Toward an Efficient Query Processor for a Deductive Database System

Chi Hang Yim

A Thesis
in
The Department
of
Computer Science

Presented in Partial Fulfilment of the Requirements for
the Degree of Master of Computer Science at
Concordia University
Montréal, Québec, Canada

March 1991

ISBN 0-315-64720-5

Abstract

Toward an Efficient Query Processor for a Deductive Database System

Chi Hang Yim

Bottom-up processing, a popular paradigm for query processing in deductive databases, consists of two phases – rewriting and evaluation. One of the most efficient rewriting strategies is the (*generalized*) *Magic Sets* method, while the *Semi-Naive Evaluation* is one of the most efficient evaluation methods known in the sense of avoiding duplicate inferences. Recently, a substantial amount of work has been done on improving these methods. We propose a rewriting method, called *Magic Filters*, which improves on the Magic Sets method, by cutting down the size of the magic predicate, while retaining the tight filtering property of Magic Sets. Our method overcomes some serious problems experienced by other claimed improvements on Magic Sets in the literature. We also propose an evaluation method, called *Forward-Semi-Naive Evaluation*, which improves on the existing Semi-Naive Evaluation method by removing half the number of auxiliary relations required, while introducing a small overhead. Finally, we integrate *Magic Filters* and *Forward-Semi-Naive* in a way that they can take advantage of each other. We also provide analyses on the above methods and compare them with existing methods. The results show significant improvements over the existing methods.

Acknowledgments

I would like to express my sincere gratitude to Dr. V.S. Lakshmanan my thesis supervisor, for his guidance and valuable insight throughout this research. His support and patience were invaluable in the preparation of this thesis.

I thank Dr. V.S. Lakshmanan and Prof. V.S. Alagar for their financial support.

I thank R. Ramakrishnan for stimulating discussion which helped clarify certain concepts, and led to an improvement of this thesis.

I would like to express my sincere thanks to Prof. C.Y. Suen and Prof. T. Radhakrishnan for providing computer equipment to carry out the experiments in this research.

Technical support from R. Bruton, C. Grossner, and W. Wong are gratefully acknowledged.

I also like to take this opportunity to thank all my friends and colleague, in particular Y.C. Cheung, S.Y. Leung, K.L. Ma, and D. Pao for their generous support and encouragement.

This research was supported in part by a grant from the Natural Sciences and Engineering and Research Council of Canada and by a grant from the Fonds Pour La Formation De Chercheurs Et L'Aide Á La Recherche of Quebec.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Recently there has been considerable interest in *deductive databases* in both the database and AI communities, with apparently different goals. The database community, upon recognizing the inadequacy of the relational database technology for many applications, has taken an active interest in deductive databases with a view to move toward a more powerful data model allowing powerful query languages. On the other hand, the search for a viable technology for supporting knowledge base systems has sparked the interest of the AI community in this field.

Recursion is at the heart of the enhanced expressive power enjoyed by deductive database query languages. However, there is a price for this enhanced expressive power, in the form of high complexity of recursive queries. This observation has sparked substantial research into the development of efficient query processing strategies for recursive queries. The reader is referred to [Sag90,Ull89,BR86] for a survey. [BR86] provides a performance evaluation of many of these strategies. It is customary to picture query processing in deductive databases as comprising two phases: (*i*) a *rewriting phase* in which the given query program is transformed into an equivalent program which exploits and propagates the bindings in the given query and hence executes faster, and (*ii*) an *evaluation phase* in which the transformed program is evaluated bottom-up using some evaluation method.

The *(generalized) magic sets* method [BR87,Ull89] and its variants (e.g., see [Ker89,Ram88,Sek89,Vie89]) are among the most popular rewriting methods in wide use. In fact, these methods are known to be equivalent [Bry89,Ull89a] in that they all enforce the same amount of tightness in filtering out the generation of useless tuples with respect to the given query. The traditional (tacit) assumption that the size of the magic predicates will be much smaller than the size of the IDB predicates (see chapter 2 for their definition) has been questioned and examples are now known where the size of the magic predicates can dominate the overall query computation. This has led researchers to investigate ways of implementing the filtering performed by the magic predicate using a number of filters, obtained by breaking down the magic predicates (filters) over subset of the arguments of the magic predicate [Sag90,SS88]. However, a close inspection reveals that while the size of the filters may be much smaller than that of the magic predicate, the restriction imposed by the filters can be much less tight than that imposed by the magic predicate. After a careful study of this problem, we realize the loss of tightness of filtering is the result of a loss of connectivity among the individual filters which are maintained independently (see section 3.3). Two fundamental questions surrounding this problem are: *(i) when* is it desirable to dissect the magic predicate into a number of smaller filters? and (*ii*) given a situation where the magic predicate is thus broken down into smaller filters, *how* can we realize the same tightness of filtering enjoyed by Magic Sets? In general, it is impossible to completely characterize when the decomposition of the magic predicate is desirable because this issue is often data dependent. However, we show in this thesis that using a *structural approach* we can characterize those situations when it is "safe" to decompose the magic predicate. We also address the second question by showing how to "synchronize" the various filters in order to achieve the tight filtering property of magic sets. We propose a *Magic Filters* [LY91] transformation that realizes these goals.

We note that there have been numerous works on improving on the performance achieved by Magic Sets on some special subclasses of programs. These works essentially fall into two categories. Those in the first category employ conceptually different methods [Agr87,Cha81,HN88,Nau87,Nau88,RHDM86]. Those in the second, also applicable to special classes of programs, modify the Magic Sets method, possible integrating with other methods [AGNS90,Nai89,NRSU89]. Although the *Magic*

Filters method would fall into the second category, it should be emphasized that the applicability is as general as Magic Sets, and the improvement is not restricted to any subclass of programs.

On the evaluation side, *Semi-Naive Evaluation* [Ull89] is one of the most efficient evaluation methods known, in that it avoids duplicate inferences. Indeed, a nice property of Semi-Naive Evaluation is that it does not repeat inferences. The main argument in favor of bottom-up methods in the literature is their inherent set-orientedness as opposed to the tuple-oriented nature of top-down methods [Ull89a]. An interesting question in the wake of this is, "is it possible to make Semi-Naive more set-oriented?". In an independent research, [RSS90] presents a conceptual framework called *Generalized Semi-Naive Evaluation*, where the central idea is to make the facts generated during evaluation of a rule immediately available to the evaluation of subsequent rules. This has been shown to reduce the number of rule applications and cut down some other overheads under a variety of assumptions. In this thesis our concern has been the investigation of *(i)* the overheads associated with rule applications, *(ii)* the relationship between rule applications and set-orientedness, and *(iii)* the overheads associated with bookkeeping, *viz.*, in maintaining numerous temporary predicates in a Semi-Naive type of evaluation. Our objective has been to reduce the overheads and increase the extent of set-orientedness. We discuss a *Forward-Semi-Naive Evaluation* embodying these principles and compare it with Semi-Naive Evaluation.

Finally, we analyze the performance of the methods proposed in this thesis and compare their performance with that of their ancestors, using simulation. We also study the impact of data selectivity and structure on the performance of the methods. Our results bring out the savings achieved by the Magic Filters over Magic Sets over the range of data tested in the simulation. Our results also show (according to the metrics for the overheads and set-orientedness used by us) Forward-Semi-Naive Evaluation indeed reduces the overheads stated above and exhibits considerably more set-orientedness compared to Semi-Naive Evaluation.

The thesis is organized as follows. In Chapter 2, we review the preliminary notions. In Chapter 3, we describe the Magic Filters transformation with examples. We identify the problem with a basic method and show how it is solved in the improved method. Chapter 4 discusses the Forward-Semi-Naive evaluation method. In Chapter 5, we discuss the performance analysis of the methods and discuss the results. Finally, in Chapter 6, we summarize our contributions and discuss directions for further research.

Chapter 2

Background

In this chapter, we briefly review the basic notions of Datalog related to this thesis. The reader is referred to [BR86,Llo86,Ull88,Ull89] for a complete description of the fundamental concepts of deductive databases and logic programming. Datalog can be viewed as a simplified version of general Logic programming, corresponding to the language of function-free Horn clauses. A Datalog program is a collection of rules and facts which are in the form of (function-free) Horn clauses. A Horn clause has the general form H :- $G_1, \ldots, G_n$, where $H, G_1, \ldots, G_n$ are all atomic formulas of the form $p(t_1, \ldots, t_l)$, where p is a predicate symbol and t_i are *terms* in the form of constants or variables. The left hand side (H) is commonly called the *rule head* or *goal*. The right hand side is the *rule body* and the atoms G_i are called subgoals. The rule is to be interpreted as a logical implication, meaning that H is true whenever all the subgoals G_i are true. If the rule body is empty, then the clause is called a *fact*, else it is a (proper) rule. We use the following notation: predicate symbols and constants are strings starting with a lower case letter, while variables are strings starting with an upper case letter. For example, consider the following Datalog program:

$r_1 : father(john, bill)$:-

$r_2 : father(george, john)$:-

$r_3 : par(X,Y)$:- $father(X,Y)$.

$r_4 : grandpar(X,Y)$:- $par(X,Z), par(Z,Y)$.

The strings *father, par* and *gandpar* are predicate symbols, the strings *john,bill,* and *george* are constants, and the symbols X, Y and Z are variables. r_1 represents the

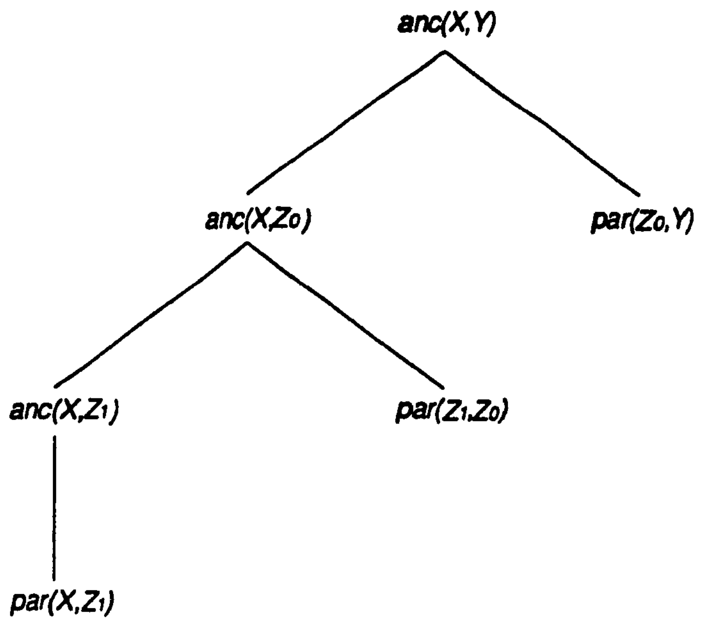


Figure 2.1: A proof tree

fact "John is the father of Bill", and similarly r_2 is the fact "George is the father of John". The other rules r_3 and r_4 represent the rules "if X is the father of Y, then X is a parent of Y", and "if X is a parent of Z and Z is a parent of Y, then X is a grandparent of Y". A rule is called *range restricted* if all variables appearing in the head also appear in the body. The range restrictedness assumption is commonly made and is a reasonable assumption necessary for ensuring that a Datalog program always generates finite answers from finite databases. This is related to the so called notion of *safety* and is discussed in detail in [Ull89]. We henceforth assume that all rules in our Datalog programs are range restricted.

There are two types of predicates in a Datalog program. They are the IDB (Intensional Database), and the EDB (Extensional Database) predicates. IDB predicates are predicates that appear in the heads of rules, while EDB predicates are predicates that appear only in rule bodies. Associated with each predicate (IDB/EDB), there is a corresponding relation, consisting of a finite set of tuples of arity corresponding to the arity of the predicate.

In a rule of the form $p(t_1, \ldots, t_l)$:- $q_1(u_1, \ldots, u_m), \ldots, q_n(v_1, \ldots, v_k)$, we say that the predicate p *depends on* each of the predicates $q_1, \ldots, q_n$. Clearly, EDB predicates do not depend on any predicate. The *depends on* relation between predicates can be transitively extended in the obvious manner. We say that a predicate is *recursive* if it depends on itself, either directly or indirectly. A Datalog program is called *recursive* if it contains at least one recursive predicate. A rule containing a subgoal which depends on the head is a *recursive rule*. Otherwise it is an *exit rule*. Given a Datalog program, defining various (IDB) predicates in terms of the database (EDB) predicates, one can query the database in terms of the predicates defined by the program using a *denial (goal)* clause of the form :- $p(t_1, \ldots, t_l)$? where p is defined in the program and t_i are variables or constants. E.g., the query :- grandpar(X,bill)? asks for the set of all grandparents of *bill*. The reader is referred to [CGT89] for an excellent informal introduction to Datalog. In the sequel, by a (logic) *query program* we mean a Datalog program together with a query. The formal semantics of Datalog program is the well known least Herbrand model semantics which is equivalent to the least fixpoint semantics [Llo86]. Thus the set of answers for a query :- $p(t_1, \ldots, t_l)$? against a program P are those tuples p which are true in the least Herbrand model of P.

Given a Datalog program P (assumed recursive without loss of generality) and predicate p defined by P, we can construct a *proof tree* for a given tuple $p(t_1, \ldots, t_l)$ as follows. Unify $p(t_1, \ldots, t_l)$ with the head of any rule defining p and generate corresponding subgoals from the rule body. If there are any IDB subgoals generated, the same process is recursively repeated. We can terminate this construction if at some point we use exit rules in the generation of subgoals, for each of the IDB subgoals generated. Consider the well-known ancestor program.

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

Here $anc(X, Y)$ denotes "X is an ancestor of Y" and $par(X, Y)$ denotes "X is a parent of Y". Fig. 2.1 shows an example proof tree for $anc(X, Y)$, of height 3. By mapping the various variables in the atoms corresponding to the leaves to constants, we actually obtain a database. It follows from the construction that the atom corresponding to each internal node of this proof tree will be derived by the ancestor Datalog program, given this database. Proof trees are used for constructing databases with special properties in Chapter 5 .

