alpha'\theta = \alpha$, and for all $i \in [1, n]$ $\beta'_i\theta = \beta_i$ and β_i is an instance of an answer in T or a fact in P.*

For instance, consider the query `reach(0,X)` over the reachability program in Figure 1(a) and `edge/2` relation in Figure 3(a).

Note that each support for an answer represents one step in some derivation of that answer. We can construct a derivation of an answer by picking a support and constructing derivations for all the atoms in that support. However, the choice of support picked at each step is crucial to the construction of a valid (i.e. finite) derivation. For instance, picking the support ⟨`2`,{`reach(0,1)`,`edge(1,1)`}⟩ each time to derive `reach(0,1)` will not lead to a finite derivation; for finiteness we must eventually pick the support ⟨`1`,{`edge(0,1)`}⟩. The key to quickly determining whether an answer is still derivable is to distinguish supports which can be selected without regard to the history and yet build finite derivations. This is done using the notion of a primary support, defined below.

Definition 2 (Primary Support) *Let $\alpha \text{ :- } \beta_1, \beta_2 \ldots, \beta_n$ be the instance of a rule k that is used by tabled resolution to derive the answer α for the first time. Then $\langle k, \{\beta_1, \beta_2 \ldots, \beta_n\}\rangle$ is called the primary support of α, and is denoted by $ps(\alpha)$.*

In Figure 3(b) the first support listed for each answer is its primary support. We use primary supports to (over)approximate the set Δ of answers to be deleted as follows.

Definition 3 (Candidates for Deletion) *Let P be a program, γ be a query, and A be the answers computed during the evaluation of γ. Let $ps(\alpha) = \langle k, S\rangle$ be the primary support $\alpha \in A$. The set of candidates for deletion due to the deletion of the program δ_p from P, denoted by $\Gamma(P, \delta_p)$ is the smallest set such that $\alpha \in \Gamma(P, \delta_p)$ whenever $\exists\beta \in S$ such that $\beta \in F \cup \Gamma(P, \delta_p)$ or rule $k \in R_d$.*

It is easy to establish that the set of candidates for deletion overapproximates the set of deleted answers. Formally,

Proposition 1. *The set of answers for a query γ over a program P, $ans_P(\gamma)$ is such that $ans_P(\gamma) - ans_{P-\delta_p}(\gamma) \subseteq \varGamma(P, \delta_p)$.*

Traditional delete-rederive algorithms such as [7, 17] use a coarser approximation. Also the effect of deletion of rules is not addressed in any view maintenance literature. The answers they delete in the first phase, which we denote by $\varGamma^\sharp$, can be characterized as follows. The set $\varGamma^\sharp$ which is the smallest set such that $\alpha \in \varGamma^\sharp(P, \delta_p)$ if there is some support $s = \langle r_k, S\rangle$ of α such that $\exists \beta \in S$ and $\beta \in \delta_p \cup \varGamma^\sharp(P, \delta_p)$. It can be easily shown that $\varGamma(P, \delta_p) \subseteq \varGamma^\sharp(P, \delta_p)$. Note that the coarser approximation has a cascading effect: an answer marked incorrectly as a candidate, in turn, leads to (incorrect) marking of other answers. Our approximation reduces such propagation and hence is considerably less coarse. Note that the notion of primary support is not specific to tabled evaluation and can be easily extended to any least fixed point computation.

Although only primary supports are used to obtain candidates for deletion, we still keep the set of all supports for an answer. First of all, note that due to the approximation, some candidates for deletion may be still derivable. We check this in the second *rederivation* phase. Traditional algorithms in the view maintenance literature pose rederivation in terms of rule evaluation. In contrast, we avoid the proof search using an algorithm based on keeping counts and the set of all supports with each answer. Secondly, when a primary support is removed in incremental evaluation but the answer is still valid, we need to identify the new primary support; the new support can be easily generated from the set of all supports. Lastly, we can improve our approximation by finding all supports of an answer which are not dependent on the answer itself and falsify the answer when all such supports are falsified. Below we describe the data structures and algorithms for computing the candidates for deletion and for rederiving answers.

Data Structures: Supports for an answer are maintained using a bipartite graph, called the support graph. Its vertices are partitioned into two sets: *or-nodes* and *and-nodes*. Every or-node in a support graph corresponds to an answer, a fact or a rule and every and-node corresponds to a support.

