NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Optimization of Logic Queries in Knowledge Base Systems

Premchand Sukumaran Nair

A Thesis

in

The Department

of

Computer Science

Presented in Partial fulfillment of the Requirements
for the Degree of Doctor of Philosophy at
Concordia University
Montréal, Québec, Canada

May 1989

ISBN 0-315-51346-2

ABSTRACT

Optimization of Logic Queries in Knowledge Base Systems

Premchand Sukumaran Nair, Ph.D.
Concordia University, 1989

One of the ways for implementing a Knowledge Base System is to augment a traditional Relational Data Base System with deductive capabilities. A major issue has been to find an efficient implementation supporting recursive queries expressed in first order logic. An efficient optimization strategy, called *Integrated Magic Set (IMS) method*, is introduced and is shown to be superior to other known query optimization methods.

Two parallel algorithms for efficient query evaluation of Datalog programs have been developed which can be implemented in any distributed architecture. The performance comparisons between the parallel algorithms and a serial algorithm are presented using an analytical model. The analysis establishes the superiority of the parallel algorithms over the semi-naive serial algorithm.

Optimization techniques based on the concept of permutation dependency are presented. This achieves optimization at different levels: (1) for storing a base predicate, (2) for removing redundant rules from a Datalog program, (3) for rewriting a Datalog program so that the resultant program is more suitable for magic set optimization and (4) for efficient computation of a derived predicate satisfying a permutation dependency.

A classification of chain rule programs based on their computational complexity is provided. A non-regular chain rule

program is shown to be computationally more complex than a chain rule program. An algorithm to determine whether or not a given binary rule program has an equivalent chain rule program is presented.

Dedicated to our parents

Mr. S. Sukumaran Nair

Ms. A. Sarada Devi

Mr. P. N. Achutha Kurup

Ms. M. J. Sarasamma

Acknowledgements

I gratefully acknowledge the invaluable assistance of my thesis supervisors Prof. V. S. Alagar and Prof. F. Sadri in the course of the preparation of this thesis. I have received financial support, technical assistance and encouragement from them throughout my studies at Concordia University. My thanks are also to Prof. P. Goyal and Dr. J. Opatrny for the many stimulating discussions I had with them.

I thank the Quebec Ministry of Education for two fee remission awards, Mr. R. Narayanan, Head, Computer Division, Indian Space Research Organization for granting study leave enabling this research, Dr. N. P. V. Nair for his encouragements and the Faculty members of the Department of Computer Science, Concordia University for providing a stimulating environment to conduct this research.

I thank my wife Ms. Suseela for supporting my dreams.

Table of Contents

List of Figures

Chapter 1

INTRODUCTION

1.1 Knowledge Base Systems

Recent developments in *Knowledge Base Systems* clearly demonstrate a unifying confluence of three research areas: *artificial intelligence, logic,* and *database management systems*. Typically, a Knowledge Base System processes large volumes of explicit information (stored facts) and implicit information (facts derived by the application of rules). Several applications in medicine, engineering and business require Knowledge Base Systems for research and development.

From a database perspective, Knowledge Base Systems are database systems endowed with deductive reasoning capability. On the other hand from an artificial intelligence systems point of view, a Knowledge Base System is a reasoning system with increased processing efficiency achieved by using well-developed database technology. Knowledge base system technology is essentially the merging of two technologies, database system technology and Artificial intelligence technology, toward the realization of a new breed of information management systems with great processing power and high processing efficiency.

We have used the term *knowledge base* without precisely defining its intended meaning. Unfortunately, at present, there is no widely accepted definition. In this dissertation, the term *knowledge base* is used to mean "a collection of simple facts and general rules representing some universe of discourse" [Fros86]. The meaning of the terms *simple fact* (hereafter referred as *fact*), *general rule* (hereafter

referred as *rule*) and *universe of discourse* will be made clear later on in the thesis.

A knowledge base may consist of more facts than rules as, for example, in a knowledge base representing a flight schedule. On the other hand, it may consist of more rules than facts as, for example, in a knowledge base representing good chess moves. A knowledge base is distinct from a conventional database in the sense that it consists of *explicitly* stated rules as well as explicitly stated facts. A database on the other hand, typically consists of a small number of *implicitly* stored rules and a large number of explicitly stated facts. For example, a requirement such as "a student record must contain his date of birth" is enforced implicitly in a database. There would be data entry and validation rules that would require the database administrator to input some value for the date of birth when a student record is created. In contrast, in a Knowledge Base System the above requirement would be expressed explicitly, possibly using first order predicate logic. Further, Knowledge Base Systems include components which can make inference over the knowledge base, thereby providing a deductive capability to the system.

1.2 Architecture : The Deductive Augmentation of Relational Database Systems

1.2.1 Knowledge Base System Architectures

The research in artificial intelligence has lead to several kinds of reasoning system architectures such as *rule-based systems, logic-based systems* and *frame-based systems*. There are also several models of database systems. Most notable ones are *relational, hierarchical, network, entity-relationship and object-oriented* models. Thus there is

more than one way of combining a database system with a reasoning system. That is, several different Knowledge Base System architectures can be constructed. At present there are numerous proposals [Smit84, Fros86] on Knowledge Base System architectures and some of them are briefly reviewed below:

(1) One of the approaches is to start from an existing reasoning system and add database capabilities such as indexing, clustering, database accessing, authorization, concurrency control and recovery. A representative work of this kind is by Warren [Warr84] where a Prolog logic programming system is extended to include the functions and features of relational database systems.

(2) Another approach is to add deductive capabilities to an existing database system, in most cases, a relational database system. Some work has been done on augmenting relational database systems with a Prolog front-end [Jark84], but there are still many problems in "bridging" the gap between logic programming and relational database systems [Brod84, Zani84]. Many researchers are interested in using general Horn clauses, not constrained by Prolog syntax, semantics and control mechanisms, to enhance relational database systems with the power of deduction. This thesis is essentially a contribution to this area of research.

(3) In this approach, a Knowledge Base System is developed from scratch. Even though this approach may require more time and resources than the other two methods, this approach is expected to produce more fine tuned systems than the other two proposals.

1.2.2 The Deductive Augmentation of Relational Database Systems

The architecture considered in this thesis is deductive augmentation of relational database systems. In this architecture, we start with a core database system around which various kinds of high-level deductive query interfaces, natural language interfaces, and more application oriented modules can be added.

A simple architecture for Knowledge Base Systems is presented next:

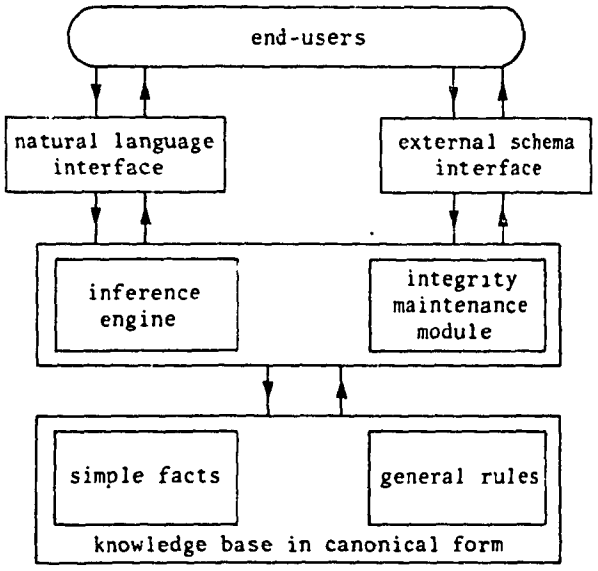


Figure 1.1 A simple architecture for knowledge base systems

Our study is focused on the inference engine module; in particular the emphasis here is on the issue of efficient processing of queries expressed using first order logic. This study is believed by many researchers to be one of the most important issues in Knowledge Base Systems research.

1.3 Goals of this Research

The research reported in this thesis is on the efficient realization of inference engines that can process queries expressed using Horn Clause rules. We study the compilation and processing of Horn clause programs without function symbols, called *Datalog* programs and investigate several optimization techniques.

1.4 Organization of the Dissertation

The research contribution in this dissertation is organized into seven chapters.

Chapter 1 introduces the concept of Knowledge Base Systems, their applications, and associated research issues. An outline of the research and the Knowledge Base System architecture - *deductive augmentation of relational database systems* - that will be investigated is presented.

Chapter 2 is a general discussion on logic databases. After stating preliminary assumptions and definitions three strategies for recursive database query processing are introduced. These methods are necessary for our discussion in later chapters. Interested reader can find a comprehensive survey of query processing strategies in [Banc86b].

There exist several strategies [Hens84, Banc85, Banc86a, Banc86b, Sacc87, Agra88, Kife88] to optimize linear recursive queries in deductive databases. Among these proposals some are intended to process linear recursive queries with some variables bound and other proposals are aimed at processing general linear queries with no variables bound. In Chapter 3, we propose a new algorithm to process linear recursive queries with some variables bound. This method can be viewed as an integration of well known magic set method with the approach taken in [Agra88]. One of the reasons for optimizing queries with some variables bound is that, in the real world such queries are encountered most frequently.

In Chapter 4, we focus on the issue of parallel processing in a multiprocessor system environment. We begin our discussion considering the class of algorithms to optimize a Datalog program with no variables bound. The best known algorithm of this group is the semi-naive evaluation method proposed in [Banc85]; efficient implementation of algorithms for processing general linear queries still remains an open problem. Note that one of the means for achieving the speed-up is to exploit the possible parallelism. It is quite surprising that, until recently, no attempt has been made to transform a general linear recursive query so that the result of the bottom up evaluation of the transformed parallel program is the same as that of the original program. In this chapter we propose two parallel algorithms for evaluating a large class of linear recursive programs, which we call *vertically partitionable programs*. Consequently, the load of evaluating the predicates can be divided among a number of processors with minimal need for inter-process communication. Thus the method is suitable for a large class of architectures. Further the class of programs that can be vertically partitioned include among

others, the same generation program [Banc86a, Banc86b], which is a canonical example of linear Datalog programs and, hence, form an interesting subclass of the class of linear Datalog programs. We also address the issue of selection propagation in the presence of bound variables. One of the parallel algorithms proposed in this chapter is capable of pushing selections before join and thus make effective use of bound variables appearing in the query.

A new data dependency, namely, the permutation dependency, is introduced in the Chapter 5. An efficient scheme for maintaining minimal knowledge in the presence of a permutation dependency is presented. Since this scheme eliminates the redundancy in the initial value of the recursive predicate, cost of query processing is reduced by a constant factor. It is shown that this method improves over other conventional methods in both space and time.

It is felt that complexity theory can provide a suitable framework for better understanding of algorithms that process recursive queries. One such framework for determining the lower and upper bounds of logic queries is presented in Chapter 6. In particular, we show that the well known same generation program is computationally more difficult than the transitive closure program. A simple algorithm to transform a binary logic program to a chain rule program is presented.

This dissertation is concluded in Chapter 7, with suggestions for some possible future research directions.

Chapter 2

LOGIC DATABASES

This chapter is a brief summary of basic definitions and notations on logic databases as described in the survey paper by Bancilhon and Ramakrishnan [Banc86b].

2.1 Syntax of a Logic Database

We define four sets of names: *variable* names, *constant* names, *predicate* or *relation* names and *evaluable predicates* names. The variables are denoted by strings of characters starting with lower case letters and constants are denoted by strings of characters starting with an upper case letter or integer. Identifiers starting with upper case letters are used for relation or predicate (evaluable and non-evaluable) names, allowing both subscripts as well as superscripts.

The terms relation (from database terminology) and predicate (from logic terminology) are used to represent the same object. There is a fixed dimension associated with each predicate. The set of evaluable predicate names is a subset of the set of predicate names. We will not be concerned with their syntactic recognition; in the examples it will be clear from the name we use.

A *literal* is of the form $P(x_1, \cdots, x_n)$ where P is a predicate name of dimension n and each x_i is a variable or a constant. An *instantiated* literal is one which does not contain any variables. We write evaluable literals using functions and equality for the purpose of clarity. For instance, $z = x + y$ denotes SUM(x, y, z), $j = i + 1$ denotes SUM(i, 1, j), and $x > 0$ denotes GT(x, 0). If $P(x_1, \cdots, x_n)$ is

a literal, then we call $(x_1, \cdots, x_n)$ a *tuple*.

A rule is a statement of the form

$P :- Q_1, Q_2, \cdots, Q_k.$

where P and Q_i's are literals such that the P is a non-evaluable predicate. P is called a *head* of the rule, and each of the Q_i's is called a goal. The conjunction of the Q_i's is the body of the rule.

A ground clause is a rule in which the body is empty. A *fact* is a ground clause which contains no variables. A *logic database (logic program)* is an unordered set of rules. Below is an example of a logic program taken from [Banc86b].

Example 2.1

r_1 : Parent(Cain, Adam).
r_2 : Parent(Abel, Adam).
r_3 : Parent(Cain, Eve).
r_4 : Parent(Abel, Eve).
r_5 : Parent(Sem, Abel).
r_6 : Ancestor(x, y) :- Parent(x, z), Ancestor(z, y).
r_7 : Ancestor(x, y) :- Parent(x, y).
r_8 : SG(x, x).
r_9 : SG(x, y) :- Parent(x, x_1), Parent(y, y_1), SG(x_1, y_1).
r_{10} : Gen(x, i) :- Gen(y, j), Parent(x, y), j = i-1.
r_{11} : Gen(x, i) :- Gen(y, j), Parent(y, x), j = i+1. □

2.2 Semantics of a Logic Database

Up to now our definitions have been purely syntactical. Let us now give an *interpretation* of a database. This will be done by associating with each relation in the database a set of instantiated tuples. We first assume that with each evaluable predicate P is associated a set natural(P) of instantiated tuples which we call its *natural interpretation*. For instance, with the predicate SUM(x, y, z) is associated an infinite set of all 3-tuples (x, y, z) of integers such that the sum of x and y is z. In general the natural interpretation of an evaluable predicate is infinite.