2.1 Semi-Naive Evaluation

Two basic approaches of processing logic query programs are the so-called ***top-down*** (also called ***backward chaining***) and the ***bottom-up*** (or ***forward chaining***) approaches. The reader is referred to [BR86,Ull88,Ull89,Llo86] for a through discussion of various top-down and bottom-up methods as well as a comparison of their relative efficiency. A basic edge of top-down methods over (traditional) bottom-up methods is their goal-orientedness. Recent advances in the bottom-up technology have made available numerous bottom-up methods which can effectively simulate the goal-orientedness of the top-down methods. In addition, bottom-up methods possess a set-oriented feature (*i.e.* a set of answers is generated at a time) as opposed to the tuple-oriented nature (*i.e.* one answer is generated at a time) of top-down methods. This is significant because in the database context, the number of disk accesses forms a sizeable proportion of the overall query processing cost. It has been widely recognized [Ull89a] that for database applications bottom-up methods are more advantageous than top-down methods. We limit ourselves to bottom-up evaluations in the sequel. One way to compute the least fixed point in a bottom-up fashion is to repeatedly fire all the rules until no new facts can be generated. This is known as the *Naive evaluation* method. Obviously, Naive Evaluation is an extremely inefficient method in the sense that inferences are being made repeatedly. That is, at the i^{th} iteration, all the inferences made from the 1^{st} to $(i-1)^{th}$ iteration are made over and over again. A more widely used bottom-up evaluation is the Semi-Naive [Ull89,BR86] method. The idea behind the Semi-Naive Evaluation is to "differentiate" the rules. That is, for each IDB predicate p we have an auxiliary predicate Δp to represent the change of p in the current iteration, defined as $\Delta p = p_i - p_{i-1}$, where p_i denotes the set of tuples of p generated in i iterations. We can differentiate a rule in a manner reminiscent of the product rule of differential calculus. We replace a rule

$$h \text{ :- } g_1, \ldots, g_n.$$

having one or more IDB subgoals, by one rule for each IDB subgoal. If g_i is an IDB subgoal, then we have the rule

$$\Delta h \text{ :- } g_1, \ldots, g_{i-1}, \Delta g_i, g_{i+1}, \ldots, g_n.$$

For example, consider the rule

$$p(X,Y) \text{ :- } q(X,Y), r(X,Y).$$

where q, and r are both IDB predicates We then have two rules for Δp.

$$\Delta p(X,Y) \text{ :- } \Delta q(X,Y), r(X,Y).$$
$$\Delta p(X,Y) \text{ :- } q(X,Y), \Delta r(X,Y).$$

The intuitive idea behind Semi-Naive Evaluation is to have the incremental relations at the start of iteration i contain only the new tuples generated from the $(i-1)^{th}$ iteration. The technique is summarized in Fig 2.2. The algorithm expects a collection of n safe rules with EDB relations $R_1, \ldots, R_k$, IDB relations $P_1, \ldots, P_m$, with associated incremental relations $\Delta P_1, \ldots, \Delta P_m$. We also use a set of temporary relations $Q_1, \ldots, Q_m$ to save the new results for the next iteration. The function *eval* accepts a rule, a set of EDB relations, a set of IDB relations, and a set of incremental relations as inputs and returns the result of evaluating the input rule.

2.2 The Magic Sets Method

A direct bottom-up evaluation of a query program, using Semi-Naive Evaluation is often wasteful, for most of the tuples generated may have no bearing on the query. Typically, the least fixpoint associated with a program is huge. Given a query, only tuples which constitute answers for the query are of interest. For example, consider the program in Fig. 2.3 for computing the set of same generate cousins. One may be

```
for $i := 1$ to $n$ do
begin
        if $r_i$ is *exit* rule then
          $\Delta P_j := \Delta P_j \cup \mathbf{eval}(r_i, R_1, \ldots, R_k, \emptyset, \ldots, \emptyset, \emptyset, \ldots, \emptyset)$;
          /* $P_j$ is defined by $r_i$, where $1 \leq j \leq m$ */

        $P_j := P_j \cup \Delta P_j$;
end;

repeat
         for $i := 1$ to $n$ do
         begin
                 if $r_i$ is not *exit* rule then
                    $Q_j := Q_j \cup \mathrm{eval}(r_i, R_1, \ldots, R_k, P_1, \ldots, P_m, \Delta P_1, \ldots, \Delta P_m)$;
         end;

         for $i := 1$ to $m$ do
         begin
                 $\Delta P_i := Q_i - P_i$;
                 $P_i := P_i \cup \Delta P_i$;
         end;
until $\Delta P_i = \emptyset$ for all $i$;
```

Figure 2.2: Semi-Naive Evaluation

only interested in knowing the set of same generation cousins of, say "Ann". Thus, we pose a query "$:\text{-}sgc(ann, X)?$" to the program. Thus, the program becomes a query program – a program with a query. It is desirable to rewrite a query program into a more effective form before evaluating it by a method such as the Semi-Naive Evaluation. The underlying idea of rewriting is to generate a rewritten program P_2 from a given a logic program P_1 and a query $:\text{-}G?$ such that P_2 is equivalent[1] to P_1 with respect to G, and P_2 is more efficient than P_1 when evaluated in a bottom-up fashion. One of the most popular rewriting methods is the *(generalized) Magic Sets* method, referred to simply as the *Magic Sets* method in the sequel. Before we discuss how the Magic Sets method rewrites a program with respect to a given query, we need several preliminary notions. A ***binding pattern*** of an n-ary predicate p is a string of length n over the alphabet $\{b, f\}$, where the occurrence of b (f) in a position indicates the corresponding argument of p is *bound (free).* A binding pattern of p signifies the pattern with which p is "called" during query evaluation. For instance, the binding pattern bf for the query predicate sgc denotes that the first argument is bound, and the second argument is free for predicate sgc. There are two types of information passing. The first type is the ***backward information passing*** also known as ***backward propagation*** [KL86]. This corresponds to the propagation of binding information in a predicate in a rule body to the head of a rule with the same predicate. The second type of information passing is known as ***sideways information passing*** or ***sideways propagation*** [KL86]. In this process binding information is passed from the rule head to a predicate in the rule body, or from one predicate to another predicate in the rule body via common variables. Note that sideways information passing is local to a particular rule. As an example, consider the query program of Fig. 2.3. Note that the query there corresponds to the binding pattern bf. Query evaluation begins by a backward propagation which passes on the query binding to the heads of rules r_1 and r_2 which propagate that information across the rule bodies using sideways propagation. Specifically, this leads to the binding ann to the first arguments of $person$, and par in the body of r_1 and r_2 respectively. Upon evaluation,

[1]This means that both P_1 and P_2 generate the same set of answers for the query $:\text{-}G?$.

$r_1 : sgc(X, X)$:- $person(X)$.
$r_2 : sgc(X, Y)$:- $par(X, X_1), sgc(X_1, Y_1), par(Y, Y_1)$.
:- $sgc(ann, X)$?

Figure 2.3: Same Generation Cousins Query Program

one obtains bindings for the second argument of *par* which are propagated sideways to the occurrence of *anc* which is thus called with the (same) binding pattern sgc^{bf}. The cycle repeats when this binding for *sgc* is "hooked" back to the heads of r_1 and r_2 via a backward propagation.

Basically, the Magic sets method implements this information passing by transforming the original query program to a more efficient program. This transformation can be viewed as mimicking the top-down approach which restricts the generation of *useless* tuples – tuples not relevant to the computation of the answer. We shall briefly discuss the Magic Sets method with an example shown in Fig. 2.3. In Fig. 2.4, which shows the Magic transformed program, the *magic predicate mg* can be viewed as a filter, used to restrict the generation of useless tuples. The *supplementary predicates* sup_1 and sup_2 are essentially used for an efficient realization of sideways propagation and avoiding repeated join computations. Rule r'_6 initializes the filter *mg*, and r'_1 updates the filter via backward propagation. Rules r'_2 and r'_3 implement sideways propagation while r'_4 and r'_5 compute tuples of *sgc* after applying the filters to restrict the generation of useless tuples. The reader is referred to [BMSU86,BR87,Ram88,Ull89] for a comprehensive discussion of the (generalized) Magic Sets method and its various extensions. [AGNS90,Nai89,NRSU89] describe some improvements on the Magic Sets method for the class of linear Datalog programs.

r_1' : $mg(X_1)$:- $sup_1(X, X_1)$.
r_2' : $sup_1(X, X_1)$:- $mg(X), par(X, X_1)$.
r_3' : $sup_2(X, X_1, Y_1)$:- $sup_1(X, X_1), sgc(X_1, Y_1)$.
r_4' : $sgc(X, Y)$:- $sup_2(X, X_1, Y_1), par(Y, Y_1)$.
r_5' : $sgc(X, X)$:- $mg(X), person(X)$.
r_6' : $mg(ann)$:-.

Figure 2.4: Same Generation Cousins Program Transformed Using Magic Sets

Chapter 3

The Magic Filters Method

One of the most popular methods of bottom-up query evaluation for deductive databases is the *magic sets method* [BMSU86,BR87]. Indeed the Magic Sets method, (or one of its variants [Ker89,Ram88,Sek89,Vie89]) has more or less become the standard facility for recursive query processing. It is known that the magic sets method and its variants are equivalent in the sense that they generate the same set of facts for the IDB predicates [Ull89,Bry89]. All these methods essentially mimic top-down evaluation with memoing. However, in the overall time spent on processing a query, the time for computing the magic facts should also be accounted for. While a formal analysis of this has never been made - it is intrinsically hard - it appears that there is some tradeoff between the efficiency of computation of the magic facts and the effectiveness of the restriction imposed by the magic predicates.

Kifer and Lozinskii [KL86] proposed a framework for efficient evaluation of recursive queries, which introduces the idea of restricting data flow using *dynamic filtering*. In [KL86], corresponding to each argument of each predicate, there is a filter which is updated dynamically at run time. [SS88] uses a unary filter for each bound argument of each predicate. Computation of these filters can be performed relatively efficiently. However, these filters can be much less tight than the magic predicates in restricting the generation of "useless" tuples. Sagiv [Sag90] describes the *envelope method* which uses several different envelopes to restrict the evaluation of rules. Envelopes are always of the order of the EDB in size and thus smaller than magic predicates, but are less tight than magic predicates. Although dynamic filtering [KL86] is proposed as an evaluation framework, while the methods in [SS88,Sag90] are rewriting methods, it can be seen that [SS88,Sag90] are particular realizations of the ideas presented in the general framework in [KL86].

In comparing the dynamic filtering of [KL86] with magic sets we observe that the dynamic filtering method has the advantage that the filters computed are small in size compared to magic predicates. On the other hand, the restriction imposed by these filters can be far less tight than the one imposed by magic predicates. In this chapter, we show that there is a structural way to approach this tradeoff between filter size and filter effectiveness. Specifically, we show that under certain circumstances, it is possible to keep the sizes of filters small while still preserving the effectiveness of restriction achieved by magic predicates. In the next section, we motivate our method with an example. In Section 3.2, we describe the basic magic filters method in detail. In Section 3.3, we identify the problems associated with the basic method. In Section 3.4 we discuss the improved method which overcomes these problems. Finally, we discuss some pertinent implementation issues and conclude the chapter.

3.1 Motivation

The objective of a rewriting method of query processing is to transform the given query program into a new equivalent program, which has restrictions, in the form of filters, imposed on original rules, together with rules for computing these filters. Ideally, we would like filters which are small compared to the original IDB predicates, yet these filters should be as tight as possible in restricting the generation of useless tuples for the IDB predicates. It is well known that the restriction imposed by the magic predicates is by far the tightest among such "filters" produced by rewriting methods. Although imposing tight restrictions on rule evaluation is of obvious importance, the amount of work required for computing the restrictions, as well as their sizes should be considered in the overall time for query processing. Indeed, there are examples in which the computation of the magic predicates dominates the overall query processing time. The natural question to ask is whether it is possible to keep the sizes of filters small while preserving the tightness of restriction enjoyed by

$r_0 : p(X,Y,Z)$:- $flat(X,Y,Z)$.
$r_1 : p(X,Y,Z)$:- $upa(X,X_1), upb(Y,Y_1), p(X_1,Y_1,Z_1), down(Z,Z_1)$.
:- $p(1,2,Z)$?

Figure 3.1: A Simple Program With a Query

magic predicates. Our thesis is that the magic sets method sometimes keeps essentially *unrelated* sets of bindings together in the form of one magic predicate, which unnecessarily increases the size of the filter (*i.e.,* the magic predicate). To understand the considerations involved let us consider an example. Fig. 3.1 shows a simple program and a query. Notice that if subgoals are processed in the order given, then the binding pattern associated with the IDB predicate p is unique. Fig. 3.2 shows the magic transformed program corresponding to Fig. 3.1. Notice that the predicates *upa* and *upb* are not connected at the time they are processed. Thus, the bindings for X_1, Y_1 in $sup_2(X,Y,X_1,Y_1)$ are computed by performing a Cartesian product between the bindings of X_1 in $sup_1(X,Y,X_1)$ and those of Y_1 in $upb(Y,Y_1)$[1]. Indeed an optimal way to compute the join of the three relations $upa(X,X_1)$, $upb(Y,Y_1)$, and $p(X_1,Y_1,Z_1)$ is to first join *upa* (or *upb*) with p and join the result with *upb* (or *upa*). The mechanism of the magic sets method has the effect of forcing the order $(upa \bowtie upb) \bowtie p$ on the above join expression. The consequences are: (*i*) an unnecessary Cartesian product $upa \times upb$ (since *upa* and *upb* are not connected at this point, $upa \bowtie upb = upa \times upb$) is computed, and (*ii*) the second join $(upa \bowtie upb) \bowtie p$ involves a substantially large relation corresponding to the Cartesian product. Let us next consider how the dynamic filtering method would handle this situation. For the case of comparison, we cast the method of [KL86] as a rewriting method, which essentially implements the same idea. Fig. 3.3 shows the rewritten program according to the method of dynamic filtering. The dynamic filtering method essentially maintains one filter for each bound argument of each predicate. For example, there are two filters associated with p, namely $\dagger p^1$ and $\dagger p^2$. Note that the rules in the transformed

[1]Even though sup_1 and upb share the variable Y, a quick reflection will reveal that sup_2 essentially corresponds to the Cartesian product $upa \times upb$, with restrictions imposed by mp

$r_0 : mp(1,2) \text{ :- }.$
$r_1 : mp(X_1,Y_1) \text{ :- } sup_2(X,Y,X_1,Y_1).$
$r_2 : sup_1(X,Y,X_1) \text{ :- } mp(X,Y), upa(X,X_1).$
$r_3 : sup_2(X,Y,X_1,Y_1) \text{ :- } sup_1(X,Y,X_1), upb(Y,Y_1).$
$r_4 : sup_3(X,Y,Z_1) \text{ :- } sup_2(X,Y,X_1,Y_1), p(X_1,Y_1,Z_1).$
$r_5 : p(X,Y,Z) \text{ :- }$ mp(X,Y), flat(X,Y,Z).
$r_6 : p(X,Y,Z) \text{ :- } sup_3(X,Y,Z_1), down(Z,Z_1).$

Figure 3.2: A Magic Sets Transformed Program

program in Fig. 3.3 are divided into five groups. The rules in Group 1 initialize the filters with the query constants. The rules in Group 2 filter the various relations using the associated filters, to create the filtered relations. For instance, the rules r_9 and r_{10} define the filtered relation $\ddagger p$, using the two filters associated with p. Group 3 contains the transformed original rules, making use of the filtered predicates. Group 4 rules implement sideways propagation. Finally, the rules in Group 5 update the filters, via a backward propagation. Notice th-+ the problem of Cartesian product is clearly avoided by the dynamic filtering method. In addition, the sizes of filters are in general much smaller than the size of the magic predicate which carries the Cartesian product of two independent bindings, and expands this Cartesian product through recursion. However, we can create instances of the relations $upa, upb, down$, and $flat$ showing that the restriction imposed by dynamic filters can be much less tight than the one imposed by the magic predicate. This motivates the question: can we combine the ideas of magic sets and dynamic filters in order to realize the advantages of both? We propose the method of *magic filters* as a partial answer to this question.

