On the Power of Magic

Catriel Beeri [1]
Raghu Ramakrishnan [2]

1: The Hebrew University
2: University of Wisconsin-Madison

ABSTRACT

This paper considers the efficient evaluation of recursive queries expressed using Horn Clauses. We define *sideways information passing* formally and show how a query evaluation algorithm may be defined in terms of sideways information passing and control. We then consider a class of information passing strategies that suffices to describe most query evaluation algorithms in the database literature, and show that these strategies may always be implemented by rewriting a given program and evaluating the rewritten program bottom-up. We describe in detail several algorithms for rewriting a program. These algorithms generalize the Counting and Magic Sets algorithms to work with arbitrary programs. Safety and optimality of the algorithms are also considered.

1. Introduction

The evaluation of recursive queries expressed as sets of Horn Clauses over a database has recently received much attention. Consider the following program:

$anc(X, Y) :- par(X, Y)$
$anc(X, Y) :- par(X, Z), anc(Z, Y)$

and let the query be

$Query : anc(john, Y)\ ?$

Assume that a database contains a parenthood relation *par*. Then the program defines a derived relation describing ancestors, and the query asks for the ancestors of *john*. A well known strategy for evaluating logic programs is the *bottom−up* strategy. It serves to define the least fixed point semantics, and is known to be complete [Lloyd 84]. While the strategy is reasonably efficient when the query does not contain instantiated variables [Bancilhon 85], this example shows that it is very inefficient when bindings for some variables are given in the query. The reason is that it computes the complete *anc* relation and then applies selection to it. Thus, all ancestors are computed, even though only the ancestors of john are needed. A

(1) Department of Computer Science, The Hebrew University, Jerusalem, Israel. Work partially supported by the USA-Israel BSF Grant # 85-00082/1. Part of the work was done while visiting MCC.
(2) This work was done while at the University of Texas at Austin and visiting MCC. It was partially supported by grants from MCC and an IBM Graduate Fellowship.

top-down strategy (as used, for example, by Prolog), may do much better by computing only the ancestors of john. The first rule computes the nodes reachable from *john* in one step. Then the second rule generates the same query for these nodes, and the first rule is used again to find the nodes reachable in two steps, and so on.

There are two modes of *information passing* in the evaluation of a query. The first is unification. Given a constant in a goal, unification with a rule head will cause some of the variables in the head to be bound to that constant. These bindings are valid in the rule's body as well. (This is normally seen as part of top-down evaluation.) The second mode is *sideways information passing*. Given bindings for some variables of a predicate, we can solve the predicate with these bindings and thus obtain bindings for some of its other variables. These new bindings can be "passed" to another predicate in the same rule to restrict the computation for that predicate. In the example above, in the top-down evaluation, unification with the query binds *X* in the second rule to *john*; these bindings are passed from the rule's head to the base predicate *par*. The values obtained by evaluating *par* with these bindings are then passed to *anc*, to generate yet another subgoal.

Several strategies have been proposed for evaluating recursive queries expressed using sets of Horn Clauses (rules). ([Henschen and Naqvi 84], [Kifer and Lozinskii 85], [Lozinskii 85], [McKay and Shapiro 81], [Rohmer and Lescoeur 85], [Sacca and Zaniolo 86a], [Van Gelder 86], [Vieille 86], etc. See [Bancilhon and Ramakrishnan 86] for a comprehensive survey.) The main thrust of the above strategies is to improve efficiency by restricting the computation to tuples that are related to the query. They all use information passing in some form, but they also use other ideas, intended, for example, to avoid repeatedly computing the same fact, using the same derivation, several times. However, so far there is no uniform framework in terms of which these strategies may be described and compared, and the basic ideas that are common to these strategies remain unclear.

It is our thesis that each of these strategies has two distinct components - a *sideways information passing strategy* (henceforth abbreviated to *sip*) for each rule (or even several such strategies per rule) and a control strategy. A sip represents a decision on how information gained about tuples in some predicates in a rule is to be used in evaluating other predicates in the rule. The control strategy implements this decision, possibly using additional optimization techniques. Thus, a given sip collection may be implemented by several control strategies, and a given control strategy may be used to implement distinct sip collections. In particular, we show that simple bottom-up evaluation may be used to implement a wide class of sip collections by first rewriting the given set of Horn Clauses and then evaluating the rewritten set.

To what extent is it justified to claim that the collection of sips and the control strategy are distinct, possibly independent, components of a query evaluation strategy? The research reported here can be seen as a step towards answering this question. We provide a formal definition of a sip. Then we present several program transformation strategies that can be applied to an arbitrary (Horn Clause) program and a query to produce a program that is equivalent to the given program with respect to the query, and that uses the bindings in the query to direct the computation, and hence is usually more efficient. Each of these transformations uses a collection of sips that are attached to the rules of the given program. In a sense, the sip collection serves as a definition of what facts are *relevant* to the given query. Each transformation produces a program that computes only these facts. The fact that the transformed programs use information passing, yet can be computed bottom-up, shows that there is no inherent relationship of information passing to top-down evaluation, thus supporting the claim that sips and control are independent. The transformations are generalizations of strategies that have been proposed in earlier work, namely the *Magic Sets* and the *Counting* strategies [Bancilhon et al. 86], but which, as presented there, were of quite limited applicability. We note that the notion of a sideway information passing graph has been introduced previously, although it is less general than our definition [Kifer and Lozinskii 86]. The work reported in [Van Gelder 86] can also be viewed as supporting our claim that information passing and control are to a large extent independent, although this claim is not explicit there.

The paper is organized as follows. Notation and definitions are introduced in the rest of this section. We define sips in Section 2, and describe how a query and a collection of sips are used to obtain an *adorned* program from a given program in Section 3. The transformation of a program into an adorned program is the first step in each of the strategies presented here. In Sections 4 through 7, we present a number of algorithms for rewriting the adorned program, making further use of the sip collection, into an equivalent program whose bottom-up evaluation implements the sip collection. We describe an important optimization of some of these algorithms in Section 8. We discuss optimality and safety in subsequent sections, before concluding with a discussion of the various methods presented in earlier sections. Readers who are not familiar with previous versions of the methods described in this paper may prefer to omit Sections 6-8 in the first reading.

1.1. Basic definitions and notation

A Horn Clause, or *rule*, has the form

$$p_0(\bar{X}_0) \;:\!-\; p_1(\bar{X}_1), \cdots, p_n(\bar{X}_n)$$

where each p_i is a predicate name, and each $\bar{X}_i$ is a vector of terms involving variables and/or constants. The :– symbol stands for 'implies', the commas stand for 'and', the part to the left of :– is the *head* of the rule, and the part to the right is its *body*. We assume that rules satisfy the following *well formedness* condition:

(WF) Each variable that appears in the head of the rule, also appears in its body.

A rule with an empty body is a *fact*. By (WF), a fact contains no variables (it is ground). A *query* is a rule without a head. We will assume here that a query is a single predicate occurrence, with some of its variables possibly bound to constants. Our methods generalize to multiple-predicate queries.

We use the following conventions. A *predicate occurrence* is a predicate name followed by a list of arguments, in the right hand side (body) of a rule. Where no confusion is possible, we simply use ‘predicate’ for ‘predicate occurrence’. An *argument* is a *term*, where a term is a constant, a variable or an n-ary *function symbol* followed by n terms. We use upper case letters for variable names, lower case letters for predicate names, and numerals or lower case letters for constants. A query is written as $q(\ \cdots\ )$? The notation $q(\overline{c}, \overline{X})$? is used to denote the fact that some of the positions in the query are bound to constants, while others contain variables. No specific ordering is implied.

A *program* is a finite set of rules, $P = \{ r_1, ..., r_n \}$. A database D is a finite set of finite relations. A tuple t in a relation p can be viewed as a fact $p(t)$, and conversely, a fact can be represented as a tuple of a relation. Hence $P \cup D$ is a Logic Program. Without loss of generality, we will assume that P contains no facts - all facts are part of the database D. Predicates that name database relations are called *base* predicates; all other predicates are called *derived*. Recursion exists when derived predicates in a program depend on each other as well as on base predicates. A program can be transformed so that it contains no base predicate as a head of a rule, and we will so assume.

Given a program-query pair $(P,\ Q = q(\overline{c}, \overline{X}))$ and a database D, the result of applying $(P,\ Q)$ to D, which we also refer to as the *answer* for the query on D, is the set of bindings to the vector of variables X that make the query expression true with respect to the program and the given database. The problem that we discuss in this paper is how to evaluate efficiently the answer of a query against a database.

Note that although our approach is somewhat different from that of the Logic Programming community, the basic theory is still applicable. Since $P \cup D$ is a program in the Logic Programming sense, basic results like the definition of semantics in terms of least fixed points, that is the completeness of bottom-up evaluation, hold. The separation of program from database allows us, however, to consider equivalence of programs with respect to all possible databases. Two programs with queries $(P,\ Q)$ and $(P‘,\ Q‘)$ are *equivalent*, if for every database D, $P \cup D$ and $P‘ \cup D$ produce the same answers for their respective queries. In other words, we can view a program with a query as defining a function from databases to finite relations. Two programs with queries are equivalent if they define the same function.

By the completeness of the bottom-up approach, the answer for a query can be computed as follows: We start with the database relations, and with empty derived predicates. The values for the derived predicates are computed in stages. In each stage, we add to each derived predicate all the tuples whose membership in it is implied by the program, given the values for the predicates in the previous stage. The sequence of values of the derived predicates is monotonically increasing, and its limit is the final values for these predicates. The answer is obtained by applying the appropriate selection to the query’s predicate. Notice that the sequence is infinite, and when the program contains function symbols, the limit may differ from the union of any finite prefix of the sequence. However, each fact that belongs to the limit is obtained after a finite number of iterations.

It follows from this discussion that for each fact that belongs to a derived predicate, there exists a finite *derivation tree*, that describes how it is derived from base facts using rules of the program. Let $p(c)$ be a fact in the derived predicate p. Then the tree has $p(c)$ at its root, the leaves are base facts, and each internal node is labeled by a fact, and by a rule that generates this fact from the facts labeling its children. A base fact may be viewed as a derivation tree of height one.

We will assume one more restriction on the from of rules. In a given rule, two variables are called *connected* if they have occurrences in the same predicate. This is extended in the obvious way to connection through a chain of variables, where each adjacent pair shares a predicate. Similarly, two predicates are connected if they each contain one of a pair of connected variables.

Connectivity is an equivalence relation (both on variables and predicates). The set of predicate occurrences in a rule is therefore the union of connected components. One of these contains the rule's head. Other components, if they exist, are actually existential subqueries, that are solved independently of any bindings for the rule's head variables. Information passing strategies are limited in scope to a connected component. Since we are primarily interested in information passing, starting at the head, we assume, without loss of generality:

(C) The predicate occurrences of a rule form a single connected component.

2. Sideways Information Passing

A *sideways information passing strategy*, henceforth referred to as a *sip*, is an inherent component of any query evaluation strategy. Informally, for a rule of a program, a sip represents a decision about the order in which the predicates of the rule will be evaluated, and how values for variables are passed from predicates to other predicates during evaluation. Sip strategies for the various rules of a program are not enough to specify an evaluation strategy. A control component that specifies, for example, whether to obtain all solutions for a goal when it is first called, or whether to obtain them one at a time (as for example in Prolog) is required. Control is a separate, often independent, component; here, we only discuss sips.

Intuitively, a sip describes how bindings passed to a rule's head by unification are used to evaluate the predicates in the rule's body. Thus, a sip describes how we evaluate a rule when a given set of head arguments are bound to constants. Consider, for example, the ancestor query presented in Section 1. The first argument is bound to *john*, and by unification, the variable X in the second rule is bound to *john*. We can evaluate *par* using this binding, and obtain a set of bindings for Z. These are passed to *anc* to generate subgoals, which in this case have the same binding pattern. This is in fact the way in which a top-down strategy like Prolog would compute this query.

Generalizing from this example, we may say that the basic step of sideways information passing is to evaluate a set of predicates (possibly with some arguments bound to constants), and to use the results to bind variables appearing in another predicate. The order in which predicates are solved and the bindings are passed is determined as a consequence of the control strategy in top-down methods. We try to separate this order from the flow of control, leading to the definition of a sip as a labeled graph, below.

Let r be a rule, with head predicate $p(\theta)$, and let θ^b denote a subset of θ, that we assume will be bound in invocations of the rule. Let p_h be a special predicate, denoting the head predicate restricted to its bound arguments. Thus, the arguments of p_h are θ^b. (If no bindings are given, that is, θ^b is empty, then p_h is a 0-ary predicate, the constant FALSE. In such a case, we may consider it not to exist at all, as far as the following discussion is concerned.) If a predicate appears in r's body more then once, we number its occurrences, starting from 0. (The numbering is just to identify the positions in the rule. It is irrelevant when unification with heads of other rules is considered.) Let $P(r)$ denote the set of predicate occurrences in the body. A sip for r is a labeled graph that satisfies the following conditions:

1. Each node is either a subset or a member of $P(r) \cup \{p_h\}$.

2. Each arc is of the form $N \rightarrow q$, with label χ, where N is a subset of $P(r) \cup \{p_h\}$, q is a member of $P(r)$, and χ is a set of variables, such that

(i) Each variable of χ appears in N.

(ii) Each member of N is connected to a variable in χ.

(iii) For some argument of q, all its variables appear in χ. Further, each variable of χ appears in an argument of q that satisfies this condition.

These two conditions define the nature of nodes and arcs of a sip. They are explained below. The following condition provides a consistency restriction on a sip. For a graph with nodes and arcs as above, define a precedence relation on the members of $P(r) \cup \{p_h\}$ as follows:

(i) p_h precedes all members of $P(r)$.

(ii) A predicate that does not appear in the graph, follows every predicate that appears in it.

(iii) If $N \rightarrow q$ is an arc, and $q' \in N$, then q' precedes q.

We can now state the last condition defining a sip:

3. The precedence relation defined by the sip is acyclic, that is, its transitive closure is a partial order.

An alternative and equivalent formulation is:

3'. There exists a total ordering of the predicates of $P(r) \cup \{p_h\}$ in which p_h is the first, such that for each arc, all predicates at its tail precede the predicate at its head, and such that the predicates that do not appear in the sip follow all others.

We explain the meaning of such a graph, by first explaining how the computation of a rule uses one arc, then dealing with the complete computation of the rule. Assume we want to use the rule r, with some arguments of the head predicate bound to constants. The special node p_h may be thought of as a base relation with positions corresponding to the bound arguments of the head predicate. Each tuple in it corresponds to the vector of bindings that is given for these arguments. (Intuitively, for those familiar with Prolog, each tuple contains the vector of constants for the bound arguments in some call of this adorned predicate.) An arc labeled χ from a set of predicates N to a predicate q means that by evaluating the join of the predicates in N (with some arguments possibly bound to constants), values for the variables in χ are obtained, and these values are passed to the predicate q, and used to restrict its computation. Thus, for each such arc, the variables in its label must be bound when the goals corresponding to the predicates in N are solved, and any control strategy that implements the sip must ensure this. This explains condition 2(i).

As stated above, we separate the issue of control from the sip. Thus, we allow predicates to be computed (for given bindings) all at once, or in stages. We can imagine a box associated with each predicate in which its tuples are collected. For an arc $N \rightarrow q$, with label χ, attach a filter that performs a join of the tuples in the boxes of the predicates of N, and for each qualifying tuple, its projection on χ is sent along the arc. (Note that if arguments are complex terms with function symbols, then the arguments are evaluated, and these are converted into values for the variables before the join is performed. See [Ullman 85] for details.) A predicate in N that is not connected to a variable of χ does not serve any useful role in the join; such predicates are excluded by condition 2(ii).

In general, there may exist several arcs entering a predicate q. The tuples arriving along these arcs are joined , and the resulting tuples passed as bindings for the evaluation of q. A binding for q is useful, however, only if it is a binding for an argument of q. That is, the methods described in this paper all treat an argument as free, if one or more of its variables are not bound, even if some other variables in it are bound. (In this decision, we follow [Ullman 85].) This explains condition 2(iii).

