BOTTOM-UP BEATS TOP-DOWN FOR DATALOG

Jeffrey D. Ullman
Stanford University

ABSTRACT

We show that for any safe datalog program $\mathcal{P}_1$ and any query Q (predicate of $\mathcal{P}_1$ with some bound arguments), there is another safe datalog program $\mathcal{P}_2$ that produces the answer to Q and takes no more time when evaluated by semi-naive evaluation than when $\mathcal{P}_1$ is evaluated top-down.

I. Introduction

You may think you know this result already. First, Beeri and Ramakrishnan [1987] claim something like it. However, as we shall show by example, their magic-sets construction can produce rules whose bottom-up evaluation is much more expensive than a straightforward top-down evaluation.

Then, Ramakrishnan [1988], and independently Seki [1988], show that for any logic program $\mathcal{P}_1$ at all, we can find a logic program $\mathcal{P}_2$, whose semi-naive evaluation is no slower than the top-down evaluation of $\mathcal{P}_1$. However, both these results involve bottom-up computation for relations whose tuples contain variables, *even when $\mathcal{P}_1$ is a datalog program.* With variables in tuples, joins become unifications, and it is unclear that reasonable efficiency can be obtained in all cases. Our construction does not use variables in tuples for any datalog program.

In what follows, we assume the reader is familiar with a number of common concepts from Ullman [1988, 1989], including:

1. Datalog, that is, Horn-clause logical rules with no function symbols and no negated subgoals.
2. Safety of rules, that is, every variable in the head appears in some ordinary (not a built-in arithmetic comparison) subgoal.
3. Semi-naive evaluation, that is, differential computation of the least fixed point.
4. Adornments, or binding patterns, which are sequences of b's (bound) and f's (free), indicating the status of arguments of a predicate.
5. Top-down evaluation. Here, we have in mind specifically a version of rule/goal tree expansion (Ullman [1985]) with left-to-right sideways information passing and a breadth-first expansion strategy to guarantee reaching all nodes in the tree eventually, as described in Ullman [1989] and sketched in Section III. However, this algorithm is easily seen to mimic the search performed by Prolog's SLD resolution strategy, and in fact, it converges to an answer in situations where Prolog would enter an infinite loop and fail to return an answer. We do not consider arbitrary SIPS (sideways information-passing strategies), as proposed by Beeri and Ramakrishnan [1987]. However, the results extend naturally, as long as we keep the same SIPS for $\mathcal{P}_2$ as is used in the evaluation of $\mathcal{P}_1$.

II. A Problem With Magic Sets

Consider the rules

```
r1:  p(X,Y,W) :- a(X,Y,W).
r2:  p(X,Y,W) :- b(W,Y,Z) & p(X,X,Z).
```

First, observe that if we start with query $p(X, Y, 1)$, then the only binding pattern we ever get for p is ffb according to the algorithm of Beeri and Ramakrishnan [1987]. That is, when we come to the recursive subgoal in r_2, $p(X, X, Z)$, the last argument is bound because Z appears in a previous subgoal, but X is free, because it appears neither in a previous subgoal nor in a bound position of the head.

Suppose that the database consists of the tuples $b(1, 2, 3)$ and $b(3, 4, 5)$, as well as a large number of tuples of the form $a(m, n, 5)$, where m and n are not

Work supported by NSF grant IRI-87-22886, a grant of IBM Corp., Air Force grant AFOSR-88-0266, and a Guggenheim fellowship.