3.2 Magic Filters — The Basic Method

Before discussing the magic filters method, we need several notions. We assume the reader is familiar with the usual terminology associated with bottom-up processing, such as binding patterns, sideways information passing, etc. as discussed in Chapter 2. The first notion we need for our method is a more general concept of a binding

* Group 1 - Initialization of Filters

r_1 : $\dagger upa^1(1)$:-.

r_2 : $\dagger upb^1(2)$:-.

r_3 : $\dagger flat^1(1)$:-.

r_4 : $\dagger flat^2(2)$:-.

* Group 2 - Filtering Original Relations

r_5 : $\ddagger flat(X,Y,Z)$:- $\dagger flat^1(X), flat(X,Y,Z)$.

r_6 : $\ddagger flat(X,Y,Z)$:- $\dagger flat^2(Y), flat(X,Y,Z)$.

r_7 : $\ddagger upa(X,X_1)$:- $\dagger upa^1(X), upa(X,X_1)$.

r_8 : $\ddagger upb(Y,Y_1)$:- $\dagger upb^1(Y), upb(Y,Y_1)$.

r_9 : $\ddagger p(X_1,Y_1,Z_1)$:- $\dagger p^1(X_1), p(X_1,Y_1,Z_1)$.

r_{10} : $\ddagger p(X_1,Y_1,Z_1)$:- $\dagger p^2(Y_1), p(X_1,Y_1,Z_1)$.

r_{11} : $\ddagger down(Z,Z_1)$:- $\dagger down^2(Z_1), down(Z,Z_1)$.

* Group 3 - Generation of Answers Using Filtered Relations

r_{12} : $p(X,Y,Z)$:- $\ddagger flat(X,Y,Z)$.

r_{13} : $p(X,Y,Z)$:- $\ddagger upa(X,X_1), \ddagger upb(Y,Y_1), \ddagger p(X_1,Y_1,Z_1), \ddagger down(Z,Z_1)$.

* Group 4 - Sideways Propagation.

r_{14} : $\dagger p^1(X_1)$:- $\ddagger upa(X,X_1)$.

r_{15} : $\dagger p^2(Y_1)$:- $\ddagger upb(Y,Y_1)$.

r_{16} : $\dagger down^2(Z_1)$:- $\ddagger p(X_1,Y_1,Z_1)$.

* Group 5 - Backward Propagation.

r_{17} : $\dagger flat^1(X)$:- $\dagger p^1(X)$.

r_{18} : $\dagger flat^2(Y)$:- $\dagger p^2(Y)$.

r_{19} : $\dagger upa^1(X)$:- $\dagger p^1(X)$.

r_{20} : $\dagger upb^1(Y)$:- $\dagger p^2(Y)$.

Figure 3.3: A Dynamic Filtering Transformed Program

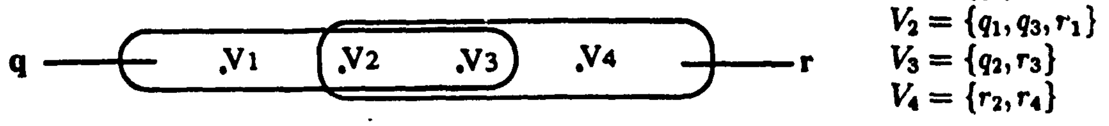
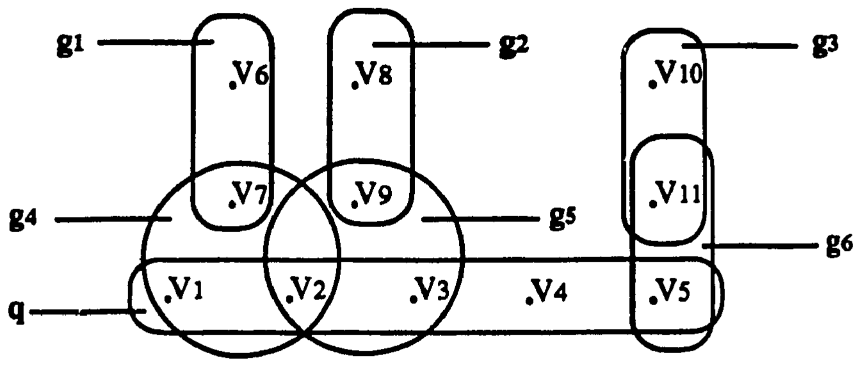


Figure 3.4: Hypergraph Representation of a Rule

pattern. Since our intention is to keep independent bindings separate, we need a notation for indicating which bindings are related. In this context, a predicate may well receive its bindings from several predicates (instead of one). Thus we need to know the connectivity between a predicate and its various sources of bindings. We choose hypergiaphs as a convenient formalism for this purpose.

A hypergraph [Ber73] is a pair $H = (N, E)$, where N is a finite set of nodes and E is a set of hyperedges, $E \subseteq 2^N$, such that $\bigcup E = N$. We represent a rule (more precisely the body of a rule) as follows. We let N be the set of distinct variables in the body of the rule. A predicate $p(X_1, \ldots, X_m)$, where X_i are (not necessary distinct) variables is represented using a hyperedge which contains exactly those nodes corresponding to the distinct variables among $X_1, \ldots, X_m$. For example, Fig. 3.4 shows the hypergraph representation corresponding to the rule $p(X, Y, Z)$:- $q(X, Y, X, W), r(X, Z, Y, Z)$. Associated with each node we have a set indicating the argument positions of predicates where the variable corresponding to the node appears. For simplicity, we use a predicate name to refer to the hyperedge it corresponds to. Next, we define the notion of a *binding group*. Consider a rule h :- $g_1, \ldots, g_m$. Suppose that the subgoals g_i are processed in the order given. Let H denote the hypergraph corresponding to this rule and H_{g_i} be the set of edges corresponding to $g_1, \ldots, g_{i-1}$. Then by a *binding group* of g_i we mean any maximal set of nodes of the hyperedge g_i that are connected in the hypergraph induced by H_{g_i}. For example, consider the rule

$$p(X, Y, Z) :- g_1(X, X_1), g_2(Y, Y_1), g_3(Z, Z_1), g_4(X_1, X_2, Y_2),$$
$$g_5(Y_1, Y_2, Z_2), g_6(Z_1, U_1), q(X_2, Y_2, Z_2, W_2, U_1).$$



$$H_q = \{1,\ldots,6\}$$
$$BG_q = \{V_1, V_2, V_3\}, \{V_5\}$$

Figure 3.5: Hypergraph Representation of Binding Groups

Fig. 3.5 shows the hypergraph representation of this rule. (For clarity, we omit the argument position sets associated with the various nodes, which can be obtained by inspection.) The binding groups of q are $\{V_1, V_2, V_3\}$ and $\{V_5\}$ which correspond to the argument positions $\{q_1, q_2, q_3\}$ and $\{q_5\}$.

We next need a generalized notation for binding patterns capable of representing binding groups. This will be used later to determine which bindings should be kept separate. We use the letter b with different subscripts to denote different binding groups. More precisely, the *binding pattern* of a predicate p is a string of length n over the alphabet $\{b_1, \ldots, b_n, f\}$. We refer to the symbols b_i as *binding symbols.* The interpretation of a binding pattern is that those argument positions for which the binding pattern has the symbol f are free and all other arguments are bound. Furthermore, two argument positions of p are in the same binding group if and only if the binding pattern contains the same symbol, say b_i, for those positions. For example, for the predicate q corresponding to Fig. 3.5 , the associated binding pattern is $q^{b_1 b_1 b_1 f b_2}$. Notice that this notation not only conveys the information about the bound/free status of arguments; it also specifies the binding groups among arguments.

With each rule we associate a set of filters. These filters are determined by the binding pattern received by the head predicate of the rule. There is exactly one filter corresponding to each binding group in the binding pattern. The arguments of a filter are those corresponding to the various arguments in a binding group. For instance, consider the rule

$$p(X,Y) \text{ :- } r(X,X_1), s(Y,Y_1), q(X_1,Y_1).$$

Suppose the rule head received the binding pattern $p^{b_1 b_2}$. Then we create two filters $\dagger p_1$ and $\dagger p_2$ corresponding to the two binding group. The filter $\dagger p_1$ corresponds to the first argument of p while $\dagger p_2$ corresponds to the second.

We next describe how to determine the binding pattern of a predicate in a rule body, from the binding pattern of the rule head. Let h :- $g_i, \ldots, g_n$ be a rule and let h receives the binding pattern h^{α}. Then g_i receives the binding pattern g_i^{β}, where β satisfies the following conditions.

(i) if a bound argument position[2] in g_i is connected to an output variable corresponding to the binding symbol b_i in α, then this argument should be assigned the symbol b_i in β, without violating (ii).

(ii) if two bound argument positions in g_i are connected to output variables corresponding to the same binding symbol b_i in α, then these argument positions should correspond to the same binding symbol b_i in β.

(iii) if two bound argument positions of g_i belong to the same binding group, then they should both correspond to the same binding symbol in β.

With these preliminary notions, we next describe the basic method of magic filters. For simplicity, we shall assume that in the programs we consider, the subgoals in rules are so ordered as to guarantee a unique binding pattern for each IDB predicate. There is no loss of generality in this assumption. The techniques of transforming an arbitrary program so the unique binding property holds are a simple extension of similar techniques for the traditional notion of binding pattern [Ull89], and will be discussed in Section 3.4, where we also discuss several heuristics for obtaining "good orders" for processing subgoals in a rule. In view of the above, it follows that each rule in the program will receive a unique binding pattern for its head predicate, during the course of query processing. In the following we describe the basic method of using magic filters, with reference to the program of Fig. 3.1. Note that for the given ordering of subgoals in the recursive rule, the predicate p is always called with the same binding pattern $p^{b_1 b_2 f}$. We rewrite the program with respect to the query

[2]Detecting these is a routine task, and can be done as discussed in [Ull89].

:- $p(1,2,Z)$? as follows.

Step 1 We create a filter corresponding to each binding group of each IDB predicate. For our example, this leads to two filters for p, which we denote $\dagger p_1$ and $\dagger p_2$. We initialize the filters using the query constants. Thus, we have the unit clauses

$\dagger p_1(1)$:-
$\dagger p_2(2)$:-

Step 2 We apply the various filters to the original rules. We implement this in a way that generalizes the idea of supplementary predicates. To make the distinction clear, we call the predicates generated by this process *filtered predicates.* An example will clarify the process. Application of the filters to the first rule yields the rule

$p(X,Y,Z)$:- $\dagger p_1(X), \dagger p_2(Y), flat(X,Y,Z)$.

For each rule containing several subgoals, we determine the predicates $q_1, \ldots, q_n$, with rank $i_1, \ldots, i_n$ in the subgoal ordering, such that (i) q_j contains at least one bound output variable[3], and (ii) q_j is disjoint from all predicates whose rank in the subgoal ordering is less than i_j. For each of the predicates q_j identified above, we create the filtered version of q_j by applying the appropriate filter(s). The filters to be applied are identified by using the argument position(s) that each filter corresponds to. For example, for the second rule in Fig. 3.1, the predicates to be filtered are easily seen to be *upa* and *upb*. The filter to be associated with *upa* is $\dagger p_1$ since it corresponds to the first argument of p and *upa* contains the output variable X which occurs in the first argument of p in the head. Thus, we generate the filtered predicates

$\ddagger upa(X,X_1)$:- $\dagger p_1(X), upa(X,X_1)$.
$\ddagger upb(Y,Y_1)$:- $\dagger p_2(Y), upb(Y,Y_1)$.

Next we implement sideways propagation (of information) in the form of rules. This process is, in principle, similar to the use of supplementary predicates in the magic sets method. The major difference is that instead of a chain of supplementary predicates, we may have several *streams* of such predicates, generated by the fact that we

[3]An output variable in a rule is any variable that appears as an argument of the head predicate.

$\dagger p_1(1)$:-
$\dagger p_2(2)$:-
$p(X,Y,Z)$:- $\dagger p_1(X), \dagger p_2(Y), flat(X,Y,Z)$.
$\ddagger upa(X,X_1)$:- $\dagger p_1(X), upa(X,X_1)$.
$\ddagger upb(Y,Y_1)$:- $\dagger p_2(Y), upb(Y,Y_1)$.
$\ddagger p(X,Y,Z_1)$:- $\ddagger upa(X,X_1), \ddagger upb(Y,Y_1), p(X_1,Y_1,Z_1)$.
$p(X,Y,Z)$:- $\ddagger p(X,Y,Z_1), down(Z,Z_1)$.
$p_1(X_1)$:- $\ddagger upa(X,X_1)$.
$p_2(Y_1)$:- $\ddagger upb(Y,Y_1)$.

Figure 3.6: A Basic Magic Filters Transformed Program

always try to maintain independent bindings separate. Another difference is that two different streams could merge whenever the streams share some arguments with some common predicate. For instances, the two filtered predicates above may be viewed as two independent supplementary streams. The next subgoal to be processed is the predicate p, and since the streams share common arguments with p, we merge them as follows.

$$\ddagger p(X,Y,Z_1) \text{ :- } \ddagger upb(X,X_1), \ddagger upb(Y,Y_1), p(X_1,Y_1,Z_1).$$

The rest of the sideways propagation is conducted in a similar manner. In our example, since there is only one subgoal left, we complete the processing of the first rule using the rule

$$p(X,Y,Z) \text{ :- } \ddagger p(X,Y,Z_1), down(Z,Z_1).$$

This completes the "filtering" of all original rules.

Step 3 In this step, we complete the definition of the filters by generating the rules for computing them. This is done by identifying the filtered predicates which contain the arguments corresponding to the filters. Thus we obtain the rules

$$\dagger p_1(X_1) \text{ :- } \ddagger upa(X,X_1).$$
$$\dagger p_2(Y_1) \text{ :- } \ddagger upb(Y,Y_1).$$

This completes the transformation. The transformed program is shown in Fig. 3.6.