The evaluation of a rule proceeds as follows. Each node with no arcs entering it is evaluated with all arguments free. (An exception is the special predicate p_h; it is treated as a base predicate and the tuples in it are those supplied for θ^b by unification.) A predicate with arcs entering it is evaluated only for values supplied through arcs. Finally, when all predicates have been evaluated, they are joined, and the result is projected on the variables of the head predicate p, to be returned as a result. (This join, like the evaluation of the predicates, can be performed in stages, as the tuples are generated, or all at once, when all tuples become available.) The third condition ensures that the sip denotes a consistent strategy for passing information through the predicates in the body. Thus, we disallow sips according to which two goals make a cyclic assumption about a variable being bound, that is, each assumes that the variable is bound by the other.

We emphasize that the above discussion of the interpretation of a sip is to be understood as an abstraction that conveys *what* is done rather than *how* it is done. For example, Prolog does not explicitly create special predicates p_h to store bound head arguments, nor does it explicitly evaluate the joins we mentioned. These operations are, however, implicit in the way Prolog maintains variable bindings through unification and backtracking.

Example 1: Consider the following rules:

sg(X,Y) :- flat(X,Y)
sg(X,Y) :- up(X,Z1), sg.1(Z1,Z2), flat(Z2,Z3), sg.2(Z3,Z4), down(Z4,Y)
Query: sg(john,?)

This is a non-linear version of the same-generation program [Bancilhon et al 86]. We have numbered the *sg* occurrences in the second rule for convenience. The earlier versions of the Magic Sets and Counting algorithms fail to rewrite the above rules.

Given the query, the natural way to use the second rule seems to be to solve the predicates in the indicated order, using bindings from each predicate to solve the next predicate. This information passing strategy may be represented by the following sip:

$\{sg_h\} \rightarrow_X up\ ; \quad \{sg_h,\ up\} \rightarrow_{Z1} sg.1$ **(I)**

$\{sg_h,\ up,\ sg.1\} \rightarrow_{Z2} flat\ ;$

$\{sg_h,\ up,\ sg.1\ ,\ flat\} \rightarrow_{Z3} sg.2$

Observe that in this sip, each set at the tail of an arc contains also predicates appearing in previous sets. Another sip that seems to represent the same order of information passing steps is:

$\{sg_h\} \rightarrow_X up\ ; \qquad \{up\} \rightarrow_{Z1} sg.1$ **(II)**

$\{sg.1\} \rightarrow_{Z2} flat\ ; \qquad \{flat\} \rightarrow_{Z3} sg.2$

[]

The difference between the two sips is that in the first, as we proceed from left to right, we carry along the bindings for all variables that have been computed so far. (This is implied by the contents of the arcs' tails and labels.) In the second, "past" information is not used. For the example as given, this does not seem to matter. The difference can be illustrated as follows. Assume that in the example a variable W is added to the predicates *up* and *flat*. The first sip can be changed by adding W to the label of the arc entering *flat*. This is possible since a predicate containing this variable appears at the tail of the arc. For the second sip we can not make such a change to the arc label, unless we add the predicate *up* to the tail.† If the second sip is not changed, we can actually compute *flat* with $Z2$ bound, and obtain some W values that are not compatible with the W values produced by the evaluation of *up*. The incompatible values will be dropped only by the final join of all body predicates.

Intuitively, it seems that methods that use all the available information are more efficient. For our purposes here, we have found it desirable to provide a definition that is as general as possible, so as to include as many strategies as possible in our framework.

To allow for the convenient representation of sips in which all previous binding information is used, we introduce *compressed* arcs, denoted ->->. Such an arc implies that the set at its tail consists of all predicates that appear there and also of all their predecessors in the partial order defined by the sip. However, the set N must still satisfy conditions 2(i, ii).

Example 1 (continued): the sip (I) can be represented in compressed form as follows:

$\{sg_h\} \rightarrow\rightarrow_X up\ ; \qquad \{up\} \rightarrow\rightarrow_{Z1} sg.1$ **(III)**

---

† If we choose to always pass all available information, then we can omit the labels altogether, since they can be deduced from the heads and tails of the arcs. Then, the change in the sets of attributes would not change the first sip at all.

$\{sg.1\} \rightarrow\rightarrow_{Z2}$ *flat* ; $\{flat\} \rightarrow\rightarrow_{Z3} sg.2$

[]

A sip in which all arcs are compressed is called *compressed*. We can either view such a sip as an abbreviation, or we may view it as a sip for which the semantics of evaluation includes remembering past information.

So far, we had arcs entering both base and derived predicates. The information passed along an arc is indeed important for both types, but for somewhat different reasons. A base predicate is always directly evaluable. Binding information is used as a selection condition (which may have a considerable influence on the method, as well as on the size of the result). Bindings passed to derived predicates influence the computation by restricting the subqueries that are generated. In this paper we are interested in binding propagation and how it can be used to improve the efficiency of evaluation in the presence of recursion. Our transformations make no use of bindings passed to base predicates. We therefore generalize our notation to allow more succint representation of sips, in which only arcs entering derived predicates are represented.

Instead of a set at the tail of an arc, we now allow an ordered pair of sets; the second set contains only base predicates. We use the notation $N_1; N_2$ --> q.The meaning is that the predicates in N_1 are evaluated and joined. This provides bindings used in the evaluation of the base predicates in N_2. (The variables that are bound can be deduced by looking at the variable sets of N_1 and N_2.) Then the join of $N_1 \cup N_2$ is used to produce bindings passed along the arc.

Example 1 (continued): The sip (I) may now be written as follows

$\{sg_h; up\} \rightarrow_{Z1} sg.1$ **(IV)**

$\{sg_h, up, sg.1; flat\} \rightarrow_{Z3} sg.2$

Note that replacing the semicolon with a comma in the first arc would produce a different sip, in which *up* is evaluated with no bindings. The sip (II) can be written now as follows:

$\{sg_h; up\} \rightarrow_{Z1} sg.1$ **(V)**

$\{sg.1; flat\} \rightarrow_{Z3} sg.2$

[]

In the sequel, when compressed arcs are used, preceding predicates (of predicates in the tail of a sip arc, according to the ordering induced by the sip) are to be added to the first component of the tail.

2.1. Partial Sips

Our definitions open the way to consider relationships between sips, in particular, when can one sip be considered to be "better" than another? As a special case, we can distinguish between *full* and *partial* sips. A *partial sip* is a sip that does not always propagate all available information. A sip G is contained in a sip G' if for each arc $N \rightarrow q$ is G, with label χ, there exists an arc $N' \rightarrow q$ in G', with label χ', such that $N \subseteq N'$ and $\chi \subseteq \chi'$. The containment is proper if at least one tail, or one label of G is properly contained in the corresponding tail or label of G' or if G' contains arcs that do not exist in G. A sip is *partial* if it is properly contained in another sip. We note the following special cases of partial sips:

1. Not all predicates appear as heads in sip arcs. (Recall that each predicate is connected to the head by a chain of variables, so in principle, it is always possible to pass information to it.)

2. An arc label does not contain all variables that appear in tail predicates and that can cover arguments of the head. (This implies that there is a free argument in the head that could have been bound using goals that were solved before it.)

3. The sip is not compressed. (Otherwise, we do not use the smallest possible set of bindings for the bound arguments.)

Example 2: Consider the sip for the second rule in the previous example. It is a full sip, but it becomes partial if we modify it as follows:

$\{sg^b\ ; up\} \rightarrow_{Z1} sg.1$

$\{flat\} \rightarrow_{Z3} sg.2$

The tail of the second arc is a base predicate. This arc does not depend on the bindings for the head predicate. It uses values from the base predicate, *flat* to restrict the computation of $sg.2$, but these values are independent of the bindings known for the head predicate. []

3. The Adorned Rule Set

An *adornment* for an n-ary predicate p is a string a of length n on the alphabet $\{b, f\}$, where b stands for *bound* and f stands for *free*. We assume a fixed order of the arguments of the predicate. Intuitively, an adorned occurrence of the predicate, p^a, corresponds to a computation of the predicate with some arguments bound to constants, and the other arguments free, where the bound arguments are those that are so indicated by the adornment. For example, p^{bbf} corresponds to computing p with the first two arguments bound and the last argument free. If $p(X, Y, Z)$ appears in the head of a rule, then we expect the rule to be invoked with X and Y bound to constants. If $p(X, f(X, Z), W)$ is the head of a rule, then the rule will be invoked with X and $f(X, Z)$ bound to constants. Note that since bindings refer to arguments (positions) of p, if X is bound but Z is not, then $f(X,Z)$ is considered to be free. For brevity, we often refer to an argument in a position designated as bound (free) by the adornment as a bound (free) argument.

Let a program P and a query $q(\bar{c},\bar{X})$ be given, where $\bar{c}$ is the vector of bound arguments and $\bar{X}$ is the vector of free arguments. q is called the *query predicate*. We construct a new, adorned version of the program, denoted by P^{ad}. In the construction we replace derived predicates of the program by adorned versions, where for some predicates we may obtain several adorned versions. For each adorned predicate p^a, and for each rule with p as its head, we choose a sip and use it to generate an adorned version of the rule (the details are presented below). Since the head of a rule may appear with several adornments, it follows that we may attach several distinct sips to versions of the same rule, one to each version.

The process starts from the given query. The query determines bindings for q, and we replace q by an adorned version, in which precisely the positions bound in the query are designated as bound, say q^e. In general, we have a collection of adorned predicates, and as each one is processed, we will mark it, so that it will not be processed again. If p^a is an unmarked adorned predicate, then for each rule that has p in its head, we generate an adorned version for the rule and add it to P^{ad}; then p is marked as processed. The adorned version of a rule contains additional adorned predicates, and these are added to the collection, unless they already appear there. The process terminates when no unmarked adorned predicates are left. Termination is guaranteed, since the number of adorned versions of predicates for any given program is finite.

Let r be a rule in P with head p. We generate an adorned version, corresponding to an (unmarked) adorned predicate p^a, as follows. The new rule has p^a as a head. Choose a sip s_r for the rule, that matches the binding a. So, the special predicate p_h is the head p restricted to arguments that are designated as bound in the adornment a. Next, we replace each derived predicate in the body of the rule by an adorned version (and if this version is new, we add it to our collection). We obtain the adorned version of a derived predicate in the body of the rule as follows. For each occurrence p_i of such a predicate in the rule let χ_i be the union of the labels of all arcs coming into p_i. (If there is no arc coming into p_i, let χ_i denote the empty label.) We replace p_i by the adorned occurrence $p_i^{a_i}$, where an argument of p_i is bound in a_i only if all the variables appearing in it are in χ_i. (For a predicate occurrence with no incoming arc, the adornment contains only f's. For our purposes here, we do not distinguish between a predicate with such an adornment and an unadorned predicate.) The arguments of the predicates in the new rule are the same as in the original rule. Since the adornments attached to a rule's predicates are determined by the sip that was chosen, the sip is attached to the rule.

Example 3: The following is the adorned rule set corresponding to the non-linear same generation example, for the sip of example 1, as presented in generalized notation (IV) with arcs entering only derived predicates:

1. sg^{bf}(X,Y) :- flat(X,Y)
2. sg^{bf}(X,Y) :- up(X,Z1), sg^{bf}(Z1,Z2), flat(Z2,Z3), sg^{bf}(Z3,Z4), down(Z4,Y)

Query: sg^{bf}(john,Y)?

We will use these adorned rules to illustrate the rule rewrite algorithms presented later. Note that if we use the partial sip of example 2 instead, we obtain the same adorned program. The difference between the sips will only become significant in the next stage of the transformation. (It is not the case, however, that all sips for the same rule generate the same adorned version.) []

Thus, in general, it is important to remember the sips that were used to generate the adorned program, since they are used in the following transformations.

Note that a single adorned version of a rule is chosen for each adorned version of the head predicate. Thus, all goals whose binding pattern matches the adornment in an adorned head predicate are solved using the same adorned version of the rule, chosen at compile time. This does not cover dynamic strategies that choose a sip for a goal at run time, that is, allow two goals with the same binding pattern to be solved with different adorned versions of the rule.

Given an unadorned program, an adorned predicate p^a can be viewed as a *query form*. It represents queries of the form $p(\chi)$, in which all arguments corresponding to b's in adornment a are assigned constants. The same view holds for an adorned program, except that now p^a is both a predicate name and a query form that represents a class of queries on itself. Keeping these slightly different viewpoints in mind, we can now consider the equivalence of adorned and unadorned programs.

For programs P_1 and P_2 (where each may be adorned or unadorned), and a query form p^a, we say that (P_1, p^a) and (P_2, p^a) are *equivalent*, if for any assignment of constants to the arguments of p (or p^a, for an adorned program) that are bound in a, the two programs produce the same answer for the resulting queries on p (p^a).

Theorem 3.1: For each p^a that appears in P^{ad}, (P, p^a) and (P^{ad}, p^a) are equivalent.
Proof: First, we note that for each rule of P^{ad}, if the adornments are dropped, we obtain a rule of P. It follows that if a rule of P^{ad} is applied to some facts to produce a new fact, then the unadorned version of the rule can be applied to the unadorned versions of those facts to generate the new unadorned fact. Thus, it is straightforward to convert a derivation tree in P^{ad} into a derivation tree in P for the same fact.

In the other direction, we note a simple invariant in a bottom-up computation of P and P^{ad}: All the adorned versions in P^{ad} of a predicate in P contain the same set of tuples within each iteration, and this is the same set contained in the corresponding unadorned predicate in P. †
[]

We can now state the main result of this section, namely that our transformation preserves equivalence.

Corollary 3.2: $(P,\ q(\overline{c}, \overline{X}))$ and $(P^{ad},\ q^e(\overline{c}, \overline{X}))$ are equivalent.
Proof: The proof follows from the definition of program-query form equivalence. []

4. Generalized Magic Sets

Henceforth, we only consider the adorned set of rules, P^{ad}. The next stage in the proposed transformation is to define additional predicates that compute the values that are passed from one predicate to another in the original rules, according to the sip strategy chosen for each rule. Each of the original rules is then modified so that it fires only when values for these additional predicates are available. These auxiliary predicates are called *magic predicates* and the sets of values that they compute are called *magic sets*. The intention is that the bottom-up evaluation of the modified set of rules simulate the sip we have chosen for each adorned rule, thus restricting the search space.

Consider, for example the ancestor example of Section 1. assume that for the second rule we choose the following sip

† We thank one of the referees for pointing out a simplification in the original proof.

$\{anc; par\} \rightarrow_Z anc.1$

The adorned program then is:

anc^{bf}(X, Y) :- par(X, Y)
anc^{bf}(X, Y) :- par(X, Z), anc^{bf}(Z, Y)
Query: *anc*(john, Y)

We want to restrict the computation only to the ancestors of john. Let $magic\_anc^{bf}$ be a new predicate, of arity one, in which we intend to store the values for which *anc* needs to be computed. We start with

$magic\_anc^{bf}$(john)

Next, from the second rule we have that if we need to compute the ancestors of say, X, and par(X, Z) holds, then we will need also to compute the ancestors of Z. Hence we have

$magic\_anc^{bf}$(Z) :- $magic\_anc^{bf}$(X), par(X, Z)

Finally, to make the second rule fire only for the appropriate values, we change it to

anc^{bf}(X, Y) :- $magic\_anc^{bf}$(X), par(X,Z), $magic\_anc^{bf}$(Z), anc^{bf}(Z, Y)

(We will see later that a simpler transformed rule suffices.) It can now be seen that in a bottom-up computation the rule will not fire unless *X* and *Z* are first computed to be in the magic predicate, and that this happens only for ancestors of john (and for all of them).

The transformation to be described follows the same general ideas. It consists of the following.

1. We create a new predicate $magic\_p^a$ for each p^a in P^{ad}. (Thus, we create magic predicates only for derived predicates, possibly only for some of them.) The arity of the new predicate is the number of occurrences of *b* in the adornment *a*, and its arguments correspond to the bound arguments of p^a.

2. For each rule *r* in P^{ad}, and for each occurrence of an adorned predicate p^a in its body, we generate a *magic rule* defining $magic\_p^a$ (see below). (Note that an adorned predicate may have several occurrences, even in one rule, so several rules that define $magic\_p^a$ may be generated from a single adorned rule.)