Given a database, an *interpretation* of this database is a mapping which associates with each relation name a set of instantiated tuples. A *model* of a database is an interpretation I such that:

(1) For each evaluable predicate P,

$$I(P) = \text{natural}(P), \text{ and}$$

(2) For any rule,

$$P(t) :- Q_1(t_1), Q_2(t_2), \cdots, Q_n(t_n).$$

and for any instantiation σ of the variables of the rule such that for $i = 1, 2, \cdots, n$, $\sigma(t_i)$ is in the interpretation of the predicate $Q_i(t_i)$ implies that $\sigma(t)$ is in the interpretation of P(t).

Notice that a given database may have many models. The nice property of Horn Clauses is that among all these models there is a *minimal* one (in the sense of set inclusion), which is chosen as the model or interpretation of a database. Notice that because of the presence of evaluable predicates the minimal model, in general, is not finite.

Let P be an n-dimensional predicate. An *adornment* of P [Ullm85] is a sequence of length n of b's and f's. For instance, bbf is an adornment of a 3-dimensional predicate. An adornment is intuitively interpreted as follows: the i^{th} variable of P is bound (respectively free) if the i^{th} element of a is b (respectively f). A *query form* is an adorned predicate and a *query* is a query form and an instantiation of bound variables. We denote it by an adorned literal where all the bound positions are filled with corresponding constants and all free positions are filled with distinct free variables. The *answer* to a query q(t) is the set:

$$\{\ q(\sigma(t)) : \sigma \text{ is an instantiation of t, and } \sigma(t) \text{ is in the interpretation of q}\}$$

2.3 Structure and Representation of a Logic Database

A logic database can be partitioned into a set of facts and the set of all other rules. The set of facts is called the *extensional database*, and the set of all other rules is called the *intensional database*. A predicate which only appears in the intensional database is called an *intensional (derived) predicate* and a predicate which appears only in the body of any rule is called an *extensional predicate*.

Most authors have chosen to describe a set of rules through some kind of graph formalism. We use *rule/goal graphs* as presented in [Ullm85] while using unadorned predicates.

The rule/goal graph has two sets of nodes: square nodes are associated with predicates, and oval nodes are associated with rules. If there is rule of the form

$r : P :- Q_1, Q_2, \cdots, Q_k.$

in the intensional database, then there is a directed edge going from node r to node P, and for all i ($1 \leq i \leq k$) there is a directed edge from node Q_i to node r.

The following example is taken from [Banc86b].

Example 2.2

Program RG

$r_1 : P_1 :- P_3, P_4.$
$r_2 : P_2 :- P_4, P_5.$
$r_3 : P_3 :- P_6, P_4, P_3.$
$r_4 : P_4 :- P_5, P_3.$
$r_5 : P_3 :- P_6.$
$r_6 : P_5 :- P_5, P_7.$
$r_7 : P_5 :- P_6.$
$r_8 : P_7 :- P_8, P_9.$

Rule/goal graph of Program RG is as follows.

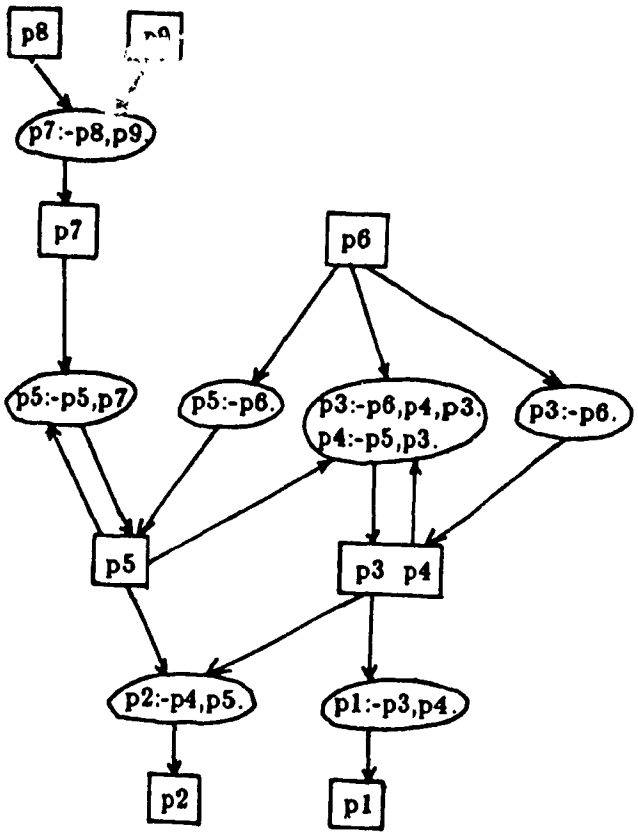


Figure 2.1 Rule/goal of Program RG.

□

2.4 Recursion

Recursion is often discussed in the single rule context. We now give the general definition of recursion in a multirule context. Let P and Q be two predicates. We say Q derives P (denoted by Q → P) if Q occurs in the body of a rule whose head predicate is P. Define →+ to be the transitive closure (*not* the reflexive transitive closure) of →. A predicate P is *recursive* if P →+ P. Two predicates P and Q are *mutually recursive* if P →+ Q and Q →+ P. It can be easily

shown that mutual recursion is an equivalence relation on the set of recursive predicates. Therefore the set of recursive predicates can be decomposed into disjoint blocks of mutually recursive predicates.

Given a set of rules, we say that the rule with head predicate P is recursive if there exists a predicate Q in the body of the rule which is mutually recursive to P and is *linear* if there is one and only one predicate in the body of the rule which is mutually recursive to P. A set of rules is *linear* if every recursive rule in the set is linear. We say two recursive rules or mutually recursive if their heads are mutually recursive. This defines an equivalence relation among the recursive rules. Since mutual recursion defines an equivalence relation among the recursive predicates and among the recursive rules, we can group together all the predicates which are mutually recursive to one another. In fact these predicates are to be evaluated as a whole. This grouping further puts together all the rules which participate in evaluating one equivalence class of predicates under mutual recursion. We now proceed to show how this can be represented in the rule/goal graph. Define the *reduced rule/goal graph* as follows:

Square nodes are associated with non-recursive predicates or with blocks of mutually recursive predicates, and the oval nodes are associated with non-recursive rules are mutually recursive rules. The graph essentially describes the non-recursive part of the database by grouping together all the predicates which are mutually recursive to one another and isolating the recursive parts. For every non-recursive rule of the form

r : P :- Q_1, Q_2, ···, Q_k.

there is a directed edge from node r to node P (if P is non-recursive),

or to node [P], the node representing the set of predicates mutually recursive to P (if P is recursive). For each non-recursive predicate Q_i there is a directed edge from node Q_i to node r and for each recursive predicate Q_j there is a directed edge from $[Q_j]$ to the node representing set of predicates mutually recursive to Q_j. Finally, each block of recursive rules [r] is uniquely associated to a set of mutually recursive predicates [P], and we draw a directed edge from [r] to [P]. We also draw a directed edge from Q (if Q is non-recursive) or from [Q] (if Q is recursive) to [r] if there is a rule in [r] which has Q in its body.

This grouping of predicates in blocks of strongly connected components is presented in [Brac86b]. We now proceed to give the reduced rule/goal graph of Program RG in Example 2.2.

Example 2.2

Reduced rule/goal graph of Program RG is given below:

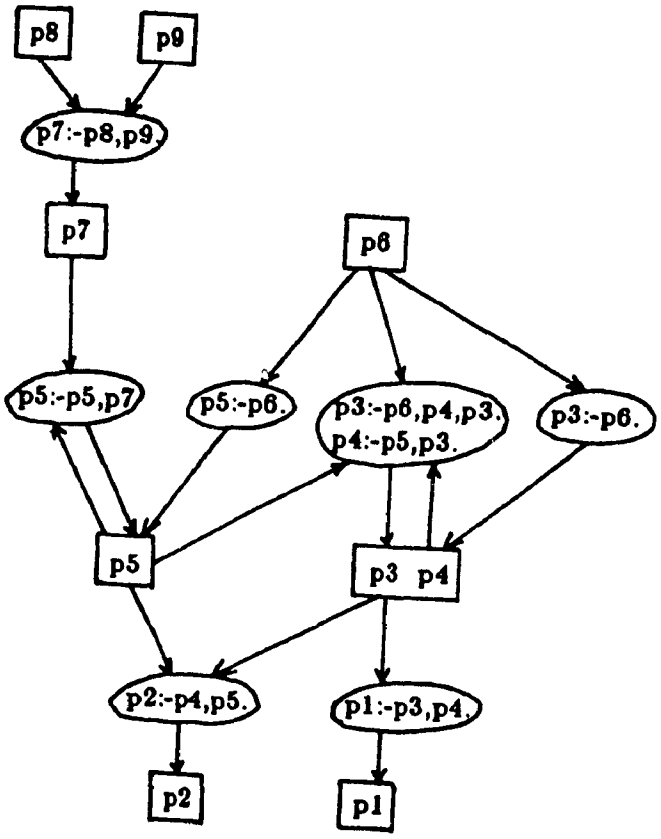


Figure 2.2 Reduced rule/goal graph of Program RG.

□

2.5 Safety of Queries

Given a query in a database D, the query is said to be *safe* in D if the answer to the query is finite. Clearly, unsafe queries are highly undesirable. The sources of unsafeness are of two kinds: (1) The evaluable arithmetic predicates are interpreted as infinite tables and they are unsafe by definition. For instance the query GT(27, x) is unsafe. (2) Rules with free variables in the head which do not

appear in the body are sources of unsafeness in the presence of evaluable arithmetic predicates. The problem of safety has received a lot of attention [Afra86, Ullm85, Zani86] and we shall not survey those results here, but merely present some simple sufficient syntactic conditions to guarantee safety.

A rule is *range restricted* if every variable of the head predicate appears somewhere in the body. A set of rules is range restricted if every rule in the set is range restricted. It is known [Reit78] that if each evaluable predicate has a finite natural interpretation and if the set of rules is range restricted, then every query defined over this set of rules is safe. This applies obviously to the case where there are no evaluable predicates. However, if there are evaluable predicates with infinite natural interpretations, safety is no longer assured.

We now proceed to present a simple sufficient condition for safety in the presence of such predicates. A rule is said to be *strongly safe* if (1) it is range restricted, and (2) every variable in an evaluable predicate term also appears in at least one base predicate.
For example, the rule

Well-Paid(x) :- Has-Salary(x, y), y $\geq$ 100K.

is strongly safe, whereas

Great-Salary(x) :- x $\geq$ 100K.

is not strongly safe.

A set of rules is strongly safe if every rule in this set is strongly safe. Any query defined over a set of strongly safe rules is safe.

2.6 Effective Computability of Queries

Safety, in general, does not guarantee that the query can be effectively computed. For instance:

$P_1(1, x, y)$:- $x \geq y$.

$P_2(x, y, 2)$:- $x \leq y$.

$P(x, y)$:- $P_1(x, z, z)$, $P_2(z, z, y)$.

Query(x, y) :- P(x, y).

The query is safe (the answer is {(1, 2)}), but there is no safe computation for it. However, strongly safe rules are guaranteed to be safe and safely computable.

We now present a sufficient condition for ensuring safe computability of recursive logic queries. First, we characterize each arithmetic predicate by a set of *safety dependencies* [Zani86]. A safety dependency is a couple $(X \rightarrow Y)$ where X and Y are sets of attributes such that "if the values of the X attributes are fixed then there is a finite number of values of the Y attributes associated with them". Thus, while their semantics are different from functional dependencies, they behave in the same fashion (and have the same axiomatization). We assume that the natural interpretation of the evaluable predicates satisfies the set of safety dependencies. For instance, the evaluable arithmetic predicate "SUM(x, y, z)" has the following safety dependencies:

$\{x, y\} \rightarrow \{z\}$

$\{y, z\} \rightarrow \{x\}$

$\{z, x\} \rightarrow \{y\}$

while the arithmetic predicate "GT(x, y)" has only trivial safety dependencies.

For any rule define recursively that each variable in the body is *safely computable* if it appears in a non-evaluable predicate in the body or if it appears in the position i in an evaluable predicate P and there is a subset I of safely computable variables of P such that $I \rightarrow \{i\}$.

A rule is *bottom-up evaluable* if

1. it is range restricted, and
2. every variable in the body is safely computable.

2.7 Language of a Logic Database

Consider the following set of rules and the query:

Ancestor(x, y) :- Parent(x, z), Ancestor(z, y).
Ancestor(x, y) :- Parent(x, y).
query(x) :- Ancestor(John, x).

We can view each of these rules as a production in a grammar. In this context, the database predicates (Parent in this example) appear as terminal symbols, and the derived predicates (Ancestor in this example) can be viewed as non-terminal symbols. Finally, to pursue the analogy, we shall take the distinguished symbols to be query(x). Of course, the analogy does not hold totally, for two reasons: (i) the presence of variables and constants in the literals and (ii) the lack of ordering between the literals of a rule. But we ignore these differences, and use the analogy informally. The language generated by this "grammar" consists of

{Parent(John, x); Parent(John, x), Parent(x, x_1);
Parent(John, x), Parent(x, x_1), Parent(x_1, x_2); ···; }.

This language has two interesting properties: (i) it consists of first order sentences involving only base predicates, that is, each word of this language can be directly evaluated against the database, and (ii) if we evaluate each word of this language against the database and take the union of all these results, we get the answer to the query.

There is however a minor problem here: the language is not finite, and we would have to evaluate an infinite number of first order sentences. To get out of this difficulty, we use termination conditions. An example of a termination condition is "one word of the language evaluates to empty set".

Two logic databases are said to be query equivalent if the answer to the query "$query_1$" of the first database is equal to the answer to the query "$query_2$" of the second database, for all possible values of base relations. In particular, if any two logic databases have the same language they are query equivalent.

From now on, no distinction is made between terms *logic database* and *logic program*. Further, a *Datalog program* is a logic program without function symbols.

2.8 Characteristics of Query Evaluation Strategies

2.8.1 Interpretation vs. Compilation

A method can be *interpreted* or *compiled*. We say that a strategy is compiled if it consists of two phases:(i) a compilation phase, which accesses only the intensional database, and which generates an "object program" of some form, and (ii) an execution phase, which executes the object program against the facts only. A second characteristic of compiled methods is that all database query

forms that occur during the query processing are generated during the compilation phase. This condition is very important, because it allows the DBMS to precompile the query forms. Otherwise the database query forms are repetitively compiled by the DBMS during the execution of the query, which is a time consuming operation. If these two conditions do not hold, we say that the strategy is interpreted. In this case no object code is produced and there is a fixed program the "interpreter", which runs against the query, the set of rules and set of facts.