Edges in the support graph are placed as follows. Whenever s is a support for answer α, we place an edge from s to α. These edges define the 'support of' relationship. The edges are represented by an attribute *support_of* of and-nodes such that $s.\mathit{support\_of} = \alpha$. Whenever a support s contains a fact or an answer or a rule β, we place an edge from β to s. These edges define the 'part of' relationship. These edges are represented by a (set-valued) attribute *part_of* of or-nodes such that $s \in \beta.\mathit{part\_of}$. At first the direction of edges may appear to be counter-intuitive. However, it coincides with the flow of information in our algorithm: we propagate deletion and rederivation from an or-node β to all and-nodes s that contain β; and from an and-node s to or-nodes α for which s is a support. For illustration, Figure 4 shows the support graph for answers to query `reach(0,X)` for the support sets listed in Figure 3(a).

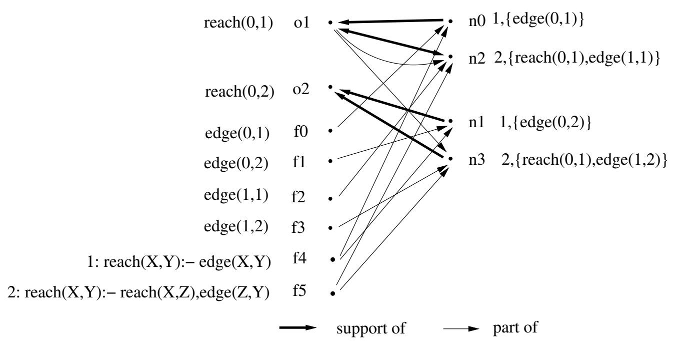


Fig. 4. Support graph for answers to query `reach(0,X)`.

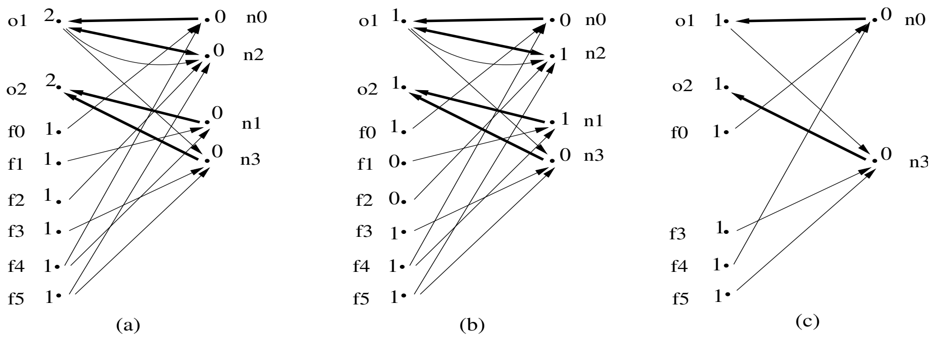


Fig. 5. Counts in support graph: at start (a); after deletion (b); after rederivation (c).

We also maintain the following additional attributes with each or-node o: (i) *supportlist*: the list of all supports of o; (ii) *answer*: the answer corresponding to o; and (iii) *primary_support*: the primary support of o.

In the rederivation phase, we need to check if candidates for deletion identified in the first phase have alternative derivations. We do so efficiently by maintaining counts with each node in the support graph. The meaning of the counts is different for or-nodes and and-nodes. For an or-node representing an answer α, its count denotes the number of supports that α must lose before it becomes false. For example, in Figure 5(a) the count of or-node $o1$ and $o2$ is 2. The count of an or-node representing true fact or a rule is initially 1; it is decreased by 1 when the fact or the rule is deleted. For an and-node representing a support s, its count denotes the number of false answers and facts in s; in other words, the count is the number of answers that must become true before the support itself becomes true. An and-node's count enables us to quickly determine the truth value of a support without evaluating its constituents.