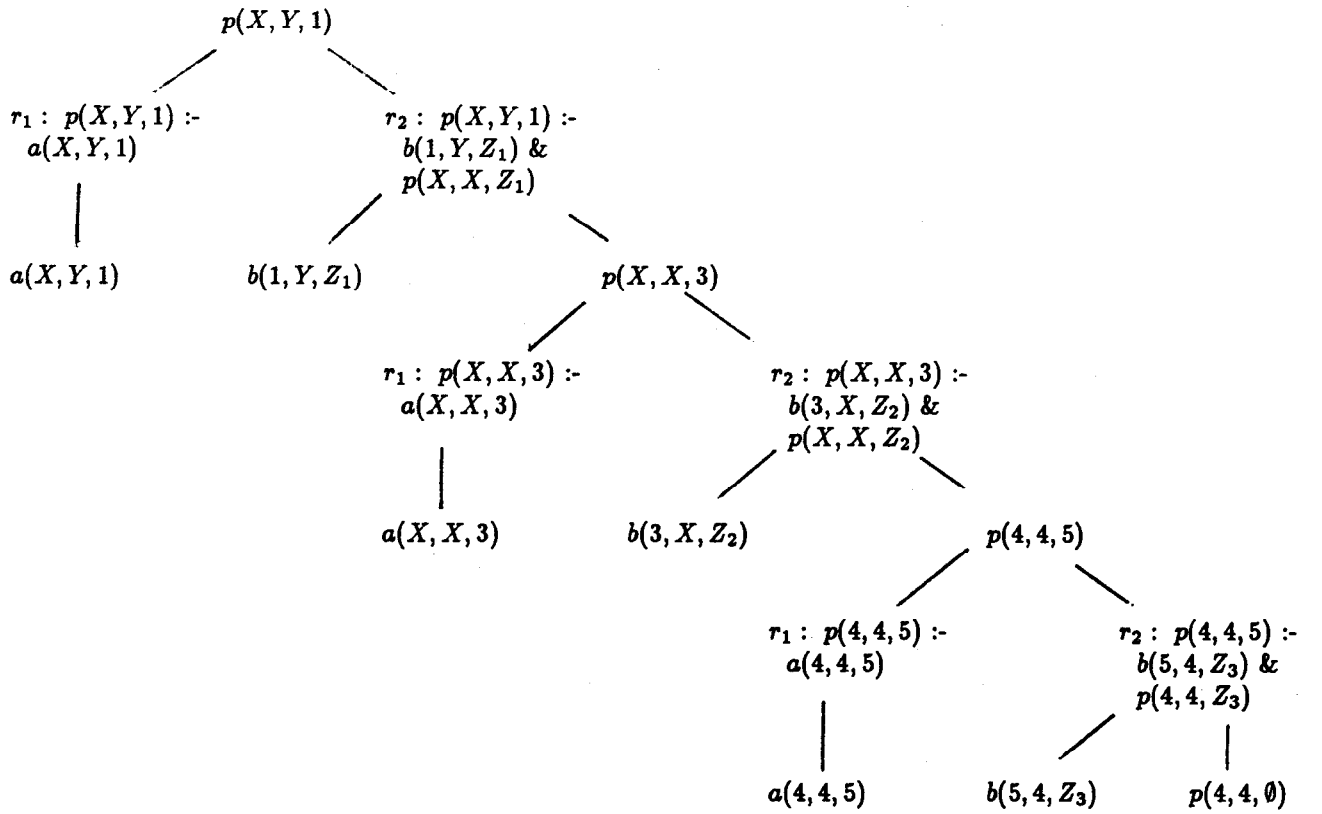

Fig. 1. Rule/goal tree with constants inserted into goals.

both 4. Figure 1 shows the rule/goal tree with root $p(X, Y, 1)$. All sideways passing of bindings fortuitously produces singletons or empty sets, so we show bindings passed explicitly as constants. For example, the rightmost grandchild of the root, $p(X, X, 3)$ is technically $p(X, X, Z_1)$, with Z_1 bound to $\{3\}$ because of the previous lookup in the b relation for subgoal $b(1, Y, Z_1)$. More details of the process for the general case are given in Section III.

Notice that the tree need not be expanded further, since the empty binding for the third argument of p at the bottom assures us that there can be no tuples produced by that branch. The remaining branches, terminating in leaves $a(X, Y, 1)$, $a(X, X, 3)$, and $a(4, 4, 5)$, can produce no tuples either, because the database relation for a has no tuple ending in 1 or 3 and has no tuple whose first two components are both 4. Thus, top-down expansion rapidly discovers that the answer to the query is empty.

Now consider what happens when we apply the magic-sets transformation to r_1 and r_2. Figure 2 shows the rules that result from applying the generalized supplementary magic sets construction (Beeri and Ramakrishnan [1987]) to rules r_1 and r_2 with the query $p(X, Y, 1)$, then simplifying by eliminating superfluous supplementary predicates as in the "minimagic" algorithm of Sacca and Zaniolo [1987].[1] More detail regarding the general magic-sets construction can be found in Section VI.

```
m_p(1).
m_p(Z) :- sup2.1(W,Y,Z).

sup2.1(W,Y,Z) :- m_p(W) & b(W,Y,Z).

p(X,Y,W) :- m_p(W) & a(X,Y,W).
p(X,Y,W) :- sup2.1(W,Y,Z) & p(X,X,Z).
```

Fig. 2. Magic-set rules.

Intuitively, m_p, the magic predicate for p, represents the set of values reachable from 1 by following a sequence of links; each link goes from the first argument of a tuple for the predicate b to the third argument of that tuple. Predicate $sup_{2.1}$, the first supplementary

[1] The later simplifications do not affect the running time of semi-naive evaluation significantly, but make the rules clearer.

relation for the second rule (r_2), is the subset of b that is relevant, because its first (and therefore third) arguments are reachable from 1.

If we simulate bottom-up evaluation of these rules, we discover that the first three rules, with the database facts $b(1,2,3)$ and $b(3,4,5)$, place 1, 3, and 5 in the relation for m_p. With $W = 5$, the fourth rule puts all of the a-facts, $(m, n, 5)$ in the relation for p. Since there is no limit on the number of these facts, we conclude that semi-naive (or any bottom-up) evaluation of the magic rules takes arbitrarily more time than top-down evaluation of the original rules.

III. The Top-Down Algorithm

The top-down algorithm we shall use as a benchmark for magic-sets algorithms is described at length in Ullman [1989], where it is called *QRGT* (Queue-Based Rule/Goal Tree expansion). Readers familiar with this algorithm should skip directly to Section IV.

The general idea is to build a rule/goal tree like Fig. 1, but that tree may be infinite, so it is built breadth-first. We also need to propagate tuples from several different kinds of relations around the tree, and at times we create new tuples by matching tuples with other tuples in a join, or by matching EDB goals[2] with their corresponding relations. We assume a fair allocation of time to the various steps, for example, a queue to service all requests, so that every tuple that can be discovered eventually is discovered. The exact discipline does not matter, as long as the algorithm does not get lost exploring some infinite path of the tree and never considers some other branch that yields an answer to the query.

The nodes of the tree are either

1. *Goal nodes*, corresponding to a predicate with arguments, or
2. *Rule nodes*, corresponding to substituted instances of rules.