2.8.2 Recursion vs. Iteration

A rule processing strategy can be recursive or iterative. The method is iterative if the "target program" (in case of a compiled approach) or the "interpreter" (in the case of a interpreted approach) is iterative and is recursive otherwise. Note that in the case of iterative methods, the data we deal with are statically determined. For instance, if we use temporary relations to store intermediate results, there are a finite number of such temporary relations. On the contrary, in recursive methods the number of temporary relations maintained by the system is unbounded.

2.8.3 Top-Down vs. Bottom-Up

The bottom-up strategies start from the terminals (or base relations) and keep assembling them to produce non-terminals (or derived relations) until they generate the distinguished symbol (or the query). The top-down strategies start from the distinguished symbol (or the query) and keep expanding it by applying the rules to the non-terminals (or the derived relations).

2.9 A Brief Discription of Three Basic Strategies

The main purpose of this section is to give a glimpse of three different strategies which we need in the sequel. For other major strategies the reader is referred to [Banc86b].

2.9.1 Naive Evaluation

Naive Evaluation is a bottom-up, compiled, iterative strategy. In the first phase, the rules which derive the query are compiled into an iterative program. The compilation process uses the rule/goal graph. It first selects all rules which derive the query. A temporary relation is assigned to each derived predicate in this set of rules. A statement which computes the value of the output predicate from the value of the input predicates is associated with each rule node in the graph. With each set of mutually recursive rules, there is associated a loop which applies the rules in that set until no new tuple is generated. Each temporary relation is initialized to the empty set. Then computation proceeds from the base predicates computing the relations at the nodes of the graph. We illustrate the method through the following example.

Example 2.4

Consider the logic database obtained by adding a query rule

r_{12} : Query(x) :- Ancestor(Abel, y).

to the rules in Example 2.1. The rules which derive the query are

r_6 : Ancestor(x, y) :- Parent(x, z), Ancestor(z, y).
r_7 : Ancestor(x, y) :- Parent(x, y).

r_{12} : Query(x) :- Ancestor(Abel, y).

and there are two temporary relations: ancestor and query. The method consists in applying r_7 to parent, producing a new value for ancestor, then applying r_6 to ancestor until no new tuple is generated, then applying r_{12}.

The program based on naive evaluation method is:

begin
initialize ancestor to the empty set;
evaluate(ancestor(x, y) :- parent(x, y)); insert the result in ancestor;
while "new tuples generated" *do*
    begin
    evaluate(ancestor(x, y) :- parent(x, z), ancestor(z, y))
        using the current value of ancestor;
    insert the result in ancestor;
    end;
evaluate(query(x) :- ancestor(Abel, x);
insert the result in query;
end. □

This algorithm has the following drawbacks: (1) the entire relation is evaluated. That is, the set of potentially relevant facts is the set of facts of the base predicates which derive the query, and (ii) while loop will be executed one more time than necessary in the sense that no new tuple is produced in the last iteration.

Naive evaluation is the most widely described method in literature. For more details, see [Banc86b].

2.9.2 Semi-Naive Evaluation

Semi-naive evaluation is a bottom-up, compiled and iterative strategy. This method uses the same approach as naive evaluation, except that it tries to cut down on the number of duplications. It behaves exactly as naive evaluation, except for the loop mechanism where it tries to be smarter.

Let us give an idea of the method as an extension of naive evaluation. Let P be a recursive predicate. Consider a recursive rule having P as a head predicate and let us write this rule:

$P :- \phi(P_1, P_2, \cdots, P_n, Q_1, Q_2, \cdots, Q_m)$.

where ϕ is a first order formula, P_i are mutually recursive to P, and Q_i are base predicates. In the naive evaluation strategy, all the Q_i's are fully evaluated when we start computing P and P_i's. On the other hand P and P_i's are all evaluated inside the same loop (together with the rest of predicates mutually recursive to P).

Let P_j^i be the value of the predicate P_j at the i^{th} iteration. At this iteration, we compute $\phi(P_1^i, P_2^i, \cdots, P_n^i, Q_1, Q_2, \cdots, Q_m)$. During that same iteration each P_j receives a set of new tuples. Let us call this set Diff-P_j^i. Thus $P_j^{i+1} = P_j^i$ + Diff-P_j^i, where + stands for union. At step (i+1) we evaluate $\phi(P_1^{i+1}, P_2^{i+1}, \cdots, P_n^{i+1}, Q_1, Q_2, \cdots, Q_m)$, Since ϕ is monotone, from the above expression it follows that the entire computation performed during the i^{th} iteration is repeated at the $i+1^{st}$ iteration. The ideal however, is to compute only the new tuples.

The basic principle of the semi-naive method is the evaluation of the differential of the head predicate, rather than the entire predicate at each step. Thus the problem is to come up with a first order expression without any difference operator. Let us assume, for a

moment that there exists such an expression, and describe the algorithm. With each recursive predicate P are associated four temporary relations P-before, P-after, Dif-P-before, Dif-P-after. Then the object program for the loop is as follows:

for all mutually recursive predicates P *do*
    initialize P-before to empty;
repeat
    for all mutually recursive predicates P *do*
        begin
        initialize Dif-P-after to empty set;
        initialize P-after to P-before;
        end;
    for each mutually recursive predicates P *do*
        begin
        evaluate the differential of P;
        add the resulting tuples to Dif-P-after and P-after;
        end;
until Dif-P-after and Dif-P_j-after are empty for all j.

The problem now reduces to finding a first order formula for the differential.

The idea of semi-naive evaluation is given in [Banc85]. In the case of linear rules, the expression for the differential is obtained by replacing the recursive predicate in the body by its differential. Hence in this case, the program for the database in Example 2.4 is written as below:

begin
initialize ancestor to the empty set;

evaluate(ancestor(x, y) :- parent(x, y)); insert the result in ancestor;
insert the result in dif-ancestor;
while dif-ancestor is not empty *do*
    begin
    evaluate(ancestor(x, y) :- parent(x, z), dif-ancestor(z, y))
    using the current value of ancestor;
    insert the result in ancestor;
    end;
evaluate(query(x) :- ancestor(Abel, x);
insert the result in query;
end.

2.9.3 Magic Set Method

The goal of the magic set optimization method is to transform a logic program into a query equivalent program so that the bottom-up evaluation can be done more efficiently. In this section, we give a brief description of magic set method. See [Banc86a, Banc86b] for a detailed discussion.

Given a system of rules, the method first obtains an *adorned rule system* as follows: Define recursively that a variable is *bound* if it is either a constant or it appears in the rule head as a bound variable (as indicated by the *adornment*) or it appears in a base predicate with at least one bound variable. The *adornment* of a predicate $P(x_1, \cdots, x_n)$ in a rule r, is a sequence of characters $a_1 \cdots a_n$ such that a_i is b (for bound) if the x_i is a bound variable and is f (for free) otherwise [Ullm85]. The adorned rule is obtained by replacing each derived literal by its adorned version. For example, consider the system of rules:

$$\begin{aligned} &Sg(x,\ x). \\ &Sg(x,\ y)\ :-\ P(x,\ x_p),\ Sg(y_p,\ x_p),\ P(y,\ y_p). \\ &query(y)\ :-\ Sg(C,\ y). \end{aligned} \tag{2.1}$$

where C is a constant. Clearly, the adorned query rule is given by

$$query^{f}(y)\ :-\ Sg^{bf}(C,\ y).$$

Since this rule contains an adorned predicate Sg^{bf} in the body, the following adorned rules are generated next.

$$\begin{aligned} &Sg^{bf}(x,\ x). \\ &Sg^{bf}(x,\ y)\ :-\ P(x,\ x_p),\ Sg^{fb}(y_p,\ x_p),\ P(y,\ y_p). \end{aligned}$$

Observe that since x is bound in the head, x is bound in P. Since P is a base predicate and x is a bound variable, x_p is bound and hence the adornment ^{fb} for the antecedent Sg. Thus the binding is propagated from one predicate in the body of the rule to another predicate in the body of the same rule. This scheme of information passing is known as the *sideways information passing*. Note that the adorned predicate Sg^{fb} appears in the body of the rule above. Therefore the following rules are included into the system of adorned rules.

$$\begin{aligned} &Sg^{fb}(x,\ x). \\ &Sg^{fb}(x,\ y)\ :-\ P(x,\ x_p),\ P(y,\ y_p),\ Sg^{bf}(y_p,\ x_p). \end{aligned}$$

Since all the adorned predicates that appeared in the body have been

considered, all the necessary adorned rules are already generated.

A brief description of the transformation is given below; for a detailed description see [Banc86a, Banc86b]

Step 1 *For* each occurrence of a derived predicate in the body of an adorned rule *do*

{ Generate—a—magic—rule }

For each adorned literal predicate P in the body of the adorned rule *r* *do*

begin

1.1. Remove all the other derived literals in the body.

1.2. In the derived predicate occurrence, replace the name of the predicate by magic.P^{a}, where a is the literal adornment; and remove variables that are free according to a.

1.3. Remove all base predicates with no bound variables.

1.4. In the head, replace the name of the predicate $S^{a'}$ by magic.$S^{a'}$, where a' is the adornment of the predicate S, and remove variables that are free according to a'.

1.5. Exchange the two magic predicates.

end

Step 2 *For* each adorned rule *do*

{ Generate—a—modified—rule }

For each rule with adorned literal predicate P^a as head *do*

begin

Include the predicate magic.$P^a(x)$ in the body where x is the list of variables that are bound according to a.

end

An application of the first step on the adorned rules

$Sg^{bf}(x, y)$:- $P(x, x_p)$, $P(y, y_p)$, $Sg^{fb}(y_p, x_p)$.

$Sg^{fb}(x, y)$:- $P(x, x_p)$, $P(y, y_p)$, $Sg^{bf}(y_p, x_p)$.

generate the magic rules:

magic.$Sg^{fb}(x_p)$:- $P(x, x_p)$, magic.$Sg^{bf}(x)$.

magic.$Sg^{bf}(y_p)$:- $P(y, y_p)$, magic.$Sg^{fb}(y)$.

Similarly, an application of step 2 on the same set of adorned rules generates the following modified rules

$Sg^{bf}(x, y)$:- magic.$Sg^{bf}(x)$, $P(x, x_p)$, $P(y, y_p)$, $Sg^{fb}(y_p, x_p)$.

$Sg^{fb}(x, y)$:- magic.$Sg^{fb}(y)$, $P(x, x_p)$, $P(y, y_p)$, $Sg^{bf}(y_p, x_p)$.

Thus the modified set of rules for (2.1) is

magic.$Sg^{fb}(x_p)$:- $P(x, x_p)$, magic.$Sg^{bf}(x)$.

magic.$Sg^{bf}(y_p)$:- $P(y, y_p)$, magic.$Sg^{fb}(y)$.

magic.$Sg^{bf}(C)$.

$Sg^{bf}(x, y)$:- magic.$Sg^{bf}(x)$, $P(x, x_p)$, $P(y, y_p)$, $Sg^{fb}(y_p, x_p)$. (2.2)

$Sg^{fb}(x, y)$:- magic.$Sg^{fb}(y)$, $P(x, x_p)$, $P(y, y_p)$, $Sg^{bf}(y_p, x_p)$.

$Sg^{bf}(x, x)$:- magic.$Sg^{bf}(x)$.

$Sg^{fb}(x, x)$:- magic.$Sg^{fb}(x)$.

$query^{f}(x)$:- $Sg^{bf}(C, x)$.

Now the query processing may take place in two stages. In the first stage, only the magic sets are computed through a bottom-up procedure. Note that during this computation. only the base predicates in the body of the recursive rule take part. The second stage uses those sets to filter the data in evaluating the query.

Chapter 3

INTEGRATED MAGIC SET METHOD

3.1 Introduction

In this chapter a new optimization method, called *Integrated Magic Set (IMS) method*, for efficient processing of logic queries is presented. This method is a harmonious integration of Magic Set (*MS*) [Banc86a] method and Agrawal-Devanbu (*AD*) method [Agra88] of "moving selections into linear least fixpoint queries". Depending on the structure of the recursive rule and query form, this rule rewriting strategy may degenerate to either *AD*-method or *MS*-method. The complexity analysis shows that *IMS*-method is superior to *MS*-method.

Numerous strategies [Hens84, Banc85, Banc86a, Banc86b, Sacc87, Agra88, Kife88] have been proposed to optimize linear recursive queries in deductive databases. Among these proposals some are intended to process linear recursive queries with some variables bound and other proposals are aimed at processing general linear queries with no variables bound. In this chapter we propose a new algorithm to process linear recursive queries with some variables bound. One of the reasons for optimizing linear recursive queries with some variables bound is that, in the real world such queries are encountered most frequently.

One optimization strategy that has proven to be the most useful and has been implemented invariably in all relational database systems is that of performing selections as early as possible in the course of processing a query. Both Magic Set and Magic Counting

(*MC*) methods [Banc86a, Sacc87] are rewrite systems capable of performing early selections; that is, the optimization consists in generating a new set of rules from the given set so that (i) the two sets are equivalent with respect to the given query and (ii) the bottom-up evaluation is more efficient for the transformed rules. Historically, the heuristic of "performing selections first" was put forward by Aho and Ullman [AhoU79] and their algorithm optimizes recursive queries by commuting selections with the Least Fixpoint (*LFP*) operator. In [Chan80] it is shown that the relational algebra without set-difference together with the *LFP* operator is equivalent in expressive power to the Horn clause programs without any function symbols. Thus the importance of *LFP* operator became quite clear. The approach in [Agra88] by Agrawal and Devanbu is an extension of the Aho-Ullman algorithm [AhoU79]. Thus, *MS*-method and *AD*-method seem to be entirely different but promising methods. Although no theoretical work to compare the efficiency of these methods has been done, there is a framework [Marc87] to compare the computational behavior of *MS*-method and the counting method [Banc86a]. Because these two methods use similar strategies for propagating bindings, they can be combined as demonstrated by the Magic Counting (*MC*) method [Sacc87]. On the other hand the strategies for propagating bindings are totally different for *MS*-method and *AD*-method.