In Figure 5(a) the count of and-nodes $n0$, $n1$, $n2$ and $n3$ are all zero. Whenever an or-node o becomes false, we increment the count of all and-nodes that

```
delete:
foreach unprocessed $o \in dlist$
   foreach $s$ in $o.part\_of$
      $s.count++$;
      if ($s.count = 1$) then
         /* $s$ just became false */
         $o' := s.support\_of$;
         $o'.count--$;
         if ($o'.primary\_support = s$) then
            $o'.leaf\_node.deleted := true$;
            $dlist := dlist + \{o'\}$;
```

```
rederive:
/* 1. Find the basis for rederivation. */
$rlist := \{\}$;
foreach $o \in dlist$
   if ($o.count > 0$) then
      $o.leaf\_node.deleted := false$;
      $rlist := rlist + \{o\}$;
      let $s \in o.supportlist$
            such that $s.count = 0$;
      $o.primary\_support := s$;
/* 2. Propagate rederived answers. */
foreach unprocessed $o \in rlist$
   foreach $s \in o.part\_of$
      $s.count--$;
      if ($s.count = 0$) then
         $o' := s.support\_of$;
         $o'.count++$;
         if ($o'.count = 1$) then
            $o'.primary\_support := s$;
            $o'.leaf\_node.deleted := false$;
            $rlist := rlist + \{o'\}$;
```

Fig. 6. Algorithm for incremental evaluation after deletion.

contain o (given by $o.part\_of$). Similarly, when an and-node s becomes false, we decrement the count of the or-node that is supported by s (given by $s.support\_of$).

The Algorithm: The algorithm for incremental evaluation after deletion of facts is shown Figure 6 and has two phases as described below.

Deletion Phase: The algorithm starts in the deletion phase, with the variable *dlist* initialized to the set of or-nodes corresponding to the deleted facts and rules. When the phase ends, the variable *dlist* represents the candidates for deletion (Γ in Definition 3). We explain the algorithm by using the support graph in Figure 5(a) as an example. Consider the deletion of two facts `edge(0,2)` (or-node $f1$) and `edge(1,1)` (or-node $f2$). We enter the deletion phase with *dlist* set to $\{f1, f2\}$. Observe from Figure 5(a) that $f1.part\_of = \{n1\}$; hence we increment $n1.count$ to 1. Now, $n1.support\_of$ is the or-node $o2$, and $o2.count$ is decremented to 1. Moreover, $n1$ is the primary support $o2$. So $o2$ is a candidate for deletion and is added to *dlist*.

We then process or-node $f2$. We increment $n2.count$ to 1. Since $n2.support\_of$ is the or-node $o1$, we decrement $o1.count$. However, $n2$ is not the primary support of $o1$, and hence nothing is added to *dlist*. Finally we pick the unprocessed node $o2$, but since $o2.part\_of$ is empty, the processing ends. It must be noted that traditional delete-rederive algorithms would have picked both $o1$ and $o2$ as candidates for deletion. In contrast, we have been able to approximate the set of deleted answers better, and not even consider $o1$ as a candidate for deletion.

Rederivation Phase: The rederivation phase, invoked with the set of candidates for deletion in *dlist*, proceeds in two steps. In the first step we determine, among all nodes in *dlist*, those that have alternative supports. In the second step we propagate the rederived answers through the support graph. We explain the rederivation phase by continuing our earlier example.

In Figure 5(b), after the deletion phase, *dlist* contains $\{f1, f2, o2\}$. Among these the counts of $f1$ and $f2$ (which correspond to the deleted facts) are zero and hence lead only to trivial processing. The count of $o2$ is 1 indicating that it is actually true. Hence we add $o2$ to *rlist*; its support $n3$ has a zero count indicating that it is valid. Hence we make $n3$ as the primary support of $o2$. We begin the second step with *rlist* set to $\{o2\}$. Note that $o2$ is not a part of any support in the graph and hence the processing ends.

Discussion: Supports for an answer are constructed based on the rule that generated the answer. In our implementation, when inserting an answer in a table we determine its supports by using the consumer choice points and other structures used to generate the answer. Hence the construction of the support graph does not increase the time complexity of query evaluation. However, the space requirements for the support graph typically exceed that of the answer tables. For instance, the number of answers for query `reach(X,Y)` is $O(n^2)$ where n is the number of vertices in the graph. For each answer, there may be up to n supports, and hence the support graph has $O(n^3)$ nodes.