The root is a goal node representing the query. Each goal node whose predicate is EDB is a leaf. Each goal node with an IDB predicate p has a rule-node child for every rule with head predicate p that unifies with the goal. If r is such a rule, and τ is the most general unifier of the goal and the head of r, then the instance of r at the rule node in question is r with substitution τ applied.

Each rule node has a goal-node child for each subgoal of the rule. The goal at such a goal node is the substituted instance of the subgoal, as it appears at the rule node. Figure 1 is essentially an example of this rule/goal tree construction process, but we have shown certain arguments bound that are technically variables (which are bound by constants discovered from the query and the database, by a process we shall discuss momentarily). Figure 3 shows the top portion of the true rule/goal tree, which is in principle infinite.

There are three kinds of relations associated with nodes of the rule/goal tree.

1. *Binding Relations.* For each goal node there is a binding relation whose attributes correspond to the bound arguments of the goal itself. For example, if the query is $p(X, Y, 1)$, then the root goal node in Fig. 3 has only the third argument bound, and its binding relation is $\{1\}$. In general, a binding relation can have any finite set of tuples, although it happens that in the example of Fig. 1, all bindings were to singletons.
2. *Answer Relations.* At each goal node there is a set of answers for that goal. These are the tuples that
   a) Are inferred by the rules and database,
   b) Unify with the goal, and
   c) Match some tuple of the binding relation for that goal.[3]

   For example, the root of Fig. 3, with binding relation $\{1\}$, will produce as an answer every p-fact with 1 in the third component.
3. *Supplementary Relations.* At each rule node for a rule with k subgoals, there are k supplementary relations, numbered $0, 1, \ldots, k-1$. The supplementary relation S_i reflects the binding of variables of the rule instance at that rule node, after the first i subgoals have been processed. The attributes of S_i are the variables that are "bound and relevant" after consideration of the first i subgoals. A variable is *bound* if it appears in a bound argument of the head or in any argument of the first i subgoals. A variable is *relevant* if it appears either in any position of the head or in a subgoal after the first i subgoals; that is, the variable will be needed later on as we evaluate the rule and return answers to the head.

[2] A goal is called EDB (extensional database) if the data for the predicate of that goal is stored in the database. Likewise, a predicate or relation can be called EDB. The opposite of EDB is IDB (intensional database), and means that the tuples for a predicate, goal, or relation are defined by rules and not stored explicitly.

[3] If there are no bound arguments, the binding relation consists of only the empty tuple, which matches every tuple.