In this chapter we combine *MS* and *AD* methods so that the size of the relevant set of facts accessed to process a query is minimal. We describe an optimization strategy, called *IMS*-method, having the following features: (i) It is a harmonious combination of *MS* and *AD* methods; (ii) Depending on the structure of the rules and query form, *IMS*-method may degenerate either to *AD* or *MS*-method; (iii) It is a

rule rewriting system which generates a system of rules equivalent to the given set of rules and (iv) The complexity of *IMS*-method, measured by the size of the facts generated, is a lower bound to the achievable complexities of *MS* and *AD* methods. A by-product of this work is a classification of logic programs based upon the structure of the recursive rules.

3.2 Preliminaries

This section gives a brief summary of definitions and basic results.

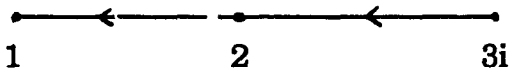
The *propagation graph* of a linear recursive rule [Agra88] can be constructed as follows:

(1) Create a node corresponding to every argument position in the consequent. Label the nodes with the corresponding argument positions.

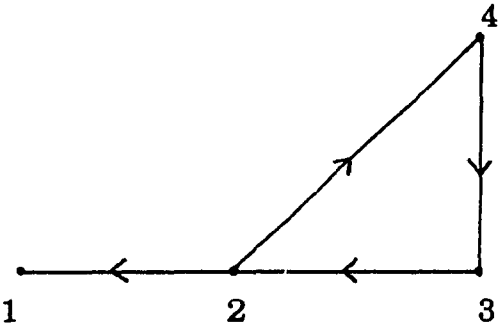
(2) Draw a directed arc from node u to node v if the argument position u of the antecedent and the argument position v of the consequent are occupied by the same variable.

(3) If a variable in the consequent does not occur in the antecedent, the corresponding node is an intruder node, and is additionally labeled i (for intruder).

Two linear rules and their corresponding propagation graphs are given next.



(a) A(x, y, z) :- A(w, x, y), B(y, z).



(b) P(x, y, z, x) :- P(w, x, y, z), Q(x, z).

Figure 3.1 Propagation Graphs

A propagation graph is a representation of how the values in the coordinates of a tuple originate and flow as the fixpoint query evaluation proceeds. For example, consider the bottom-up evaluation of A in Figure 3.1(a). It is easy to see that the value of the first coordinate of a tuple t_i in A generated at iteration level i would be the same as the value of the second coordinate of the tuple t_{i-1} in A which participated in the bottom-up evaluation of t_i. Similarly, the value of the second coordinate of the tuple t_{i-1} generated at level i−1 is the value of the third coordinate of the tuple t_{i-2} in A that participated in bottom-up evaluation of t_{i-1}. The intruder value is picked up from a non-recursive relation at each step of the iteration. In Figure 3.1(b), the value of the first coordinate of a tuple t_i (in P) generated at iteration level i is the value at the second coordinate of a tuple t_{i-1} which participated in the generation of t_i. Similarly, the value of the second coordinate of the tuple t_{i-1} generated at iteration

level i-1 is the value at the third coordinate of a tuple t_{i-2} which participated in the generation of t_{i-1}, the value of the third coordinate of a tuple t_{i-2} generated at iteration level i-2 is the value at the fourth coordinate of a tuple t_{i-3}, the value at the fourth coordinate of t_{i-3} is the value at the second coordinate of the tuple t_{i-4}, and so on.

We also borrow form [Agra88] the definition of an *ordered set of equivalence constraints* for a rule. Given a rule, the ordered set of equivalence constraints for the rule, indexed by i, is defined to be

$$\bigcup_{i=0}^{\infty} EC_i$$

where EC_i is the set of equality conditions to be satisfied among the coordinates of a tuple t in the recursive predicate so that t may fire some tuple in i iterations. Even though i is shown to vary from 0 to ∞, i can take only a finite number of values for a given rule. Each EC_i is a conjunction of a set of equality predicates; it is generated by grouping together all the coordinates that must have equal values in order to generate a tuple in i iterations. These groupings are called *equivalence groupings*; they are indexed by i, and each one is a partition of the subset of coordinate positions that are in the propagation graph. A formal description of the algorithm based upon the informal discussion in [Agra88] for computing the equivalence groupings S_i's is given below:

Algorithm Equivalence-Grouping

INPUT : The propagation graph of a linear recursive rule.

OUTPUT : The equivalence groupings; S_i, i = 0, 1, ···.

1. i ← 0;

$S_i \leftarrow$ {{k} | k is a node (in the propagation graph) with in-degree > 1 }

2. *repeat*

$S_{i+1} \leftarrow S_i$;

For each s *in* S_i *do*

begin

$\hat{s} \leftarrow$ { m | <m, n> is an edge of the propagation graph and $n \in s$};

$S_{i+1} \leftarrow S_{i+1} \cup \hat{s}$;

end;

"merge intersecting members of S_{i+1}";

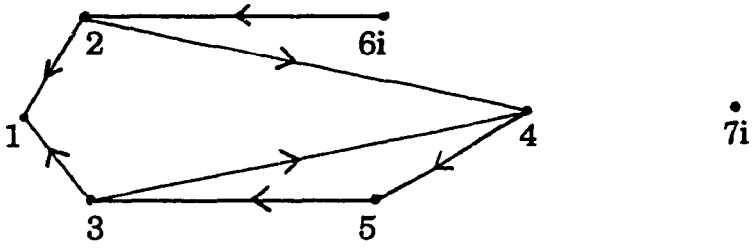
{ to form an equivalence class }

$i \leftarrow i + 1$;

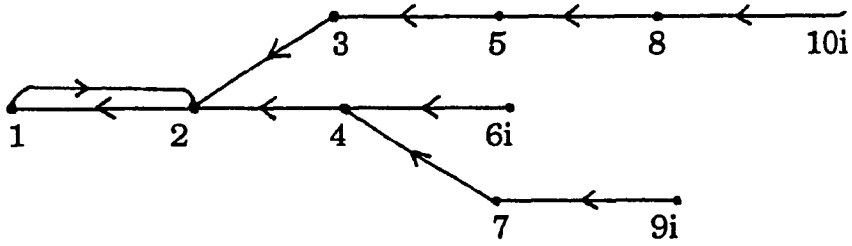
until $(S_{i+1} = S_i)$;

Since $\bigwedge(Col_p = Col_k)$, p, $k \in s \in S_i$, $p \neq k$ defines EC_i, it is easy to see that EC_i is monotone in the sense that any equality constraint in EC_{i-1} is also an equality constraint in EC_i. Due to the finiteness of the predicate arities, and the monotonocity of EC_i, there exists an integer k, called the *depth* of linear recursive rule, such that $EC_k = EC_{k+1}$. We refer by IEC_i to those equality constraints in EC_i that are not in EC_{i-1} and call IEC_i the *increment in* EC_i. It is possible to represent IEC_i in terms of the variables in the consequent as equality predicates. This we do in the following example as well as in subsequent discussions.

Example 3.1



(a) P(x, y, z, x, w, s, j) :- P(v, x, x, w, z, y, pj), B(j, pj), Q(v, s)



(b) P(x, y, z, u, v, i_1, w, m, i_2, i_3) :- P(y, x, y, y, z, u, u, v, w, m), Q(m, i_1, i_2, i_3)

Figure 3.2 Propagation Graphs

In Figure 3.2(a), the 0^{th} equivalence grouping is {{1}, {4}}, the 1^{st} grouping is {{1}, {4}, {2, 3}}, the 2^{nd} grouping is {{1}, {4}, {2, 3}, {5, 6}}. Since the 3^{rd} grouping {{1}, {4}, {2, 3}, {5, 6}} is the same as the second grouping, the algorithm stops. Hence, EC_1 is (2 = 3), EC_2 is (2 = 3) $\wedge$ (5 = 6), IEC_1 is (y = z), and IEC_2 is (w = s). Note that the depth of this rule is 2. In Figure 3.2(b), the 0^{th} equivalence grouping is {{2}, {4}}, the 1^{st} grouping is {{2}, {1, 3, 4}, {6, 7}}, the 2^{nd} grouping is {{2, 5, 6, 7}, {1, 3, 4}, {9}}, the 3^{rd} grouping is {{2, 5, 6, 7}, {1, 3, 4, 8, 9}}, the 4^{th} grouping is {{2, 5, 6, 7; 10}, {1, 3, 4, 8, 9}}. Therefore, EC_1 is (1 = 3 = 4) $\wedge$ (6 = 7),

EC_2 is $(1 = 3 = 4) \wedge (2 = 5 = 6 = 7)$, EC_3 is $(1 = 3 = 4 = 8 = 9) \wedge (2 = 5 = 6 = 7)$, EC_4 is $(1 = 3 = 4 = 8 = 9) \wedge (2 = 5 = 6 = 7 = 10)$, IEC_1 is $(x = z = u) \wedge (i_1 = w)$, IEC_2 is $(y = v = i_1)$, IEC_3 is $(x = m = i_2)$, and IEC_4 is $(y = i_3)$. The depth of the rule is 4. □

3.3 Integrated Magic Set Method

Recall that during the magic set computation the trivial part of the Datalog program is not consulted It is our thesis that since the magic set computation does not depend on the base predicates that appear in the nonrecursive rule, in the worst case the entire magic set computation may become useless. Further, even in other situations, magic set may generate tuples that may not be relevant to the query in the sense that those tuples can never participate in the recursive application of the rule ;see Example 3.4.

We also remark that another serious drawback of the *MS*-method is its inability to take advantage of situations where a collection of queries are posed on the same derived predicate. In fact, *MS*-method fails to achieve any sort of "global optimization" when both the derived predicate and the query form are the same. For example, the following k queries

$$query^{(r)}(x, y, z, u, w, s) :- P(x, y, z, u, w, s, M_r). \qquad r = 1, 2, \cdots, k,$$

are treated independently and no "global optimization" is possible as each query processing depends on its own magic set. Moreover, the magic set computation is inherently sequential in that even if more than one processor is available the *MS*-method has to wait for the completion of the magic set computation. Thus, an improvement in the *MS*-method is warranted.

The *Integrated Magic Set (IMS)* method is a powerful strategy, which will alleviate most of the above mentioned problems associated with *MS*-method. The central idea behind the *IMS*-method is to use the equality constraints and subsets of magic sets to restrict the number of tuples generated during the recursive rule application. We start with a simple example illustrating some of the major drawbacks of the *MS*-method and the achievable efficiency of *IMS*-method.

Example 3.2

Consider the following Datalog program:

Program DD_0

P(x, y, z, u, w, s, j) :- I_0(x, y, z, u, w, s, j).
P(x, y, z, x, w, s, j) :- P(v, x, x, w, z, y, pj), Q(v, s), B(j, pj).
$Query_0$(x, y, z, u, w, s) :- P(x, y, z, u, w, s, C_0).

If the query is processed using the *MS*-method, the first step is to transform the program DD_0 into the modified Program DD_1:

Program DD_1

Rule 1: magic(C_0) .
Rule 2: magic(pj) :- magic(j), B(j, pj).
Rule 3: P(x, y, z, u, w, s, j) :- magic(j), I_0(x, y, z, u, w, s, j).
Rule 4: P(x, y, z, x, w, s, j) :- magic(j), P(v, x, x, w, z, y, pj),
Q(v, s), B(j, pj).
Rule 5: $Query_1$(x, y, z, u, w, s) :- P(x, y, z, u, w, s, C_0).

Next the magic set is computed which is then used to restrict the tuples generated during the bottom-up evaluation.

However *IMS*-method first computes the IEC_i's of the recursive rule. The IEC_i's of Rule 4 are IEC_1 (y = z) and IEC_2 (w = s). Since the depth of the recursive rule is two, we define three magic sets, $magic_0$, $magic_1$ and $magic_2$ and the Program DD_0 is transformed into Program DD_2 shown below:

Program DD_2

m_0 : $magic_0(C_0)$.

m_1 : $magic_1(pj)$:- $magic_0(j)$, B(j, pj).

m_2 : $magic_2(pj)$:- $magic_1(j)$, B(j, pj).

m_3 : $magic_2(pj)$:- $magic_2(j)$, B(j, pj).

r_1 : $I_1(x, y, z, u, w, s, j)$:- $I_0(x, y, z, u, w, s, j)$, y = z.

r_2 : $I_2(x, y, z, u, w, s, j)$:- $I_1(x, y, z, u, w, s, j)$, w = s.

i_0 : $P_0(x, y, z, u, w, s, j)$:- $magic_0(j)$, $I_0(x, y, z, u, w, s, j)$.

i_1 : $P_1(x, y, z, u, w, s, j)$:- $magic_1(j)$, $I_1(x, y, z, u, w, s, j)$.

i_2 : $P_2(x, y, z, u, w, s, j)$:- $magic_2(j)$, $I_2(x, y, z, u, w, s, j)$.

rr_2 : $P_2(x, y, z, x, w, s, j)$:- $magic_2(j)$, $P_2(v, x, x, w, z, y, pj)$, Q(v, s), B(j, pj), w = s

rr_1 : $P_1(x, y, z, x, w, s, j)$:- $magic_1(j)$, $P_2(v, x, x, w, z, y, pj)$, Q(v, s), B(j, pj).

rr_0 : $P_0(x, y, z, x, w, s, j)$:- $magic_0(j)$, $P_1(v, x, x, w, z, y, pj)$, Q(v, s), B(j, pj).

q : $Query_2(x, y, z, u, w, s)$:- $P_0(x, y, z, u, w, s, C_0)$.

We claim that programs DD_1 and DD_2 are query equivalent. Before giving a formal theorem, we verify that $Query_1$ = $Query_2$ on a database instance through an example. □

Example 3.3

Consider the Datalog programs DD_1 and DD_2. For the rest of this example, let x_j, y_j, s_j, w_j, u_j and C_j (j = 0, 1, ···, 120) be some constants. Since B is a binary relation, we describe the relevant tuples of B in graph terminology. Hereafter, the statements "a tuple (y, x) is in B" and "x is a parent of y (or y is a son of x)" will be used without distinction. The subgraph of B relevant to our discussion has $\{ C_j \mid j = 0, 1, \cdots, 120\}$ as the vertex set and $\{ <C_j, C_{3j+s}> \mid j = 0, 1, \cdots, 39$ and $s = 1, 2, 3\}$ as the edge set.