3. Each rule is P^{ad} *modified* by the addition of magic predicates to its body.

4. We create a *seed* for the magic predicates, in the form of a fact, obtained from the query. The seed provides an initial value for the magic predicate corresponding to the query predicate. Using our notation above, the seed is $magic\_q^e(\overline{c})$.

We now explain the second step in more detail. We use the following notation. Greek letters (possibly subscripted) are used to denote argument lists. If χ denotes the argument list of a predicate p^a, then χ^f (χ^b) denotes χ with all arguments that are bound (free) in adornment 'a' deleted. Consider the adorned rule:

r: $p^a(\chi)$:- $q_1^{a_1}(\theta_1), q_2^{a_2}(\theta_2), \ldots, q_n^{a_n}(\theta_n)$

Let s_r be the sip associated with this rule. Assume that the predicates in the body are ordered according to the sip, as per condition 3'. (Those that participate in the sip precede those that do not, and the predicates in the tail of an arc precede the predicate at the head of the arc.)

Consider q_i. Let N --> q_i be the only arc entering q_i in the sip. We generate a magic rule defining $magic\_q_i^{a_i}$ as follows. The head of the magic rule is $magic\_q_i^{a_i}(\theta_i^b)$. If q_j, $j < i$, is in N, we add $q_j^{a_j}(\theta_j)$ to the body of the magic rule. If q_j is a derived predicate and the adornment a_j contains at least one b, we also add $magic\_q_j^{a_j}(\theta_j^b)$ to the body. If the special predicate denoting the bound arguments of the head is in N, we add $magic\_p^a(\chi^b)$ to the body of the magic rule.

If there are several arcs entering q_i, we define the magic rule defining $magic\_q_i^{a_i}$ in two steps. First, for each arc N_j --> q_i with label χ_j, we define a rule with head $label\_q_i\_j(\chi_j)$. The body of the rule is the same as the body of the magic rule in the case where there is a single arc entering q_i (described above). Then the magic rule is defined as follows. The head is $magic\_q_i^{a_i}(\theta_i^b)$. The body contains $label\_q_i\_j(\chi_j)$ for all j (that is, for all arcs entering q_i).

In the third step, we modify the original rule by inserting occurrences of the magic predicates corresponding to the derived predicates of the body and to the head predicate, into the rule body. In principle, the magic predicates can be inserted anywhere in the rule, but it helps to understand how they interact with the other predicate occurrences by considering specific positions for the insertions. The position for the insertion of $magic\_p^a(\chi^b)$ is at the left side of the rule's body, before all other predicates. The position for insertion of $magic\_q_i^{a_i}(\theta^{b_i})$ is just before the occurrence of $q_i^{a_i}$. Intuitively, $magic\_q_i^{a_i}(\theta^{b_i})$ computes the values that may be passed to $q_i^{a_i}$ from the left during evaluation of the rule. Insertion of $magic\_q_i^{a_i}$ serves as a guard. In a bottom-up evaluation, the rule does not fire, unless the appropriate values are first computed in $magic\_q_i^{a_i}$. The occurrence of $magic\_p^a$ serves the same purpose for the rule's head.

Example 4: Using the sips presented in Example 1 (IV), the Generalized Magic Sets strategy rewrites the adorned rule set corresponding to the non-linear same generation example into the following set of rules. (The rule numbers refer to the adorned rule set. Further, we simplify the rules by discarding some unnecessary occurrences of magic predicates from the rules. Proposition 4.2 describes when this can be done.)

$magic\_sg^{bf}$(john) [Seed; from the query rule]
$magic\_sg^{bf}$(Z1) :- $magic\_sg^{bf}$(X), up(X,Z1) [From rule 2, 2nd body literal]
$magic\_sg^{bf}$(Z3) :- $magic\_sg^{bf}$(X), up(X,Z1), sg^{bf}(Z1,Z2), flat(Z2,Z3)
[From rule 2, 4th body literal]
sg^{bf}(X,Y) :- $magic\_sg^{bf}$(X), flat(X,Y) [Modified rule 1]
sg^{bf}(X,Y) :- $magic\_sg^{bf}$(X), up(X,Z1), sg^{bf}(Z1,Z2),
flat(Z2,Z3), sg^{bf}(Z3,Z4), down(Z4,Y) [Modified rule 2]

If we apply the transformation to the partial sip (V), we obtain instead

$magic\_sg^{bf}$(john) [Seed; from the query rule]
$magic\_sg^{bf}$(Z1) :- $magic\_sg^{bf}$(X), up(X,Z1) [From rule 2, 2nd body literal]
$magic\_sg^{bf}$(Z3) :- $magic\_sg^{bf}$(Z1), sg^{bf}(Z1,Z2), flat(Z2,Z3)
[From rule 2, 4th body literal]
sg^{bf}(X,Y) :- $magic\_sg^{bf}$(X), flat(X,Y) [Modified rule 1]
sg^{bf}(X,Y) :- $magic\_sg^{bf}$(X), up(X,Z1), sg^{bf}(Z1,Z2), flat(Z2,Z3),
sg^{bf}(Z3,Z4), down(Z4,Y) [Modified rule 2]

Let P^{mg} denote a program obtained from P^{ad} by the transformation above. We now consider the correctness of the transformation. There is a factor we need to consider first. For the given query, we have a seed definition for the magic sets. If we choose a different query with the same query form, the same magic predicates, magic predicate definitions and modified rules will result, but the seed will be specific to the query. Therefore, let us consider the seed not to be a part of P^{mg}. We say that (P^{ad},p^a) and (P^{mg}, p^a) are equivalent if the two programs produce the same results for every instance of the query form p^a, *if the corresponding seed is added to* P^{mg}.

Theorem 4.1: Let P^{ad}, P^{mg} be as above, and let p^a be a predicate that appears in P^{ad}. Then (P^{ad}, p^a) is equivalent to (P^{mg}, p^a).

Proof: One direction is simple. Each rule of P^{mg} that is derived from a rule of P^{ad}, is more restrictive than that rule, since it has additional predicates in its body. It follows that any answer produced by P^{mg} for a query can also be produced by P^{ad}.

The other direction is proved by induction on the height of derivation trees of facts in P^{ad}. The basis of the induction is the set of derivation trees of height one. These are simply base facts, and they are also derivation trees for P^{mg}. Consider now a derivation tree of height n, and assume that the rule used to derive its root is the following

r: $p^a(\chi)$:- $q_1^{a_1}(\theta_1), q_2^{a_2}(\theta_2), \ldots, q_n^{a_n}(\theta_n)$

Let the instance of the rule at the root be

$$p^a(c) \text{ :- } q_1^{a_1}(c_1), q_2^{a_2}(c_2), \ldots, q_n^{a_n}(c_n) \qquad \text{(AD)}$$

The corresponding rule instance in P^{mg} has the form

$$p^a(c) \text{ :- } magic\_p^a(c^b), Q_1, \ldots, Q_n, \qquad \text{(MG)}$$

where each Q_i is either a pair of predicates

$magic\_q_i^{a_i}(c_i^b)$, $q_i^{a_i}(c_i)$,

or a single predicate

$q_i^{a_i}(c_i)$

(The second case occurs when Q_i is either a base predicate or a derived predicate with an adornment containing only f's, or it is a derived adorned predicate, but we have chosen to omit the corresponding magic predicate.) By the induction hypothesis, there exist derivation trees in P^{mg} for the derived predicate occurrences in the body of the rule, since each of them is the root of a derivation tree of P^{ad} tree of height less than n. Note however, that for each such derivation tree the (new) seed used in it should correspond to the fact being derived. For the fact $q_i^{a_i}(c_i)$, the seed is $magic\_q_i^{a_i}(c_i^b)$. Since these seeds are not known *a priori*, to rely on the induction hypothesis, we have to show that they can be derived in P^{mg} augmented with the seed for the original query. This will also show that all the facts in the body of the rule instance (MG) above are derivable in P^{mg}, from which it follows that the fact $p^a(c)$ is derivable as well.

The proof is by induction, where the induction is on the position of the predicate occurrence in the rule body. For $magic\_p^a$, the fact that is to be derived is $magic\_p^a(c^b)$. However, note that $p^a(c)$ is an answer for a query that is an instance of p^a. The seed for that query is, by assumption, given to us. It is precisely the desired fact, $magic\_p^a(c^b)$. We see now that each of the other magic facts that we need, $magic\_q_i^{a_i}(c_i^b)$ (for each i such that q_i is derived), is derivable. Indeed, for $i = 1$, if q_1 is derived, then

$$magic\_q_1^{a_1}(\chi_1^b) \text{ :- } magic\_p^a(\chi^b)$$

is a magic rule defining $magic\_q_1^{a_1}$. It follows that $magic\_q_1^{a_1}(c_1^b) = magic\_p^a(c^b)$ is derivable. For $i > 1$, (if q_i is derived), we use the fact $magic\_p^a(c^b)$ and the induction hypothesis to show the existence of a derivation tree for $q_j^{a_j}(c_j)$, $j < i$, and thus derive the fact $magic\_q_i^{a_i}(c_i)$. []

In constructing the magic rules and the modified rules, we added a number of magic predicates to the body. We now prove an important lemma which shows that in each rule some of these magic predicates may be dropped without loss of information. That is, the sets of values computed by the magic predicates remain unchanged, and the number of firings of the modified rules remains unchanged.

Consider an adorned rule and a sip for it. Let us define $p \Rightarrow q$ as follows. If the sip contains an arc $N \dashrightarrow q$, and N contains p, then $p \Rightarrow q$. If $p \Rightarrow l$ and $l \Rightarrow q$, then $p \Rightarrow q$. (We cannot have $p \Rightarrow q$ and $q \Rightarrow p$ simultaneously because the sip induces a total ordering.) Let us define the *order* of q to be the length of the longest chain $q_i \Rightarrow q_{i+1} \Rightarrow \ldots \Rightarrow q$. The order is 0 if there is no such chain.

Consider the set P^{mg_opt} that is obtained from P^{mg} by repeated applications of the following transformation:

Let r be an adorned rule and let r' be a (magic or modified) rule generated from r. If the body of r' contains occurrences of both $magic\_p_i^{a_i}$ and $magic\_p_j^{a_j}$, and $p_i \Rightarrow p_j$, then delete the occurrence of $magic\_p_j^{a_j}$.

We have the following proposition.

Proposition 4.2: Let P^{mg} and P^{mg_opt} be as above, and let p^a be a predicate that appears in P^{mg}. Then, (P^{mg}, p^a) is equivalent to (P^{mg_opt}, p^a). []

The previous lemma tells us that we may drop some occurrences of magic predicates while passing only the information indicated by the sip. We have stated the above lemma without proof since a simpler and stronger result can be shown to hold, as observed in [Balbin et al.

87]. †

Proposition 4.3: Let r be a rule in P, and let r' be a (magic or modified) rule in P^{mg} that is obtained from r. Let p_h denote the bound arguments of the head of r in the sip associated with r. Then, a magic predicate in the body of r' can be deleted unless it corresponds to p_h. []

The intuition is that the magic literal in the rules defining a predicate q^a already achieves the restriction that is obtained by adding the corresponding magic predicate to an occurrence of q^a. The rewriting algorithm is simplified in [Balbin et al. 87] by not introducing any of the redundant magic predicates.

5. Generalized Supplementary Magic Sets

The Generalized Magic Sets algorithm for rewriting a set of adorned rules succeeds in implementing a given set of sips, but it suffers from the drawback that many facts are evaluated repeatedly. If $p \Rightarrow q$ in some adorned rule, then the evaluation of *magic_q* repeats much of the work done in evaluating *magic_p*; and further, the evaluation of the modified rules repeats the work done in evaluating the magic sets.

Consider the non-linear same-generation example presented earlier (Example 4). The join of $magic\_sg^{bf}$ and *up* in the first magic rule is evaluated again in the second magic rule. Further, every join in the second magic rule is evaluated again in the second modified rule defining sg^{bf}.

We now present another algorithm for rewriting a set of adorned rules. This algorithm is motivated by the drawback of the previous algorithm, and by the observation that much of the duplicate work can be eliminated if we store intermediate results that are potentially useful later. We store these results in special predicates called *supplementary magic predicates*, following Sacca and Zaniolo [Sacca and Zaniolo 86], who used essentially the same idea in generalizing the versions of the Magic Sets and Counting algorithms presented in [Bancilhon et al. 86].

The algorithm is as follows. We first order the predicates in the body of each adorned rule according to the total ordering induced by the sip associated with that rule.

For each adorned rule:

1. We introduce a number of *supplementary magic predicates*, $supmagic_i^r$, associated with this rule, and define them using *supplementary magic rules*.

2. For each occurrence of an adorned predicate p^a in the body of the adorned rule, we generate a *magic rule* defining the *magic* predicate $magic\_p^a$ if the sip associated with r contains an arc $N \dashrightarrow p$.

3. We generate a *modified* rule from the adorned rule by replacing some of the predicates in the body by a supplementary magic predicate.

---

† This was also pointed out by one of the referees.

-- --

Finally, we create a *seed* for the magic predicates, in the form of a fact, from the query.

Consider the adorned rule:

r: $p^a(\chi)$:- $q_1^{a_1}(\theta_1), q_2^{a_2}(\theta_2), \ldots, q_n^{a_n}(\theta_n)$

Let the body predicates be ordered in accordance with the sip ordering, and let q_m be the last body predicate that has an incoming arc in the sip. We generate m supplementary magic rules. The first supplementary magic rule is:

$supmagic_1^r(\phi_1)$:- $magic\_p^a(\chi^b)$

where ϕ_1 is the set of variables that appear in arguments of χ^b.

Supplementary magic rule i, i = 2 to m is:

$supmagic_i^r(\phi_i)$:- $supmagic_{i-1}^r(\phi_{i-1}), q_{i-1}^{a_{i-1}}(\theta_{i-1})$

where ϕ_i is the set of variables that appear in arguments of ϕ_{i-1} or θ_{i-1}.

In generating the supplementary magic rules, the following simple optimizations may be applied. We may discard from ϕ_i, i = 1 to n, all variables that do not appear in any arguments of χ or θ_j, j = i to n. Also, the supplementary magic rule defining $supmagic_1^r(\phi_1)$ may be deleted if we replace every occurrence of this literal in a rule body by $magic\_p^a(\chi^b)$, where $p^a(\chi)$ is the head of the adorned rule r.

We generate a magic rule defining $magic\_q_i^{a_i}$ if the sip s_r contains an arc N --> q_i. (In generating the predicate name $magic\_q_i^{a_i}$, we exclude any subscripts of q_i introduced for the purpose of distinguishing different occurrences of the same predicate.) This magic rule is:

$magic\_q_i^{a_i}(\theta_i^b)$:- $supmagic_i^r(\phi_i)$

The modified rule corresponding to r is:

$p^a(\chi)$:- $supmagic_m^r(\phi_m), q_m^{a_m}(\theta_m), \ldots, q_n^{a_n}(\theta_n)$

Finally, if the query is $q^a(\eta)$, we also add a magic rule corresponding to it, to act as the seed:

$magic\_q^a(\eta^b)$

Thus, in addition to auxiliary ‘‘magic’’ predicates associated with each adorned predicate, we also define auxiliary ‘‘supplementary magic’’ predicates associated with positions in a rule.

Example 5: Continuing with the same generation example, the Generalized Supplementary Magic Sets algorithm produces the following rules by rewriting the adorned rules according to the given sip. (The rule numbers refer to the adorned rules.)

$magic\_sg^{bf}$(john)	[From the query rule]
$supmagic_2^2$(X,Z1) :- $magic\_sg^{bf}$(X), up(X,Z1)	[From rule 2]
$supmagic_3^2$(X,Z2) :- $supmagic_2^2$(X,Z1), sg^{bf}(Z1,Z2)	[From rule 2]
$supmagic_4^2$(X,Z3) :- $supmagic_3^2$(X,Z2), flat(Z2,Z3)	[From rule 2]
sg^{bf}(X,Y) :- $magic\_sg^{bf}$(X), flat(X,Y)	[Modified rule 1]

sg^{bf}(X,Y) :- $supmagic_4^2$(X,Z3), sg^{bf}(Z3,Z4), down(Z4,Y) [Modified rule 2]
$magic\_sg^{bf}$(Z1) :- $supmagic_2^2$(X,Z1) [From rule 2, 2nd body literal]
$magic\_sg^{bf}$(Z3) :- $supmagic_4^2$(X,Z3) [From rule 2, 4th body literal]

[]

Let us denote by P^{sup-mg} any program obtained by this transformation.