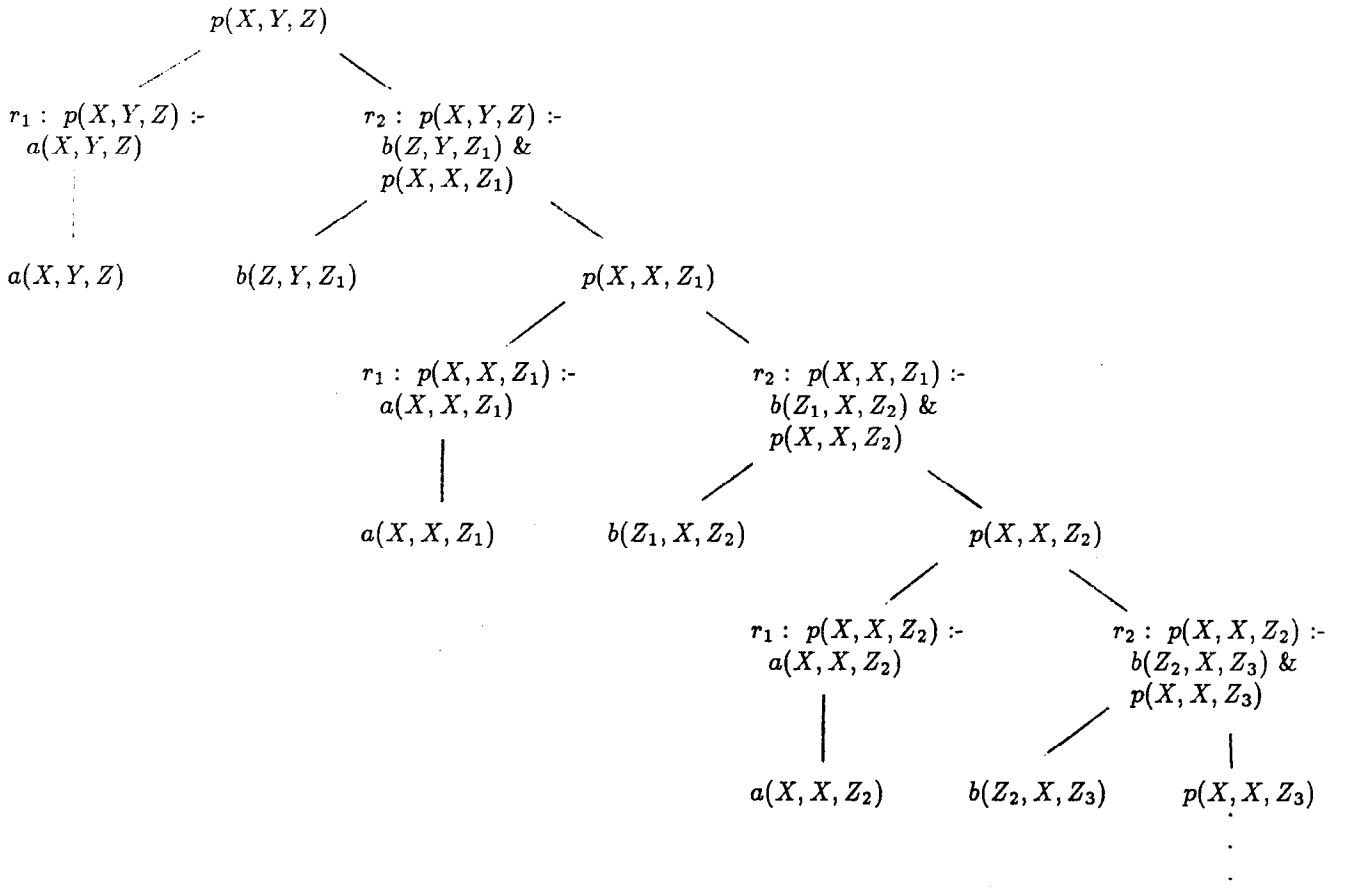

Fig. 3. True rule/goal tree.

An easy way to remember the rule for the arguments of supplementary relations is to pretend that the entire head appears after the last subgoal and the bound arguments of the head also appear before the first subgoal. then "bound and relevant" at a point between subgoals means that the variable appears both before and after that point.

The purpose of S_i at a rule node is to represent the binding relation for the predicate of the head (taken from the goal node parent of the rule node in question) joined with the answer relations for the subgoals prior to the ith subgoal, then projected onto the relevant variables. In terms of Prolog (SLD resolution), each tuple of S_i gives a possible binding for the relevant variables that can occur in some activation record for the rule in question, after i subgoals of that rule have been called. However, the correspondence is a bit subtle. Assuming no loops, Prolog will construct stacks of activation records for rules and goals, and each stack will correspond to a path from the root in the rule/goal tree. The same node of the tree may correspond to many different activation records, which Prolog constructs at different times. Each tuple of S_i for a particular tree node corresponds to the bindings that occur in one or more of these activation records. The reason the same tuple can represent more than one activation record is that irrelevant variables are dropped from S_i, but, of course, must be retained in the activation record.

Example 1: Consider the right child of the root in Fig. 3. For S_0, only variable Z is bound, because the query, in our running example, only binds the third argument of the root, and this binding in turn binds Z in the head of r_2. Z is also relevant, since it appears in the first subgoal; it also appears in the head.

For S_1, we have Z bound by the head, and Y and Z_1 bound by the first subgoal, $b(Z,Y,Z_1)$. Z_1 appears in the second subgoal, while Y and Z appear in the head. Thus, all three variables are relevant as well as bound at this point, and all three are attributes of S_1. □

Now, let us give the rules whereby tuples are added

to the various relations.

a) Initially, the binding relation for the root goal node has the tuple whose components are the constants appearing in the query.

b) Tuples are passed from supplementary relations to binding relations at goal-node children, as follows. Suppose R is a rule node, and the ith subgoal is $p(X_1, \ldots, X_k)$, where the X's need not be distinct and may be constants. Suppose $X_{i_1}, \ldots, X_{i_m}$ is the sublist of arguments that are bound, either because they are constants or because they are variables appearing as attributes of S_{i-1}; that is, they are bound by the head or a prior subgoal. Then each tuple of S_{i-1} provides a constant for each of $X_{i_1}, \ldots, X_{i_m}$, and these constants are the components of a tuple belonging to the binding relation at the ith goal-node child of rule node R.

c) Tuples are passed from binding relations to zeroth supplementary relations at rule-node children as follows. Let G be a goal node whose goal has predicate p, and let $p(X_1, \ldots, X_k)$ be the head of some rule for p. Then at the rule node R that is the child of G for this rule, S_0 has attributes corresponding to whatever variables among $X_1, \ldots, X_k$ appear in argument positions that are bound for the goal at node G. Each tuple in the binding relation for G provides bindings for the variables that are attributes of S_0 in the obvious way. If a variable appears in two or more bound arguments, then there are multiple bindings provided for that variable, and they must be identical, or this tuple of the binding relation does not produce a tuple for S_0.

d) Tuples of a binding relation for an EDB goal are turned into answer-relation tuples by looking up the EDB relation for tuples that match the bindings.

e) A tuple μ of a supplementary relation S_{i-1} at some rule node R combines with a tuple ν in the answer relation at the ith goal node child of R, to produce a tuple for S_i at R, as follows, provided the ith subgoal is not the last of the rule. First, convert ν, which is a tuple whose components are arguments of the ith subgoal, say $p(X_1, \ldots, X_k)$, into a tuple ν', whose components correspond to each of the variables that appear one or more times among $X_1, \ldots, X_k$. We construct ν' by projecting out of ν the jth component if X_j is either a constant or the same variable as some previous X_l, for $l < j$. Now, we may take the natural join of μ and ν'. Finally, we project out any attributes that correspond to variables that are no longer relevant after the ith subgoal; that is, they appear neither in the head nor in a subsequent subgoal.

f) Under the same circumstances as (e), but with the ith subgoal the last in the rule, we do the same as in (e), but the resulting tuple, say ρ, does not belong in any supplementary relation. the components of ρ correspond exactly to the variables appearing in the head, say $q(Y_1, \ldots, Y_n)$. Each Y_j is either a constant or a variable; in the latter case we substitute for Y_j the component of ρ that corresponds to Y_j. The result is a tuple that is inferred by the rule at R, and it is placed in the answer relation for the goal-node parent of R.