Relation *upa*

X	X_1
1	3
1	4
3	5
4	6

Relation *upb*

Y	Y_1
2	7
2	8
8	9
7	10

Relation *flat*

X	Y	Z
1	10	11
6	2	12
3	9	13
5	7	14

Figure 3.7: EDB Relations $upa, upb, flat$

3.3 The Column Mixing Problem

In this section, we discuss a source of inefficier ~v of the basic method described in the previous section. Consider the example of Section 3.2. Compared with the magic transformation, which would create a single filter containing the Cartesian product of the bindings for X_1 and Y_1 and hence force the join of upa, upb, and p to be evaluated as $(upa \times upb) \bowtie p$, the transformation we just described has the advantage of keeping independent bindings separate and hence avoiding unnecessary Cartesian products. However, we would ideally like to maintain the same effective restriction as is imposed by the magic predicate on the generation of useless tuples. We shall show that in general this may not be possible if we use the basic magic filters transformation above. Let us illustrate the problem involved with an example. For the program of Fig. 3.1, suppose that the EDB consists of the relations shown in Fig. 3.7. Using the rules for the filters $\dagger p_1$ and $\dagger p_2$, we see that these filters will contain the bindings $\{1,3,4,5,6\}$ and $\{2,7,8,9,10\}$ respectively. It is not hard to see that the answer to the query :- $p(1,2,Z)$? is the empty set. However, the transformed program of Fig. 3.2 generates the tuples $< 1,10,11 >, < 6,2,12 >, < 3,9,13 >, < 5,7,14 >$ for the relation p. Clearly, all these tuples are useless with respect to the given query. Using this idea, it is easy to construct examples on which the program obtained using the basic transformation produces an arbitrary number of useless tuples. We can generalize the observation from the preceding example as follows. Let P be any Datalog query program and let the associated magic predicate mp be n-ary, $n > 1$. Suppose that the associated basic magic filters transformation uses n unary filters

$\dagger p_1, \ldots, \dagger p_n$ in place of the n-ary magic predicate. (The extension of the following argument to the case where $k < n$ filters are used, possibly because of the structure of the rules, is straightforward and hence there is no loss in generality.) Consider the Semi-Naive Evaluation of the programs obtained using the Magic Sets and the basic Magic Filters transformations. Suppose that at the end of i iterations, each filter $\dagger p_j$ has $m_1^j + m_2^j + \cdots + m_i^j$ distinct tuples, where m_k^j tuples were added in iteration k, $k = 1, \ldots, i$. The total number of tuples in the various filters is $\sum_{j=1}^{n}(m_1^j + \cdots + m_i^j)$. Note that these tuples really correspond to $\prod_{j=1}^{n}(m_1^j + \cdots + m_i^j)$ bindings. On the other hand, for the same situation, the total number of tuples in the magic predicate is at most $\sum_{k=1}^{i}(m_k^1 * m_k^2 * \cdots * m_k^n)$. In this case, the number of bindings is of course exactly the number of magic tuples. In general, the number of bindings represented by the n filters can be arbitraryly larger than the number of magic tuples. For example, if each filter adds the same number m of tuples in each iteration, the number of magic tuples is $i \cdot m^n$ while the number of bindings captured by the filters is $(i \cdot m)^n$. Although the size of the filters can be smaller than the size of the magic predicate, this argument shows that the filtering employed by the filters is much less tight than is employed by the magic predicate. Thus, the basic filter method may produce a large number of useless tuples. This feature of the basic transformation is extremely undesirable. Notice that this is *not* offset by the fact that the filters are smaller than the magic predicate. We note that the argument above also applies to the methods described in [KL86,SS88]. In Section 3.4, we describe a method which solves this problem. For convenience of future reference, we refer to the problem above as the problem of *column mixing.*

3.4 Magic Filters — The Improved Method

A careful examination of the problem of the basic method, illustrated in Section 3.3, reveals the following. The reason for the lack of effectiveness of the filters is because when they are applied to the rules, we apply all pairs of possible bindings for X and Y to the rules. Thus, in addition to applying pairs of bindings for X and Y that correspond to the same iteration, we also apply binding pairs that belong to

$r_1 : \dagger p_1(0,1)$:-
$r_2 : \dagger p_2(0,2)$:-
$r_3 : p(X,Y,Z)$:- $\dagger p_1(i,X), \dagger p_2(i,Y), flat(X,Y,Z)$.
$r_4 : \ddagger upa(X,X_1)$:- $\dagger p_1(i,X), upa(X,X_1)$.
$r_5 : \ddagger upb(Y,Y_1)$:- $\dagger p_2(i,Y), upb(Y,Y_1)$.
$r_6 : \ddagger p(X,Y,Z_1)$:- $\dagger p_1(i,X), \dagger p_2(i,Y), \ddagger upa(X,X_1), \ddagger upb(Y,Y_1), p(X_1,Y_1,Z_1)$.
$r_7 : p(X,Y,Z)$:- $\ddagger p(X,Y,Z_1), down(Z,Z_1)$.
$r_8 : \dagger p_1(next(i),X_1)$:- $\dagger p_1(i,X), \ddagger upa(X,X_1)$.
$r_9 : \dagger p_2(next(i),Y_1)$:- $\dagger p_2(i,Y), \ddagger upb(Y,Y_1)$.

Figure 3.8: An Improved Magic Filters Transformed Program

different iterations. For instance, for the EDB considered in Section 3.3, the pairs $< 1,2 >, < 3,7 >, < 3,8 >, < 4,7 >, \ldots$ are examples of binding pairs of X and Y that belong to the same iteration. On the other hand, in pairs such as $< 1,10 >$, $< 6,2 >$, the bindings for X and Y are generated in different iterations. Since in the filtering method we maintain independent bindings separately, the "connection" between pairs of bindings provided by the iteration in which they are generated is lost. This is the reason why the basic method is not effective in reducing the number of useless tuples generated, in general. One way to solve this problem is to use a *time stamp*, which intuitively corresponds to the iteration in which a binding is generated, as one of the arguments of each filter. We illustrate the idea of time stamp using the example of Section 3.3. Fig. 3.8 shows the rewritten program incorporating time stamps. At the time the filters are initialized, the corresponding bindings are associated with the time stamp 0. Let us momentarily ignore the issue of how to generate successive time stamps correctly, and consider the application of filters to the original rule. Notice that in r_6 we insist that the bindings for X and Y should have been generated in the same iteration, by equating the time stamps of the filters. Recall that this is exactly the connection we need to ensure that useless combinations of bindings are not applied to the variables in rules. Thus, our next concern is how to compute the time stamps correctly when we update the filters in backward propagation. Notice that we need a way of generating unique time stamps for each set of bindings generated in the various filters. At the same time, we also need to synchronize bindings corresponding to the same iteration of different filters. A natural way of enforcing this synchronization is to associate the same time stamp with these bindings. For this purpose, we use a function *next*(i), where i is a time stamp, defined as follows. Recall that backward propagation refers to the passing of bindings from an IDB predicate in a rule body to a rule with this predicate as its head. For example, rules r_8 and r_9 in Fig. 3.8 implement backward propagation on the filters $\dagger p_1$ and $\dagger p_2$ respectively. In general, a filter may experience backward propagation more than once in a given iteration (*e.g.*, see the original query program in Fig. 3.10 and the Magic Filters transformed program in Fig. 3.12). The function *next*(i) is implemented as follows. If i is being considered during backward propagation for the first time, then *next*(i) denotes $max + 1$ where max denotes the current maximum of the number of times backward propagation has been performed[4]. Otherwise, i must have been considered at least once before. In this case, *next*(i) denotes the value *next*(i) that was generated when i was considered for the first time. We remark that the *next* function does not have to be implemented literally as described above. The description above is only a logical view of the *next* function. The same effect can be achieved by simply updating the time stamps of the set of filters corresponding to the same binding pattern with the same value.

We shall now show the algorithm of rewriting using magic filters. Some criteria for ordering the subgoals in rules are shown in Fig. 3.9. Order of processing of subgoals can have a significant impact on the performance of the system. Unfortunately, there are no general criteria that can be used for all kinds of situations, and the one in Fig. 3.9 are to be viewed as good heuristics useful for many situations. (The priorities are in descending order, and the symbol > represents *more preferable*). Next, we shall discuss the complete Magic Filters rewriting algorithm. The algorithm is summarized in Fig. 3.11. The algorithm follows a "depth-first" approach, that is, whenever the program encounters a backward propagation, it will first process the backward propagation. The significance of depth-first will be discussed in Chapter 4. We shall begin by briefly describing the algorithm. The program accepts a set of filters

[4]Notice that even within one iteration of evaluation of the rules, backward propagation may be performed (perhaps corresponding to different filters) several times.

1) predicate with binding > predicate without binding
2) *EDB* predicate > *IDB* predicate
3) more arguments bound > less arguments bound
4) less arguments > more arguments

Figure 3.9: Criterion for Ordering Subgoals

$$\begin{aligned} r_1 &: p(X,Y,Z) \text{ :- } flat(X,Y,Z). \\ r_2 &: p(X,Y,Z) \text{ :- } upa(X,X_1), upb(Y,Y_1), p(X_1,Z_1,Y_1), down(Z,Z_1). \\ & \quad\quad\quad\quad\quad \text{ :- } p(1,2,Z)? \end{aligned}$$

Figure 3.10: A Program With Non-Unique Binding Pattern

as input. For each appropriate rule with respect to the input filters, we first order the subgoals according to the heuristics suggested in Fig. 3.9. We then repeatedly select a subgoal p, and do the following until all subgoals are selected. For each p, we define the corresponding filtered predicate by generating a new rule. But if p is the last subgoal selected, we redefine the original rule. In addition, if p is an IDB, then we process the backward propagation corresponding to p first. We shall illustrate the method with reference to the program and query shown in Fig. 3.10. Using appropriate orders for processing subgoals, we find that two distinct binding patterns are generated for the predicate p namely $p^{b_1b_2f}$, and $p^{b_1fb_2}$.

We create a filter corresponding to each binding group of the query predicate. In this case, this leads to two filters for p, which we denote $\dagger p_1^{b_1b_2f}$ and $\dagger p_2^{b_1b_2f}$. We initialize the time stamps to zero, and initialize the other arguments with the corresponding query constant. Thus, we have the unit clauses

$$\begin{aligned} &\dagger p_1^{b_1b_2f}(0,1) \text{ :- } \\ &\dagger p_2^{b_1b_2f}(0,2) \text{ :- } \end{aligned}$$

Corresponding to the binding pattern $p^{b_1b_2f}$, We rewrite the rules as follows. We transform the first rule r_1 to

$$p(X,Y,Z) \text{ :- } \dagger p_1^{b_1b_2f}(i,X), \dagger p_2^{b_1b_2f}(i,Y), flat(X,Y,Z).$$

```
procedure rewrite(fil : set of filters);
/* Input : a set of filter corresponding to the same binding pattern */
/* Output : a set of rewritten rules corresponding to fil */
begin
    for each rule corresponding to fil do
    begin
        Order subgoals;
        repeat
            Select a subgoal p;
            if p is an IDB predicate then
            begin
                if the binding pattern of p is new then
                begin
                    f := create_new_filters(p);
                    rewrite(f);
                end
                else
                    generate rules to update the existing filters;
            end;
            if p is the last subgoal then
                redefine the original rule with p
            else
                Create filtered predicate corresponding to p;
        until all subgoals processed;
    end;
end;
```

Figure 3.11: Algorithm for Magic Filters

For r_2, we select *upa*, and *upb* and generate the following rules for the filtered predicates.

$$\ddagger upa^{b_1b_2f}(X, X_1) \text{ :- } \dagger p_1^{b_1b_2f}(i, X), upa(X, X_1).$$
$$\ddagger upb^{b_1b_2f}(Y, Y_1) \text{ :- } \dagger p_2^{b_1b_2f}(i, Y), upb(Y, Y_1).$$

The next subgoal we select in the ordering is p. Since p is an IDB predicate, according to the algorithm we generate the filters corresponding to p. At this moment, the first argument and the third argument of p receive bindings from $\ddagger upa^{b_1b_2f}$ and $\ddagger upb^{b_1b_2f}$ respectively. Since we want to generate the transformed rules in the "depth-first" order, we begin processing the current binding pattern $p^{b_1fb_2}$. Thus, we create the set of filters corresponding to $p^{b_1fb_2}$

$$\dagger p_1^{b_1fb_2}(next(i), X_1) \text{ :- } \dagger p_1^{b_1b_2f}(i, X), \ddagger upa^{b_1b_2f}(X, X_1).$$
$$\dagger p_2^{b_1fb_2}(next(i), Y_1) \text{ :- } \dagger p_2^{b_1b_2f}(i, Y), \ddagger upb^{b_1b_2f}(Y, Y_1).$$

We next rewrite the rule r_1 with respect to $p^{b_1fb_2}$.

$$p(X, Y, Z) \text{ :- } \dagger p_1^{b_1fb_2}(i, X), \dagger p_2^{b_1fb_2}(i, Z), flat(X, Y, Z).$$

We then rewrite r_2. The subgoals we choose with respect to $p^{b_1fb_2}$ are *upa* and *down*. We generate the rules for the filtered predicates.

$$\ddagger upa^{b_1fb_2}(X, X_1) \text{ :- } \dagger p_1^{b_1fb_2}(i, X), upa(X, X_1).$$
$$\ddagger down^{b_1fb_2}(Z, Z_1) \text{ :- } \dagger p_2^{b_1fb_2}(i, Z), down(Z, Z_1).$$

We next choose the subgoal p which is an IDB predicate. Hence we write rules for the filters corresponding to the binding pattern $p^{b_1fb_2}$. Since these filters have already been defined, we can view this process as an "update" to the existing filters. For the same reason, we continue with the processing of $p^{b_1fb_2}$.

$$\dagger p_1^{b_1b_2f}(next(i), X_1) \text{ :- } \dagger p_1^{b_1fb_2}(i, X), \ddagger upa^{b_1fb_2}(X, X_1).$$
$$\dagger p_2^{b_1b_2f}(next(i), Z_1) \text{ :- } \dagger p_2^{b_1fb_2}(i, Z), \ddagger q^{b_1fb_2}(Z, Z_1).$$

The next predicate to be defined is the filtered predicate $\ddagger p^{b_1fb_2}$. Since p is receiving bindings (via sideways propagation) from several streams (see Section 3.2), we have to use the filters corresponding to the various binding groups in order to prevent column mixing (see Section 3.3).

$$\begin{aligned}\ddagger p^{b_1fb_2}(X,Z,Y_1) :\!- \; & \dagger p_1^{b_1fb_2}(i,X), \dagger p_2^{b_1fb_2}(i,Z),\\ & \ddagger upa^{b_1fb_2}(X,X_1), \ddagger down^{b_1fb_2}(Z,Z_1), p(X_1,Z_1,Y_1).\end{aligned}$$

At this point, it only remains to generate the rule for p as far as the processing of the binding pattern $p^{b_1fb_2}$ is concerned.

$$p(X,Y,Z) :\!- \ddagger p^{b_1fb_2}(X,Z,Y_1), upb(Y,Y_1).$$

We now resume the processing of $p^{b_1b_2f}$. This yields the following rules.

$$\begin{aligned}\ddagger p^{b_1b_2f}(X,Y,Z_1) :\!- \; & \dagger p_1^{b_1b_2f}(i,X), \dagger p_2^{b_1b_2f}(i,Y),\\ & \ddagger upa^{b_1b_2f}(X,X_1), \ddagger upb^{b_1b_2f}(Y,Y_1), p(X_1,Z_1,Y_1).\\ p(X,Y,Z) :\!- \; & \ddagger p^{b_1b_2f}(X,Y,Z_1), down(Z,Z_1).\end{aligned}$$

The complete Magic transformed program is shown in Fig. 3.12.