Tuples of I_0 which will not be filtered by the magic set are $(x_j, y_j, z_j, u_j, w_j, s_j, C_j)$, for j = 0, 1, ···, 120. These tuples are such that they satisfy IEC_1 if and only if $j \equiv 1 \bmod 3$; and satisfy IEC_2 if and only if $j \equiv 2, 4, 6$ or 8 (mod 9).

The relevant tuples of relation Q can be described as follows: Given any value x from its domain there exists a unique value $\bar{x}$ such that $Q(x, \bar{x})$ is true. Further if $x = x_j$ (j = 0, 1, ···, 120) then $\bar{x} = u_j$ if and only if j is even; and if $x = y_j$ then $\bar{x} = y_j$ if and only if j = 2 or 3 (mod 4).

Trace of Program DD_1

Clearly, the magic set consists of 121 elements $C_0, C_1, \cdots, C_{120}$. During the application of the Rule 3, the predicate P is initialized with 121 tuples $(x_j, y_j, z_j, u_j, w_j, s_j, C_j)$, j = 0, 1, ···, 120. By applying Rule 4 once, the following 40 tuples

$(y_j, s_j, w_j, y_j, u_j, \bar{x}_j, C_p)$, (j = 1, 4, 7, ···, 118 and $p = \lfloor \frac{j-1}{3} \rfloor$)

are generated. Now, applying the recursive rule once more, the following 13 tuples

$(w_j, \bar{x}_j, u_j, w_j, y_j, \bar{y}_j, C_{pp})$, $(j = 4, 13, \cdots, 112 \text{ and } pp = \lfloor \frac{p-1}{3} \rfloor)$

are generated and $\bar{y}_j = y_j$ if and only if $j \equiv 2$ or 3 mod 4. During the third application of the recursive rule the 6 new tuples

$(u_j, \bar{y}_j, y_j, u_j, w_j, \bar{w}_j, C_{ppp})$, $(j = 22, 40, \cdots, 112 \text{ and } ppp = \lfloor \frac{pp-1}{3} \rfloor)$

are generated. Thus in the fourth application of the recursive rule only 2 tuples

$(y_j, \bar{w}_j, w_j, y_j, u_j, \bar{u}_j, C_{pppp})$, $(j = 58 \text{ and } 94 \text{ and } pppp = \lfloor \frac{ppp-1}{3} \rfloor)$

are generated. No more new tuples can be generated and the $Query_1$ contains the following tuples: $(x_0, y_0, z_0, u_0, w_0, s_0)$, $(y_1, s_1, w_1, y_1, u_1, \bar{x}_1)$, $(w_4, u_4, u_4, w_4, y_4, \bar{y}_4)$, $(u_{22}, y_{22}, y_{22}, u_{22}, w_{22}, \bar{w}_{22})$, $(y_{58}, \bar{w}_{58}, w_{58}, y_{58}, u_{58}, \bar{u}_{58})$ and $(y_{94}, \bar{w}_{94}, w_{94}, y_{94}, u_{94}, \bar{u}_{94})$. Note that a total of 61 tuples are generated during the recursive rule application.

Trace of Program DD_2

Note that I_1 contains all tuples of I_0 which satisfy IEC_1 (= EC_1). So I_1 consists of tuples of the form $(x_j, y_j, y_j, u_j, w_j, s_j, C_j)$, $j = 1, 4, 7, \cdots, 118$ and other tuples of I_0 which satisfy IEC_1. Similarly, I_2 contains all tuples of the form $(x_j, y_j, y_j, u_j, w_j, w_j, C_j)$, $j = 4, 13, 22, \cdots, 112$ and other tuples of I_1 which satisfies IEC_2. Note that $magic_0$ is the set $\{C_0\}$, $magic_1$ is the set $\{C_1, C_2, C_3\}$ and $magic_2$ is the set $\{C_j \mid j = 4, 5, \cdots, 120\}$. Therefore, rules i_0 and i_1 initialize P_0 and P_1 with tuples $(x_0, y_0, z_0, u_0, w_0, s_0, C_0)$ and $(x_4, y_4, y_4, u_4, w_4, s_4, C_4)$ respectively; and rule i_2 assigns P_2 the thirteen tuples, $(x_j, y_j, y_j, u_j, w_j, s_j, C_j)$, $j = 4, 13, 22, \cdots, 112$.

An Application of rule rr_2 produces the following 6 tuples:

$(y_{22}, w_{22}, w_{22}, y_{22}, u_{22}, u_{22}, C_7)$,

$(y_{40}, w_{40}, w_{40}, y_{40}, u_{40}, u_{40}, C_{13})$,

$(y_{58}, w_{58}, w_{58}, y_{58}, u_{58}, u_{58}, C_{19})$,

$(y_{76}, w_{76}, w_{76}, y_{76}, u_{76}, u_{76}, C_{25})$,

$(y_{94}, w_{94}, w_{94}, y_{94}, u_{94}, u_{94}, C_{31})$,

$(y_{112}, w_{112}, w_{112}, y_{112}, u_{112}, u_{112}, C_{37})$.

Applying rule rr_2 on these new tuples, we have 2 new tuples:

$(w_{58}, u_{58}, u_{58}, w_{58}, y_{58}, y_{58}, C_6)$,

$(w_{94}, u_{94}, u_{94}, w_{94}, y_{94}, y_{94}, C_{10})$.

Rule rr_2 cannot produce any more tuples and so rule rr_1 is invoked next and the relation P_1 gets the following 4 tuples.

$(y_4, w_4, w_4, y_4, u_4, u_4, C_1)$,

$(w_{22}, u_{22}, u_{22}, w_{22}, y_{22}, \bar{y}_{22}, C_2)$,

$(u_{58}, y_{58}, y_{58}, u_{58}, w_{58}, \bar{w}_{58}, C_1)$,

$(u_{94}, y_{94}, y_{94}, u_{94}, w_{94}, \bar{w}_{94}, C_3)$.

Now rule rr_0 is applied; and this generates the 5 tuples for the relation P_0. Since P_0 already has the tuple $(x_0, y_0, z_0, u_0, w_0, s_0, C_0)$ and since $\bar{y}_{22} = y_{22}$, the relation P_0 has the following 6 tuples:

$(x_0, y_0, z_0, u_0, w_0, s_0, C_0)$,

$(y_1, s_1, w_1, y_1, u_1, \bar{x}_1, C_0)$,

$(w_4, u_4, u_4, w_4, y_4, \bar{y}_4, C_0)$,
$(u_{22}, y_{22}, y_{22}, u_{22}, w_{22}, \bar{w}_{22}, C_0)$,

$(y_{58}, \bar{w}_{58}, w_{58}, y_{58}, u_{58}, \bar{u}_{58}, C_0)$,

$(y_{94}, \bar{w}_{94}, w_{94}, y_{94}, u_{94}, \bar{u}_{94}, C_0)$.

Thus relations $Query_1$ and $Query_2$ are equal for this database instance. □

We now proceed to formally state the *IMS*-method and provide its proof of correctness. The rest of our discussion on the general transformation rules will focus on the two logic programs D_0 and D_1 defined below:

Program D_0.

P :- I_0.
P :- P, R.
$Query_1$:- P, $Col_q = C_0$.

where R represents the collection of all base predicates appearing in the recursive rule and I_0 stands for the collection of base predicates appearing in the exit rule. The following Datalog Program D_1 is obtained from D_0 by an application of magic set transformation rules.

Program D_1.

Rule 1: magic(C_0).
Rule 2: magic(x) :- magic(y), B.
Rule 3: P :- magic(y), I_0.

Rule 4: P :- magic(y), P, R.

Rule 5: $Query_1$:- P, $Col_q = C_0$.

Here B is the collection of those base predicates in R which participate in the sideways information passing of magic set computation. Let $\pi_{-q}S$ denote the projection of a relation S on all its coordinates except q.

Theorem 3.1

Let $k > 0$ be the depth of the recursive rule in the Datalog program D_0. Then D_0 is query equivalent to the following Datalog program D_2.

Program D_2.

m_0 : $magic_0(C_0)$.

m_j : $magic_j(x)$:- $magic_{j-1}(y)$, B. ($j = 1, 2, \cdots, k$)

m_{k+1} : $magic_k(x)$:- $magic_k(y)$, B.

r_j : I_j :- IEC_j, I_{j-1} ($j = 1, 2, \cdots, k$)

i_j : P_j :- $magic_j(y)$, I_j ($j = 0, 1, \cdots, k$)

rr_k : P_k :- $magic_k(y)$, IEC_k, P_k, R

rr_j : P_j :- $magic_j(y)$, P_{j+1}, R. ($j = 0, 1, \cdots, k-1$)

q : $Query_2 = \pi_{-q}P_0$.

Proof

Since D_1 is obtained from D_0 by magic set transformation, we have $Query_0 = Query_1$ and hence it is sufficient to show that $Query_1 = Query_2$. It is easy to see that any rule in Program D_2 is a restricted version of some rule in Program D_1. Hence a tuple generated by any rule in Program D_2 is also generated by some rule in Program D_1. Hence $Query_1 \supseteq Query_2$. To prove the converse, we

introduce the following Datalog Program D_3 obtained by removing IEC_i predicates from Program D_2.

Program D_3.

m_0 : $magic_0(C_0)$.

m_j : $magic_j(x)$:- $magic_{j-1}(y)$, B. ($j = 1, 2, \cdots, k$)

m_{k+1} : $magic_k(x)$:- $magic_k(y)$, B.

r_j : I_j :- I_{j-1} ($j = 1, 2, \cdots, k$)

i_j : P_j :- $magic_j(y)$, I_j ($j = 0, 1, \cdots, k$)

rr_k : P_k :- $magic_k(y)$, P_k, R

rr_j : P_j :- $magic_j(y)$, P_{j+1}, R. ($j = 0, 1, \cdots, k-1$)

q : $Query_3 = \pi_{-q}P_0$.

It may be noted that $magic_j(x)$ ($j = 0, 1, \cdots, k-1$) computed in Program D_3 essentially consists of all ancestors of C_0 at level j under B and $magic_k(x)$ consists of all ancestors of C_0 at level k and above. Hence $magic(x) = \bigcup_{j=0}^{k} magic_j(x)$. Let t_0 be a tuple of P satisfying the condition $Col_q = C_0$. We show that t_0 is in P_0 of Program D_3. First, note that t_0 was obtained through exactly p applications of Rule 4 (for some $p \geq 0$) and exactly one application of each of the Rules 3 and 5. Hence there exist tuples t_j ($j = 1, 2, \cdots, p$) in P, r_j ($j = 1, 2, \cdots, p$) in R, and C_j ($j = 0, 1, \cdots, p-1$) in magic(x) such that C_{j-1}, t_j and r_j participate in the derivation of the tuple t_{j-1} in P. Further, t_p is in I_0. Basically, we distinguish two cases $p < k$ and $p \geq k$.

case 1. ($p < k$) Note that in this case, t_p belongs to P_p and C_j is in $magic_j$ ($j = 0, 1, \cdots, p-1$). Consequently, t_j is derived using the tuples C_j (in $magic_j$), t_{j+1} (in P_{j+1}), and r_{j+1} (in R) by an application of rule rr_j ($j = p-1, p-2, \cdots, 0$). Thus t_0 belongs to P_0.

case2. ($p \geq k$). Clearly, t_p is in P_k. Further C_j is in $magic_j$ for $0 \leq j < k$ and is in $magic_k$ for $j \geq k$. Therefore, the tuple t_j in P_k is generated using the tuples t_{j+1}, C_j, and r_{j+1} and rule rr_k (for $j = p-1, p-2, \cdots, k$). Further, t_j is derived using the tuples C_j, t_{j+1}, and r_{j+1} by an application of rule rr_j ($j = k-1, p-2, \cdots, 0$). Thus t_0 belongs to P_0. So $Query_1 \subseteq Query_3$. It is easy to see that all the tuples of $Query_3$ will also appear in $Query_2$, because program D_2 is nothing but Program D_3 together with IEC's. Note that a tuple can be used to produce a new tuple if and only if it satisfies the EC_1 and this new tuple can be used to produce yet another tuple if and only if the "first" tuple satisfies EC_2, etc. Therefore, IEC_i's will only filter those tuples which will fail to produce a tuple of $Query_2$. Therefore, any tuple of $Query_3$ produced by Program D_3 is also a tuple of $Query_2$. Hence, $Query_1$ is contained in $Query_2$. □

Next, we proceed to extend the *IMS*-method to all linear Datalog programs. Note that without any loss of generality, we may assume that the program has a unique query rule, say

q : Query :- P, Col_q = constant.

Here P is a derived predicate of the program. We distinguish between two disjoint classes of linear Datalog programs: (a) the program is devoid of any predicate mutually recursive to P and (b) the program in which there exists at least one predicate which is mutually recursive to P. First, assume that the program belongs to class (a). Since, all the rules are linear and there exists no predicate mutually recursive to P, we assume that there exists a unique exit rule of the form

e : P :- I.

where I is a base predicate. For, if there exists k ($k \geq 1$) exit rules, say P :- I_j, j = 1, ···, k, then it is possible to replace these exit rules by a single exit rule e where I is a derived predicate defined through the k rules: I :- I_1, ···, I :- I_k. Further, since there is no mutually recursive rule, all the recursive rules have P as their head predicate. Thus a typical program of class (a) can be written as follows.

Program M

e : P :- I.
r_j : P :- P, R_j. $(1 \leq j \leq s)$
q : Query :- P, Col_q = constant.

The *IMS*-method can be independently applied to logic programs M_j defined below:

Program M_j

e : P :- I.
r_j : P :- P, R_j.
q_j : $Query_j$:- P, Col_q = constant.

Denoting the resulting query equivalent logic programs by N_1, ···, N_s it is easy to see that the logic program N obtained by "unioning" N_1, ···, N_s is query equivalent to the logic program M. Thus *IMS*-method can transform any logic program belonging to class (a) into a query equivalent logic program.

Suppose the logic program belongs to class (b). Since the query depends ultimately on one derived predicate, say P, it is possible to

transform it into a class (a) program. The transformation is achieved through the elimination of other derived predicates using the rules defining them. Since the method is quite similar to the elimination of "unknowns" in a system of equations, instead of giving a formal description, we illustrate the transformation through an example.

Example 3.4

Consider the following logic program G.