The reader may question whether QRGT represents the "real" or "best possible" notion of top-down logic processing. We claim that it is at least as efficient as Prolog's SLD-resolution algorithm. Since QRGT projects out irrelevant variables, it can be much faster than SLD resolution, even in cases where the latter algorithm converges.[4] There are a number of other algorithms that can be viewed as top-down, and that sometimes combine the work of several nodes of the rule/goal tree into one. These algorithms, are often called "memoing" algorithms, and include Earley deduction (Pereira and Warren [1983], Porter [1986]), and the algorithms of McKay and Shapiro [1981], Lozinskii [1985], Neiman [1986], Van Gelder [1986], Vielle [1987, 1988], Tamaki and Sato [1987], and Dietrich [1987]. Each of these algorithms is dominated by the corresponding magic-set algorithm, either the algorithm of this paper for datalog, or the algorithm of Ramakrishnan [1988] for nondatalog. They have the property that every action [inference of a tuple or pairing of tuples in (e) or (f) above] performed at one or more nodes during QRGT is performed at least once by each of these algorithms. That is sufficient, as we shall see, to show magic-sets to be at least as efficient.

IV. Subgoal Rectification

Now, let us return to the question of why QRGT might be better than magic-sets. The problem in the example of Section II is not hard to spot. The dual use of X in the recursive subgoal $p(X, X, Z)$ causes "alias-

[4] However, Prolog *implementations* usually use a form a tail-recursion optimization that, for certain examples, such as the right-linear version of transitive closure, will avoid rippling answer tuples up the rule/goal tree, and thus can be faster than QRGT. Chapter 15 of Ullman [1989] discusses this issue.

ing" between the first and second arguments of p. At rule nodes for r_2 far down the rule/goal tree, X and Y represent the same thing, so the binding of Y by the subgoal $b(W,Y,Z)$ covertly binds X. The effect is that, even though we couldn't discover it from the rules r_1 and r_2, there are goal nodes for p in the rule/goal tree with binding pattern *bbb* instead of ffb; one of them is $p(4,4,5)$ in Fig. 1, which corresponds to $p(X,X,Z_3)$ in Fig. 3.

The solution is to modify the rules so all IDB subgoals are *rectified*, that is, their arguments are distinct variables. The important property of rectified subgoals is that when we construct the rule/goal tree, all unifications of rule heads with subgoals are trivial, and the rule instance at each rule node can be taken to be the rule itself. Notice that in Fig. 3 we get instances of r_2 like

```
p(X,X,Z2) :- b(Z2,X,Z3) & p(X,X,Z3).
```

that are not equivalent to r_2 itself. If the rule/goal tree does not create special cases of rules, then the binding pattern on each goal node can be shown to be exactly what we would expect using the algorithm in Beeri and Ramakrishnan [1987]. Then, we can see how each step of semi-naive evaluation is mirrored by at least one step of the top-down algorithm.

The following algorithm modifies a datalog program so its subgoals are rectified.

1. Find a nonrectified subgoal, for example,

   $p(X,X,Y,a)$

   which has repeated variables and/or constants as arguments.

2. Create a new predicate symbol q whose arguments correspond to the distinct variables of the subgoal in question, for example, $q(X,Y)$.

3. Replace the subgoal from (1) by the subgoal from (2), wherever it or an equivalent (up to renaming of variables) subgoal appears.

4. For each rule r for p, unify the head of the rule with the subgoal from (1) to get a most general unifier τ. Create a new rule whose head has predicate q and arguments equal to τ applied to each of the variables that were determined in (2) to be arguments of q. The body of the rule is τ applied to the body of r. For example, given the subgoal $p(X,X,Y,a)$ from (1) and the rule

   ```
   p(U,V,b,W) :- s(U,V,W).
   ```

   τ makes X, U, and V equivalent, makes W equivalent to a, and Y equivalent to b. Thus, we would obtain the rule

   ```
   q(X,b) :- s(X,X,a).
   ```

We repeat this transformation until it can no longer be applied. Note that if s is an IDB predicate in the above example, we have created another instance of a nonrectified subgoal. However, we can show that no cycles result; each new predicate has fewer arguments than the one from which it was created. Thus, we claim

Theorem 1: For every safe datalog program $\mathcal{P}_1$ there is a safe datalog program $\mathcal{P}_2$ with rectified subgoals that produces the same relation for every IDB predicate of $\mathcal{P}_1$. Further, top-down evaluation of $\mathcal{P}_2$ by the QRGT algorithm takes no more time than QRGT on $\mathcal{P}_1$.

Proof: The above construction produces $\mathcal{P}_2$ from $\mathcal{P}_1$. The rules of $\mathcal{P}_2$ are each a special case of a rule of $\mathcal{P}_1$, possibly with some predicates renamed. Thus, QRGT on $\mathcal{P}_2$ mimics QRGT on $\mathcal{P}_1$, perhaps with simplifications because variables that are bound to constants at a rule or goal node do not appear explicitly, and arguments that share a variable, and therefore are identical in all tuples, are replaced by a single copy in the rule/goal tree expansion of $\mathcal{P}_2$. □

Example 2: The rules r_1 and r_2,

```
r1:  p(X,Y,W) :- a(X,Y,W).
r2:  p(X,Y,W) :- b(W,Y,Z) & p(X,X,Z).
```

with which we started our discussion, become the rules in Fig. 4, when we replace $p(X,X,Z)$ by $q(X,Z)$. Note that it is unnecessary (and generally impossible) to rectify the EDB subgoals like a. □

```
p(X,Y,W) :- a(X,Y,W).
p(X,Y,W) :- b(W,Y,Z) & q(X,Z).

q(X,W) :- a(X,X,W).
q(X,W) :- b(W,X,Z) & q(X,Z).
```

Fig. 4. Rules with rectified subgoals.