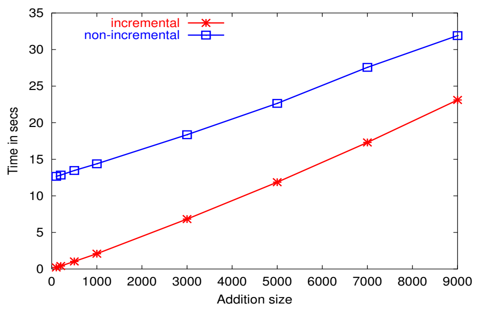
4 Experimental Results

We implemented our incremental algorithms by modifying XSB (version 2.5). In the following we present results of our preliminary experiments designed to measure (i) the effectiveness of our techniques, (ii) their overheads, (iii) the effect of repeated incremental evaluation, and (iv) the effectiveness of using supports to control deletion. All measurements were made on an Intel Xeon 1.7GHz machine with 2GB RAM running RedHat Linux 7.2. For nonincremental evaluation, we used the standard release of XSB version 2.5. We used left-recursive reachability and same generation predicates over trees and complete graphs as benchmarks.

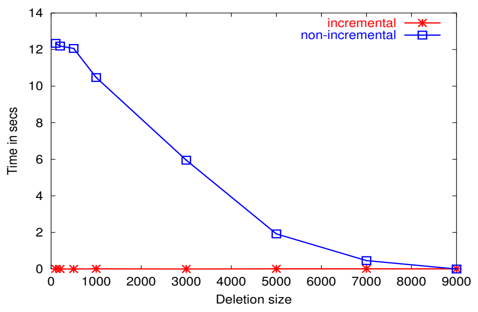
Effectiveness: Figures 7(a) compare the incremental and non-incremental evaluation time of one query after the *addition* of a set of facts to the edge relations. The figures do not include the times for evaluating the initial query. Observe that when the size of addition is small (less than 5% of the total size of the edge relation), incremental evaluation is 20–60 times faster than nonincremental evaluation. Moreover, as the size of the addition increases, incremental evaluation time approaches that of nonincremental evaluation, but remains lower even when the size of addition is over 90% of the original edge relation.

Figure 7(b) compares the times of one query after the *deletion* of a set of facts from the edge relation. Incremental deletion on this benchmark takes very little time (less than 0.1s for all deletion set sizes) and far outperforms its non-incremental counterpart. The two are comparable only when the input graph becomes very small due to deletion of a large number of edges.

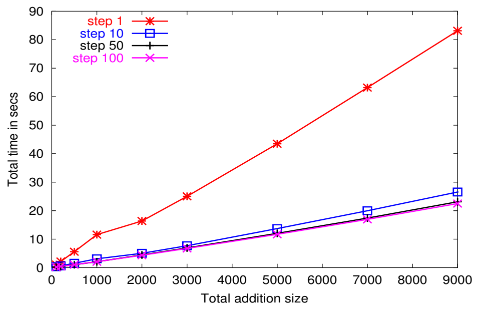
Overheads: We find that the initial query evaluation time for incremental evaluation is at most 8% greater than that for nonincremental evaluation. How-



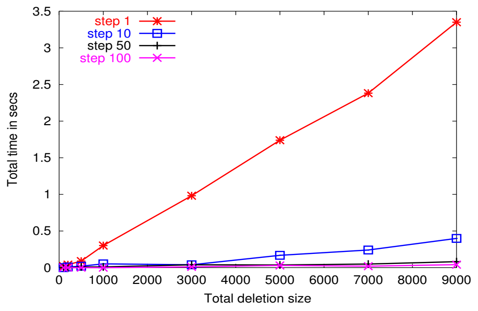
(a) Addition [reach(a,X);tree(9999 edges)]



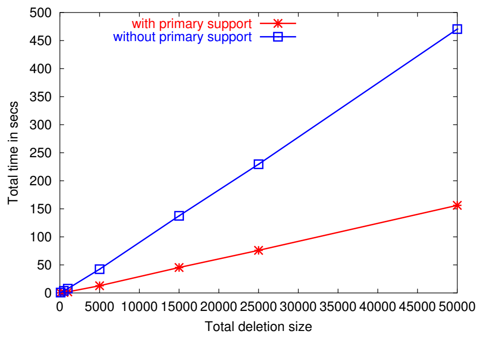
(b)Deletion [reach(a,X);tree(9999 edges)]



(c) Repeated additions
reach(a,X);tree(9999 edges)



(d) Repeated deletions
reach(a,X);tree(9999 edges)



(e) Deletions with and without primary support [reach(a,X);complete graph(89700 edges);step 1]

Fig. 7. Experimental results

ever, since the support graph size may be much larger than the answer set size, we do observe significant memory overheads. Memory overheads range from a factor of 1.9 (49MB incremental vs. 26MB nonincremental) for same generation queries over binary trees, to as much as 131 (2.5MB incremental vs. 19KB nonincremental) for reachability over complete graphs. Note that the space overhead is due to the support graph alone, and is incurred only if we handle deletions.