Program G

e_1 : $P_1(x, y, z)$:- $I_1(x, y, z)$.
e_2 : $P_2(x, y, z)$:- $I_2(x, y, z)$.
e_3 : $P_3(x, y, z)$:- $I_3(x, y, z)$.
r_1 : $P_1(x, y, z)$:- $P_2(x_1, y_1, z_1)$, $R_1(x, y, z, x_1, y_1, z_1)$.
r_2 : $P_2(x, y, z)$:- $P_3(x_1, y_1, z_1)$, $R_2(x, y, z, x_1, y_1, z_1)$.
r_3 : $P_3(x, y, z)$:- $P_1(x_1, y_1, z_1)$, $R_3(x, y, z, x_1, y_1, z_1)$.
q : Query(y, z) :- $P_1(x, y, z)$, x = constant.

It can be verified that the program G is query equivalent to the following program GG obtained by first eliminating P_3 and then P_2.

Program GG

ee_1 : $P_1(x, y, z)$:- $I_1(x, y, z)$.
ee_2 : $P_1(x, y, z)$:- $I_2(x_1, y_1, z_1)$, $R_1(x, y, z, x_1, y_1, z_1)$.
ee_3 : $P_1(x, y, z)$:- $I_3(x_2, y_2, z_2)$, $R_2(x_1, y_1, z_1, x_2, y_2, z_2)$,
$R_1(x, y, z, x_1, y_1, z_1)$.
rr : $P_1(x, y, z)$:- $P_1(x_3, y_3, z_3)$, $R_3(x_2, y_2, z_2, x_3, y_3, z_3)$,
$R_2(x_1, y_1, z_1, x_2, y_2, z_2)$, $R_1(x, y, z, x_1, y_1, z_1)$.

qq : Query(y, z) :- P_1(y, z), x = constant.

Clearly, GG belongs to class (a). Thus *IMS*-method is applicable to any general linear Datalog program. □

3.4 Comparison with Magic Set Method

The purpose of this section is to compare the *IMS*-method with the *MS*-method. Since the in-degree of node 7 in the propagation graph of the recursive rule in Example 3.2 is zero, it is easy to see that *AD*-method cannot be applied to such problems. Therefore, the efficiency achieved by *IMS*-method cannot be achieved by performing *AD* and *MS* methods in any order. Note that *IMS*-method uses the same set of equality constraints as *AD*-method and possibly more information from magic sets to filter the tuples generated. This implies that *IMS*-method can never generate more tuples than AD-method. Therefore, we restrict our comparative study to *MS*-method versus *IMS*-method.

Recall that any rule of Program D_2 is a restricted version of some rule of Program D_1. Therefore, any tuple generated by program D_2 in the course of query processing is also generated by Program D_1. So, it is intuitively clear that Program D_2 cannot perform any worse than Program D_1. In Example 3.3, it is shown that the number of tuples generated by programs DD_1 and DD_2 are 61 and 17 respectively, and we claim that the above figures are typical.

It is easy to observe that in order to compare programs D_1 and D_2 it is sufficient to "count" the number of tuples derived by Rule 4 of Program D_1 and rules rr_i (i = 1, 2, ···, k) of D_2. For, the cost

due to other rules is either common to both methods or the rule need to be applied only once (independent of the query). We model the database instance as follows:

For the sake of simplicity, assume that B is a binary relation. Thus in Rule 2 (of Program D_1), x is an *ancestor* of y; and y is a *son* of x. Let n be the least positive integer such that the bottom-up evaluation of the recursive rule in Program D_0 will not produce any more new tuples at the n+1-st iteration. An ancestor C′ of C_0 is at *level* j if there exists C_i (i = 1, 2, ···, j) such that (C_i, C_{i+1}) is in B (i = 0, 1, ···, j-1) and C′ = C_j. The parameters of the database features are given below:

ρ_j : Number of ancestors of C_0 at level j.

α_j : The probability that a tuple of I_0 satisfies EC_j, j = 1, 2, ···, k.

η : The average number of tuples generated by the natural join of P and R (as indicated in Rule 4; but with no equality constraints on the tuples of P) for one value of the magic set.

τ : The average number of sons of an ancestor of C_0.

μ : The average number of sons in the magic set of an ancestor of C_0.

γ : The conditional probability that a tuple t′ generated through the firing of a tuple t satisfies EC_k given that t satisfies EC_k.

We introduce the following parameters for notational convenience.

α_j : $= \alpha_k \gamma^{j-k}$, j = k+1, k+2, ···, n.

δ : $= \frac{\gamma\eta\mu}{\tau}$

β : $= \frac{\eta\mu}{\tau}$

S_j : The set of all ancestors of C_0 at level j.

N_1 : The number of tuples generated by Rule 4 of Program D_1.

N_{21} : The number of tuples generated by rules rr_j (j = 1, 2, ···, k−1) of Program D_2.

N_{22} : The number of tuples generated by rule rr_k of Program D_2.

N_2 : $N_{21} + N_{22}$

We first obtain an expression for N_{22}, the number of tuples generated by rule rr_k of Program D_2. To begin with consider ρ_i ancestors of C_0 at level i (i > k). Hence $\rho_i\eta$ is the number of tuples generated by taking the natural join of I_0 and R. Since α_k is the probability that a tuple of I_0 is in P_k, the number of tuples obtained by the natural join of P_k and R is given by $\alpha_k\rho_i\eta$. Since each ancestor of C_0 has τ sons and out of which only μ of them are in the magic set, $\frac{\mu}{\tau}\alpha_k\rho_i\eta$ is the number of tuples produced by the natural join of $magic_k$ with the natural join of P_k and R. Now, from the definition of γ, the number of tuples generated by the first application of rule rr_k is $\gamma\frac{\mu}{\tau}\alpha_k\rho_i\eta = \rho_i\alpha_k\delta$. Applying rr_k on these newly generated tuples, $\rho_i\alpha_k\delta^2$ new tuples are generated; and so on. Thus the total number of tuples generated by rule rr_k, due to level i ($k < i \leq n$) ancestors of C_0 is $\rho_i\alpha_k\delta^{i-k} + \rho_i\alpha_k\delta^{i-k-1} + \cdots + \rho_i\alpha_k\delta = \delta\rho_i\alpha_k\left[\frac{\delta^{i-k}-1}{\delta-1}\right]$. Since i varies from k + 1 to n, we have

$$N_{22} = \sum_{i=k+1}^{n} \delta\, \rho_i \alpha_k \left[\frac{\delta^{i-k}-1}{\delta-1}\right] = \alpha_k \delta \sum_{i=k+1}^{n} \rho_i \left[\frac{\delta^{i-k}-1}{\delta-1}\right].$$

The number of tuples generated by rr_k and used by rule rr_{k-1} for firing is $N_{221} = \sum_{i=k+1}^{n} \rho_i \alpha_k \delta^{i-k} = \sum_{i=k+1}^{n} \rho_i \alpha_i \beta^{i-k}$. Since the number of ancestors of C_0 at level k is ρ_k, the number of tuples generated by rule rr_{k-1} is $\left(N_{221} + \rho_k \alpha_k\right)\beta$. Similarly, the number of tuples generated by rule rr_{k-2} is $\left(N_{221} + \rho_k \alpha_k\right)\beta^2 + \rho_{k-1}\alpha_{k-1}\beta$; and so on. Thus an expression for N_{21} is

$$N_{21} = N_{221}\beta \left[\frac{\beta^k-1}{\beta-1}\right] + \sum_{i=1}^{k} \beta \rho_i \alpha_i \left[\frac{\beta^i-1}{\beta-1}\right].$$

Hence $N_2 = N_{21} + N_{22}$

$$= \beta\left(\left[\frac{\beta^k-1}{\beta-1}\right]\left[\sum_{i=k+1}^{n} \rho_i \alpha_i \beta^{i-k}\right] + \sum_{i=1}^{k} \rho_i \alpha_i \left[\frac{\beta^i-1}{\beta-1}\right]\right)$$

$$+ \delta \alpha_k \sum_{i=k+1}^{n} \rho_i \left[\frac{\delta^{i-k}-1}{\delta-1}\right].$$

An expression for N_1 can be derived in a similar fashion. Recall that α_i is the probability that a tuple t satisfies the i-th equality constraint. So the number of tuples generated by one application of the recursive rule due to ρ_n ancestors of C_0 at level n is $\rho_n \alpha_1 \eta \frac{\mu}{\tau} = \rho_n \alpha_1 \beta$. Since α_2 is the probability that a tuple of I_0 satisfies EC_2, the number of tuples satisfying EC_1 after one application of the recursive rule among $\rho_n \alpha_1 \beta$ tuples are given by $\rho_n \alpha_2 \beta$. Therefore, the number of tuples generated by firing of newly generated $\rho_n \alpha_1 \beta$ tuples using rule 4 are $\rho_n \alpha_2 \beta^2$; and so on. Thus the number of newly

generated tuples after k application of rule 4 is $\rho_n\alpha_k\beta^k$. Further, from definition of γ, it follows that the number of tuples generated by the repeated application of rule 4 are given by $\rho_n\alpha_k\gamma\beta^{k+1}$, $\rho_n\alpha_k\gamma^2\beta^{k+2}$, ..., $\rho_n\alpha_k\gamma^{n-k}\beta^n$. Thus ρ_n ancestors of C_0 at level n derives a total of $\rho_n \sum_{i=1}^{n} \alpha_i\beta^i$ tuples. Similar expressions are valid for all levels; and thus $N_1 = \sum_{p=1}^{n} \rho_p\left[\sum_{i=1}^{p} \alpha_i\beta^i\right]$.

Example 3.5

For empirical comparisons we consider the database instance in Example 3.3. Notice that while the *MS*-method derived 61 tuples to answer the query, the *IMS*-method generated only 17 tuples.

The parameter values for the database instance are: $n = 4$, $k = 2$, $\alpha_1 = \frac{1}{3}$, $\alpha_2 = \frac{1}{9}$, $\eta = 3$, $\tau = 3$, $\mu = 1$, $\gamma = \frac{1}{2}$. Therefore $\delta = \frac{1}{2}$ and $\beta = 1$. Further, $\alpha_3 = \frac{1}{18}$ and $\alpha_4 = \frac{1}{36}$. Thus the computed values for N_1 and N_2 are 61.25 and 18.75 respectively. Thus the approximate expressions we derived in Section 3.4 are quite close to the exact values. □

Theorem 3.2

If $\alpha_i < 1$, $1 \leq i \leq n$, the integrated magic set method is more efficient than magic set method.

Proof.

We claim that N_2 is less than N_1 and show this by comparing the coefficient of ρ_j, $j = 1, 2, \cdots, n$. Suppose $j \leq k$. Then the coefficient

of ρ_j in $N_1 = \sum_{i=1}^{j} \alpha_i\beta^i > \sum_{i=1}^{j} \alpha_j\beta^i > \alpha_j \sum_{i=1}^{j} \beta^i$ = coefficient of ρ_j in N_2. On the other hand, if $j > k$, then the coefficient of ρ_j in $N_1 = \sum_{i=1}^{j} \alpha_i\beta^i = \sum_{i=j-k+1}^{j} \alpha_i\beta^i + \sum_{i=1}^{j-k} \alpha_i\beta^i > \sum_{i=j-k+1}^{j} \alpha_j\beta^i + \alpha_k \sum_{i=1}^{j-k} \delta^i =$ coefficient of ρ_j in N_2. Thus $N_1 < N_2$. □

In addition to the above mentioned advantage, *IMS*-method is highly suitable for situations where queries with the same query forms are encountered. For example, the following k queries

$query_r(x, y, z, u, w, s)$:- $P(x, y, z, u, w, s, C_r)$. $(r = 1, 2, \cdots, k)$

need the same I_1 and I_2; and this computation is independent of the query.

Even if the query forms are not the same, *IMS*-method can perform some global optimization which *MS*-method cannot perform. Note that $I_1, I_2, \cdots, I_k$ need to be computed only once in Program D_2. Further, if I_j is empty for some j ($1 \leq j \leq k$), one can ignore all the rules in D_2 with subscript value greater than j. This optimization is independent of the query; in fact it depends only on the recursive rule and the trivial part of the program. On the other hand *MS*-method cannot perform any such preprocessing.

3.5 Conclusion

Among the rule rewriting methods both *MS* and *MC* methods are the most significant. *AD*-method, which is a generalization of Aho and Ullman's algorithm of commuting selections with LFP operator, works very well for a limited class of recursive rules. The *IMS*-method described in this chapter inherits the best of these two

methods and improves the processing efficiency of Datalog programs by a significant factor.

An important advantage of the *IMS*-method is its ability to do some global optimization independent of the given query. Further if the architecture permits, one can perform this global optimization in parallel with magic set computation.

This method can be viewed either as an improvement on the *MS*-method or as an integrated method involving *MS* and *AD* methods. To be convinced of this, notice that Datalog programs of type D_0 can be classified on the basis of the propagation graphs: (a) node q has at least one closed backtrace; that is, there exists a path from some node q_1 to q with at least one loop. (b) node q does not have a closed backtrace. It is easy to see that for programs in group (a) the *AD*-method is best suited. However, in such situations the *AD*-method is the same as the *IMS*-method, with no sideways information passing. The *AD*-method cannot be applied to programs in group (b). Programs in group (b) can be divided into two groups: (b_1) There exists an output variable v which appears at least twice in the antecedent of the rule. As we noted earlier, the *MS*-method processes a lot of data that may not be used in the final query evaluation. So the *IMS*-method will perform better for all programs in this group. (b_2) The rules do not satisfy the criteria in b_1. Hence both *IMS* and *MS*-methods behave identically. Thus it is sufficient to implement only the *IMS*-method in a query processing environment of Datalog programs. A by-product is a classification of Datalog programs based solely on the nature of the rules and query form as applied to these processing strategies.

Chapter 4

PARALLEL EVALUATION OF DATALOG PROGRAMS

4.1 Introduction

Two parallel algorithms for efficient query evaluation of Datalog programs are presented in this chapter. These algorithms can be implemented in a large class of parallel architectures. The performance comparisons between the parallel algorithms and a serial algorithm are presented using a simple analytical model. The analysis establishes the superiority of the parallel algorithms over the serial algorithm.