3.5 Duplicate Elimination

In the bottom-up evaluation using a Semi-Naive type of evaluation method, one of the most important functions is the elimination of duplicate tuples. This is necessary not only for making the evaluation efficient, but for the termination of the evaluation as well. Indeed the termination condition of the Semi-Naive Evaluation is that no *new* tuples are generated for any IDB predicates. Since in the Magic Filters method a number of individual filters in synchrony realize the effect of the magic predicate, they have to be updated (with new tuples) synchronously in a bottom-up evaluation. Suppose that there are k such filters $\dagger p_1, \ldots, \dagger p_k$. Recall that each filters carry an extra column for the time stamp, which implements the synchronization. Suppose that in some iteration the incremental relations corresponding to tuples generated in that iteration are $\Delta\dagger p_1, \ldots, \Delta\dagger p_k$. The duplicate check to be implemented should logically correspond to the duplicate check on the magic predicate in that we should report duplicates exactly when the magic predicate gets duplicates. Using this observation, we note that we should report duplicates exactly when there are duplications in all incremental relations, for the same value of the time stamp. More precisely,

$\dagger p_1^{b_1b_2f}(0,1) \text{ :-}$

$\dagger p_2^{b_1b_2f}(0,2) \text{ :-}$

$p(X,Y,Z) \text{ :- } \dagger p_1^{b_1b_2f}(i,X), \dagger p_2^{b_1b_2f}(i,Y), \mathit{flat}(X,Y,Z).$

$\ddagger \mathit{upa}^{b_1b_2f}(X,X_1) \text{ :- } \dagger p_1^{b_1b_2f}(i,X), \mathit{upa}(X,X_1).$

$\ddagger \mathit{upb}^{b_1b_2f}(Y,Y_1) \text{ :- } \dagger p_2^{b_1b_2f}(i,Y), \mathit{upb}(Y,Y_1).$

$\dagger p_1^{b_1fb_2}(\mathit{next}(i),X_1) \text{ :- } \dagger p_1^{b_1b_2f}(i,X), \ddagger \mathit{upa}^{b_1b_2f}(X,X_1).$

$\dagger p_2^{b_1fb_2}(\mathit{next}(i),Y_1) \text{ :- } \dagger p_2^{b_1b_2f}(i,Y), \ddagger \mathit{upb}^{b_1b_2f}(Y,Y_1).$

$p(X,Y,Z) \text{ :- } \dagger p_1^{b_1fb_2}(i,X), \dagger p_2^{b_1fb_2}(i,Z), \mathit{flat}(X,Y,Z).$

$\ddagger \mathit{upa}^{b_1fb_2}(X,X_1) \text{ :- } \dagger p_1^{b_1fb_2}(i,X), \mathit{upa}(X,X_1).$

$\ddagger \mathit{down}^{b_1fb_2}(Z,Z_1) \text{ :- } \dagger p_2^{b_1fb_2}(i,Z), \mathit{down}(Z,Z_1).$

$\dagger p_1^{b_1b_2f}(\mathit{next}(i),X_1) \text{ :- } \dagger p_1^{b_1fb_2}(i,X), \ddagger \mathit{upa}^{b_1fb_2}(X,X_1).$

$\dagger p_2^{b_1b_2f}(\mathit{next}(i),Z_1) \text{ :- } \dagger p_2^{b_1fb_2}(i,Z), \ddagger q^{b_1fb_2}(Z,Z_1).$

$\ddagger p^{b_1fb_2}(X,Z,Y_1) \text{ :- } \dagger p_1^{b_1fb_2}(i,X), \dagger p_2^{b_1fb_2}(i,Z),$
$\qquad\qquad \ddagger \mathit{upa}^{b_1fb_2}(X,X_1), \ddagger \mathit{down}^{b_1fb_2}(Z,Z_1), p(X_1,Z_1,Y_1).$

$p(X,Y,Z) \text{ :- } \ddagger p^{b_1fb_2}(X,Z,Y_1), \mathit{upb}(Y,Y_1).$

$\ddagger p^{b_1b_2f}(X,Y,Z_1) \text{ :- } \dagger p_1^{b_1b_2f}(i,X), \dagger p_2^{b_1b_2f}(i,Y),$
$\qquad\qquad \ddagger \mathit{upa}^{b_1b_2f}(X,X_1), \ddagger \mathit{upb}^{b_1b_2f}(Y,Y_1), p(X_1,Z_1,Y_1).$

$p(X,Y,Z) \text{ :- } \ddagger p^{b_1b_2f}(X,Y,Z_1), \mathit{down}(Z,Z_1).$

Figure 3.12: Non-Unique Binding Pattern Program Transformed By Magic Filters

let us picture each filter $\dagger p_i$ divided into distinct compartments, each holding tuples corresponding to the same time stamp, with each compartment corresponding to a distinct time stamp. Once the relations $\Delta\dagger p_1, \ldots, \Delta\dagger p_k$ are generated, we test whether the tuples in $\Delta\dagger p_i$ are *all* contained in one single compartment of $\dagger p_i$, such that all compartments correspond to the same time stamp, $i = 1, \ldots, k$. We conclude that there are no tuples generated for the filter predicates exactly when this condition holds. Otherwise, we update all filters $\dagger p_i$ with the tuples in $\Delta\dagger p_i$ synchronously and associate them with a new time stamp. Since our termination condition is thus equivalent to that of Magic Sets, the termination condition can be seen to be correct.

Chapter 4

Forward-Semi-Naive Evaluation Method

Semi-Naive Evaluation is one of the well-known evaluation methods for bottom-up evaluation of logic query programs. Since the fundamental edge of bottom-up processing over top-down processing is its set-oriented nature (see Chapter 2), any improvements in the "extent" of set-orientedness of an evaluation method can be expected to cut down the overhead in query processing. In this chapter, we discuss an evaluation method, called *Forward-Semi-Naive Evaluation* (FSN) which seems to be an improvement of the Semi-Naive method in this direction. In Section 4.1 we motivate the quest for such an improvement. In Section 4.2 we describe the FSN method and illustrate it with an example. In Section 4.3 we present an efficient implementation of this method. Results of performance evaluation of this method compared with the Semi-Naive method are discussed in Chapter 5. Finally, we briefly discuss the issue of rule ordering relevant in the context of the FSN method.

4.1 Motivation

As seen in Chapter 2, one of the advantages of bottom-up evaluation of logic programs is its set-oriented feature. When sets of tuples are being fetched rather than a tuple at a time, the amount of disk I/O is reduced. This suggests the more set-oriented a method is the better in terms of disk I/O and related overhead. The Semi-Naive Evaluation method is a widely used bottom-up evaluation method. Before considering how to improve the extent of set-orientedness, it would be useful to have a measure for this property. The first measure that suggests itself is the number of tuples generated per rule application. A careful consideration, however suggests this measure is far from accurate and could mislead us to the conclusion that to improve set-orientedness is to reduce the number of rule applications, where the number of tuples that must be generated is assumed to be fixed. The problem is that there could be numerous "non-effective" rule applications during an evaluation (see Section 5.4 for an idea of the number of such applications for a typical query program). A rule application is considered *effective* provided it generates some nonempty set of tuples. Thus, a correct measure of set-orientedness seems to be the number of tuples generated per effective rule application. Besides increasing set-orientedness, it would be desirable to cut down the number of duplicate inferences[1] generated during an evaluation. In Semi-Naive Evaluation method, rules can only make use of the tuples generated from the previous iteration. Many rule applications cannot generate any tuples due to this reason, even though new facts are actually generated in the current iteration but are just made *invisible* to the subsequent rules. This seems to violate the fundamental principle of set-orientedness in a bottom-up evaluation. In order to improve the set-orientedness of Semi-Naive Evaluation method, we must be able to make an application of a rule generate more tuples[2]. In addition, as seen above Semi-Naive Evaluation also has the side effect of making many rule applications non-effective. To look at this problem, let us consider a segment of a program shown in Fig 4.1. There are n rules $r_1, \ldots, r_n$ in this program segment. In each rule r_i, an IDB predicate p_i is defined. There is a subgoal p_{i-1} in each rule r_i except for r_1, which contains the subgoal p_n instead. Consider the Semi-Naive Evaluation of the program of Fig. 4.1. Suppose for simplicity that the only IDB subgoal in r_i is p_{i-1} and all other subgoals are EDB predicates. Since each rule in Fig 4.1 has to wait for the preceding rule to generate the required *new* facts for the corresponding IDB relation,

[1]Duplicate inferences signify the same tuple(s) being redundantly generated several times during the course of an evaluation.

[2]Notice that the number of useful tuples that must be generated by the method is considered fixed. The idea, then is to generate as many of those as possible in each iteration with the objective of generating the entire set of tuples quickly.

$$\begin{array}{lll}
r_1 & : p_1 & :\!- \cdots, p_n, \cdots. \\
r_2 & : p_2 & :\!- \cdots, p_1, \cdots. \\
 & \vdots & \\
r_{n-1} & : p_{n-1} & :\!- \cdots, p_{n-2}, \cdots. \\
r_n & : p_n & :\!- \cdots, p_{n-1}, \cdots.
\end{array}$$

Figure 4.1: A Segment of a Typical Magic-Transform Program

it is not hard to see that each rule only generates (new) facts every n iterations. In other words, we fire n rules in each iteration, but only one rule is able to generate new facts. The applications of rules that are incapable of generating new facts are obviously non-effective and essentially from an overhead. This overhead becomes serious when n is large or when a program contains many segments similar to the one shown in Fig 4.1. Notice that such program segments may be generated as a result of the transformation of one rule in the original program using Magic Sets method (or its variants, including Magic Filters[3]), in which case n is of the order of the number of subgoals in this original rule. An example of this typical Magic-transformed structure can be found in the Magic-transformed program shown in Fig 3.2 in the previous chapter.

It is quite clear that, there is a need to reduce the number of non-effective rule applications, and increase the extent of set-orientedness. One approach to achieve this goal is to make the newly generated facts available as soon as they are generated. An independent research introducing this idea can be found in [RSS90]. We note that a major concerns there, is to order the rules in such a way as to minimize the number of rule applications. The authors propose and study a number of rule orderings and show that a so-called *cycle preserving ordering* of rules always minimizes the number of rule applications. It should be noted that for certain classes of rules, cycle-preserving order does not exist [RSS90]. In addition, the only known algorithm for deciding the existence of this ordering requires $O(n!)$ time, where n is the number of rules in the program. In this chapter, we take a simple and realistic approach which

[3]Magic Filters will generate several smaller segments, depending on the number of binding groups.

uses a reasonable rule ordering, and our major concerns here are: *i*) to reduce all the avoidable overheads through an efficient implementation, and *ii*) to increase the set-orientedness of Semi-Naive Evaluation. To this end, we propose a method called *Forward-Semi-Naive (FSN) Evaluation*, and discuss an efficient implementation of this method.

4.2 The Forward-Semi-Naive Evaluation Method

The fundamental idea involved in the FSN method is essentially this: make *new* facts generated by a given rule during Semi-Naive Evaluation available to subsequent rules, possibly referring to that predicate, as soon as they are generated. While this appears extremely simplistic, certain issues have to be addressed before this method can be implemented effectively. In the following, we describe these considerations. Although it seems to be a definite advantage to reduce the number of rule application during a bottom-up evaluation, it would be undesirable if the simplicity of Semi-Naive Evaluation were lost in the course of this improvement. The most difficult part of implementing FSN is to update the *incremental relations* (see Chapter 2). If we simply replace the temporary relation used in the Semi-Naive Evaluation (Q_j in Fig 2.2) by the corresponding incremental relations, the incremental relations will become as big as the original IDB relations. In order to implement the idea of FSN efficiently, we must be able to remove tuples from the incremental relations such that the tuples removed have already been used by all rules in the program. We shall illustrate the difficulties involved via an example. Consider the following program

$$r_1 : s(X,Y) \text{ :- } p(X,Z), t(Z,Y).$$
$$r_2 : p(X,Y) \text{ :- } s(X,Z), t(Z,Y).$$
$$r_3 : t(X,Y) \text{ :- } s(X,Z), p(Z,Y).$$

Conceptually, the FSN representation of the program is

$$r'_1 : \Delta s^{i+1}(X,Y) \text{ :- } \Delta p^i(X,Z), t^i(Z,Y) \cup p^i(X,Z), \Delta t^i(Z,Y).$$
$$r'_2 : \Delta p^{i+1}(X,Z) \text{ :- } \Delta s^{i+1}(X,Z), t^i(Z,Y) \cup s^{i+1}(X,Z), \Delta t^i(Z,Y).$$
$$r'_3 : \Delta t^{i+1}(X,Y) \text{ :- } \Delta s^{i+1}(X,Z), p^{i+1}(Z,Y) \cup s^{i+1}(X,Z), \Delta p^{i+1}(Z,Y).$$

(For convenience we use the Union operator in the rules.)

Here, superscripts denote the most recent iteration in which tuples were generated. E.g., Δp^{i+1} contains tuples generated in iteration $(i+1)$ (in addition, possibly, to tuples generated in iteration i and so on). Let us concentrate on how Δp is updated. First, we execute r_1'. The tuples in Δp^i become "old" with respect to r_1'. At this point, the tuples in Δp^i are "new" with respect to r_3'. Therefore we cannot ıemove any tuples from Δp at this moment. We then execute r_2' and generate some new tuples for Δp^i, updating it to Δp^{i+1}. Finally, executing r_3', all tuples in Δp^{i+1} are old with respect to r_3'. But only some of the tuples in Δp^{i+1} are old with respect to r_1. Thus, we know that there are some tuples in Δp^{i+1} which are *old* with respect to *all* rules, and hence have to be removed. Some bookkeeping is necessary to decide exactly *which* tuples in Δp^{i+1} can be deleted.