Effect of repeated evaluation: We now compare the performance of the incremental engine for sequences of query evaluations and changes (additions/ deletions) such that the changes are interspersed between evaluations. In all the runs, the total number of changes is the same; the number of changes between two query evaluations is the step size of the run. For instance, a run with step size 10 means that after every 10 changes we issue an incremental query to refresh the table. Figure 7(c) and (d) show the total evaluation times for additions and deletions, respectively, for runs with different step sizes. Observe from the figure that batching changes together (i.e. querying infrequently, and hence allowing tables to go stale) usually takes less time than maintaining the tables fresh all the time (i.e. step size 1). Nevertheless, consistently maintaining freshness is only only 3 to 5 times slower than refreshing the table only after all changes are done. This reflects the low overheads for incremental query evaluation.

Role of support-based control of deletion: Finally, we compare the effectiveness of selecting the candidate answers for deletion based on primary supports. Figure 7(e) compares the performance of the incremental engine with control based on primary support turned off, with that of the full algorithm. Both versions used support graphs for rederivation. We measured the performance of the two versions for a sequence of deletions from a complete graph, issuing a query after each deletion. Observe from Figure 7(e) that use of primary supports results in a more than 3-fold reduction in evaluation time.

5 Related Work

The materialized view maintenance problem has been extensively researched (see, e.g. [8, 12] for surveys). Most of the works in recursive views maintenance generate rules that are similar in spirit to those of DRed [7] and are subsumed by DRed (as compared in [8]).

The relationship of our work to the DRed algorithm [7] has been explained in sections 2 and 3. We handle programs with stratified negation in the same way as DRed algorithm. It should be noted that the idea of counts has been used in other works such as [6, 17] but has not been used to avoid recomputing subgoals in recursive rules. The transformation rules for supporting addition are similar to those of [7, 18] but we evaluate the incremental rules top-down and on-demand. We also use specialized data structures to efficiently generate new answers and avoid propagation of generation of old answers.

Our techniques are also closely related to the incremental model checking algorithm (MCI) of [17]. We have adopted MCI's use of counts to efficiently compute truth values of nodes during incremental evaluation. The MCI algorithm constructs a *product graph* where each node is associated with a truth value, and the edges denote the dependencies between the nodes' values. Its product graph corresponds to propositional logic program. MCI algorithm falsifies and rederives same number answers as DRed does with respect to this program. Hence, our improvement over DRed carries over to MCI also.

Straight Delete (StDel) algorithm [11] eliminates the rederivation phase of DRed by keeping the entire proof with every answer. While such an approach may be feasible for constraint databases, it is prohibitively expensive for logic programs. For instance, while the support set size for a context free grammar parser is cubic in the length of the string, the number of distinct proofs may be exponential. Thus a succinct representation such as a support graph is essential. However, since we do not keep all the proofs, we cannot avoid rederivation.

A top-down algorithm for incrementally checking integrity constraints (which can be seen as views) is presented in [16]. This algorithm first computes the set of integrity constraints that are possibly affected by the changes to the facts. It then evaluates the integrity constraints top-down. The method works only for non-recursive predicates. However, the idea of using a bottom-up propagation phase to mark the goals that may need to be evaluated is an interesting aspect that we plan to study in the future.

6 Concluding Remarks

We presented, to the best of our knowledge, the first techniques for incrementally evaluating tabled logic programs. Our implementation shows that incremental evaluation in the presence of addition of facts and rules can be added without any overhead whereas there is a tradeoff between memory overhead and performance in presence of deletion of facts and rules. This work opens up numerous interesting research questions, a few of which are enumerated below.

We handle deletions in a purely bottom-up fashion. It would be interesting to propagate answer deletions lazily, only on demand. Furthermore, the support sets that we use to handle deletion can be very large, and we seek ways to reduce the storage requirement. First of all, we need to design clever data structures to share components of different supports. Secondly, we can let the support sets be incomplete (i.e. not store all the supports), thereby trading space for time (for proof searches on rederivation). Thirdly, certain base facts may be "indelible": i.e., can never be deleted. Answers derived solely based on such facts can be also be simply marked "indelible", and we need not keep any supports for them.