In Chapter 3, we concentrated on optimizing a Datalog Program with some variables bound. In this chapter we focus our attention to the class of algorithms proposed to optimize a Datalog program with no variables bound. The best known algorithm of this group is the semi-naive evaluation method proposed in [Banc85]. An efficient implementation of algorithms for processing general logic queries still remains an open problem.

One of the means for achieving the speed-up is to exploit the possible parallelism. It is quite surprising that, until recently, no attempt has been made to transform a general linear recursive query so that the result of the bottom-up evaluation of the transformed parallel program is the same as that of the original program. However recently [Vald88] two parallel algorithms are discussed to evaluate the transitive closure of a binary relation. Empirical results are provided to show the attained speed up. On the other hand the transformation scheme proposed in [Wolf88] can be applied only to a

very limited class of rules: one-sided recursive chain rules [Naug87]. Moreover no analysis is given.

In this chapter we propose two parallel algorithms for evaluating a large class of linear recursive programs, which we call *vertically partitionable programs*. Consequently, the load of evaluating the predicates can be divided among a number of processors with minimal need for inter-process communication. Thus the method is suitable for a large class of architectures. Further the class of programs that can be vertically partitioned include among others, the same generation program [Banc86a, Banc86b], which is a canonical example for linear Datalog programs and, hence, form an interesting subclass of the class of linear Datalog programs.

In Section 4.2, we discuss the class of Datalog programs that can be partitioned vertically and introduce two different transformation schemes for the efficient processing of such programs. A performance analysis based upon a simple analytical model is carried out in Section 4.3. This section clearly indicates the superiority of the parallel algorithms over the semi-naive algorithm. In Section 4.4, we consider the same-generation program as a canonical example and expressions obtained in Section 4.3 are used to compare the efficiency of parallel programs. A simple criteria to choose between the two parallel algorithms is also provided. The issue of pushing selections in parallel algorithms is discussed in Section 4.5. In Section 4.6, we consider a very special case of Datalog programs where all the rules, including the exit rules, of the program can be partitioned vertically. We conclude this chapter in Section 4.7.

4.2 Parallel Algorithms for Evaluating Linear Datalog Programs

In this section, we first introduce the concept of *vertically partitionable* logic programs. As a motivating example, consider the Same—Generation Program.

Program SG

Sg(x, y) :- I(x, y).

Sg(x, y) :- A(x, x_1), Sg(x_1, y_1), B(y, y_1).

query(x, y) :- Sg(x, y).

It may be noted that query(x, y) = $\cup_j A^j(x, x_1)I(x_1, y_1)B^j(y, y_1)$.

Now if $query_1(x, y)$ and $query_2(x, y)$ are defined as below, then we claim that for all values of I, A and B the relations query(x, y), $query_1(x, y)$, $query_2(x, y)$ are equal under relational algebra.

Program SG_1

$S_1(0, x, x)$:- I(x, y).

$S_2(0, y, y)$:- I(x, y).

$S_1(j, x, x_1)$:- A(x, x_1), $S_1(j-1, x_1, z)$.

$S_2(j, y, y_1)$:- B(y, y_1), $S_2(j-1, y_1, z)$.

$S_0(0, x, y)$:- I(x, y).

$S_0(j, x, y)$:- $S_1(j, x, x_1)$, $S_0(j-1, x_1, y_1)$, $S_2(j, y, y_1)$.

$query_1(x, y)$:- $S_0(j, x, y)$.

Program SG_2

$SS_1(0, x, x)$:- I(x, y).

$SS_2(0, y, y)$:- I(x, y).

$SS_1(j, x, x_1)$:- A(x, z), $SS_1(j-1, z, x_1)$.

$SS_2(j, y, y_1)$:- B(y, z), $SS_2(j-1, z, y_1)$.

$SS_0(j, x, y)$:- $SS_1(j, x, x_1)$, $I(x_1, y_1)$, $SS_2(j, y, y_1)$.
$query_2(x, y)$:- $SS_0(j, x, y)$.

A formal proof for the most general situation is given later in this section. It may be observed that a simple naive evaluator might enter into an infinite loop. However, if we assure termination, S_1 and S_2 (or SS_1 and SS_2) can be computed in parallel with no data communication between them. Informally, we say S is vertically partitionable. This concept is formalized next.

4.2.1 Vertically Partitionable Datalog Programs

The variables appearing in the head of a rule are called output variables and those appearing in the antecedent of a rule are called input variables. All other variables appearing in a rule are called local variables. A variable can be both an input as well as an output variable; but an input (output) variable cannot be a local variable. Note that a non-recursive rule is devoid of any input variables. Two variables x and y of a rule are *connected* if there are variables $x = u_1, u_2, ..., u_{k-1}, u_k = y$ in the rule such that every adjacent pair u_i, u_{i+1} , $i = 1, ..., k-1$ appear in a base predicate. It is easy to see that connectedness is an equivalence relation on the set of variables in a rule. This definition can be extended in a natural way to base predicates. Two base predicates A and B of a rule are said to be connected if there exist variables x in A and y in B such that x and y are connected. Once again it is clear that connectedness defined on the set of base predicates of a rule is an equivalence relation. Notice that under this relation there is a one-one correspondence between equivalence classes of variables and the equivalence classes of base predicates. We shall refer to each equivalence class as a *component*. There are four types of

components: *input component* containing no output variable and at least one input variable; *output component* that contains no input variable and at least one output variable; *mixed component* that contains both input and output variables and *local component* containing only local variables. A recursive rule that contains a local component can be rewritten as

$$S(x) :- B(x, y), S(y), C(z). \quad (4.1)$$

where x is the vector of output variables, y is the vector of input variables, z is the vector of local variables, B represents all base predicates appearing in input, output and mixed components and C represents all base predicates appearing in local components. It may be noted that if C is empty, then (4.1) can be removed and if C is not empty, then (4.1) can be replaced by

$$S(x) :- B(x, y), S(y). \quad (4.2)$$

in any logic program. Hence for the rest of our discussion, we assume rules do not have local components.

An input variable y in the antecedent of a rule is said to *correspond* to the output variable x in the head of the same rule if x and y occur in the same coordinate position. A set $B = \{A_1, A_2, ..., A_k\}$ of base predicates in a rule is said to be *closed* (with respect to input-output variables) if for x in A_i, the component (possibly empty) containing the variable y corresponding to x is contained in B.

A closed set that contains all base predicates of mixed components and is minimal under set inclusion is called a *minimal closed set.* Notice that a rule has one and only one minimal closed set. An input (output) predicate is *independent* if it is not a member

of the minimal set. The set of base predicates of a recursive rule without local components can be partitioned into three sets; a minimal closed set B = $\{A_1, A_2, ..., A_k\}$, I = $\{C_1, C_2, ..., C_r\}$ of independent input predicates and O = $\{D_1, D_2, ..., D_s\}$ of independent output predicates. Thus we can rewrite a recursive rule without local components as

$$S(x, y) :- A(x, z)\ C(w)\ D(y)\ S(z, w) \qquad (4.3)$$

where

$A :- A_1\ A_2\ ...\ A_k$,
$C :- C_1, C_2\ ...\ C_r$,
$D :- D_1, D_2\ ...\ D_s$,
x : vector of output variables appearing in A_1 , ..., A_k,
y : vector of output variables appearing in D_1 , ..., D_s,
z : vector of input variables corresponding to x,
and w : vector of input variables corresponding to y.

Lemma 4.1

The Datalog program D is equivalent to Program D′ (in the sense that both D and D′ generate exactly the same tuples for the derived predicate S) if the base predicates B and C have at least one common tuple.

Program D: $S(x, y) :- P(x, y)$.
$S(x, y) :- A(x, x_1), B(y_1), C(y), S(x_1, y_1)$.

Program D′: $S'(x) :- A(x, x_1), B(y_1), P(x_1, y_1)$.
$S'(x) :- A(x, x_1), S'(x_1)$.
$S(x, y) :- P(x, y)$.

$S(x, y)$:- $S'(x)$, $C(y)$.

If B and C contain no common tuples then D is equivalent to the (non-recursive) program D".

Program D": $S(x, y)$:- $P(x, y)$.

$S(x, y)$:- $A(x, x_1)$, $B(y_1)$, $C(y)$, $P(x_1, y_1)$. □

Due to Lemma 4.1, for the rest of our discussion, we assume that the recursive rule is free of independent input/output components.

Next, we define the relation *linked*, a generalization of connected relation, on the set of variables occurring in a rule. Two variables x and y of a rule are said to be linked if either x and y are connected or x is linked to u and y is linked to v where u and v are corresponding variables in the rule. We can view "linkedness" as a relation on the components generated by the "connectedness" relation. Since in a non-recursive rule there is no input variable, the relations connectedness and linkedness mean the same. It is easy to see that each linked component is a union of connected components and further that the set of predicates appearing in a linked component is closed.

Given a rule r with n output variables x_1, x_2, ···, x_n, the *Partition Set (PS)* of r is $PS(r) = \{ s \mid i, j \in s$ if and only if x_i is linked to $x_j \}$. Note that PS(r) is a partition of the set {1, 2, ···, n}. A partition P of {1, 2, ···, n} is a cover of $PS(r_1)$ if every $s_1 \in PS(r_1)$ is contained in some $s \in P$. A partition is nontrivial if it induces a nontrivial partition on the set of base predicates in each recursive rule.

A Datalog program is *vertically partitionable* if there exists a nontrivial partition P covering the partition set of all recursive rules in the program. Note that if a program is vertically partitionable, there exists a *minimum cover* M in the sense that if M' is also a cover for all recursive rules in the program then M' is a cover of M.

We explore the possibility of rewriting a Datalog program through vertical partitioning. To begin with consider a Datalog program with one recursive rule and one nonrecursive rule and let $M = \{m_1, m_2, \cdots, m_n\}$ be its minimum cover. With out loss of generality, we may assume that if i and j are linked and if k lies between i and j then k is also linked to i. Hence any recursive rule can be rewritten as

$$S(x_1, \ldots, x_n) :- A_1(x_1, y_1), \ldots, A_n(x_n, y_n), S(y_1, \ldots, y_n).$$

where y_i denotes the vector of output variables corresponding to x_i and A_i is the collection of base predicates having variables in x_i and y_i. Since any rule can be brought into the above form the entire program can be written according to the minimum cover. We, hereafter, assume that the program has already been rewritten in this manner. The main result of this section is stated next.

Theorem 4.1

Let D be the following Datalog program rewritten according to the minimum cover.

Program D

$S(x_1, \ldots, x_n) :- I(x_1, \ldots, x_n).$

$S(x_1, \ldots, x_n) :- A_1(x_1, y_1), \ldots, A_n(x_n, y_n), S(y_1, \ldots, y_n).$

$query(x_1, \ldots, x_n) :- S(x_1, \ldots, x_n).$

Then the relation query(x_1, ..., x_n) in Program D is equal to the relation $query_1(x_1, ..., x_n)$ of Program D′ and $query_2(x_1, ..., x_n)$ of program D″, where D′ and D″ are as follows:

Program D′

$S_1(0, x_1, x_1)$:- $I(x_1, ..., x_n)$.

. . .

$S_1(0, x_n, x_n)$:- $I(x_1, ..., x_n)$.

$S_1(j, x_1, y_1)$:- $A_1(x_1, y_1)$, $S_1(j\text{-}1, y_1, z_1)$.

. . .

$S_n(j, x_n, y_n)$:- $A_n(x_n, y_n)$, $S_n(j\text{-}1, y_n, z_n)$.

$S_0(0, x_1, ..., x_n)$:- $I(x_1, ..., x_n)$

$S_0(j, x_1, ..., x_n)$:- $S_0(j\text{-}1, y_1, ..., y_n)$, $S_1(j, x_1, y_1)$, ..., $S_n(j, x_n, y_n)$.

$query_1(x_1, ..., x_n)$:- $S_0(j, x_1, ..., x_n)$.

Program D″

$SS_1(0, x_1, x_1)$:- $I(x_1, ..., x_n)$.

. . .

$SS_1(0, x_n, x_n)$:- $I(x_1, ..., x_n)$.

$SS_1(j, x_1, y_1)$:- $A_1(x_1, z)$, $SS_1(j\text{-}1, z, y_1)$.

. . .

$SS_n(j, x_n, y_n)$:- $A_n(x_n, z)$, $SS_n(j\text{-}1, z, y_n)$.

$SS_0(j, x_1, ..., x_n)$:- $I(y_1, ..., y_n)$, $SS_1(j, x_1, y_1)$, ..., $SS_n(j, x_n, y_n)$.

$query_2(x_1, ..., x_n)$:- $SS_0(x_1, ..., x_n)$.

Proof.

(a) $query(x_1, ..., x_n) = query_1(x_1, ..., x_n)$.

Note that

$$query(x_1, ..., x_n) = \bigcup_j \{A_1^j(x_1, y_1), \cdots, A_n^j(x_n, y_n), I(y_1, \cdots, y_n)\}. \quad (4.4)$$

For $1 \leq i \leq n$, the projection of $S_i(j, x_i, y_i)$ on attributes x_i and y_i produces a subset of A_i containing all tuples of A_i that will participate in the indicated join in (4.4) during the j^{th} iteration. Hence, from the definition of S_0, the result follows.

(b) $query(x_1, ..., x_n) = query_2(x_1, ..., x_n)$.

Note that the projection of $SS_i(j, x_i, y_i)$ on attributes x_i and y_i produces the relation $A_i^j(x_i, y_i)$ and hence the result. □

The main advantage in partitioning a linear recursive rule into n linear recursive rules is that these n rules do not share a common variable and hence they can be evaluated in parallel using different processors with no communication between them. Let us assume that we have a multiprocessor configuration of n+1 processors U_0, U_1, ..., U_n, where U_0 is the master processor and is star-connected to the remaining n 'slave' processors U_1, ..., U_n. The master processor U_0 is responsible for the evaluation of S_0, the synchronization of the slave processors, and the final termination decision. Each slave processor communicates the new results from an iteration to the master. The acknowledgement from master can be treated as a signal for the next iteration to start; the synchronization, therefore, does not require manipulation of any counters. That is, U_i can ignore the variable j and just compute other attributes in S_i (SS_i) and send the generated tuples to U_0 after each iteration.

With this parallel architecture, we now consider the termination of the parallel algorithms in the presence of cyclic data.