V. Predicate Splitting

Having rectified subgoals, we still may not have a unique binding pattern for each IDB predicate, because recursive calls to a predicate may occur with bindings other than those of the head.

Example 3: Suppose we run the rules of Fig. 4 with the query p^{ffb}. Then the call to q is made with adornment q^{fb}. However, when we apply the recursive rule for q, we get the adornment q^{bb}, because both X and Z appear in a previous subgoal. We must therefore distinguish between these two binding patterns for q, and to do so, we create two predicates q_fb and q_bb. The rules become as shown in Fig. 5. Now, each IDB predicate has a unique adornment for the given query. □

```
p(X,Y,W) :- a(X,Y,W).
p(X,Y,W) :- b(W,Y,Z) & q_fb(X,Z).

q_fb(X,W) :- a(X,X,W).
q_fb(X,W) :- b(W,X,Z) & q_bb(X,Z).

q_bb(X,W) :- a(X,X,W).
q_bb(X,W) :- b(W,X,Z) & q_bb(X,Z).
```

Fig. 5. Rectified subgoals and split predicates.

The technique of splitting predicates was used by Beeri and Ramakrishnan [1987] in their magic sets construction. The basic idea is to construct a rule/goal graph (Ullman [1985, 1989]) containing the query goal with its adornment.

Theorem 2: Given a safe datalog program $\mathcal{P}_1$ and a query goal Q, we can find $\mathcal{P}_2$, a safe datalog program with rectified subgoals and the *unique binding property*, that is, every IDB predicate p has a binding pattern α such that every goal node for p in the rule/goal tree expansion with root Q has exactly the bound arguments that α says are bound. Furthermore, the running time of QRGT on $\mathcal{P}_2$ is at worst proportional to the running time of the same algorithm on $\mathcal{P}_1$.

Proof: We apply the subgoal rectification and predicate-splitting constructions sketched above. We must show that predicate splitting preserves safety and subgoal rectification, both of which are easy. Then, an induction on the number of steps performed in a breadth-first expansion of the rule/goal tree shows that the binding at every goal node is what we would expect it to be from analyzing the rules.

We saw in Theorem 1 that the running time of QRGT does not degrade when we rectify subgoals. It is easy to see that predicate splitting does not affect the running time at all, since the only changes are in the names of the predicates. □

VI. The Generalized Supplementary Magic-Sets Algorithm

We shall now review the generalized supplementary magic-set construction of Beeri and Ramakrishnan [1987]. Given a set of datalog rules with the unique binding property, which we suppose are numbered $r_1, \ldots, r_n$, we introduce several new predicates.

1. For every IDB predicate p there is a *magic predicate* m_p. The arguments of m_p correspond to the bound arguments of p, according to the unique binding pattern for p. Intuitively, $m\_p(a_1, \ldots, a_k)$ is true exactly when $(a_1, \ldots, a_k)$ is a tuple in the binding relation at some goal node with predicate p in the rule/goal tree constructed for a given query.
2. For every rule r_i having k subgoals, there are *supplementary predicates* $sup_{i,0}, \ldots, sup_{i,k-1}$. The arguments of $sup_{i,j}$ are the variables of rule r_i that are bound and relevant after the first j subgoals of r_i. Intuitively, $sup_{i,j}(a_1, \ldots, a_m)$ is true exactly when tuple $(a_1, \ldots, a_m)$ appears in the jth supplementary relation at some rule node for rule r_i.

The given rules and query are rewritten as follows. The new rules closely mirror the way tuples are passed from node to node in the QRGT algorithm of Section III. Of the six tuple-construction rules (a) through (f), only (d), which involves EDB lookup, does not yield corresponding magic-set rules. Thus, we shall give the rule construction algorithm in a manner that parallels the QRGT algorithm.

a) If the query involves predicate p and has constants $a_1, \ldots, a_k$ for the bound arguments of p, then we have rule

$$m\_p(a_1, \ldots, a_k).$$

b) Suppose $p(X_1, \ldots, X_k)$ is the jth subgoal of rule r_i, and p is an IDB predicate. Let $Y_1, \ldots, Y_n$ be the variables of r_i that are arguments of $sup_{i,j-1}$; that is, these are the bound and relevant variables of r_i prior to the jth subgoal. Finally, let $i_1, \ldots, i_t$ be the bound arguments of p. Then we have rule

$$m\_p(X_{i_1}, \ldots, X_{i_t}) \text{ :- } sup_{i,j-1}(Y_1, \ldots, Y_n).$$

to represent the passing of bindings to goal node children. Note, however, that in the magic-sets algorithm, we only pass to IDB goals.

c) Suppose $p(X_1, \ldots, X_k)$ is the head of rule r_i, and $i_1, \ldots, i_t$ are the bound arguments of p. Let $Y_1, \ldots, Y_m$ be the variables of r_i that appear among $X_{i_1}, \ldots, X_{i_t}$, in the order that they appear as arguments of $sup_{i,0}$. Then we have rule

$$sup_{i.0}(Y_1, \ldots, Y_m) \text{ :- } m\_p(X_{i_1}, \ldots, X_{i_t}).$$

to reflect the passing of bindings from binding relations to zeroth supplementary relations.