Theorem 5.1: Let P^{ad}, P^{sup-mg} be as above, and let p^a be a predicate in P^{ad}. Then (P^{ad}, p^a) is equivalent to (P^{sup-mg}, p^a).

Proof: As in the previous case, the proof in one direction is simple. The other direction is proved by induction on the height of derivation trees of facts in P^{ad}. The proof is similar to the previous proof. It suffices to note that the seeds have the same form as in the previous case, and that the seed for $q_i^{a_i}(c_i)$ can be obtained by first generating $supmagic_i^r(d)$, where d is a vector of constants, and then using the auxiliary rule $magic\_q_i^{a_i}(..) :- supmagic_i^r(..)$ to generate the required seed $magic\_q_i^{a_i}$. []

The Alexander strategy, described in [Rohmer and Lescoeur 85], is essentially the Generalized Supplementary Magic Sets strategy, although they only consider Datalog.

6. Generalized Counting

Counting is a further elaboration on the theme of restricting the search by auxiliary predicates. Using magic predicates, we were able to restrict the invocation of a predicate to values that were reachable from those given in the query. The new idea here is to keep track of which rules and which predicate occurrence in each rule were used to reach a vector of values that is now used in the invocation of a predicate. This is done using indices that essentially encode the structure of the computation used to generate a fact. These indices allow us to perform certain powerful optimizations in some cases. The Counting method was in fact originally presented [Bancilhon et al. 86, Sacca and Zaniolo 86a] with these optimizations as an integral part of the transformation. (Like the Magic Sets method, the original version was of restricted applicability, and for example, could not handle the same generation example that we have used as a running example.)

We have chosen to separate the indexing feature of the method from these further optimizations in order to show the underlying relationship of this method to the Generalized Magic Sets method. In this paper, therefore, the Generalized ‘‘Counting’’ method refers to only a part of the transformation usually associated with this name. The optimizations that comprise the second part of the transformation (and provide the raison d'etre for the elaborate indexing mechanisms introduced in this section) are presented in a separate section (Section 8). Before that, we also show how the Generalized Counting method (essentially Generalized Magic Sets with indices) can be refined by using auxiliary ‘‘supplementary’’ predicates, just like the Generalized Magic Sets method.

Since computing these indices represents additional work, we expect these methods to be used only when the further optimizations described in Section 8 are applicable. (The indices by themselves do not provide additional selectivity. That is, if a program is rewritten according to Generalized Counting, the set of facts computed by the rewritten program, after projecting out the index fields, is identical to the set of facts computed by the rewritten program produced by Generalized Magic Sets.)

We now describe the Counting transformation. We first replace each adorned derived predicate p^a in the adorned rules, where a contains at least one bound argument, by an *indexed version* of this predicate, p_ind^a, that has three new arguments. These arguments are used for constructing indices, and we assume that they are the first three arguments. Note that the adornment a refers only to the non-index fields. For each rule r in P^{ad}:
1. For each occurrence of an adorned predicate p^a in the body of the adorned rule, we generate a *counting rule* defining the *counting* predicate cnt_p^a if the sip associated with r contains an arc N --> p.
2. The rule is *modified* by the addition of counting predicates to its body.

Finally, we create a *seed* for the counting predicates, in the form of a fact, from the query.

Before each of these steps is explained, let us consider how the rules and predicates used in a derivation may be encoded in a predicate, together with the derived value(s). Assume we have m rules† [1], numbered r_0 to r_{m-1}. Then the sequence of rules used in a derivation can be represented by a sequence of numbers, each in the range $[0 .. m-1]$. Any standard encoding can be used to represent such sequences by numbers. We will use the encoding that represents the sequence $i_0, i_1, ..., i_k$ by the value $(...((i_0 \times m) + i_1) \times m + ...) \times m + i_k$. In other words, given a number that represents a sequence, to concatenate an element to the sequence we multiply by m and add the element. The last element is the remainder modulo m, and previous elements can be obtained by repeated use of the modulus operation. A similar encoding is used for predicate occurrences. Assuming that there are at most t occurrences of predicates in any rule's body, we use t as the base for the encoding. † [2] These two encodings occupy the second and third positions of the counting predicates. The first position is used to record the number of rules that were applied so far (starting from the seed). (Note that this number can be computed from the encodings; it is convenient to have an explicit representation for it.)

We now describe each step of the Counting algorithm in detail. *It is understood in the following that the index fields are omitted from arguments of base predicates and adorned predicates p^a where a contains no bound arguments.*

Consider the adorned rule:

$$r_i\colon p^a(\chi) \text{ :- } q_1^{a_1}(\theta_1), q_2^{a_2}(\theta_2), \ldots, q_n^{a_n}(\theta_n)$$

We generate a counting rule defining $cnt\_q\_ind_j^{a_j}$ if the sip associated with r_i contains an arc† N --> q_j. The head of the counting rule is $cnt\_q\_ind_j^{a_j}(I+1, K\times m+i, H\times t+j, \theta_j^b)$. If q_k is in N,

---

† 1,2 We call a rule an *exit rule* if all the predicates in its body are base predicates. We need to encode only non-exit rules. Similarly, we need only keep track of invocation of derived predicates. For clarity, we do not use these optimizations in our examples.

we add $q\_ind_k^{a_k}(I+1, K\times m+i, H\times t+k, \theta_k)$ to the body of the counting rule. If q_k is a derived predicate and the adornment a_k contains at least one b, we also add $cnt\_q\_ind_k^{a_k}(I+1, K\times m+i, H\times t+k, \theta_k^b)$ to the body. If the special predicate denoting the bound arguments of the head is in N, we add $cnt\_p\_ind^a(I, K, H, \chi^b)$ to the body of the counting rule.

The modified rule corresponding to adorned rule r_i is:

$$p\_ind^a(I, K, H/t, \chi) \text{ :- } cnt\_p\_ind^a(I, K, H/t, \chi^b),\ q\_ind_1^{a_1}(I+1, K\times m+i, H+1, \theta_1)$$
$$, \ldots, q\_ind_n^{a_n}(I+1, K\times m+i, H+n, \theta_n)$$

If there is no arc entering q_j in the sip, the three index fields in the modified rule given above are omitted. Further, we could add the counting predicates corresponding to the predicates $q_j^{a_j}$ to the body of the modified rule, but we have a lemma (see below) which tells us that they are unnecessary, and so we have omitted them altogether for simplicity.

Finally, if the query is $q^a(\eta)$, we add the fact $cnt\_q\_ind^a(0, 0, 0, \eta^b)$ to serve as the seed for the counting predicates.

Example 6: Continuing our running example, we present below the rewritten rules produced by the Generalized Counting method.

cnt_sg_ind^{bf}(I+1, k*2+2, h*5+2, Z1) :- *cnt_sg_ind*^{bf}(I, k, h, X), up(X,Z1)
[From rule 2, second body literal]

cnt_sg_ind^{bf}(I+1, k*2+2, h*5+4, Z3) :- *cnt_sg_ind*^{bf}(I, k, h, X), up(X,Z1),
sg_ind^{bf}(I+1, k*2+2, h*5+2, Z1,Z2), flat(Z2,Z3)
[From rule 2, fourth body literal]

sg_ind^{bf}(I, k, h/5, X,Y) :- *cnt_sg_ind*^{bf}(I, k, h/5, X), flat(X,Y) [Modified rule 1]

sg_ind^{bf}(I, k, h/5, X,Y) :- *cnt_sg_ind*^{bf}(I, k, h/5, X), up(X,Z1),
sg_ind^{bf}(I+1, k*2+2, h+2, Z1,Z2), flat(Z2,Z3),
sg_ind^{bf}(I+1, k*2+2, h+4, Z3,Z4), down(Z4,Y) [Modified rule 2]

cnt_sg_ind^{bf}(0, 0, 0, john) [From the query rule]

[]

Let us denote a program obtained from P^{ad} by the transformation above by P^{count}. We now need a convention for comparison of the queries in P^{ad} and P^{count}. We will say that (P^{ad}, p^a) is equivalent to (P^{count}, p_ind^a) are equivalent if for any vector θ of appropriate arity, with constants in the positions bound by a, the program-query $(P^{ad}, p^a(\theta))$ has the same set of answers as the program-query pair $(P^{count}, p\_ind^a(I, K, H, \theta))$, for any values of I, K, H. (Note that we are not restricting attention to queries with a triple of 0's. We need arbitrary triples for the induction hypothesis of the theorem below. Intuitively, it should not matter which numerical values are supplied with the query, so this is not a real generalization.) We will also use the same conventions as in the previous sections, regarding seeds. We now have the following theorem.

† We assume that there is at most one such arc. The generalization to the case where there are several such arcs is similar to that in the Generalized Magic Sets strategy.

-- --

Theorem 6.1: Let P^{ad} and P^{count} be as above and let p^a be a query form. Then (P^{ad}, $p^)$ and (P^{count}, p_ind^a) are equivalent.

Proof: Again, one direction is simple, since each rule of P^{count} is more restricted then the corresponding rule of P^{ad}. The other direction is proved by induction on the height of derivation trees of facts in P^{ad}.

Derivation trees of height zero are base facts, and are also derivation trees of P^{count}. For a derivation tree of height n, assume the rule used in P^{ad} to derive its root is the rule

$$r_i{:}p^a(\chi) :\text{-}\ q_1^{a_1}(\theta_1), q_2^{a_2}(\theta_2), \ldots, q_n^{a_n}(\theta_n)$$

and the instance of the rule that is used is

$$p\_ind^a(c) :- \ q\_ind_1^{a_1}(c_1), \ldots, q\_ind_n^{a_n}(c_n)$$

By the induction hypothesis, there exist derivation trees in P^{count} for the extended versions of the derived predicate occurrences that appear in the body, for any values of the numerical triples that may be added. Here again, we have to show, however, that appropriate seeds can be generated first. Let us consider the case where $p\_ind^a(I, K, H, c)$ is the instance of the root for which we want to show the existence of a derivation tree. The seed, $cnt\_p\_ind^a(I, K, H, c^b)$ is by assumption given to us. We now prove by induction on the position of a predicate occurrence in the body that the appropriate seed for each predicate, which is an instance of the corresponding counting predicate, can be generated. The details follow the line of the proof of Theorem 4.1 and are omitted. []

Let P^{cnt_opt} be defined similarly to P^{mg_opt}. We have the following lemma that allows us to delete unnecessary occurrences of counting predicates from rule bodies.

Lemma 6.2: Let P^{cnt} and P^{cnt_opt} be as above, and let p^a be a predicate that appears in P^{cnt}. Then, (P^{cnt}, p^a) is equivalent to (P^{cnt_opt}, p^a).

Proof: The proof is similar to the proof of Proposition 4.2 and is omitted. []

7. Generalized Supplementary Counting

The Generalized Counting algorithm suffers from duplicate work since, like in the Generalized Magic Sets algorithm, several the bodies of several rules contain the same joins. We use the same idea - of eliminating this duplication by storing potentially useful intermediate results - to define the Generalized Supplementary Counting method.

The algorithm is as follows. We first order the predicates in the body of each adorned rule according to the total ordering induced by the sip associated with that rule. We also replace each adorned derived predicate p^a in the adorned rules by an *extended version* of this predicate, p_ind^a, that has three new arguments. These arguments are used for indices, and we assume that they are the first three arguments. Now, for each adorned rule:

1. We introduce a number of *supplementary counting predicates*, $supcnt_i^r$, associated with this rule, and define them using *supplementary counting rules*.

2. For each occurrence of an adorned predicate p^a in the body of the adorned rule, we generate a *counting rule* defining the *counting* predicate cnt_p^a if the sip associated with r contains an arc N --> p.

3. We generate a *modified* rule from the adorned rule by replacing some of the predicates in the body by a supplementary counting predicate and appropriately indexing the remaining predicates.

Finally, we create a *seed* for the counting predicates, in the form of a fact, from the query.

We now explain each of the above steps in detail. It is understood in the following that the three index fields are omitted from arguments of base predicates.

Consider the adorned rule:

$$r:\ p^{a}(\chi)\ :\!-\ q_1^{a_1}(\theta_1),\, q_2^{a_2}(\theta_2),\, \ldots,\, q_n^{a_n}(\theta_n)$$

Let the body predicates be ordered in accordance with the sip ordering, and let q_m be the last body predicate that has an incoming arc in the sip. We generate m supplementary counting rules. The first supplementary counting rule is:

$$supcnt_1^r(I,K,H,\phi_1)\ :\!-\ cnt\_p\_ind^{a}(I,K,H,\chi^b)$$

where ϕ_1 is the set of variables that appear in arguments of χ^b, and I, K and H are running indices.

Supplementary counting rule j, j = 2 to m is:

$$supcnt_j^r(I,K,H,\phi_j)\ :\!-\ supcnt_{j-1}^r(I,K,H,\phi_{j-1}),\, q\_ind_{j-1}^{a_{j-1}}(I+1, K\times m+i, H\times t+j-1, \theta_{j-1})$$

where ϕ_j is the set of variables that appear in arguments of ϕ_{j-1} or θ_{j-1}.

In generating the supplementary counting rules, the following simple optimizations may be applied. We may discard from ϕ_j, j = 1 to m, all variables that do not appear in any arguments of χ or θ_k, k = j to m. Also, the supplementary counting rule defining $supcnt_1^r(I,K,H,\phi_1)$ may be deleted if we replace every occurrence of this literal in a rule body by $cnt\_p\_ind^{a}(I,K,H,\chi^b)$, where $p^a(\chi)$ is the head of the adorned rule r.

We generate a counting rule defining $cnt\_q\_ind_j^{a_j}$ if the sip associated with the adorned rule contains an arc N --> q_j. (In generating the predicate name $cnt\_q\_ind_j^{a_j}$, we exclude any subscripts of q_j introduced for the purpose of distinguishing different occurrences of the same predicate.) This counting rule is:

$$cnt\_q\_ind_j^{a_j}(I+1, K\times m+i, H\times t+j, \theta_j^b)\ :\!-\ supcnt_j^r(I,K,H,\phi_j)$$

The modified rule corresponding to r_i is:

$$p\_ind^{a}(I,K,H,\chi)\ :\!-\ supcnt_m^r(I,K,H,\phi_m),\, q\_ind_m^{a_m}(I+1, K\times m+i, H\times t+m, \theta_m),\, \ldots,\, q\_ind_n^{a_n}(\theta_n)$$

(Note that none of the predicates following $q\_ind_m^{a_m}$ contain index fields. This follows from the fact that no sip arc enters any of them, and so they have no bound arguments.)

Finally, if the query is $q^a(\eta)$, we also add a counting rule corresponding to it, to act as the seed:

$$cnt\_q\_ind^{a}(0,0,0,\eta^b)$$

Example 7: The same non-linear same generation query is rewritten as follows by the Generalized Supplementary Counting algorithm.

$supcnt_2^2$(I, k, h, X,Z1) :- $cnt\_sg\_ind^{bf}$(I, k, h, X), up(X,Z1) [From rule 2]
$supcnt_3^2$(I, k, h, X,Z2) :- $supcnt_2^2$(I, k, h, X,Z1), sg_ind^{bf}(I+1, k*2+2, h*5+2, Z1,Z2) [From rule 2]
$supcnt_4^2$(I, k, h, X,Z3) :- $supcnt_3^2$(I, k, h, X,Z2), flat(Z2,Z3) [From rule 2]
sg_ind^{bf}(I, k, h, X,Y) :- $cnt\_sg\_ind^{bf}$(I, k, h, X), flat(X,Y) [Modified rule 1]
sg_ind^{bf}(I, k, h, X,Y) :- $supcnt_4^2$(I, k, h, X,Z3),
    sg_ind^{bf}(I+1, k*2+2, h*5+4, Z3,Z4), down(Z4,Y) [Modified rule 2]
$cnt\_sg\_ind^{bf}$(I+1, k*2+2, h*5+2, Z1) :- $supcnt_2^2$(I, k, h, X,Z1) [From rule 2, second body literal]
$cnt\_sg\_ind^{bf}$(I+1, k*2+2, h*5+4, Z3) :- $supcnt_4^2$(I, k, h, X,Z3) [From rule 2, fourth body literal]
$cnt\_sg\_ind^{bf}$(0, 0, 0, john) [From the query rule]

Let us denote by $P^{sup-cnt}$ any program obtained by this transformation.