For handling deletion in the presence of non-tabled predicates, we can build supports for a non-tabled answer, say β, on the stack, and include these supports whenever a tabled answer α is generated based on β. This approach mimics the way delay lists are propagated across non-tabled predicates when evaluating the well-founded model of a non-stratified program [2].

Finally, we have not considered *updates* to rules and facts as an independent operation. For instance, consider the problem of incremental parsing. A change in the input string can be considered in terms of a set of deletions and additions. However, such a formulation results in the reparsing of a large segment of the string: deletions "cut" the string into pieces which has a profound effect on the parse tree; additions "join" the pieces back, making large-scale changes to the parse tree again. Designing incremental evaluation techniques by considering update as a basic operation is an interesting and important open problem.

References

1. R. Bol and L. Degerstadt. Tabulated resolution for well-founded semantics. In *ILPS*, 1993.
2. W. Chen, T. Swift, and D. S. Warren. Efficient implementation of general logical queries. *J. Logic Prog.*, 1995.
3. W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, 1996.
4. L. Damas and V. S. Costa. The YAP prolog system, 2002. `http://www.ncc.up.pt/~vsc/Yap/`.
5. S. Dawson, C. R. Ramakrishnan, and D. S. Warren. Practical program analysis using general purpose logic programming systems — a case study. In *PLDI*, 1996.
6. A. Gupta, D. Katiyar, and I. S. Mumick. Counting solutions to the view maintenance problem. In *Workshop on Deductive Databases, JICSLP*, pages 185–194, 1992.
7. A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of ACM SIGMOD*, pages 157–166, 1993.
8. A. Gupta and I.S. Mumick. Maintenance of materialized views: Problems, techniques, and appfications. *IEEE Data Engineering Bulletin*, 18(2):3–18, 1995.
9. G. Gupta H-F. Guo. A simple scheme for implementing tabled logic programming systems based on dynamic reordering of alternatives. In *ICLP*, volume 2237 of *LNCS*, pages 181–196, 2001.
10. Y. Liu and S. Stoller. From Datalog rules to efficient programs with time and space guarantees. In *PPDP*, 2003.
11. J. Lu, G. Moerkotte, J. Schue, and V. S. Subrahmanian. Efficient maintenance of materialized mediated views. In *ACM SIGMOD*, pages 340–351, 1995.
12. E. Mayol and E. Teniente. A survey of current methods for integrity constraint maintenance and view updating. In *ER Workshops*, pages 62–73, 1999.
13. R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM TOPLAS*, 4(3):402–454, 1982.
14. C.R. Ramakrishnan, I.V. Ramakrishnan, S.A. Smolka, Y. Dong, X. Du, A. Roychoudhury, and V.N. Venkatakrishnan. XMC: A logic-programming-based verification toolset. In *CAV*, volume 1855 of *LNCS*, pages 576–580, 2000.
15. I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren. Efficient tabling mechanisms for logic programs. In *ICLP*, pages 697–711, 1995.
16. R.R. Seljee and H.C.M. de Swart. Three types of redundancy in integrity checking; an optimal solution. *Journal of Data and Knowledge Enigineering*, 30:135–151, 1999.
17. O. V. Sokolsky and S. A. Smolka. Incremental model checking in the modal mu-calculus. In *CAV*, volume 818 of *LNCS*, pages 351–363, 1994.
18. M. Staudt and M. Jarke. Incremental maintenance of externally materialized views. In *The VLDB Journal*, pages 75–86, 1996.
19. H. Tamaki and T. Sato. OLDT resolution with tabulation. In *ICLP*, pages 84–98, 1986.
20. XSB. The XSB logic programming system. Available from `http://xsb.sourceforge.net`.
21. G. Yang and M. Kifer. Flora: Implementing an efficient dood system using a tabling logic engine. In *Computational Logic*, volume 1861 of *LNCS*, pages 1078–1093, 2000.
22. N.-F. Zhou, Y.-D. Shen, L.-Y. Yuan, and J.-H. You. Implementation of a linear tabling mechanism. *Journal of Functional and Logic Programming*, 2001(10), 2001.