d) As mentioned, there are no rules that correspond to part (d) of the QRGT algorithm (EDB lookup).

e) Suppose $p(X_1, \ldots, X_k)$ is the jth subgoal of r_i, and is not the last subgoal of r_i. Let $Y_1, \ldots, Y_m$ be the variables bound and relevant just prior to the jth subgoal of r_i and let $Z_1, \ldots, Z_n$ be the variables bound and relevant just after; each list is in the order of appearance as arguments of their respective supplementary predicates. Then we have rule

$$sup_{i.j}(Z_1, \ldots, Z_n) \text{ :- } sup_{i.j-1}(Y_1, \ldots, Y_m) \ \& \ p(X_1, \ldots, X_k).$$

to reflect joining the $(j-1)$st supplementary relation with the jth subgoal to obtain tuples of the jth supplementary relation.

f) Suppose all is as in (e), but j is the last subgoal of r_i, and $q(Z_1, \ldots, Z_n)$ is the head of r_i. The we have the rule

$$q(Z_1, \ldots, Z_n) \text{ :- } sup_{i.j-1}(Y_1, \ldots, Y_m) \ \& \ p(X_1, \ldots, X_k).$$

to reflect the joining of the last supplementary predicate and the last subgoal to get answer tuples.

Incidentally, the reader may wonder what happened to the database lookup operations from (d) in the QRGT algorithm of Section III. Notice that an EDB subgoal can only be the second subgoal of a rule of type (e) or (f); the first subgoal in each such rule has a supplementary predicate. If we perform semi-naive evaluation of the magic rules, these supplementary predicates will occasionally get new tuples, and on the next round, a lookup will occur to find the matching tuples of the EDB subgoal. In effect, type (d) operations in QRGT are combined with a type (b) operation, where tuples of a supplementary predicate are converted to binding tuples. The difference does not affect the order-of-magnitude running time of either algorithm.

Example 4: Let us apply the above construction to the rules of Fig. 5. To simplify the notation, we shall use q for q_fb and s for q_bb. The query, as in our running example, is p^{ffb}. Thus, m_p has one argument, corresponding to the third argument of p; m_q has one argument corresponding to the second of q, and m_s has two arguments, since the unique binding pattern for s is bb. The resulting rules, grouped according to the five parts, (a), (b), (c), (e), and (f), above, are shown in

```
m_p(1).

m_q(Z) :- sup2.1(W,Y,Z).
m_s(X,Z) :- sup4.1(W,X,Z).
m_s(X,Z) :- sup6.1(W,X,Z).

sup1.0(W) :- m_p(W).
sup2.0(W) :- m_p(W).
sup3.0(W) :- m_q(W).
sup4.0(W) :- m_q(W).
sup5.0(X,W) :- m_s(X,W).
sup6.0(X,W) :- m_s(X,W).

sup2.1(W,Y,Z) :- sup2.0(W) & b(W,Y,Z).
sup4.1(W,X,Z) :- sup4.0(W) & b(W,X,Z).
sup6.1(W,X,Z) :- sup6.0(X,W) & b(W,X,Z).

p(X,Y,W) :- sup1.0(W) & a(X,Y,W).
p(X,Y,W) :- sup2.1(W,Y,Z) & q(X,Z).
q(X,W) :- sup3.0(W) & a(X,X,W).
q(X,W) :- sup4.1(W,X,Z) & s(X,Z).
s(X,W) :- sup5.0(X,W) & a(X,X,W).
s(X,W) :- sup6.1(W,X,Z) & s(X,Z).
```

Fig. 6. Magic rules.

Fig. 6. □

Note that the rules of Fig. 6 can be simplified. For example, we could replace each predicate $sup_{i.0}$, for $1 \leq i \leq 6$, by an equivalent magic predicate, and then eliminate the third group of rules.

VII. A Performance Guarantee for Magic Sets

We wish to compare semi-naive evaluation and QRGT on a given query and logic program. Recall that the *semi-naive* logic evaluation is a bottom-up algorithm that computes the least fixed point of logical rules with respect to a given database. On the first pass, we evaluate the heads of those rules that have only EDB subgoals, to infer some tuples for the head predicate. On subsequent passes, we evaluate the bodies of all rules, but we require that at least one of the IDB subgoals use a tuple that was newly inferred on the previous pass. That restriction guarantees that when we match tuples from the relations of the various subgoals, we never match the same pair twice (although we may infer the same tuple several times by using different rules and/or different tuples).

When the rules are constructed by the algorithm of Section VI, semi-naive evaluation takes a simple form. First, note that just one rule has only EDB subgoals — the rule generated by (a) from the query itself; this rule has no subgoals at all. All other generated rules

have a supplementary or magic predicate, which are IDB predicates, in their bodies. This observation is important, since if there were nontrivial rules with only EDB subgoals, the semi-naive algorithm would copy, or even join, whole database relations on the first pass.

To compare the running times of semi-naive and QRGT, we identify *events*, which are of two types:

1. A tuple is inserted into some relation.
2. Tuples from a supplementary relation and a subgoal are found to match in the body of a magic rule (for semi-naive evaluation) or at a rule node (for QRGT).

We assume suitable index structures make each of these events take $O(1)$ time,[5] and make it possible to neglect other costs of the two algorithms. Specifically, we do not waste significant time looking for matches that don't exist.