Theorem 7.1: Let P^{ad} and $P^{sup-cnt}$ be as above, and let p^a be a query form. Then (P^{ad}, p^a) is equivalent to ($P^{sup-cnt}$, $\bar{p}^a$).

Proof: As in Theorem 6.1, the proof in one direction is simple. The other direction is proved by induction on the height of derivation trees of facts in P^{ad}. The proof is similar to the previous proof. []

8. Further Optimizations of Counting Methods

We now present some optimizations that may sometimes be used in further rewriting rules produced by the Generalized Counting or Generalized Supplementary Counting Strategies. They do not apply to the Magic Sets strategies since they rely on the indices generated by the Counting strategies. As we remarked earlier, since computing the indices represents an additional overhead, we expect the Counting methods to be used only when the optimizations described in this section are applicable. The improvement can be considerable, as the analysis in [Bancilhon and Ramakrishnan 87] indicates.

Consider the rewritten program produced by the Counting method for the same generation example (Example 6).

If we examine the second rule defining $cnt\_sg\_ind^{bf}$:

$cnt\_sg\_ind^{bf}$(I+1, k*2+2, h*5+4, Z3) :- $cnt\_sg\_ind^{bf}$(I, k, h, X), up(X,Z1),
    sg_ind^{bf}(I+1, k*2+2, h*5+2, Z1,Z2), flat(Z2,Z3)

we find that it is equivalent to the following rule:

$cnt\_sg\_ind^{bf}$(I+1, k*2+2, h*5+4, Z3) :- sg_ind^{bf}(I+1, k*2+2, h*5+2, Z1,Z2), flat(Z2,Z3)

The deleted literals influence the rest of the rule only by joining with the first (non-index) argument of sg_ind^{bf}, which is Z1. However, we can identify every value for Z1 for which this join succeeds using only the joins on index fields, by construction of the indices. Thus, deleting these literals does not change the tuples computed for $cnt\_sg\_ind^{bf}$, and the two rules are equivalent.

This observation leads to our first optimization.

Lemma 8.1: Consider a rule r in the given program, with an arc $N \rightarrow q_k$ in the corresponding sip. Consider the modified rule produced from r by the (Supplementary) Counting method, or the (supplementary) counting rule generated from this sip arc. Denote this rule r_1.

If no variable appearing in a predicate in N (or an associated counting predicate) appears outside N in r_1, except possibly in bound arguments of $q\_ind_k^{a_k}$, then all predicates in N (and their counting predicates) may be deleted from the given rule.

Proof: The predicates in N (and their counting predicates) represent a join with the bound arguments of $q\_ind_k^{a_k}$. By construction of the counting predicates, the projection of N that participates in this join is a subset of the counting predicate for $q\_ind_k^{a_k}$. The indices identify the subset that belongs in this projection. Since only facts that agree with the tuples in this counting predicate are computed in $q\_ind_k^{a_k}$, the join with the predicates in N is satisfied for every tuple in $q\_ind_k^{a_k}$, with the appropriate index values. We may therefore delete the predicates in N from the rule. []

The intuition behind the above lemma is as follows. The deleted literals' only purpose is to provide values for the bound arguments of θ_k. This is done in the counting rule for the corresponding counting predicate. Thus they may be dropped in the given rule.

If the variables in N appear outside N, the deleted literals still play a role in the modified rule, and deleting them is not correct (if we do so, we will generate some tuples that might be ruled out by the deleted literals). For example, if the body has:

... b1(X,Z), b2(Z), p(X,Y), ...

and X is bound in p by an arc $\{b1\} \rightarrow_X p$, then the literal b1(X,Z) is needed in the modified rule to effect the join on Z.

We reproduce below the rewritten rules for the same generation example (from Example 6) with the modifications allowed by the previous lemma.

$cnt\_sg\_ind^{bf}$(I+1, k*2+2, h*5+2, Z1) :- $cnt\_sg\_ind^{bf}$(I, k, h, X), up(X,Z1)
[From rule 2, second body literal]
$cnt\_sg\_ind^{bf}$(I+1, k*2+2, h*5+4, Z3) :- sg_ind^{bf}(I+1, k*2+2, h*5+2, Z1,Z2), flat(Z2,Z3)
[From rule 2, fourth body literal]
sg_ind^{bf}(I, k, h/5, X,Y) :- $cnt\_sg\_ind^{bf}$(I, k, h/5, X), flat(X,Y) [Modified rule 1]
sg_ind^{bf}(I, k, h/5, X,Y) :- $cnt\_sg\_ind^{bf}$(I, k, h/5, X), up(X,Z1),
sg_ind^{bf}(I+1, k*2+2, h+2, Z1,Z2), flat(Z2,Z3),
sg_ind^{bf}(I+1, k*2+2, h+4, Z3,Z4), down(Z4,Y) [Modified rule 2]
$cnt\_sg\_ind^{bf}$(0, 0, 0, john) [From the query rule]

[]

Let us again examine the second rule defining $cnt\_sg\_ind^{bf}$. We observe that a further optimization is possible - we can replace Z1 in the body literal sg_ind^{bf} by an anonymous variable. The value in this argument does not influence the rule since the variable Z1 does not appear anywhere else in the rule. What is important here is that only tuples of sg_ind^{bf} that have certain values in this argument should be used to join with *flat*. This is ensured by the indices since they identify the appropriate sg_ind^{bf} facts.

This leads to our second optimization.

Lemma 8.2: Consider a rule in the rewritten set of rules produced by the Counting or Supplementary Counting method, either before or after applying Lemma 8.1. Consider a predicate $q\_ind_k^{a_k}$ in the body. If for each variable in a bound argument of $q\_ind_k^{a_k}$, the variable does not appear anywhere else in the rule, then the bound arguments of $q\_ind_k^{a_k}$ in that occurrence may be designated as anonymous variables.

Proof: If each of the deleted arguments is a (distinct) variable, then none of them represents a join with any other predicate in the rule. Since we are only interested in the free arguments, and the values in the free arguments of a given tuple are determined by the indices, the value in the bound argument is not a constraint on the values in the free arguments. So, we may drop the bound arguments without changing the set of tuples computed by the rule. If one of these arguments is a constant, or if the same variable appears in two bound arguments, then it is necessary to observe that the counting predicate for $q\_ind_k^{a_k}$ is computed using these restrictions. Thus, every tuple with the appropriate indices will have the same constants in the corresponding argument places, and identical values in argument places corresponding to the same variable. []

If Lemma 8.2 applies to every occurrence of an adorned predicate $q\_ind_k^{a_k}$ in the rewritten set of rules, we may conclude that the values for these arguments are not needed. We may drop these arguments, that is, decrease the arity of $q\_ind_k^{a_k}$, in all rules of the program. In particular, the bound arguments may then be dropped from the heads of those rules with this adorned predicate ($q\_ind_k^{a_k}$) in the head.

Now assume that some of the variables in bound arguments of $q\_ind_k^{a_k}$ in a body of a rule appear in the bound arguments of the head of the rule. Then, it appears that we cannot omit these arguments. Assume that the head predicate is $q\_ind_j^{a_j}$. It may be the case that the same phenomenon happens with the bound arguments of $q\_ind_j^{a_j}$, where it appears in the body of another rule. Namely, in that other rule the bound arguments are only needed for the bound arguments in the head. We thus obtain a chain of justifications for the need to keep arguments in predicates. If the chain ‘‘closes’’ by arriving again at $q\_ind_k^{a_k}$, then, intuitively, the support is circular and we can drop all of these bound arguments.

Let us call a maximal set of mutually recursive predicates a *block*.

Theorem 8.3: (The Semijoin Optimization) Consider a block B of mutually recursive (adorned) predicates in the rewritten set of rules produced by the Counting or Supplementary Counting method. Suppose the following conditions hold for every predicate *p* in B, in every rule defining a predicate in block B:

1. No variable in a bound argument of (a body literal) p appears anywhere else in the rule, except possibly in bound arguments of the head, or in some other bound arguments of the same literal p, or in arguments of predicates in N, where there is an arc $N \rightarrow p$ in the corresponding sip.
2. For each arc $N \rightarrow p$ encountered in (1), the variables appearing in N (or in associated counting predicates) appear nowhere else in the rule, except possibly in bound arguments of p_ind^{a}.

Then, all the bound arguments of the predicates in B may be deleted (that is, their arities are decreased) and for each rule defining a predicate in B, and each arc $N \rightarrow p$ encountered above, the predicates in N may be deleted from the rule.

Proof: First, we note that Lemma 8.1 can be used to remove the predicates of N, for each arc $N \rightarrow p$ that satisfies the conditions of the theorem. Now, what remains is a collection of argument positions that are bound positions of the predicates in B, such that the values assigned to them cannot changed the values in any argument position outside this collection (of bound arguments of predicates in B). They can only affect the set of values in the bound arguments that occur in some tuple along with a given set of values for the free arguments. That is, if we generate all possible tuples for predicates in block B using these rules, and then take the projection of the free argument positions, the result is identical to first deleting all bound arguments from these rules and then computing all tuples. Since we are only interested in values that appear in free arguments, we can therefore delete the bound arguments entirely. []

We refer to the above optimization as the semijoin optimization, because the intuition essentially is that we perform a sequence of joins proceeding from one direction and at each stage projecting out unnecessary columns. Since the join is recursively defined, we can only do this by taking advantage of the indices.

Example 8:

The semijoin optimization applies to all occurrences of sg_ind^{bf} in the rewritten rules produced by the Generalized Counting algorithm. In particular, it applies to the second occurrence of sg_ind^{bf} in modified rule 2. So we delete all body literals to the left of this literal in optimizing the above rules using the semijoin optimization:

$cnt\_sg\_ind^{bf}$(I+1, k*2+2, h*5+2, Z1) :- $cnt\_sg\_ind^{bf}$(I, k, h, X), up(X,Z1)
[From rule 2, second body literal]

$cnt\_sg\_ind^{bf}$(I+1, k*2+2, h*5+4, Z3) :- sg_ind^{bf}(I+1, k*2+2, h*5+2, Z2),
flat(Z2,Z3) [From rule 2, fourth body literal]

sg_ind^{bf}(I, k, h/5, Y) :- $cnt\_sg\_ind^{bf}$(I, k, h/5, X), flat(X,Y) [Modified rule 1]

sg_ind^{bf}(I, k, h/5, Y) :- sg_ind^{bf}(I+1, k*2+2, h+4, Z4), down(Z4,Y)
[Modified rule 2]

$cnt\_sg\_ind^{bf}$(0, 0, 0, john) [From the query rule]

It is possible to reduce the number of indices in some cases. The first index can be recovered from the other two, and is only included for convenience. Further, if in each rule, there is at most one occurrence of a given derived predicate, then we do not need the third index, which encodes which predicate occurrence in a rule body was expanded. Similarly, if there is only one recursive rule, the second index, which encodes the number of the recursive rule used to expand a literal, can be omitted. These refinements are straightforward, and we do not discuss them here.

9. On the Power of Magic

In this section, we present some results characterizing the Generalized Magic Sets rule rewriting algorithm with respect to other strategies for implementing a sip. Recall our assumption (Section 1) that the predicates in a rule form a connected component. This is, effectively, an assumption that we begin each computation by first evaluating, for each rule, the components that are not connected to the head. If the component can be satisfied, we delete it from the rule body, else we discard the rule. This seems to be a reasonable assumption about how an intelligent strategy would work, and the results in this section are subject to it.

Our main result concerns the optimality of the Generalized Magic Sets strategy, in the sense that it implements a given sip by computing a minimal number of facts. We first define the class of strategies for which this claim of optimality is made. Essentially, this definition seeks to capture the work that must be done to establish that every answer has been computed; and to preclude strategies that behave like ‘‘oracles’’, in that they work with knowledge other than the logical consequences of the rules and the facts in the database. (In doing so, it also precludes some strategies that work only with the logical consequences of the program, but do not proceed by simply generating subgoals. We discuss this point further after Theorem 9.1.) It limits consideration to strategies that follow the given set of rules, as per the given collection of sips, and only utilize fully bound arguments. (Note that it is possible to extend the notion of a sip to include strategies that utilize partially bound arguments, as is done in [Ramakrishnan 88].)

Accordingly, we define a *sip-strategy* for computing the answers to a query expressed using a set of Datalog rules, and a set of sips, one for each adornment of a rule head, as follows. We use the notation $p(\overline{c},\overline{X})$ to denote a query with predicate name *p*, a list of bound arguments $\overline{c}$, and a list of free arguments $\overline{X}$. We assume that a strategy constructs queries, and for each query it constructs answers by computing facts. The set of queries and the set of facts generated during a computation must satisfy certain conditions, which express the fact that the strategy follows the sips in computing the answer.

A sip-strategy takes as input

i. A query, and

ii. A program with a collection of sips, where for each rule, there is exactly one sip per head adornment.

The computation must satisfy the following conditions:

1. If $p(\overline{c},\overline{X})$? is a query, and $p(\overline{c},\overline{d})$ holds, then $p(\overline{c},\overline{d})$ is computed.
2. If $p(\overline{c},\overline{X})$? is a query, then for every rule with head predicate p, a query is constructed for every predicate in the rule body according to the sip for the rule.

A sip-strategy is initially called with the given set of rules, the facts in the database, and the given query. The first condition requires that it computes all answers to each query that it generates. The second condition describes how answers are generated for a query. For every rule head matching the query, we invoke the rule, thus determining an adornment, and selecting a sip to follow. Next, the rule's body is evaluated. For every body literal, subqueries are generated according to the sip. That is, each subquery contains values for the bound arguments that are passed through the sip arcs entering the node corresponding to that literal. For each subquery generated, there is a set of answers. These are used to pass bindings, as per the sip, to create additional subqueries. By combining the answers to all these subqueries, we generate answers for the original query involving the rule head.

In defining adornments and passing bindings, an argument *must* be considered bound if all variables in it are bound, and an argument is considered free if any variable in it is free. The latter restriction essentially limits us to the class of strategies that make no use of partially instantiated arguments. The strategy commonly used by Prolog is an example that does not belong in this class. However, if we consider only Datalog programs, this distinction does not arise, and our definition of a strategy includes all methods that infer facts solely by following the rules.

A *sip-optimal* strategy is defined to be a sip-strategy that generates only the facts and the queries required by the above definition for the predicates in the program. Sip-optimality does not imply that facts and queries are not generated more than once, or that the computation is efficient in the resources that it consumes. However, we do believe that it is an important property of a strategy.

We have the following theorem.

Theorem 9.1: Consider a query over a set of connected rules P, where a sip is associated with each adornment of a rule's head. Let P^{mg} be the set of rewritten rules produced by the Generalized Magic Sets method. The bottom-up evaluation of P^{mg} is sip-optimal.

Proof: Denote the collections of queries and facts in conditions 1 and 2 in the definition of a method by Q and F respectively.

Let us consider a bottom-up computation of P^{mg}. First, we need to define the facts generated in such a computation. We use the following definition. The magic seed is a generated fact. Suppose that f_1, ... f_m are generated facts corresponding to derived predicates in the body of a rule, and g_1, ... , g_l are facts in base predicates in the body, such that the body is satisfied and generates the fact f for the head. Then f is also a generated fact.

It remains to show that every fact generated for a predicate in a bottom-up computation of P^{mg} is an answer to a query in Q, or denotes a query in Q. More precisely, we claim that for each generated fact, if it is a magic fact $magic\_p(\overline{c})$ or a fact $p^a(\overline{c},\overline{d})$, then there exists a query $p^a(\overline{c},\overline{X})$? in Q.

The proof is by induction on the number of steps needed to derive the fact. This number is 0 for the seed. If a rule with derived facts $f_1, \ldots, f_m$ (and possibly some base facts) in the body is used to derive a fact f, then the number for f is the maximal number for any of the f_i plus 1.