4.3 An Efficient Implementation of Forward-Semi-Naive Evaluation

As seen in the last section, the difficulty in updating incremental relations is that for each IDB predicate in each rule, we need to know *which* tuples have been used by all rules in the evaluation, from the corresponding incremental relation. In this section, we propose an effective implementation method to solve this problem. For the sake of clarity, we first describe the method from a logical point of view and then describe an actual implementation written in the framework of a database management system. Let us use the example consisting of the three rules r_1', r_2', r_3' of the previous section as a vehicle. We shall show not only it is possible to identify exactly which tuples in an incremental relation have been used by a particular rule, but it is also possible to do away with the need for physically maintaining the incremental relations as separate relations. Notice that since each IDB relation is updated using the new tuples collected in its incremental relation, there is an obvious storage redundancy in maintaining an incremental relation separately. Thus, there is an obvious gain in eliminating separate storage of incremental relations. Fig. 4.3 represents a logical viewpoint of our implementation method corresponding to the example of the previous section. We use logical lines to delimit the sets of tuples of the relation p which have been used by given rules. E.g., in Fig. 4.3(b) all tuples above the line marked 3 have been used by r'_3 while those above the lines marked 1 have been used by r'_1. In particular the tuples between line 1 and 3 have been used by r'_3 but not r'_1. Thus the tuples above line 1 have been used by *all* rules. Let us next consider how to "update" or move these lines as rules are fired. Initially, the relation p is initialized (using any initialization rules or *exit* rules). Suppose, without loss of generality, that this leads to a nonempty relation for p. The top portion of Fig. 4.3(a) represents the corresponding state, signifying no tuple of p has been used by r'_1 or r'_3,[4] which is obvious since none of the (recursive) rules has been fired so far. After r'_1 is fired, we obtain the state corresponding to the second (from top) portion of Fig. 4.3(a). At this point, all (no) tuples in p have been used by r'_1 (r'_3). After r'_2 is fired, the lines corresponding to r'_1 and r'_3 do not change. However, r'_2 could have generated some tuples for p. This yields the corresponding state in Fig. 4.3(a). Finally, after r'_3 is fired, the line for r'_1 is not changed, although the line for r'_3 is pushed down to the bottom, since all tuples in p have been used by r'_1 at this time. This conceptual method can be generalized to any logic program in the obvious manner.

We next address the actual implementation of these logical lines. The major question is how to implement these lines without affecting the physical organization of the relations. In order to do this, we add an extra argument called *tag*, to each IDB predicate. This argument takes a natural number as its value. Whenever new tuples are generated for a relation, we associate them with a new value for tag. For each IDB predicate p and for each rule r_i, we maintain the maximum value of tag max_i signifying the set of tuples of that IDB predicate which have been used by r_i. Notice that this maximum has to be computed once *before* every application of r_i. The next time we fire r_i, we can select tuples of p with tag value greater than max_i to get exactly those tuples of p which have not been used by r_i. An example of maintaining this maximum with respect to r'_2 from the previous example is shown in Fig 4.2. For clarity, we choose to illustrate the idea with SQL embedded in the C language.

[4]Since r'_2 does not depending on p, we omit r'_2 from consideration, while discussing the use of tuples of p by rules.

The predicates $p(X,Y), s(X,Y)$, and $t(X,Y)$ become $p(tag,X,Y), s(tag,X,Y)$, and $t(tag,X,Y)$ respectively. We also employ a global variable i serving as the counter for updating tag in all IDB relations. For each IDB predicate q in each rule r_j, we maintain two variables $newq_j$, and $oldq_j$ for representing the new, and the current tag value respectively. Just as the logical implementation outlined before, the implementation described here extends to arbitrary logic programs. We see that by using this simple idea we are able to achieve significant savings - *(i)* reducing the number of auxiliary relations by half, *(ii)* identifying exactly which tuples in the incremental relations (which are implicitly represented inside IDB relation now) have been used by the rules, so these tuples do not have to be considered in further rule applications. These savings are achieved at a relatively low cost of adding an extra column to the IDB predicate.

4.4 Rule Ordering

The order in which the rules are fired in a FSN type of evaluation is quite significant. Indeed, in the worst case, the performance of FSN can degenerate to that of Semi-Naive Evaluation if the ordering of rules is not "right". To understand this, let us consider the following rules.

$r_1 : s(X,Y) \text{ :- } q(X,Z), s(Z,Y).$
$r_2 : q(X,Y) \text{ :- } q(X,Z), p(Z,Y).$
$r_3 : p(X,Y) \text{ :- } a(X,Z), p(Z,Y).$

Suppose that the rules are evaluated in the order given above. In this case, it is not hard to see that FSN registers no improvement over Semi-Naive Evaluation. The reason is that the tuples used by each rule are those from a previous iteration. E.g., r_1 needs tuples of q and s, however, they have not yet been generated in the current iteration. For the given rules the order $r_3 \rightarrow r_2 \rightarrow r_1$ can be seen to be optimal in the sense of maximizing set-orientedness. In [RSS90], it is shown that a so-called *cycle-preserving order* always guarantees optimality in a formal sense. However, as noted earlier this order need not exist for certain classes of rules and even when it does exist,

```
select $max(tag)$
into :$news_2$
from $s$;

select $max(tag)$
into :$newt_2$
from $t$;

insert into $p$
select :$i, X, Y$
from  $s, t$
where $s.Y = t.X$  and $s.tag >$ :$olds_2$
union
select :$i, X, Y$
from  $s, t$
where $s.Y = t.X$  and $t.tag >$ :$oldt_2$
minus
select :$i, X, Y$
from  $p$;

$olds_2 = news_2$;
$oldt_2 = newt_2$;
i++;
```

Figure 4.2: SQL Representation of FSN

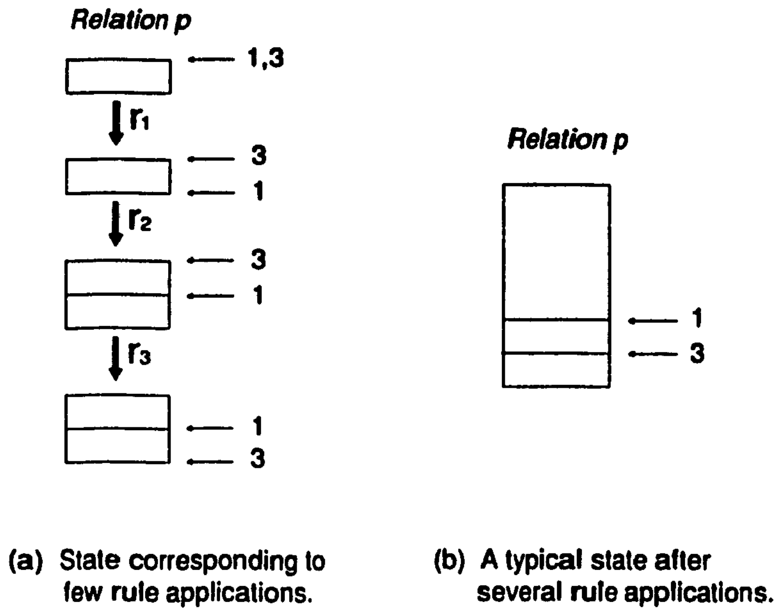

Relation p
1,3
r1
3
1
r2
3
1
r3
1
3
Relation p
1
3
(a) State corresponding to few rule applications.
(b) A typical state after several rule applications.

Figure 4.3: A Logical Representation of the Incremental Relations

generating it can impose an enormous overhead (it takes $O(n!)$ time, where n is the number of rules). A promising practical approach in rule ordering is the following. A predicate p ***depends*** on a predicate q whenever q occurs as a subgoal in some rule of p. The idea then is to try to order all rules for q before all rules for p whenever p depends on q. In the event of a mutual recursion, the tie has to be broken arbitrarily. We call this the ***depth-first ordering.*** The Magic Filters transformation algorithm has actually been implemented incorporating the depth-first ordering of the rewritten rules. It should be emphasized that the depth-first ordering is always possible for all programs and generating such an ordering can be done quite efficiently.

Chapter 5

Performance Analysis

In this chapter, we discuss the performance evaluation of the Magic Sets and Magic Filters method as well as Semi-Naive and FSN evaluation method. A comprehensive analysis of all methods is a separate topic of research by itself. In this thesis, our concern has been understanding *(i)* how the gains achieved by the Magic Filters over Magic Sets by avoiding the Cartesian product, vary as a result of changes in various sets of data, and *(ii)* similarly, how the improved set-orientedness of FSN over Semi-Naive Evaluation is affected by the data parameters. Consequently, we have taken a "typical" query program which brings out the structural improvement of Magic Sets on Magic Filters, and studied the performance of Magic Sets and Magic Filters on this program over different sets of data. For Semi-Naive Evaluation and FSN we have taken a similar approach. In Section 5.1, we describe the data set. In Section 5.2, we describe the cost metrics used in the comparison of various methods. Section 5.3 compares Magic Sets and Magic Filters rewriting methods while Section 5.4 compares Semi-Naive Evaluation and FSN methods.

5.1 The Test Data

We recall that the Magic Filters transformation applies to all Datalog programs. However, since one of our objectives has been to study the variation of the performance gains of Magic Filters method over the Magic Sets method, we have chosen one of the simplest programs on which the Magic Filters transformation would differ from the Magic Sets transformation. The query program is shown in Fig. 5.1. The

$p(X,Y,Z)$:- $flat(X,Y,Z)$.
$p(X,Y,Z)$:- $upa(X,X_1), upb(Y,Y_1), p(X_1,Y_1,Z_1), down(Z,Z_1)$.
:- $p(1,2,Z)$?

Figure 5.1: The Test Program

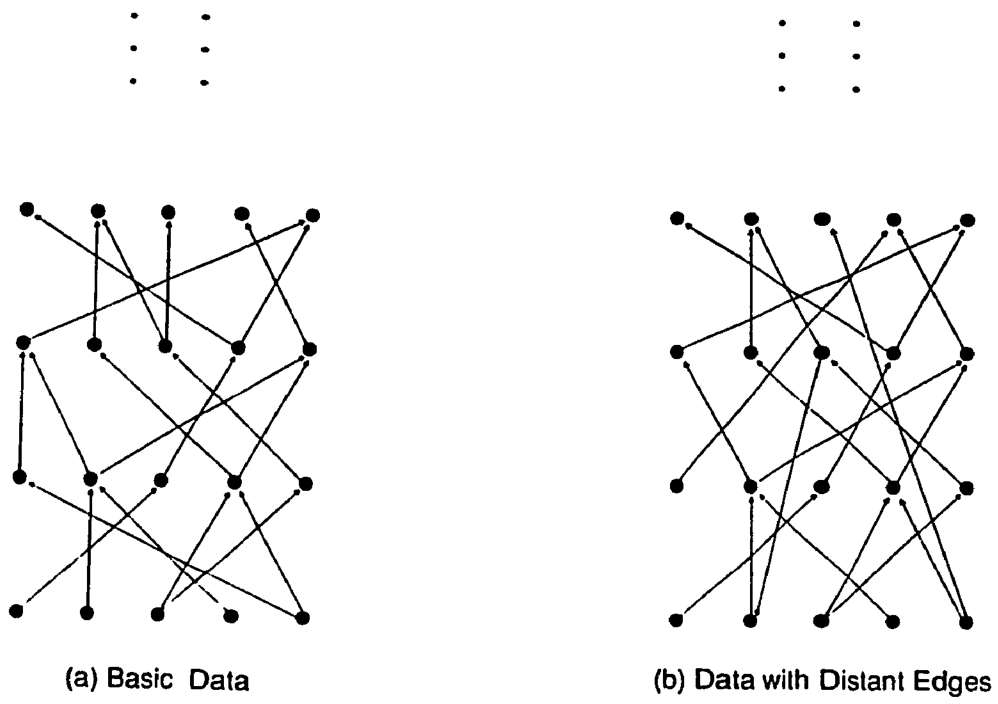


Figure 5.2: Sample Data Sets

choice of binary relations for EDB predicates lets us associate them with directed graphs and allows us to relate results of our performance analysis to the structural properties of the input data.

Bancilhon and Ramakrishnan [BR86] provide a detailed performance analysis of various methods. For our input (EDB) data, we start with the type of data corresponding to that used in [BR86] as our basic data and consider many structural variations on this basic data set, in our performance evaluation. Let us briefly describe the basic data next. For each EDB relation, the basic data set corresponds to a digraph, conceptually divided into layers with the property that all tuples correspond to edges which always connect a node in a given layer to a node in the next layer. Fig. 5.2(a) shows an example of such a basic data set. Note in particular that nodes in the first (*i.e.* bottom-most) layer have only outgoing edges, and nodes in the last (*i.e.* top-most) layer have only incoming edges. The edges are chosen at random from a uniform distribution.

The basic data set has some special properties which may be considered too "pure" from a practical point of view. For instance, all edges always go from one layer to the next layer. In particular, there are no cycles or "short-cuts" (*i.e.* edges connecting a node in a given layer to a node several layers away and thus providing a shortcut between these two layers). The basic data set seems to be somewhat "artificial" from a practical point of view in that such a clearcut layer separation with *all* edges connecting *only* nodes in *adjacent* layers need not always exist. It seems reasonable to expect that in practice there may indeed be edges connecting two layers which are not necessarily adjacent giving rise to cycles and shortcuts in the process. Fig. 5.2(b) illustrates this situation. We use the terminology of *distant edges* to denote edges connecting nodes in non-adjacent layers. It is useful to distinguish between two types of distant edges structurally as they exhibit different behavior - *up-edges* and *down-edges*. It is not hard to see that up-edges essentially constitute shortscuts while down-edges could give rise to cycles. For convenience of discussion, let us use the generic term *size of recursion* to denote the overall number of tuples generated during an evaluation of a recursive query. It should be recognized that cycles and shortcuts in the input data affect the size of the recursion to varying degrees. Before describing this difference, let us note that the way in which we use distant edges on top of the basic data set is a *replacement* of normal edges. More precisely, suppose we use 10% distant edges in the basic data set. This then means that for every 100 edges going out of a given layer, 10 are distant. This also explains why up-edges (acting as replacements of some "regular" edges) are indeed shortcuts between the layers connected by them. Thus, shortcuts always tends to reduce the overall size of the recursion, since potentially long paths in the data have been replaced by shorter ones. On the other hand, cycles (caused by down-edges) intuitively expand the size of the recursion. In fact, the number and the lengths of the cycle in the data can have a rather serious effect on the overall recursion size. Indeed, unless some limits are imposed on these parameters their effect could lead to a combinatorial explosion (in terms of the number of tuples generated) which in turn would produce practically intractable results.

With all these considerations in mind, we have made the proportion of up- and down- edges somewhat lopsided favoring up-edges. We have thus chosen a ratio of 4 to 1 for up- and down- edges. This ratio is held constant across all parametric variations on the data. We define the *selectivity* σ of data as the average out-degree of a node in the data.

We vary the percentage of distant edges d from 0 to 10 in steps of 2, where 0% distant edges correspond to the basic data set. We vary the selectivity over the range [1.67, 2.50]. Too low a selectivity leads to an extremely sparse data (digraph) which causes the recursion to die down in a very few iterations, producing atypical results. We believe such results are atypical because unless the recursion is allowed to build up and "stabilize" over several iterations, the results produced really correspond to degenerate cases. On the other hand, too high a selectivity produces a combinatorial explosion of the number of tuples generated. This is particularly serious when the number of tuples in the EDB relations is large. Finally, we use 6 layers and 10,000 tuples in each EDB relation and hold these parameters constant through out the performance evaluation experiment. For this size of the EDB relations, the chosen selectivities 1.67, 1.82, 2.00, 2.22, and 2.50 correspond respectively to 800, 900, 1,000, 1,100, and 1,200 nodes per layer in the associated digraph whose edges are chosen at random from a uniform distribution.