Theorem 3: For every safe datalog program $\mathcal{P}_1$ and query Q there is a logic program $\mathcal{P}_2$ that produces the same answer on Q and that takes time, when evaluated by the semi-naive algorithm, that is no more than proportional to the time taken by QRGT on $\mathcal{P}_1$.

Proof: We construct $\mathcal{P}_2$ by applying to $\mathcal{P}_1$, in order,

1. Subgoal rectification,
2. Predicate splitting, and
3. The generalized supplementary magic sets transformation.

Let $\mathcal{P}_3$ be the result of steps (1) and (2) only. By Theorems 1 and 2 it suffices to compare the running times of semi-naive on $\mathcal{P}_2$ with QRGT on $\mathcal{P}_3$. The proof is an identification of each event of semi-naive evaluation with a distinct event of QRGT, and proceeds by induction on the number of rounds of the bottom-up semi-naive evaluation algorithm that we have performed.

The details of the proof are found in Ullman [1989], and here we shall give only the important observations. The most difficult part occurs when, during semi-naive evaluation, we apply a rule involving supplementary predicates, for example,

$$\mathrm{sup}_{r,i+1}(\mathtt{X,Z,U}) \ \mathtt{:-}\ \mathrm{sup}_{r,i}(\mathtt{X,Y,Z})\ \&\ \mathtt{p(X,Y,U,V).}$$

and have a type-2 event where we pair a tuple μ from $sup_{r,i}$ with a tuple ν from p. By the inductive hypothesis, we can identify a rule node R for rule r, whose ith supplementary relation has tuple μ. Note that the rectified-subgoal property is essential to guarantee that the instance of rule r at this node is r itself, rather than some substitution on r, in which case the arguments of the supplementary relations at the node might not even correspond to the arguments of $sup_{r,i}$. We need the following lemma, which is easily proved by induction on the number of inferences (e.g., by semi-naive evaluation) that it takes to infer $p(\nu)$ from the database, using the given rules.

Lemma: If ν satisfies p and matches some binding for p provided to a particular goal node G for p during QRGT, then the answer relation for G will include ν. □

Then, we note that the presence of μ in the ith supplementary relation at R causes G, the child of R corresponding to the p-subgoal in question, to receive a binding that matches ν. The reason is that μ and ν agree where both are defined, or else they would not have resulted in a type-2 event during semi-naive evaluation. Hence, G returns answer ν to R, and μ and ν are matched at R. □

In fact, the same argument proves that the program $\mathcal{P}_2$ evaluates the query as fast, using semi-naive evaluation, than any algorithm **A** that performs the same set of events as QRGT does, even if **A** consolidates the same event executed at different nodes into one event. We claim that all the algorithms mentioned at the end of Section III, at least when restricted to datalog programs, are in this class.

[5] $O(\log n)$, where n is the size of the database, would be more realistic, but the same factor would be lost by either algorithm, so there is no bias inherent in this assumption.

References

Beeri, C. and R. Ramakrishnan [1987]. "On the power of magic," *Proc. Sixth ACM Symposium on Principles of Database Systems*, pp. 269–283.

Dietrich, S. W. [1987]. "Extension tables: memo relations in logic programming," *Proc. IEEE Symp. on Logic Programming*, pp. 264–272.

Lozinskii, E. L. [1985]. "Evaluating queries in a deductive database by generating," *Proc. Ninth ICJAI*, pp. 173–177.

McKay, D. and S. Shapiro [1981]. "Using active connection graphs to reason with recursive rules," *Proc. Seventh IJCAI*, pp. 368–374.

Neiman, V. S. [1986]. "Deduction search with single consideration of subgoals," *Dokl. Akad. Nauk SSSR*, pp. 251–254.

Pereira, F. C. N. and D. H. D. Warren [1983]. "Parsing as deduction," *Proc. Twenty-first Annl. Meeting of the*

Assn. for Computational Linguistics, pp. 137–144.

Porter, H. H. III [1986]. "Earley deduction," TR CS/E-86-002, Oregon Grad. Center, Beaverton, OR.

Ramakrishnan, R. [1988]. "Magic templates: a spell-binding approach to logic programs," *Proc. Fifth Intl. Symp. on Logic Programming*, pp. 140–159.

Seki, H. [1988]. "On the power of continuation passing, Part I: an analysis of recursive query processing methods," unpublished manuscript, ICOT, Tokyo.

Tamaki H. and T. Sato [1986]. "OLD resolution with tabulation," *Proc. Third Intl. Conf. on Logic Programming*, pp. 84–98.

Ullman, J. D. [1985]. "Implementation of logical query languages for databases," *ACM Trans. on Database Systems* **10**:3, pp. 289–321.

Ullman, J. D. [1988]. *Principles of Database and Knowledge-Base Systems*, Volume I, Computer Science Press, Rockville, Md.

Ullman, J. D. [1989]. *Principles of Database and Knowledge-Base Systems*, Volume II, Computer Science Press, Rockville, Md.

Van Gelder, A. [1986]. "A message-passing framework for logical query evaluation," *ACM SIGMOD International Conf. on Management of Data*, pp. 155–165.

Vielle, L. [1987]. "Recursive axioms in deductive databases: the query/subquery approach," in Kerschberg, L. (ed.) *Expert Database Systems*, Benjamin-Cummings, Menlo Park, CA, pp. 253–268.

Vielle, L. [1988]. "From QSQ towards QoSaQ: global optimization of recursive queries," *Proc. Second Intl. Conf. on Expert Database Systems.*