For the basis of the induction, we have the seed, $magic\_q^e(\overline{c})$, which corresponds to the given query.

Suppose the claim holds for all facts generated in N or fewer steps. Consider a fact f, generated using a rule r with derived facts $f_1, \ldots, f_m$ in the body that are all derived in N or fewer steps. If the head of this rule is a non-magic predicate p^a, then one of the f_i, say f_1, must be $magic\_p^a(\overline{c})$, and the fact f must be $p^a(\overline{c},\overline{d})$. By induction, since $magic\_p^a(\overline{c})$ is a fact generated in N or fewer steps, it corresponds to a query $p^a(\overline{c},\overline{X})$? in Q. The fact f, which is derived in N+1 steps, also corresponds to this query.

If the fact f is a magic fact $magic\_p^a(\overline{c})$, then consider the adorned rule in P^{ad}, say $r1$, and sip $N \to p$, that generated the magic rule r defining $magic\_p^a$. By construction of the magic rule, if q is a predicate in N, and corresponds to the literal $q^{a1}(theta)$ in $r1$, then $q^{a1}(\theta)$ and $magic\_q^{a1}(\theta^b)$ appear in the body of $r^\dagger$. Since each of the facts f_i corresponds to a query in Q or an answer to a query in Q, by the hypothesis, it follows by construction of the magic rule that $\overline{c}$ must be passed into the node denoting p according to the sip. By condition (2) in the definition of a method, the query $p^a(\overline{c},\overline{Y})$ must be in Q, This completes our proof. []

We consider the significance of the result. First, our definition tries to capture the intuitive idea of a strategy that evaluates a program using a given sip collection. A method that does not generate some of these queries or facts cannot be considered as using the given collection of rules and sips. For if it does, then there must be a stage in the computation (corresponding to the missing queries or facts) where it is ‘‘guessing’’, or using an oracle. A strategy may generate additional queries, or facts, in addition to those that must be generated by conditions 1 and 2, and then we have good reason to consider it inferior to the Generalized Magic Sets strategy.

We remark that there exist methods in the literature that are sometimes better than Generalized Magic (e.g. [Chang 81, Henschen and Naqvi 84]) in terms of the number of distinct facts generated. These usually work only for certain classes of programs, and do not proceed by invoking subgoals using the program rules and sips. Utilizing special knowledge about the form of rules, these methods do not follow the rules, but rather, use special techniques. For example, in the ancestor example, one may use an infinite expansion, expressing *anc* as the union of powers of *par*. This infinite union can be evaluated in finite time, by using a suitable halting condition. Thus, no query on *anc* is generated. If *john* has n ancestors, then such methods compute only n facts, namely the relationships of *john* to his ancestors, whereas our method computes n^2 facts, the relationships of each ancestor to his/her ancestor For methods that are generally applicable, our notion of a sip-strategy seems reasonable. Such methods include QSQ [Vieille 85], Extension Tables [Dietrich and Warren 85], Apex [Lozinskii 85], Static and Dynamic Filtering [Kifer and Lozinskii 86, 87], Prolog (on Datalog programs) and several parallel evaluation strategies proposed in the logic programming literature (on Datalog programs) [Kale 86, etc.] Thus, our definition of a method is sufficiently general to include a significant class of proposed strategies.

In the definition above, we have not included details of the implementation We make no claims about the number of times a fact is (re)computed, and in fact, it is in this area that the approach must be refined. The other variants of the Magic Sets and Counting strategies presented in this paper attempt to address this issue. We also do not consider the differing costs of inferring a fact or generating a subquery under different evaluation strategies. Depending on the implementation, a strategy may ‘‘generate queries’’ by introducing auxiliary predicates and computing additional facts in these predicates. This is the case for the Generalized Magic Sets method. Thus, the cost of generating a subquery is one fact inference. In other implementations, subqueries may be generated by maintaining a variable environment, as, for example, in Prolog. These quite different costs associated with the generation of a subquery (and, similarly, the generation of a fact) are not captured in the notion of sip-optimality. As an approximation, we may choose to simply count the number of facts produced, and ignore other costs. This favors strategies such as Prolog since the cost of generating queries is not measured, at the expense of strategies like Generalized Magic Sets, which generate additional facts (the facts in magic predicates) in order to generate subqueries. This was, in fact, the approach taken in [Bancilhon and Ramakrishnan 87], and the results of that study indicate that the number of magic facts is, in general, a small fraction of the generated facts.

We note that any method proceeding according to a given collection of sips must evaluate all queries in Q and all facts in F, and the Magic Set method does not generate any facts that do not correspond to queries in Q or facts in F. Since the convergence of the fixpoint evaluation of the rewritten program is assured if this set of facts is finite, we have the following corollary.

Corollary 9.2: Consider a query over a set of connected rules P, where a sip is associated with each rule. Let P^{mg} be the set of rewritten rules produced by the Generalized Magic Sets method. The bottom-up evaluation of P^{mg} is safe if any safe method exists for evaluating P according to the associated sips. []

Let s_1 and s_2 be two sips for a given rule. We say that $s_1 > s_2$ if for every arc $N \dashrightarrow p$ in s_2 with label χ, we have an arc $M \dashrightarrow p$ in s_1 with label ϕ, where $N \subseteq M$ and $\chi \subseteq \phi$. So s_2 is a partial sip, and s_1 is a (partial or full) sip that does all the information passing in s_2 (and possibly more). For a given set of bound arguments in the head of the rule, a full sip is thus a sip s_1 such that for every possible sip s_2 with the same set of bound arguments in the head, $s_1 > s_2$. We have the following lemma.

Lemma 9.3: Given a connected rule, and two sips s_1 and s_2 for this rule, the set of facts computed by a sip-optimal strategy for s_1 is contained in the set of facts computed by a sip-optimal strategy for s_2 if $s_1 > s_2$.

Proof: Let us denote the set of facts and queries generated using sip s_i, i = 1,2, by Fi and Qi respectively. By examining the subqueries generated from the given rule using either sip s_1 or s_2, we show that for every subquery $q^{a1}(\overline{c1},\overline{X1})$? generated using s_1, there is a subquery $q^{a2}(\overline{c2},\overline{X2})$? generated using s_2 where the set of bound arguments in $a2$ is a subset (possibly a proper subset) of the set of bound arguments in $a1$, and the values of the bound arguments in $\overline{c2}$ are identical to the values of the corresponding arguments in $\overline{c1}$. Let $N1 \rightarrow q$ be the arc entering q in s_1 and let $N2 \rightarrow q$ be the arc entering q in s_2, and let the labels be ϕ_1 and ϕ_2 respectively. If the set of arguments of q bound by the variables in ϕ_1 is strictly larger than the set of arguments bound by the variables in ϕ_2, then the set of bound arguments in $a2$ is a proper subset of the set of bound arguments in $a1$. The converse cannot be the case since $s_1 > s_2$. Further, since every predicate in N_2 also appears in N_1, each vector of bindings, say $v1$, for variables in ϕ_1 computed using s_1 has a corresponding vector of bindings, say $v2$, computed for the variables in ϕ_2 using s_2 such that $v2$ is a projection of $v1$. Thus, if a magic fact $magic\_q^{a1}(\overline{c1})$ is generated using s_1, a magic fact $magic\_q^{a2}(\overline{c2})$ is also generated if the computation is carried out using s_2. The corresponding subqueries are therefore added to Q1 and Q2 respectively.

Since every fact that is an answer to $q^{a1}(\overline{c1},\overline{X1})$? is also an answer for $q^{a2}(\overline{c2},\overline{X2})$?, this concludes the proof. []

10. Safety

We now consider the issue of safety, that is, does the bottom up evaluation of the rewritten rules terminate after computing all answers? In the previous section, we observed that this is indeed the case for the Magic Sets method if there exists any safe implementation of the program according to the given sips. Thus, the rewriting algorithms we discussed are really orthogonal to the issue of safety. The problem of safety can thus be stated at the level of sip collections, and this provides a more general approach to safety. However, this does not tell us whether such a program exists. (Also, such a result does not hold for the Counting method due to the fact that the index fields may grow indefinitely - for example, if the data contains cycles.) In this section, we present some sufficient conditions for recognizing that the bottom up evaluation of a rewritten set of rules is safe.

We first present a generalization of the safety condition described in [Sacca and Zaniolo 86b]. The *binding graph* of a query is defined to be a directed graph whose nodes are adorned predicates p^a. We draw an arc $[r_i, j]$ from $p_1^{a_1}$ to $p_2^{a_2}$ if $p_1^{a_1}$ is the head of the ith adorned rule and $p_2^{a_2}$ is the jth predicate in the body. The root of the graph is the query node q^a.

We define the *length* of a term t, denoted |t|, to be 1 if t is a constant, and to be the sum of the lengths of the arguments of t plus 1, if t is an n-ary term $f(t_1, \ldots, t_n)$. This allows us to compute the length of constant terms. If variables are present, we can express the length of the term in terms of the lengths of these variables. In general, we have no information about the length of a variable X, except that |X| ≥ 1. For example, |X.X| = |X| + |X| + 1 = 2 |X| + 1. Thus, in general, |X.X| ≥ 3.

Let $(p_1^{a_1}, p_2^{a_2})$ be an arc in the binding graph. The *arc length* of this arc is defined to be the difference between the total length of the bound arguments in $p_1^{a_1}$ and the total length of the bound arguments in $p_2^{a_2}$. The length of a path is defined to be the sum of the lengths of its arcs.

We have the following theorem.

Theorem 10.1: The Generalized Magic Sets and Counting algorithms terminate after computing all answers if the length of every cycle in the binding graph associated with the query is positive.

Proof: This result is established for the Magic Sets and Counting algorithms in [Sacca and Zaniolo 86b]. The above generalization of their result covers sips that could not be handled by those rewriting algorithms. The proof is a straightforward extension of their proof, and is omitted. []

Often, we have some information on the upper bound of the length of a variable. For example, if b is a base relation containing only constants, and $b(X)$ appears in the body of a rule, |X| = 1. If b is a base relation containing terms, we may know that the size of terms in b is less than some number n. In that case, |X| < n. This kind of information is often critical in determining safety, as pointed out by Sacca and Zaniolo.

Now consider the case when queries are expressed over Datalog rules. The above theorem does not apply since all cycle lengths become 0. We have, however, the following theorem.

Theorem 10.2: The Magic Sets strategies are safe for Datalog programs.

Proof: This follows immediately from the fact that the number of all possible facts that can be constructed using the constants in the query and in tuples of base predicates is finite. Thus the number of possible derived and magic facts is finite. []

The above theorem does not hold for the Counting strategies. It is well known that the Counting strategies may not terminate if the data is cyclic, since the same value may be computed periodically at several levels (of indexing).

There are also some Datalog rules for which the Counting strategies will not terminate, regardless of the data. Consider a Datalog program. Construct the binding graph for the query. Construct an *argument* graph from the binding graph as follows. Consider an arc $p_1^{a_1} \rightarrow p_2^{a_2}$ in the binding graph, with label $[r_i, j]$. If a variable X appears in the mth argument of p_1 and the nth argument of p_2, and these are bound arguments, then add the arc $p_1^{a_1}(m) \rightarrow p_2^{a_2}(n)$ to the argument graph. The adorned query predicate is the root. We have the following theorem.

Theorem 10.3: The Generalized Counting and Generalized Supplementary Counting strategies will not terminate for Datalog programs with cyclic reachable argument graphs.

Proof: This follows from the observation that the magic fact corresponding to the query is repeatedly generated with monotonically increasing indices by traversing some cycle in the argument graph. The reader is referred to the non-linear ancestor example. []

11. Discussion

We have presented the following rule rewriting strategies:

1. Generalized Magic Sets (GMS)
2. Generalized Supplementary Magic Sets (GSMS)
3. Generalized Counting (GC)
4. Generalized Supplementary Counting (GSC)

We have also presented an important optimization called the semijoin optimization.

In this section, we discuss their relative merits informally. The main point we make is that for each of these strategies, with or without semijoin optimizations, there is some set of rules and data such that it is the best strategy. Therefore, we need to consider all of them in deciding on a rule rewrite strategy.

In the following discussion, we refer to the strategies by their acronyms. GMS suffers from the fact that it duplicates the work it does in computing the magic sets when computing the corresponding predicates (that is, when firing the modified rules). GC suffers from the same drawback.

This problem is addressed in GSMS and GSC by storing all results that are potentially useful later on. Thus, they tradeoff additional memory (and possibly, increased lookup times) for the time gained in avoiding some duplicate firings of rules.

GC and GSC refine the notion of a *relevant fact* by essentially numbering the magic sets. This means that they avoid many unnecessary firings by starting at the query node and working outwards. They do this at the cost of maintaining a system of indices (and of course, are applicable only for a restricted set of data and rules).

The semijoin optimization offers two benefits. It reduces the number of joins (by deleting some literals) in the optimized rule, and reduces the width (number of arguments) of the optimized predicate. It is a powerful optimization that could significantly improve performance. If the semijoin optimization is not applicable for an occurrence of an adorned predicate, it might become applicable if we consider some of the bound arguments to be free. (For example, if there is a variable in a bound argument of a body literal that also appears to the right of the literal, the conditions for applying the semijoin optimization are violated. If we consider the argument to be free, this violation is removed.) This might lead us to choose a partial sip: We use a less restrictive counting set (which leads to more duplicate firings of rules), and in exchange, we obtain the benefits of the semijoin optimization.

We now consider an important problem associated with GC and GSC. The indices essentially encode the path in the derivation tree by which a fact is inferred. If there are r recursive rules and l literals per rule, in the worst case, there are $(r.l)^n$ derivation paths of length n. Since the same fact could be generated using different derivations, there could be a large number of facts that agree on the non-index fields but have differing values in the index fields. This has two important consequences. First, the number of facts inferred could be much larger than that for the Magic Set methods (or even Naive). Second, if there are two facts that differ only on the index fields, there could be a cycle in the data, thus leading to an unbounded number of such facts. (On the other hand, there might be just two distinct and acyclic paths to the same fact, but in general it is not possible to distinguish the two cases.) It may thus be appropriate to use Counting only when there are no pairs of facts that disagree only on the index fields, that is, there is a unique derivation for each fact in the corresponding Magic Sets program. (Further, under these conditions, we would expect Counting to improve on Magic Sets only if several counting facts are produced by the same derivation path. To see this, consider a simple example: If each counting fact generates m facts as ‘‘answers’’, and there are n counting facts generated by a given derivation path (i.e. the same values in the index fields), then $m.n$ facts in the Magic Sets version are represented by the n counting facts and m facts with the same indices in the corresponding adorned predicate.) This issue is addressed further in [Sacca and Zaniolo 87]. Another approach is to use a dynamic encoding of paths instead of the static encoding of index fields that we have used. Vieille has suggested such a scheme in [Vieille 88], in conjunction with a different evaluation method, where (in effect) each magic fact is given a unique identifier when it is generated (along with a pointer to the magic fact that was used to generate it). Each non-magic fact is given the same identifier as the magic fact that was used to generate it. Finally, properties such as commutativity of rules can be used to reduce the number of paths that must be encoded.

Finally, in GMS and GC, in the rules defining the cnt and magic predicates, we may drop some of the literals in the body (so long as the remaining literals contain all variables that appear in the head) without altering the correctness of the rewriting algorithm. This would cause us to compute a larger relation for the corresponding cnt or magic predicate, that is, be less selective about the rules we fire subsequently. However, if the increase is not significant, it might be worthwhile to drop some of the literals to save on the number of joins.

There are many possible variations of the rewrite strategy, and it is important to understand how to choose between them. This is an important problem that needs to be addressed.

An important issue is the generalization of these algorithms for dealing with negation in rule bodies, and we address this in [Beeri et al. 87]. The problem is also studied in [Balbin et al. 87].

12. Acknowledgements

We wish to thank Francois Bancilhon for stimulating our interest in this work, and Oded Shmueli, Jeffrey Ullman and Carlo Zaniolo for many useful discussions.