5.2 The Cost Metrics

In this section, we describe the cost metrics used to assess the performance of the methods in our evaluation. For the comparison of the rewriting method, we used three metrics – *(i)* the number of tuples generated T, *(ii)* the number of successful inferences SI, and *(iii)* the estimated cost of the query evaluation. The parameter T

refers to the total number of *distinct* tuples (of various predicates including the query predicate and any auxiliary predicates introduced by a rewriting method) generated in the evaluation. To a certain extent, this parameter reflects the amount of work needed for query processing including overheads. However, it does not account for overheads due to duplicate tuple generation. For this purpose, we use the parameter SI. Following [BR86] we define SI to be the total number of tuples (not necessarily distinct) ever generated during the evaluation. This can be computed by counting the tuples generated *before* any duplicates are removed. Consider the computation of a simple natural join of two relations. In this context, SI essentially corresponds to the *output cost* [Ull89] in the join computation. Thus, the *input cost* must be accounted for in order to obtain overall cost. This consideration extends to recursive query evaluation in a natural manner, upon the realization that it is essentially iterated computation of relational algebraic expressions. Since our objective has been to study the effect of join costs (occurring during recursive evaluation), we mainly focuse on joins in our estimated cost. Following [Ull89], we compute the input cost of a join as follows. Suppose the join computed is $r_1 \bowtie \cdots \bowtie r_n$, where $r_1, \ldots, r_n$ are any relations[1]. Suppose that the join is computed taking r_1 as the "driving" relation. That is, the join is computed by fetching in tuples of r_1 and then using them to determine tuples from other relations to be fetched. Using a primitive model, we account for the input cost by counting the number of tuples fetched in from the various relations in this context. For every join performed in the course of the recursive evaluation, we thus compute the overall cost of the join as the sum of the input and output costs of join computation. The overall estimated cost of the evaluation is computed by mainly focusing on the join cost, since join is one of the costliest operations. All results were obtained by running the tests on 3 independently chosen samples of random data and then taking the averages of the individual results.

[1]Since the Magic Filters transformation produces many filters, it is necessary to consider multi-way joins rather than binary joins alone.

5.3 Magic Sets *vs.* Magic Filters

In this section, we describe the results of the performance studies of Magic Sets and Magic Filters on the query program of Fig. 5.1 for which the transformed programs corresponding to Magic Sets and Magic Filters are given in Fig. 3.2, and Fig. 3.6 respectively. We test these transformed programs based on the Semi-Naive Evaluation method described in the previous chapter.

We begin with the simple case. The simplest case is the basic data set with no distant edges. The results are shown in Table 5.1, where *Tuples, S.I.*, and *Cost* represent the three metrics defined in the previous section. It is quite clear that the Magic Filters method performs significantly better than the Magic Sets method. With reference to the Magic Sets transformed program in Fig. 3.2, rules r_0 - r_2 create and propagate the Cartesian product computation between bindings on X and bindings on Y (which are independent) and rules r_3 - r_5 use the Cartesian product directly, while r_6 uses it indirectly. This imposes a substantial amount of overhead for computing the answer. As the selectivity increases, this problem becomes more serious, since higher selectivity increases the size of the Cartesian product. This can be verified by studying the variation of the performance with respect to the selectivities in Table 5.1. On the other hand, the Magic Filters method performs reasonably well even when the selectivity increases. The results show that the performances always differs by an order of magnitude.

Next, we shall see the effects of distant edges on Magic Sets and Magic Filters. In Table 5.2 through Table 5.6, the effects of 2%, 4%, 6%, 8%, and 10% of distant edges are shown respectively. In each case, we see the same relative performance between Magic Sets and Magic Filters and a similar variation of the performances across the range of selectivities, as in the case of 0% distant edges. In all cases, Magic Filters continues to out perform Magic Sets by an order of magnitude. Let us next look at the performance variation with respect to the proportion distant edges. Let us recall that distant edges are a combination of edges causing shortcuts and edges possibly causing cycles. Also recall that these two types of edges have opposite effects on the size of the recursion. As a whole increasing the proportion of distant edges increases the various metrics of cost. This is because, as we discussed earlier, cycles tend to dominate the overall effect of distant edges and hence lead to an overall increase in the cost. However, notice that the increase in the cost does not seem to be strictly monotonic, which is not surprising, considering the opposing effects of shortcuts and cycles. As a final remark, we note that even with respect to distant edges, Magic Filters exhibits a more gradual variation compared to Magic Sets, in terms of the various performance metrics. Recall that with respect to the number of tuples fetched from the EDB, Magic Sets and Magic Filters have the same performance, since they enforce the same restriction on useless tuple generation. However, with respect to the tuples generated for the additional predicates (*viz.*, the magic predicate and the supplementary predicates) the Magic Sets method generates extra tuples by storing the Cartesian product of independent bindings, using this Cartesian product through the supplementary predicates, and propagating it through recursion. The size of the Cartesian product increases with the selectivity and the percentage of distant edges. This is avoided in the Magic Filters method. We have summarized the results in the form of plots (of *Cost vs.* selectivity and percentage of distant edges). Fig. 5.3 and Fig. 5.4 show the individual plots for Magic Sets and Magic Filters respectively. In Fig. 5.5 they are combined for easy comparison.

Some remarks about the test program in Fig. 5.1 are in order. Notice that even though the test program is linear, the improvement in performance of Magic Filters over Magic Sets is not peculiar to the linear structure of the program. Indeed, in the test program, one or both the predicates upa, upb could well be IDB predicates, possibly mutually recursive with p. Even in this case, the important thing is that the bindings coming into the predicate p are structurally independent (in the sense of this thesis) and on such occasions, we can expect Magic Filters to improve on the performance of Magic Sets.

5.4 Semi-Naive Evaluation *vs.* Forward-Semi-Naive Evaluation

In this section, we compare the two evaluation methods based on the performance

Selectivity	Magic Sets			Magic Filters		
	Tuples	S.I.	Cost	Tuples	S.I.	Cost
1.67	1026	1035	2712	129	129	385
1.82	903	911	2363	192	192	547
2.00	6755	6909	17649	345	345	991
2.22	31362	32382	75232	715	715	2063
2.50	45636	48769	109359	890	892	2616

Table 5.1: Relative Performance For 0% of Distant Edges

Selectivity	Magic Sets			Magic Filters		
	Tuples	S.I.	Cost	Tuples	S.I.	Cost
1.67	4632	4669	12114	310	310	879
1.82	4181	4243	10956	288	288	857
2.00	8070	8167	21161	401	401	1178
2.22	12466	12839	32637	504	504	1427
2.50	35620	37393	87118	754	754	2206

Table 5.2: Relative Performance For 2% of Distant Edges

Selectivity	Magic Sets			Magic Filters		
	Tuples	S.I.	Cost	Tuples	S.I.	Cost
1.67	1895	1898	4978	233	233	632
1.82	3179	3201	8316	277	277	765
2.00	3621	3663	9470	273	273	813
2.22	10378	10500	27021	456	460	1264
2.50	43645	45988	104300	1082	1091	3066

Table 5.3: Relative Performance For 4% of Distant Edges

Selectivity	Magic Sets			Magic Filters		
	Tuples	S.I.	Cost	Tuples	S.I.	Cost
1.67	1616	1624	4953	206	206	602
1.82	1412	1437	3731	208	208	617
2.00	3246	3271	8601	273	273	766
2.22	17723	18252	46669	693	700	1950
2.50	105377	108907	268833	2669	2890	7390

Table 5.4: Relative Performance For 6% of Distant Edges

Selectivity	Magic Sets			Magic Filters		
	Tuples	S.I.	Cost	Tuples	S.I.	Cost
1.67	654	654	1717	130	130	383
1.82	3882	3940	10182	298	298	873
2.00	4729	4775	55739	336	337	936
2.22	17708	17939	46630	807	863	2340
2.50	101421	104860	265686	2943	3414	8562

Table 5.5: Relative Performance For 8% of Distant Edges

Selectivity	Magic Sets			Magic Filters		
	Tuples	S.I.	Cost	Tuples	S.I.	Cost
1.67	2026	2029	5346	228	228	664
1.82	1625	1639	4271	226	226	676
2.00	6157	6230	16253	386	394	1122
2.22	6289	6381	16476	554	554	1554
2.50	99094	101651	260022	2406	2740	7121

Table 5.6: Relative Performance For 10% of Distant Edges

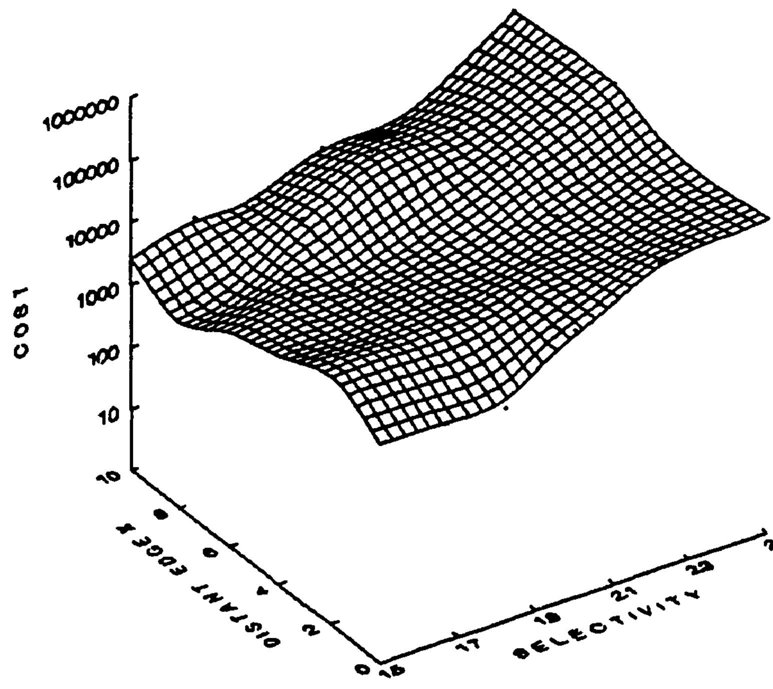


Figure 5.3: Overall Performance of Magic Sets

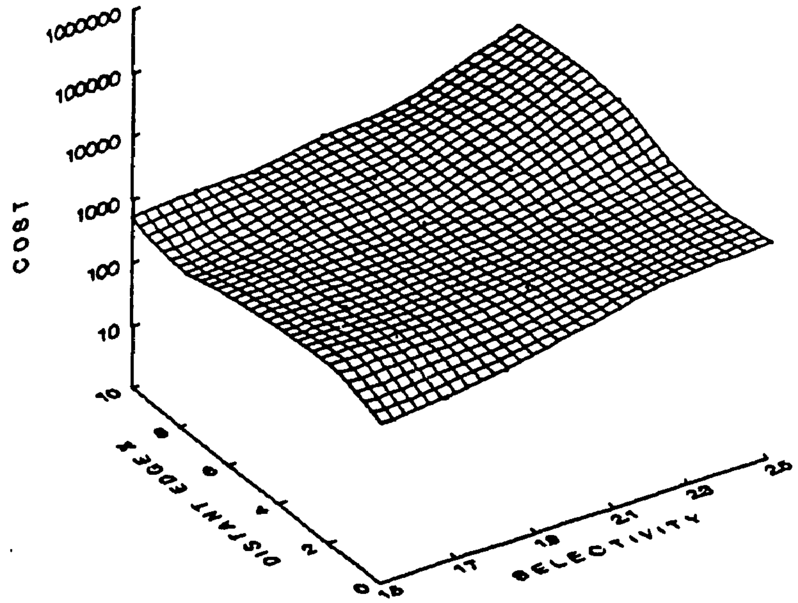


Figure 5.4: Overall Performance of Magic Filters

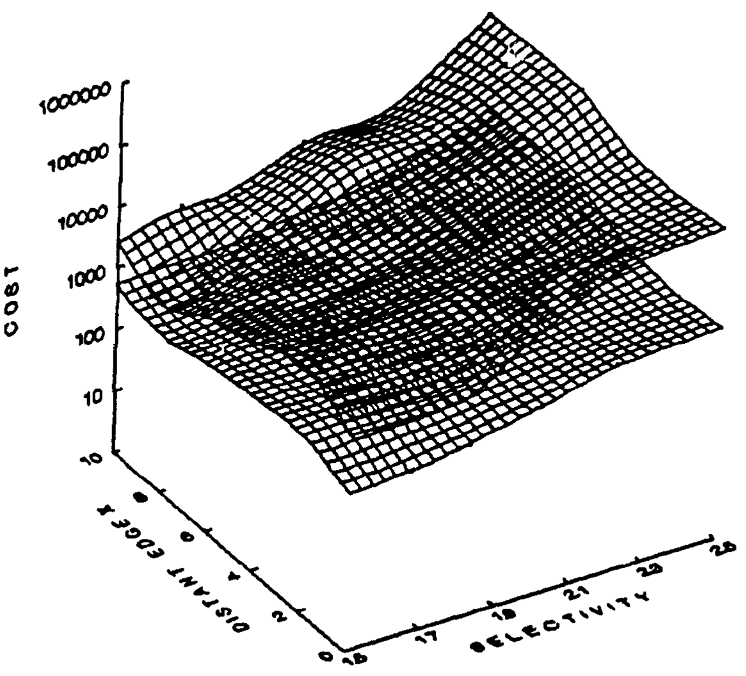


Figure 5.5: Overall Performance of Magic Sets and Magic Filters

analysis. Let us recall that FSN tries to improve on Semi-Naive Evaluation by reducing the number of non-effective rule applications, and increasing the extent of set-orientedness (see Section 4.2). As was clarified in Section 4.2, it is important to distinguish between the activities of reducing the number of non-effective rule applications, and increasing the number of tuples generated per effective rule application. To this end, we separate these two issues in our performance analysis. In the next section, we compare the relative performance of Semi-Naive and FSN evaluation methods as regards the number of non-effective rule applications. This is done by running the tests against the same basic query program of Fig. 5.1. In Section 5.4.2 we compare Semi-Naive Evaluation and FSN evaluation with the number of tuples generated per effective rule application in mind. In order to avoid the interaction of the reduction of the number of non-effective rule applications and set-orientedness (which is necessary for an accurate estimate of the improvement in set-orientedness), we use a new query program and test data where the data is so constructed that all rule applications are effective. The details are discussed in Section 5.4.2. We note that the improvements of Magic Filters on Magic Sets and the improvement of FSN over Semi-Naive Evaluation are conceptually independent. More precisely, the interaction of FSN with Magic Filters is essentially identical to the interaction of FSN with Magic Sets. Besides, our major concern is to study the improvements of Magic Filters over Magic Sets (FSN over Semi-Naive) independently of any evaluation methods (rewriting method). For this reason, we compare Semi-Naive and FSN based on the magic transformation program.

5.4.1 Non-effective Rule Application

We use the same query program as the one used in the previous section and compare the performance of the Semi-Naive Evaluation and FSN on the magic transformed program. The main effect of selectivity is in deciding the number of tuples generated. In particular, it does not influence set-orientedness or non-effective rule application. Since selectivity does not play any essential role in here, we use a constant value 2.50 for selectivity. We then vary the amount of distant edges from 0% to 10% in steps of 2%. We measure the amount of non-effective rule applications by counting the number of relational operations which produce no tuples. We also include the number of iterations required for the evaluation as part of the measurements. In some sense, the number of iterations seems to represent the overall effect of reducing non-effective rule applications. The parameters are denoted as *Iteration*, and *Null-result.* Table 5.7 shows the results. We can see that FSN reduces the number of iterations by 2/3 compared to Semi-Naive. Indeed, the evaluation of the program starts by generating new tuples for sup_1 using the tuples in mp. Then new sup_2 tuples are generated using the tuples in sup_1. Finally, mp is updated using new tuples in sup_2. In general, these steps will take three iterations for Semi-Naive Evaluation, but it will only take one iteration for FSN to accomplish that. Therefore, FSN only requires one third of the number of iterations as compared to the Semi-Naive Evaluation in this example. The number of iterations reduced because of this reason in general depends on the predicate dependency of the program. Table 5.7 also indicates that FSN reduces many of the non-effective rule applications by reducing the number of

Distant Edge %	Semi-Naive		Forward-Semi-Naive	
	Iteration	Null-result	Iteration	Null-result
10	46	267	17	67
8	49	279	17	60
6	44	258	16	58
4	22	126	9	34
2	16	91	6	22
0	16	88	6	20

Table 5.7: Number of Iteration/Null-result

iterations.

5.4.2 Set-Orientedness

As outlined in the beginning of Section 5.4, improvements of FSN on Semi-Naive can come from reducing overheads such as number of non-effective rule applications or from genuinely increasing the extent of set-orientedness of the evaluation method. In order to assess the latter accurately, it would be preferable to prevent its interaction with the former in contributing to an overall improvement. To arrange this, we start with the following simple program

$$
\begin{aligned}
r_1 &: q(X,Y) \text{ :- } p(X,Z), r(Z,Y).\\
r_2 &: r(X,Y) \text{ :- } a(X,Z), p(Z,Y).\\
r_3 &: p(X,Y) \text{ :- } q(X,Z), r(Z,Y).\\
&\quad \text{:- } p(1,2)?
\end{aligned}
$$

The rules are intended to signify some arbitrary transformed rules produced using a method such as Magic Sets or Magic Filters. We implicitly assume appropriate exit rules for the predicates p, q, r with exit relations p_e, q_e, r_e (The exit rules are assumed to be evaluated in the beginning of an evaluation). We consider a query :-$p(1,2)$? which corresponds to the binding pattern p^{bb}. We shall first describe the synthetic database we use in measuring the metric of interest to us effectively. Corresponding to the tuple $p(1,2)$ we create a proof tree of height n. Notice that the leaves of the proof tree will consist of EDB predicates. The proof tree can be associated with a database upon mapping each distinct variable with a distinct constant. More precisely, the EDB consists of the set of all tuples corresponding to the leaves of the proof tree, obtained from the mapping above. It follows from this construction that each atom corresponding to an internal node can be derived in exactly one way. In addition, each rule application against this database is necessarily effective. This property of the database lets us separate the effects due to a reduction of the number of non-effective rule applications, and an increase in the number of tuples generated per effective rule application, and lets us investigate whether an evaluation method is genuinely more set-oriented than Semi-Naive in the sense of increasing the number of tuples generated per effective rule application. We should remember that the performance of FSN is sensitive to the ordering of rules. For the program above we use the order r_1, r_2, r_3. A quick reflection will reveal that Semi-Naive needs n iterations in order to generate the atom $p(1,2)$ from the database generated above (from a proof tree of height n for $p(1,2)$). We studied the number of iterations needed by FSN as a function of the height of the proof tree for $p(1,2)$. Fig. 5.6 compares the numbers of iterations needed by Semi-Naive and FSN in order to generate $p(1,2)$. FSN needs about 2/3 of the number of iteration needed by Semi-Naive. Notice that this reduction in the number of iteration needed was *not* achieved by cutting down non-effective rule applications. Indeed, such a reduction can only come from being genuinely more set-oriented than Semi-Naive method. Since the total number of tuples generated by Semi-Naive and FSN is the same, this immediately implies a corresponding increase in the number of tuples generated per effective rule application. We thus find that these results and this experiment provide a justification for our intuitive argument that FSN is inherently more set-oriented than Semi-Naive. We believe this improved set-orientedness of FSN advances one of the fundamental advantages of bottom-up evaluation.

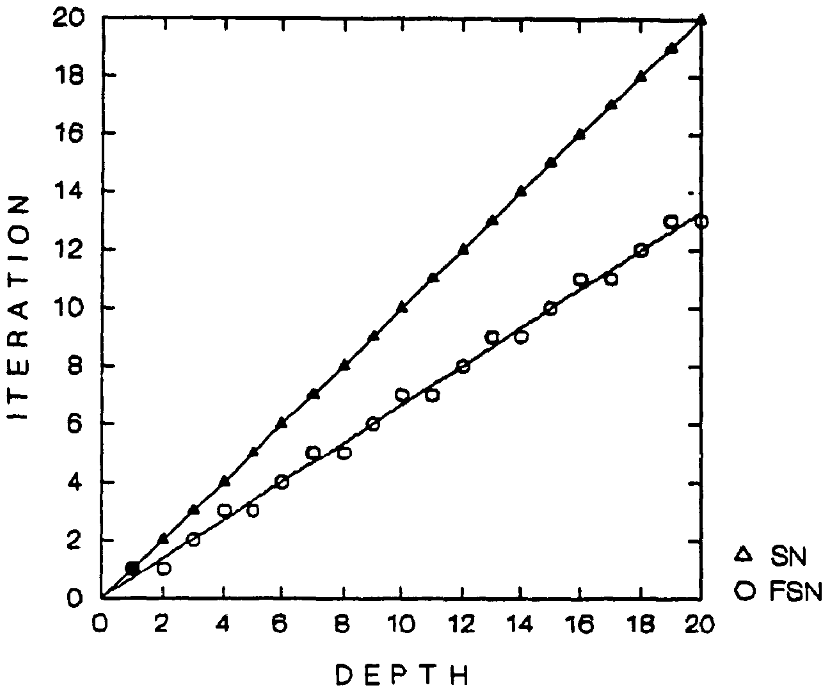


Figure 5.6: Number of Iterations Required by FSN

Chapter 6

Conclusions and Further Research Directions

Deductive database query languages such as Datalog, while far more powerful than conventional query languages based on relational algebra, introduce a high complexity for query processing. Recognizing this, a substantial amount of work has been done on efficient query processing for deductive databases. For database applications, bottom-up query processing seems to be more advantageous than top-down processing. Under this paradigm, query processing consists of two phases – rewriting and evaluation. Magic Sets and Semi-Naive Evaluation are among the most efficient rewriting and evaluation methods respectively. In this thesis, we have proposed improvements to both these methods. For Magic Sets, we identified certain conditions (based on the structure of the query program) under which the Cartesian product computed and propagated by Magic Sets can be avoided. We proposed a Magic Filters method which overcomes this problem and improves on Magic Sets. We first developed a basic method, which, while avoiding Cartesian products, experiences the problem of column mixing like some similar methods in the literature. The Magic Filters method is obtained upon solving the column mixing problem of the basic method. We compared Magic Filters and Magic Sets through performance evaluations. In addition to showing an order of magnitude improvement over Magic Sets, Magic Filters are seen to exhibit a more gradual performance variation as the data selectivity increases and as the data contains relatively more shortcuts and cycles.

On the evaluation side, we noted that it is possible to make Semi-Naive Evaluation more set-oriented, by making the results of a rule application available immediately to all subsequent rules referring to that predicate. Based on this, we developed the Forward-Semi-Naive Evaluation method. We studied the relative performance of FSN and Semi-Naive evaluation by focusing on the number of non-effective rule applications and the number of tuples generated per effective rule application. On both counts, our results show a significant improvement of FSN over Semi-Naive evaluation. We showed that it is possible to incorporate a rule ordering that enhances the performance of an FSN evaluation directly in the way the Magic Filters transformed rules are generated. We also discussed an efficient implementation of the FSN method that cuts down the number of auxiliary predicates needed by a half.

More research is needed for a more detailed comparison between Magic Sets and Magic Filters. Also, as remarked earlier, several improvments on Magic Sets for special subclasses of programs have been proposed in the literature (*e.g.*, see [AGNS90,Nai89,NRSU89]). Since the direction of these improvements is orthogonal to the direction of this thesis, a comparison with such methods has not been made here. In fact, because of the independence between the directions of Magic Filters and those in [AGNS90,Nai89,NRSU89], there seems to be potential for integrating Magic Filters with some of these methods. In particular, combining Magic Filters with the *Integrated Magic Sets* method of [AGNS90] seems promising.

It would be desirable to extend the Magic Filters method to handle negated subgoals in rules, say for the class of stratified logic programs. Another important direction for extension is function symbols. Finally, a comparison of Semi-Naive evaluation and FSN on arbitrary programs based on a knowledge of predicate dependencies would be interesting. Ramakrishnan *et al* [RSS90] report some results along this direction. We believe an analytical comparison bringing out the relative extents of set-orientedness is useful from a practical point of view and is an important open problem.

Bibliography

[Agr87] R. Agrawal, "Alpha: An extension of relational algebra to express a class of recursive queries," *IEEE Conf. Data Engineering*, Feb. 1987, pp. 580-590.

[AGNS90] V.S. Alagar, P. Goyal, P.S. Nair, and F. Sadri, "Integrated magic set method: a rule rewrite scheme for optimizing linear Datalog programs," *The Computer Journal* (in Press).

[BMSU86] F. Bancilhon, D. Maier, Y. Sagiv, and J.D. Ullman, "Magic sets and other strange ways to implement logic programs," *ACM Symp. Principles of Database Systems*, 1986, pp. 1-15.

[BR86] F. Bancilhon and R Ramakrishnan, "An amateur's introduction to recursive query-processing strategies," *ACM SIGMOD Intl. Conf. on Management of Data*, 1986, pp. 16-52.

[BR87] C. Beeri and R. Ramakrishnan, "On the power of magic," *ACM Symp. Principles of Database Systems*, 1987, pp. 269-283.

[Ber73] C. Berge, *Graphs and Hypergraphs,* North Holland, 1973.

[Bry89] F. Bry, "Query evaluation in recursive databases: top-down and bottom-up reconciled," *Proc. 1st Int. Conf. Deductive and Object-Oriented Databases*, Japan, 1989, pp. 20-39.

[Cha81] C. Chang, "On the evaluation of queries containing derived relations in a relational database," *Advances in Data Base Theory*, Vol. 1, Plenum Press, 1981.

[CGT89] S. Ceri, G. Gottlob, and L. Tanca, "What you always wanted to know about Datalog (and never dared to ask)," *IEEE Trans. on Knowledge and Data Engineering*, Vol 1, No 1, March 1989, pp.146-166.

[HN88] R.W. Haddad and J.F. Naughton, "Counting methods for cyclic relations," *ACM Symp. Principles of Database Systems*, 1988, pp. 333-340.

[Ker89] J.M. Kerisit, "A relational approach to logic programming: the extended Alexander method," *Theoretical Computer Science*, 69:1(1989), 55-68.

[KL86] M. Kifer and E.L. Lozinskii, "A framework for an efficient implementation of deductive databases," *Proc. Advanced Database Symp.*, Japan, 1986, pp. 109-116.

[LY91] V.S. Lakshmanan and C.H. Yim, "Can filters do magic for deductive databases?," To appear in *UKALP, Assoc. of Logic Programming, Conf. on Logic Programming*, U.K., April, 1991.

[Llo86] J.W. Lloyd, *Foundations of Logic Programming,* 2nd edn., Springer-Verlag, New York.

[NRSU89] J.F. Naughton, R. Ramakrishnan, Y. Sagiv, and J.D. Ullman, "Efficient evaluation of right-, left-, and multi-linear rules," *ACM SIGMOD Intl. Conf. on Management of Data*, 1989, pp. 235-242.

[Nai89] P.S. Nair, *Optimization of Logic Queries in Knowledge Base Systems,* Ph.D. Thesis, Department of Computer Science, Concordia University, Montréal, May 1989.

[Nau87] J.F. Naughton, "One sided recursions," *ACM Symp. Principles of Database Systems*, 1987, pp. 340-348.

[Nau88] J.F. Naughton, "Compiling separable recursions," *ACM SIGMOD Intl. Conf. on Management of Data*, 1988, pp. 312-319.

[Ram88] R. Ramakrishnan, "Magic templates: a spellbinding approach logic programs," *Proc. Int. Conf. Logic Programming*, 1988, pp. 140-159.

[RSS90] R. Ramakrishnan, D. Srivastava, and S. Sudarshan, "Rule ordering in bottom-up fixpoint evaluation of logic programs," Manuscript, October 1990.

[RHDM86] A. Rosenthal, S. Heiler, U. Dayal, and F. Manola, "Traversal recursion: A practial approach to supporting recursive applications," *ACM SIGMOD Intl. Conf. on Management of Data*, 1986.

[Sag90] Y. Sagiv, "Is there anything better than magic?," *Proc. North Amer. Conf. Logic Programming*, 1990, pp. 235-254.

[Sek89] H. Seki, "On the power of Alexander templates," *ACM Symp. Principles of Database Systems*, 1989, pp. 150-159.

[SS88] S. Sippu and E. Soisalon-Soininen, "An optimization strategy for recursive queries in logic databases," *IEEE Conf. Data Engineering*, 1988, pp. 470-477.

[Vie89] L. Vielle, "Recursive query processing: the power of logic," *Theoretical Computer Science*, 69:1(1989), 1-53.

[Ull88] J.D. Ullman, *Principles of Database and Knowledge-Base Systems,* vol. I, Computer Science Press, MD, 1988.

[Ull89] J.D. Ullman, *Principles of Database and Knowledge-Base Systems,* vol. II, Computer Science Press, MD, 1989.

[Ull89a] J.D. Ullman, "Bottom-up beats Top-down for Datalog," *ACM Symp. Principles of Datubose Systems,* 1989, pp. 140-149.