13. References

[Balbin et al. 87]

‘‘Magic Set Computation for Stratified Databases,’’ I. Balbin, G.S. Port and K. Ramamohanarao, *Dept. of Computer Science, University of Melbourne, Australia, TR 87/3.*

[Bancilhon 85]
‘‘A Note on the Performance of Rule Based Systems,’’ F. Bancilhon, *MCC Technical Report DB-022-85, 1985.*

[Bancilhon et al. 86]
‘‘Magic Sets and Other Strange Ways to Implement Logic Programs,’’ F. Bancilhon, D. Maier, Y. Sagiv and J. Ullman, *Proc. 5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems, 1986.*

[Bancilhon and Ramakrishnan 86]
‘‘An Amateur’s Introduction to Recursive Query Processing Strategies,’’ F. Bancilhon and R. Ramakrishnan, *Proc. SIGMOD, Invited paper, 1986.*

[Bancilhon and Ramakrishnan 87]
‘‘Performance Evaluation of Data Intensive Logic Programs,’’ F. Bancilhon and R. Ramakrishnan, *To appear in Foundations of Deductive Databases and Logic Programming, Ed. J. Minker, Morgan Kaufman, 1987.*

[Beeri et al. 87]
‘‘Sets and Negation in a Logic Database Language (LDL1),’’ C. Beeri, S. Naqvi, R. Ramakrishnan, O. Shmueli and S. Tsur, *Proc. 6th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems, 1987.*

[Chang 81]
‘‘On the Evaluation of Queries Containing Derived Relations in Relational Databases,’’ C. Chang, In *Advances in Data Base Theory, Vol.1, H. Gallaire, J. Minker and J.M. Nicolas, Plenum Press, New York, 1981, pp 235-260.*

[Gallaire et al. 84]
‘‘Logic and Data Bases: A Deductive Approach,’’ H. Gallaire, J. Minker and J.-M. Nicolas, *Computing Surveys, Vol. 16, No 2, June 1984.*

[Henschen and Naqvi 84]
‘‘On Compiling Queries in Recursive First-Order Data Bases,’’ L. Henschen and S. Naqvi, *JACM, Vol 31, January 1984, pp 47-85.*

[Kifer and Lozinskii 85]
‘‘Query Optimization in Logic Databases,’’ M. Kifer and E. Lozinskii, *Technical Report, SUNY at Stonybrook, June 1985.*

[Kifer and Lozinskii 86] ‘‘ A Framework for an Efficient Implementation of Deductive Databases,’’ M. Kifer and E. Lozinskii, *Proc. Advanced Database Symposium, Tokyo, 1986.*

[Lloyd 84]
‘‘Foundations of Logic Programming,’’ J. W. Lloyd, *Springer-Verlag, 1984.*

[Lozinskii 85]
‘‘Evaluating Queries in Deductive Databases by Generating,’’ E. Lozinskii, *Proc. 11th*

International Joint Conference on Artificial Intelligence, 1985.

[McKay and Shapiro 81]
‘‘Using Active Connection Graphs for Reasoning with Recursive Rules,’’ D. McKay and S. Shapiro, *Proc. 7th International Joint Conference on Artificial Intelligence, 1981.*

[Ramakrishnan 88]
‘‘Magic Templates: A Spellbinding Approach to Logic Programs,’’ R. Ramakrishnan, *Proc. Joint Symposium on Logic Programming and International Conference on Logic Programming, 1988.*

[Rohmer and Lescoeur 85]
‘‘La Methode Alexandre: une solution pour traiter les axiomes recursifs dans les bases de donnees deductives ,’’ Rohmer and Lescoeur, *Colloque Reconnaissance de Formes et Intelligence Artificielle, Grenoble, November 1985.*

[Roussel 75]
‘‘PROLOG, Manuel de Reference et de Utilisation,’’ P. Roussel, *Groupe Intelligence Artificielle, Universite Aix-Marseille II, 1975.*

[Sacca and Zaniolo 86a]
‘‘On the Implementation of a Simple Class of Logic Queries for Databases,’’ D. Sacca and C. Zaniolo, *Proc. 5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems, 1986.*

[Sacca and Zaniolo 86b]
‘‘The Generalized Counting Method for Recursive Logic Queries,’’ D. Sacca and C. Zaniolo, *Proc. First International Conference on Database Theory, 1986.*

[Sacca and Zaniolo 87]
‘‘Magic Counting Methods,’’ D. Sacca and C. Zaniolo, *Proc. SIGMOD, 1987.*

[Ullman 85]
‘‘Implementation of Logical Query Languages for Databases,’’ J. Ullman, *TODS, Vol. 10, No. 3, pp. 289-321, 1985.*

[Van Gelder 86]
‘‘A Message Passing Framework for Recursive Query Evaluation,’’ A. Van Gelder, *Proc. SIGMOD, 1986.*

[Vieille 86]
‘‘Recursive axioms in Deductive Databases: The Query/Subquery Approach,’’ L. Vieille, *Proc. First Intl. Conference on Expert Database Systems, 1986.*

[Vieille 88]
‘‘From QSQ towards QoSaQ: Global Optimization of Recursive Queries,’’ L. Vieille, *Proc. Second Intl. Conference on Expert Database Systems, 1986.*

Appendix: Examples

14. The Problems

1. The ancestor example:

    a(X,Y) :- p(X,Y)

    a(X,Y) :- p(X,Z), a(Z,Y)

    Query: a(john,?)

2. Non-linear version of the ancestor example:

    a(X,Y) :- p(X,Y)

    a(X,Y) :- a(X,Z), a(Z,Y)

    Query: a(john,?)

3. Nested version of the same generation example:

    p(X,Y) :- b1(X,Y)

    p(X,Y) :- sg(X,Z1), p(Z1,Z2), b2(Z2,Y)

    sg(X,Y) :- flat(X,Y)

    sg(X,Y) :- up(X,Z1), sg(Z1,Z2), down(Z2,Y)

    Query: p(john,?)

4. List Reverse:

    append(V, [], V|[]) :-

    append(V, W|X, W|Y) :- append(V,X,Y)

    reverse([], []) :-

    reverse(V|X, Y) :- reverse(X,Z), append(V,Z,Y)

    Query: reverse(list, ?)

15. The Adorned Rule Sets

The following are the sets of adorned rules for each of the example problems. The literals in each adorned rule have been ordered so as to make the rule canonical.

The ancestor example:

1. a^{bf}(X,Y) :- p(X,Y)
2. a^{bf}(X,Y) :- p(X,Z), a^{bf}(Z,Y)

Query: a^{bf}(john,?)

Non-linear version of the ancestor example:

1. a^{bf}(X,Y) :- p(X,Y)
2. a^{bf}(X,Y) :- a^{bf}(X,Z), a^{bf}(Z,Y)

Query: a^{bf}(john,?)

Nested version of the same generation example:

1. p^{bf}(X,Y) :- b1(X,Y)
2. p^{bf}(X,Y) :- sg^{bf}(X,Z1), p^{bf}(Z1,Z2), b2(Z2,Y)
3. sg^{bf}(X,Y) :- flat(X,Y)
4. sg^{bf}(X,Y) :- up(X,Z1), sg^{bf}(Z1,Z2), down(Z2,Y)

Query: p^{bf}(john,?)

List Reverse:

1. $append^{bbf}$(V, [], V|[]) :-
2. $append^{bbf}$(V, W|X, W|Y) :- $append^{bbf}$(V,X,Y)
3. $reverse^{bf}$([], []) :-
4. $reverse^{bf}$(V|X, Y) :- $reverse^{bf}$(X,Z), $append^{bbf}$(V,Z,Y)

Query: $reverse^{bf}$(list, ?)

16. Generalized Magic Sets (GMS)

We present the rewritten set of rules for each of the example problems, using the Generalized Magic Sets algorithm.

16.1. Ancestor

$magic\_a^{bf}$(Z) :- $magic\_a^{bf}$(X), p(X,Z) [From rule 2]
a^{bf}(X,Y) :- $magic\_a^{bf}$(X), p(X,Y) [Modified rule 1]
a^{bf}(X,Y) :- $magic\_a^{bf}$(X), p(X,Z), a^{bf}(Z,Y) [Modified rule 2]
$magic\_a^{bf}$(john) [From the query rule]

16.2. Non-linear Ancestor

$magic\_a^{bf}$(X) :- $magic\_a^{bf}$(X) [From rule 2, first body literal; can be deleted]
$magic\_a^{bf}$(Z) :- $magic\_a^{bf}$(X), a^{bf}(X,Z) [From rule 2, second body literal]
a^{bf}(X,Y) :- $magic\_a^{bf}$(X), p(X,Y) [Modified rule 1]
a^{bf}(X,Y) :- $magic\_a^{bf}$(X), a^{bf}(X,Z), a^{bf}(Z,Y) [Modified rule 2]

$magic\_a^{bf}$(john) [From the query rule]

16.3. Nested Same Generation

$magic\_p^{bf}$(Z1) :- $magic\_p^{bf}$(X), sg^{bf}(X,Z1) [From rule 2]
$magic\_sg^{bf}$(X) :- $magic\_p^{bf}$(X) [From rule 2]
$magic\_sg^{bf}$(Z1) :- $magic\_sg^{bf}$(X), up(X,Z1) [From rule 4]
p^{bf}(X,Y) :- $magic\_p^{bf}$(X), b1(X,Y) [Modified rule 1]
p^{bf}(X,Y) :- $magic\_p^{bf}$(X), sg^{bf}(X,Z1), p^{bf}(Z1,Z2), b2(Z2,Y) [Modified rule 2]
sg^{bf}(X,Y) :- $magic\_sg^{bf}$(X), flat(X,Y) [Modified rule 3]
sg^{bf}(X,Y) :- $magic\_sg^{bf}$(X), up(X,Z1), sg^{bf}(Z1,Z2), down(Z2,Y) [Modified rule 4]
$magic\_p^{bf}$(john) [From the query rule]

16.4. List Reverse

$magic\_append^{bbf}$(V,X) :- $magic\_append^{bbf}$(V, W|X) [From rule 2]
$magic\_append^{bbf}$(V,Z) :- $magic\_reverse^{bf}$(V|X), $reverse^{bf}$(X,Z) [From rule 4]
$magic\_reverse^{bf}$(X) :- $magic\_reverse^{bf}$(V|X) [From rule 4]
$append^{bbf}$(V, [], V|[]) :- $magic\_append^{bbf}$(V, []) [Modified rule 1]
$append^{bbf}$(V, W|X, W|Y) :- $magic\_append^{bbf}$(V, W|X), $append^{bbf}$(V,X,Y)
[Modified rule 2]
$reverse^{bf}$([], []) :- $magic\_reverse^{bf}$([]) [Modified rule 3]
$reverse^{bf}$(V|X, Y) :- $magic\_reverse^{bf}$(V|X), $reverse^{bf}$(X,Z),
$append^{bbf}$(V,Z,Y) [Modified rule 4]
$magic\_reverse^{bf}$(list) [From the query rule]

17. Generalized Supplementary Magic Sets (GSMS)

We present the rewritten set of rules for each of the example problems, using the Generalized Supplementary Magic Sets algorithm.

17.1. Ancestor

$supmagic_1^1$(X) :- $magic\_a^{bf}$(X) [From rule 1]
$supmagic_1^2$(X) :- $magic\_a^{bf}$(X) [From rule 2]
$supmagic_2^2$(X,Z) :- $supmagic_1^2$(X), p(X,Z) [From rule 2]

a^{bf}(X,Y) :- $supmagic_1^1$(X), p(X,Y) [Modified rule 1]

a^{bf}(X,Y) :- $supmagic_2^2$(X,Z), a^{bf}(Z,Y) [Modified rule 2]

$magic\_a^{bf}$(Z) :- $supmagic_2^2$(X,Z) [From rule 2]

$magic\_a^{bf}$(john) [From the query rule]

These rules may be optimized. The rules defining $supmagic_1^1$(X) and $supmagic_1^2$(X) may be omitted if we replace occurrences of these literals with $magic\_a^{bf}$(X). The optimized rules are given below. This optimization is discussed in the text, and in subsequent examples, we will always perform this optimization.

$supmagic_2^2$(X,Z) :- $magic\_a^{bf}$(X), p(X,Z) [From rule 2]

a^{bf}(X,Y) :- $magic\_a^{bf}$(X), p(X,Y) [Modified rule 1]

a^{bf}(X,Y) :- $supmagic_2^2$(X,Z), a^{bf}(Z,Y) [Modified rule 2]

$magic\_a^{bf}$(Z) :- $supmagic_2^2$(X,Z) [From rule 2]

$magic\_a^{bf}$(john) [From the query rule]

17.2. Non-linear Ancestor

$supmagic_2^2$(X,Z) :- $magic\_a^{bf}$(X), a^{bf}(X,Z) [From rule 2]

a^{bf}(X,Y) :- $magic\_a^{bf}$(X), p(X,Y) [Modified rule 1]

a^{bf}(X,Y) :- $supmagic_2^2$(X,Z), a^{bf}(Z,Y) [Modified rule 2]

$magic\_a^{bf}$(X) :- $magic\_a^{bf}$(X) [From rule 2, first body literal; can be deleted]

$magic\_a^{bf}$(Z) :- $supmagic_2^2$(X,Z) [From rule 2, second body literal]

$magic\_a^{bf}$(john) [From the query rule]

17.3. Nested Same Generation

$supmagic_2^2$(X,Z1) :- $magic\_p^{bf}$(X), sg^{bf}(X,Z1) [From rule 2]

$supmagic_2^4$(X,Z1) :- $magic\_sg^{bf}$(X), up(X,Z1) [From rule 4]

p^{bf}(X,Y) :- $magic\_p^{bf}$(X), b1(X,Y) [Modified rule 1]

p^{bf}(X,Y) :- $supmagic_2^2$(X,Z1), p^{bf}(Z1,Z2), b2(Z2,Y) [Modified rule 2]

sg^{bf}(X,Y) :- $magic\_sg^{bf}$(X), flat(X,Y) [Modified rule 3]

sg^{bf}(X,Y) :- $supmagic_2^4$(X,Z1), sg^{bf}(Z1,Z2), down(Z2,Y) [Modified rule 4]

$magic\_p^{bf}$(Z1) :- $supmagic_2^2$(X,Z1) [From rule 2]

$magic\_sg^{bf}$(X) :- $magic\_p^{bf}$(X) [From rule 2]

$magic\_sg^{bf}$(Z1) :- $supmagic_2^4$(X,Z1) [From rule 4]

$\mathit{magic\_p}^{bf}$(john) [From the query rule]

17.4. List Reverse

$\mathit{supmagic}_2^4$(V,X,Z) :- $\mathit{magic\_reverse}^{bf}$(V|X), $\mathit{reverse}^{bf}$(X,Z) [From rule 4]

append^{bbf}(V, [], V|[]) :- $\mathit{magic\_append}^{bbf}$(V, []) [Modified rule 1]

append^{bbf}(V, W|X, W|Y) :- $\mathit{magic\_append}^{bbf}$(V, W|X), append^{bbf}(V,X,Y)
[Modified rule 2]

$\mathit{reverse}^{bf}$([], []) :- $\mathit{magic\_reverse}^{bf}$([]) [Modified rule 3]

$\mathit{reverse}^{bf}$(V|X, Y) :- $\mathit{supmagic}_2^4$(V,X,Z), append^{bbf}(V,Z,Y) [Modified rule 4]

$\mathit{magic\_append}^{bbf}$(V,X) :- $\mathit{magic\_append}^{bbf}$(V, W|X) [From rule 2]

$\mathit{magic\_append}^{bbf}$(V,Z) :- $\mathit{supmagic}_2^4$(V,X,Z) [From rule 4]

$\mathit{magic\_reverse}^{bf}$(X) :- $\mathit{magic\_reverse}^{bf}$(V|X) [From rule 4]

$\mathit{magic\_reverse}^{bf}$(list) [From query rule]

18. Generalized Counting (GC)

We present the rewritten set of rules for each of the example problems, using the Generalized Counting algorithm.

18.1. Ancestor

$\mathit{cnt\_a\_ind}^{bf}$(I+1, k*2+2, h*2+2, Z) :- $\mathit{cnt\_a\_ind}^{bf}$(I, k, h, X), p(X,Z) [From rule 2]

$\mathit{a\_ind}^{bf}$(I, k, h/2, X,Y) :- $\mathit{cnt\_a\_ind}^{bf}$(I, k, h/2, X), p(X,Y) [Modified rule 1]

$\mathit{a\_ind}^{bf}$(I, k, h/2, X,Y) :- $\mathit{cnt\_a\_ind}^{bf}$(I, k, h/2, X), p(X,Z),
$\mathit{a\_ind}^{bf}$(I+1, k*2+2, h+2, Z,Y) [Modified rule 2]

$\mathit{cnt\_a\_ind}^{bf}$(0, 0, 0, john) [From the query rule]

The only occurrence of $\mathit{a\_ind}^{bf}$ is in the second modified rule, and the semijoin optimization applies. Thus, we have:

$\mathit{cnt\_a\_ind}^{bf}$(I+1, k*2+2, h*2+2, Z) :- $\mathit{cnt\_a\_ind}^{bf}$(I, k, h, X), p(X,Z) [From rule 2]

$\mathit{a\_ind}^{bf}$(I, k, h/2, Y) :- $\mathit{cnt\_a\_ind}^{bf}$(I, k, h/2, X), p(X,Y) [Modified rule 1]

$\mathit{a\_ind}^{bf}$(I, k, h/2, Y) :- $\mathit{a\_ind}^{bf}$(I+1, k*2+2, h+2, Y) [Modified rule 2]

$\mathit{cnt\_a\_ind}^{bf}$(0, 0, 0, john) [From the query rule]

18.2. Non-linear Ancestor

We generate the following rule:

$cnt\_a\_ind^{bf}$(I+1, k*2+2, h*2+1, X) :- $cnt\_a\_ind^{bf}$(I, k, h, X)

[From rule 2, first body literal]

So the counting strategy does not terminate in this example either.

18.3. Nested Same Generation

$cnt\_p\_ind^{bf}$(I+1, k*4+2, h*3+2, Z1) :- $cnt\_p\_ind^{bf}$(I, k, h, X),
sg_ind^{bf}(I+1, k*4+2, h*3+1, X,Z1) [From rule 2]
$cnt\_sg\_ind^{bf}$(I+1, k*4+2, h*3+1, X) :- $cnt\_p\_ind^{bf}$(I, k, h, X) [From rule 2]
$cnt\_sg\_ind^{bf}$(I+1, k*4+4, h*3+2, Z1) :- $cnt\_sg\_ind^{bf}$(I, k, h, X), up(X,Z1)
[From rule 4]
p_ind^{bf}(I, k, h/3, X,Y) :- $cnt\_p\_ind^{bf}$(I, k, h/3, X), b1(X,Y) [Modified rule 1]
p_ind^{bf}(I, k, h/3, X,Y) :- $cnt\_p\_ind^{bf}$(I, k, h/3, X), sg_ind^{bf}(I+1, k*4+2, h+1, X,Z1),
p_ind^{bf}(I+1, k*4+2, h+2, Z1,Z2), b2(Z2,Y) [Modified rule 2]
sg_ind^{bf}(I, k, h/3, X,Y) :- $cnt\_sg\_ind^{bf}$(I, k, h/3, X), flat(X,Y) [Modified rule 3]
sg_ind^{bf}(I, k, h/3, X,Y) :- $cnt\_sg\_ind^{bf}$(I, k, h/3, X), up(X,Z1),
sg_ind^{bf}(I+1, k*4+4, h+2, Z1,Z2), down(Z2,Y) [Modified rule 4]
$cnt\_p\_ind^{bf}$(0, 0, 0, john) [From the query rule]

We can verify that the semijoin optimization applies to all occurrences of p_ind^{bf} and sg_ind^{bf} in the above rules. Thus, applying the semijoin optimization, we have the optimized set of rules:

$cnt\_p\_ind^{bf}$(I+1, k*4+2, h*3+2, Z1) :-
sg_ind^{bf}(I+1, k*4+2, h*3+1, Z1) [From rule 2]
$cnt\_sg\_ind^{bf}$(I+1, k*4+2, h*3+1, X) :- $cnt\_p\_ind^{bf}$(I, k, h, X) [From rule 2]
$cnt\_sg\_ind^{bf}$(I+1, k*4+4, h*3+2, Z1) :- $cnt\_sg\_ind^{bf}$(I, k, h, X), up(X,Z1)
[From rule 4]
p_ind^{bf}(I, k, h/3, Y) :- $cnt\_p\_ind^{bf}$(I, k, h/3, X), b1(X,Y) [Modified rule 1]
p_ind^{bf}(I, k, h/3, Y) :- p_ind^{bf}(I+1, k*4+2, h+2, Z2), b2(Z2,Y) [Modified rule 2]
sg_ind^{bf}(I, k, h/3, Y) :- $cnt\_sg\_ind^{bf}$(I, k, h/3, X), flat(X,Y) [Modified rule 3]
sg_ind^{bf}(I, k, h/3, Y) :- sg_ind^{bf}(I+1, k*4+4, h+2, Z2), down(Z2,Y)
[Modified rule 4] $cnt\_p\_ind^{bf}$(0, 0, 0, john) [From the query rule]

18.4. List Reverse

$cnt\_append\_ind^{bbf}$(I+1, k*4+2, h*2+1, V,X) :- $cnt\_append\_ind^{bbf}$(I, k, h, V, W|X) [From rule 2]

$cnt\_append\_ind^{bbf}$(I+1, k*4+4, h*2+2, V,Z) :- $cnt\_reverse\_ind^{bf}$(I, k, h, V|X), $reverse\_ind^{bf}$(I+1, k*4+4, h*2+1, X,Z) [From rule 4]

$cnt\_reverse\_ind^{bf}$(I+1, k*4+4, h*2+1, X) :- $cnt\_reverse\_ind^{bf}$(I, k, h, V|X) [From rule 4]

$append\_ind^{bbf}$(I, k, h/2, V, [], V|[]) :- $cnt\_append\_ind^{bbf}$(I, k, h/t, V, []) [Modified rule 1]

$append\_ind^{bbf}$(I, k, h/2, V, W|X, W|Y) :- $cnt\_append\_ind^{bbf}$(I, k, h/2, V, W|X), $append\_ind^{bbf}$(I+1, k*4+2, h+1, V,X,Y) [Modified rule 2]

$reverse\_ind^{bf}$(I, k, h/2, [], []) :- $cnt\_reverse\_ind^{bf}$(I, k, h/2, []) [Modified rule 3]

$reverse\_ind^{bf}$(I, k, h/2, V|X, Y) :- $cnt\_reverse\_ind^{bf}$(I, k, h/2, V|X), $reverse\_ind^{bf}$(I+1, k*4+4, h+1, X,Z), $append\_ind^{bbf}$(I+1, k*4+4, h+2, V,Z,Y) [Modified rule 4]

$cnt\_reverse\_ind^{bf}$(0, 0, 0, list) [From the query rule]

19. Generalized Supplementary Counting (GSC)

We present the rewritten set of rules for each of the example problems, using the Generalized Supplementary Counting algorithm.

19.1. Ancestor

$supcnt_1^1$(I, k, h, X) :- $cnt\_a\_ind^{bf}$(I, k, h, X) [From rule 1]

$supcnt_1^2$(I, k, h, X) :- $cnt\_a\_ind^{bf}$(I, k, h, X) [From rule 2]

$supcnt_2^2$(I, k, h, X,Z) :- $supcnt_1^2$(I, k, h, X), p(X,Z) [From rule 2]

a_ind^{bf}(I, k, h, X,Y) :- $supcnt_1^1$(I, k, h, X), p(X,Y) [Modified rule 1]

a_ind^{bf}(I, k, h, X,Y) :- $supcnt_2^2$(I, k, h, X,Z), a_ind^{bf}(I+1, k*2+2, h*2+2, Z,Y) [Modified rule 2]

$cnt\_a\_ind^{bf}$(I+1, k*2+2, h*2+2, Z) :- $supcnt_2^2$(I, k, h, X,Z) [From rule 2]

$cnt\_a\_ind^{bf}$(0, 0, 0, john) [From the query rule]

These rules may be optimized. The rules defining $supcnt_1^1$(X) and $supcnt_1^2$(X) may be omitted if we replace occurrences of these literals with $magic\_a^{bf}$(X). The optimized rules are given below. This optimization is discussed in the text, and in subsequent examples, we will always perform this optimization.

$supcnt^2_2$(I, k, h, X,Z) :- $cnt\_a\_ind^{bf}$(I, k, h, X), p(X,Z) [From rule 2]

a_ind^{bf}(I, k, h, X,Y) :- $cnt\_a\_ind^{bf}$(I, k, h, X), p(X,Y) [Modified rule 1]

a_ind^{bf}(I, k, h, X,Y) :- $supcnt^2_2$(I, k, h, X,Z), a_ind^{bf}(I+1, k*2+2, h*2+2, Z,Y) [Modified rule 2]

$cnt\_a\_ind^{bf}$(I+1, k*2+2, h*2+2, Z) :- $supcnt^2_2$(I, k, h, X,Z) [From rule 2]

$cnt\_a\_ind^{bf}$(0, 0, 0, john) [From the query rule]

The only occurrence of a_ind^{bf} is in the second modified rule, and the semijoin optimization applies. Thus, we can further optimize the rules to:

$supcnt^2_2$(I, k, h, X,Z) :- $cnt\_a\_ind^{bf}$(I, k, h, X), p(X,Z) [From rule 2]

a_ind^{bf}(I, k, h, Y) :- $cnt\_a\_ind^{bf}$(I, k, h, X), p(X,Y) [Modified rule 1]

a_ind^{bf}(I, k, h, Y) :- a_ind^{bf}(I+1, k*2+2, h*2+2, Y) [Modified rule 2]

$cnt\_a\_ind^{bf}$(I+1, k*2+2, h*2+2, Z) :- $supcnt^2_2$(I, k, h, X,Z) [From rule 2]

$cnt\_a\_ind^{bf}$(0, 0, 0, john) [From the query rule]

We note that the first (non-index)argument of the *supcnt* predicate may now be dropped.

19.2. Non-linear Ancestor

We generate the following rule:

$cnt\_a\_ind^{bf}$(I+1, k*2+2, h*2+1, X) :- $cnt\_a\_ind^{bf}$(I, k, h, X)
[From rule 2, first body literal]

So the supplementary counting strategy does not terminate in this example either.

19.3. Nested Same Generation

$supcnt^2_2$(I, k, h, X,Z1) :- $cnt\_p\_ind^{bf}$(I, k, h, X),
sg_ind^{bf}(I+1, k*4+2, h*3+1, X,Z1) [From rule 2]

$supcnt^2_3$(I, k, h, X,Z2) :- $supcnt^2_2$(I, k, h, X,Z1),
p_ind^{bf}(I+1, k*4+2, h*3+2, Z1,Z2) [From rule 2]

$supcnt^4_2$(I, k, h, X,Z1) :- $cnt\_sg\_ind^{bf}$(I, k, h, X), up(X,Z1) [From rule 4]

$supcnt^4_3$(I, k, h, X,Z2) :- $supcnt^4_2$(I, k, h, X,Z1),

sg_ind^{bf}(I+1, k*4+4, h*3+2, Z1,Z2) [From rule 4]

p_ind^{bf}(I, k, h, X,Y) :- $cnt\_p\_ind^{bf}$(I, k, h, X), b1(X,Y) [Modified rule 1]

p_ind^{bf}(I, k, h, X,Y) :- $supcnt_3^2$(I, k, h, X,Z2), b2(Z2,Y) [Modified rule 2]

sg_ind^{bf}(I, k, h, X,Y) :- $cnt\_sg\_ind^{bf}$(I, k, h, X), flat(X,Y) [Modified rule 3]

sg_ind^{bf}(I, k, h, X,Y) :- $supcnt_3^4$(I, k, h, X,Z2), down(Z2,Y) [Modified rule 4]

$cnt\_p\_ind^{bf}$(I+1, k*4+2, h*3+2, Z1) :- $supcnt_2^2$(I, k, h, X,Z1) [From rule 2]

$cnt\_sg\_ind^{bf}$(I+1, k*4+2, h*3+1, X) :- $cnt\_p\_ind^{bf}$(I, k, h, X) [From rule 2]

$cnt\_sg\_ind^{bf}$(I+1, k*4+4, h*3+2, Z1) :- $supcnt_2^4$(I, k, h, X,Z1) [From rule 4]

$cnt\_p\_ind^{bf}$(0, 0, 0, john) [From the query rule]

The semijoin optimization applies to all occurrences of sg_ind^{bf} and p_ind^{bf} in the above rules, and thus we have the optimized rule set:

$supcnt_2^2$(I, k, h, Z1) :- sg_ind^{bf}(I+1, k*4+2, h*3+1, Z1) [From rule 2]

$supcnt_3^2$(I, k, h, Z2) :- p_ind^{bf}(I+1, k*4+2, h*3+2, Z2) [From rule 2]

$supcnt_2^4$(I, k, h, Z1) :- $cnt\_sg\_ind^{bf}$(I, k, h, X), up(X,Z1) [From rule 4]

$supcnt_3^4$(I, k, h, Z2) :- sg_ind^{bf}(I+1, k*4+4, h*3+2, Z2) [From rule 4]

p_ind^{bf}(I, k, h, Y) :- $cnt\_p\_ind^{bf}$(I, k, h, X), b1(X,Y) [Modified rule 1]

p_ind^{bf}(I, k, h, Y) :- $supcnt_3^2$(I, k, h, Z2), b2(Z2,Y) [Modified rule 2]

sg_ind^{bf}(I, k, h, Y) :- $cnt\_sg\_ind^{bf}$(I, k, h, X), flat(X,Y) [Modified rule 3]

sg_ind^{bf}(I, k, h, Y) :- $supcnt_3^4$(I, k, h, Z2), down(Z2,Y) [Modified rule 4]

$cnt\_p\_ind^{bf}$(I+1, k*4+2, h*3+2, Z1) :- $supcnt_2^2$(I, k, h, Z1) [From rule 2]

$cnt\_sg\_ind^{bf}$(I+1, k*4+2, h*3+1, X) :- $cnt\_p\_ind^{bf}$(I, k, h, X) [From rule 2]

$cnt\_sg\_ind^{bf}$(I+1, k*4+4, h*3+2, Z1) :- $supcnt_2^4$(I, k, h, Z1) [From rule 4]

$cnt\_p\_ind^{bf}$(0, 0, 0, john) [From the query rule]

19.4. List Reverse

$supcnt_2^4$(I, k, h, V,X,Z) :- $cnt\_reverse\_ind^{bf}$(I, k, h, V|X),
$reverse\_ind^{bf}$(I+1, k*4+4, h+2+1, X,Z) [From rule 4]

$append\_ind^{bbf}$(I, k, h, V, [], V|[]) :- $cnt\_append\_ind^{bbf}$(I, k, h, V, [])
[Modified rule 1] $append\_ind^{bbf}$(I, k, h, V, W|X, W|Y) :-
$cnt\_append\_ind^{bbf}$(I, k, h, V, W|X),
$append\_ind^{bbf}$(I+1, k*4+2, h+1, V,X,Y) [Modified rule 2]

$reverse\_ind^{bf}$(I, k, h, [], []) :- $cnt\_reverse\_ind^{bf}$(I, k, h, []) [Modified rule 3]

$reverse\_ind^{bf}$(I, k, h, V|X, Y) :- $supcnt_2^4$(I, k, h, V,X,Z),
$append\_ind^{bbf}$(I+1, k*4+4, h+2, V,Z,Y) [Modified rule 4]

$cnt\_append\_ind^{bbf}$(I+1, k*4+2, h*2+1, V,X) :- $cnt\_append\_ind^{bbf}$(I, k, h, V, W|X) [From rule 2]

$cnt\_append\_ind^{bbf}$(I+1, k*4+4, h*2+2, V,Z) :- $supcnt_2^4$(I, k, h, V,X,Z) [From rule 4]

$cnt\_reverse\_ind^{bf}$(I+1, k*4+4, h*2+1, X) :- $cnt\_reverse\_ind^{bf}$(I, k, h, V|X) [From rule 4]

$cnt\_reverse\_ind^{bf}$(0, 0, 0, list) [From query rule]