Lemma 4.2

If no new tuples can be generated for S_0 (SS_0) in the j^{th} iteration then no new tuples can be generated in the $j+1^{st}$ iteration.□

Lemma 4.3

In programs D′ (D″) if any of the S_i's (SS_i's) fail to generate a tuple in the j^{th} iteration then S_0 (SS_0) cannot generate a tuple in the j^{th} iteration. □

Due to Lemmas 4.2 and 4.3, the termination of the modified program is assured even if the data is cyclic. Whenever U_0 discovers, either during the data collection phase or after the evaluation of S_0 (SS_0) in a given iteration, that no new tuples have been generated, it broadcasts a 'HALT' message to all slave processors.

The extension of Theorem 4.1 to any general program is quite straight forward. If there are more than one recursive rule, apply Theorem 4.1 on each of the rules independently to obtain the modified program. Further, if there are any mutually recursive rules, one may first remove such rules as explained Section 3.3. For the sake of completeness we recall the method briefly as follows: Note that the query depends ultimately on one derived predicate, say P. Hence a mutually recursive rule r is rewritten using the rules defining the other predicates appearing in r. Since the method is quite similar to the elimination of "unknowns" in a system of equations, instead of giving a formal description, we illustrate the transformation through an example.

Example 4.1

Consider the following logic program G.

Program G

e_1 : $P_1(x, y, z)$:- $I_1(x, y, z)$.

e_2 : $P_2(x, y, z)$:- $I_2(x, y, z)$.

e_3 : $P_3(x, y, z)$:- $I_3(x, y, z)$.

r_1 : $P_1(x, y, z)$:- $P_2(x_1, y_1, z_1)$, $R_1(x, y, z, x_1, y_1, z_1)$.

r_2 : $P_2(x, y, z)$:- $P_3(x_1, y_1, z_1)$, $R_2(x, y, z, x_1, y_1, z_1)$.

r_3 : $P_3(x, y, z)$:- $P_1(x_1, y_1, z_1)$, $R_3(x, y, z, x_1, y_1, z_1)$.

q : Query(y, z) :- $P_1(x, y, z)$, x = constant.

It can be verified that program G is query equivalent to the following program GG obtained by first eliminating P_3 and then P_2.

Program GG

ee_1 : $P_1(x, y, z)$:- $I_1(x, y, z)$.

ee_2 : $P_1(x, y, z)$:- $I_2(x_1, y_1, z_1)$, $R_1(x, y, z, x_1, y_1, z_1)$.

ee_3 : $P_1(x, y, z)$:- $I_3(x_2, y_2, z_2)$, $R_2(x_1, y_1, z_1, x_2, y_2, z_2)$,
$R_1(x, y, z, x_1, y_1, z_1)$.

rr : $P_1(x, y, z)$:- $P_1(x_3, y_3, z_3)$, $R_3(x_2, y_2, z_2, x_3, y_3, z_3)$,
$R_2(x_1, y_1, z_1, x_2, y_2, z_2)$, $R_1(x, y, z, x_1, y_1, z_1)$.

qq : Query(y, z) :- $P_1(y, z)$, x = constant. □

4.3 Performance Analysis

There are several ways of measuring the performance of an algorithm in a parallel environment. In this chapter, we consider the *total time* and *response time* of programs D, D′ and D″ of Theorem 4.1. We assume that semi-naive algorithm is used to evaluate S in

Program D, S_0 in Program D′ and SS_0 in Program D″. We also make the assumption that the order of computing n joins indicated in the body of the recursive rule in Program D and Program D′ are given by $((\cdots((SA_1)A_2)\cdots)A_n)$ and $((\cdots((S_0S_1)S_2)\cdots)S_n)$. Similarly, the order of evaluation of n joins in Program D″ is given by $((\cdots((SS_0SS_1)SS_2)\cdots)SS_n)$. Note that there is no loss of generality, as we assume the same order for all algorithms.

It may be noted that the number of iterations required is the same for all three algorithms and we make the assumption that the required data is available in the corresponding nodes. That is, entire data is available at one node for Program D and in the case of program D′ (D″) the data is distributed among n nodes: $S_i(0, x_i, x_i)$ $(SS_i(0, x_i, x_i))$ and A_i at node i. Thus there is no communication time at the beginning of the execution.

The total time is the sum of IO, CPU and communication time and therefore gives a fair estimate of machine resources usage. The response time is the time elapsed from the initiation to the completion of the algorithm. Parallel algorithms, in general, minimize the response time at the expense of total cost. Therefore both measures are complementary to understand the performance trade-offs of parallel algorithms.

Local processing time is incurred by reading and comparing the operand tuples and producing new tuples using join and union. It may be noted that both join and union can be implemented through hashing with complexity almost linear in the size of the operands [Brat84, Vald88]. Thus we assume that the time to join two relations C and D is given by $\alpha(c|C| + d|D|)$, where α is a constant and c and d are the sizes of tuples in C and D respectively. The

communication time is incurred due to moving the tuples fr m slaves to the master node, and sending the signal from the master node to slaves. In addition to α, the following parameters are also required to evaluate the algorithms:

l maximum number of iterations performed by any one of the processors.

δ_{ij} the expected value of $\frac{|S_i(j, x_i, y_i)|}{|A_i|}$, for i = 1, ···, n and j = 1, ···, l. For the sake of convenience, let δ_{ij} be zero if j $\notin$ [1, l] and i $\notin$ [1, n].

δ'_{ij} the expected value of $\frac{|SS_i(j, x_i, y_i)|}{|A_i|}$, for i = 1, ···, n and j = 1, ···, l. For the sake of convenience, let δ_{ij} be zero if j $\notin$ [1, l] and i $\notin$ [1, n].

γ_j the number of new tuples generated in S_0 of Program D′ (or in S in Program D) during the j^{th} iteration. For notational convenience, let γ_0 be |I| and γ_j be zero if j $\notin$ [0, l].

κ number of unit size tuples per packet. Messages are send in terms of a fixed sized packets.

μ time to process a message (includes both send and receive time and is the overhead involved irrespective of the message length).

ν time to transfer a packet between two nodes. It is assumed that all communication links are identical.

s_i size of the attribute x_i in A_i.

We introduce the following parameters for notational convenience.

$$\beta = \frac{\nu}{\kappa}$$

$$\phi_i = |A_i|.$$

$$s = \sum_{i=1}^{n} s_i$$

The parameter μ is measured in number of CPU instructions; ν is the constant time required to transfer a data packet from master(slave) to slave(master). Messages are of variable size, that is, of multiple packets. The communication time incurred in sending and routing an N packet message is $(\mu + N \times \nu)$. We assume that there is always sufficient buffer space available for holding the packets sent by slave processors.

4.3.1 Analysis of Program D

Note that D being a serial algorithm, both total cost and response time are equal in this case.

The cost for the j^{th} iteration of the recursive rule is $\sum_{i=1}^{n} \alpha(s\gamma_{j-1} + 2s_i\phi_i)$. Hence the total cost TT_u of the uniprocessor algorithm is $TT_u = \sum_{j=1}^{l} \sum_{i=1}^{n} \alpha(s\gamma_{j-1} + 2s_i\phi_i)$.

4.3.2 Analysis of Program D′

This section is divided into two subsections: in the first we compute the total time and in the second we compute the response time. In both subsections we distinguish between two different environments.

By a *distributed environment* we refer to any architecture where one has to include the communication time involved in sending the messages. On the other hand the term *tightly coupled environment* is used to refer any architecture where the communication time is negligible.

4.3.2.1 Analysis of total time

(a) *Distributed environment*

The number of tuples at the beginning of the j^{th} iteration at node i is δ_{ij-1}. Hence the computational cost at node i during the j^{th} iteration is given by $2s_i\alpha(\phi_i + \delta_{ij-1})$. Hence the total computational cost at all the slaves for l iterations is given by $\sum_{j=1}^{l}\sum_{i=1}^{n} 2s_i\alpha(\phi_i + \delta_{ij-1})$. Similarly, the communication cost is the cost involved in sending the tuples generated at each slave processor to the master processor immediately after each iteration. Hence the total communication cost of the i^{th} node to send the result of the j^{th} iteration is $\mu + 2\beta s_i\delta_{ij}$. Therefore the communication cost for the j^{th} iteration is $\sum_{i=1}^{n}(\mu + 2\beta s_i\phi_{ij})$ and the total communication cost is $\sum_{j=1}^{l}\sum_{i=1}^{n}(\mu + 2\beta s_i\delta_{ij})$.

Now the only other cost to be computed is due to joining that takes place at the master node. After each iteration the master processor has to perform n joins involving S_1, S_2, ..., S_n. The time for the j^{th} iteration is $\sum_{i=1}^{n}\alpha(s\gamma_{j-1} + 2s_i\delta_{ij})$. Hence the total time at the master node is $\sum_{j=1}^{l}\sum_{i=1}^{n}\alpha(s\gamma_{j-1} + 2s_i\delta_{ij})$. Thus an expression for the response time in a Distributed environment is given by

$$TT_d = \sum_{j=1}^{l} \sum_{i=1}^{n} \left(\alpha(s\gamma_{j-1} + 2s_i\phi_i + 2s_i(\delta_{ij} + \delta_{ij-1})) + \mu + 2\beta s_i\delta_{ij} \right).$$

(b) *Tightly coupled environment*

An expression for TT_m, the total cost involved in a tightly coupled environment is obtained by ignoring the communication cost. Thus

$$TT_m = \sum_{j=1}^{l} \sum_{i=1}^{n} \alpha(s\gamma_{j-1} + 2s_i\phi_i + 2s_i(\delta_{ij} + \delta_{ij-1})).$$

4.3.2.2 Analysis of Response Time

Assuming pipelining to overlap concurrent tasks we compute the response time of Program D′.

(a) *Distributed environment*

In this case, we assume the following pipelining strategy. At any instance three operations are taking place in parallel: (1) j^{th} iteration of the slave processors; (2) the tuples produced during the $j\text{-}1^{st}$ iteration at each slave processor is being transferred to the master processor and (3) the master processor is performing the join of relations produced by the slave processor during the $j\text{-}2^{nd}$ iteration. Therefore, the response time for distributed processing is

$$RT_d = \sum_{j=1}^{l} \max\left[\max_i\{ 2s_i\alpha(\phi_i + \delta_{ij-1})\}, \max_i\{\mu + 2s_i\beta\delta_{ij-1}\}, \sum_{i=1}^{n} \alpha(s\gamma_{j-2} + 2s_i\delta_{ij-2}) \right]$$

(b) *Tightly coupled environment*

In this case, we assume the following pipelining strategy. At any instance the following two operations are taking place in parallel: (1) j^{th} iteration of the slave processors; (2) the master processor is performing the join of relations produced by the slave processor during the $j\text{-}2^{nd}$ iteration. Therefore, the response time is

$$RT_m = \sum_{j=1}^{l} \max\left[\max_i\{ 2s_i\alpha(\phi_i + \delta_{ij-1})\}, \sum_{i=1}^{n} \alpha(s\gamma_{j-2} + 2s_i\delta_{ij-2})\right]$$

4.3.3 Analysis of Program D"

Since the analysis is very similar to the one performed for Program D', we summarize the results without giving any details.

4.3.3.1 Analysis of total time

(a) *Distributed environment*

$$TT'_d = \sum_{j=1}^{l}\sum_{i=1}^{n} \left(\alpha(s\gamma_{j-1} + 2s_i\phi_i + 2s_i(\delta'_{ij} + \delta'_{ij-1})) + \mu + 2\beta s_i\delta'_{ij}\right).$$

(b) *Tightly coupled environment*

$$TT'_m = \sum_{j=1}^{l}\sum_{i=1}^{n} \alpha(s\gamma_{j-1} + 2s_i\phi_i + 2s_i(\delta'_{ij} + \delta'_{ij-1})).$$

4.3.3.2 Analysis of Response Time

(a) *Distributed environment*

$$RT'_d = \sum_{j=1}^{l} \max\left[\max_i\{ 2s_i\alpha(\phi_i + \delta'_{ij-1})\}, \mu + \max_i\{2s_i\beta\delta'_{ij-1}\}, \sum_{i=1}^{n} \alpha(s\gamma_{j-2} + 2s_i\delta'_{ij-2})\right]$$

(b) *Tightly coupled environment*

$$RT'_m = \sum_{j=1}^{l} \max\left[\max_i\{ 2s_i\alpha(\phi_i + \delta'_{ij-1})\}, \sum_{i=1}^{n} \alpha(s\gamma_{j-2} + 2s_i\delta'_{ij-2})\right].$$

The above analysis shows that the distributed architecture is more expensive than the tightly coupled one. Further, we assume that the join at master is the most expensive one, then the speed-up ratios are

$$SG_1 : \sum_{j=1}^{l}\sum_{i=1}^{n}(s\gamma_{j-1} + 2s_i\delta_{ij}) / \sum_{j=1}^{l}\sum_{i=1}^{n}(s\gamma_{j-1} + 2s_i\phi_i).$$

$$SG_2 : \sum_{j=1}^{l}\sum_{i=1}^{n}(s\gamma_0 + 2s_i\delta'_{ij}) / \sum_{j=1}^{l}\sum_{i=1}^{n}(s\gamma_{j-1} + 2s_i\phi_i).$$

4.4 Performance Comparisons

In this section we compare the cost expressions obtained in the previous section. In particular, we analyse the typical "same generation" Program SG. Our motivation to fix the parameters, as explained below, comes from the recent empirical analysis reported in [Vald88] for parallel evaluation of transitive closure of a binary predicate. Let κ = 20, μ = 5000 instructions, ν = 100 microseconds and α = 1000 instructions. Further, since l plays only a marginal role, as for as our comparative study is concerned, we fix l as 50. It is worth noticing that there exists a large class of relations for which δ_{ij} and δ'_{ij} are almost the same. In fact, if the relation can be modeled as a "tree" or an "inverted tree" or a "cylinder" then $\delta_{ij} = \delta'_{ij}$. Note that these are the only types of relations considered in [Banc86b]. Accordingly we also assume that $\delta_{ij} = \delta'_{ij} = \delta$ for all i and j and let sizes of relations I, A and B be 100,000 tuples [Banc86b] . Finally, set $\frac{\gamma_j}{\gamma_0}$ to be same for all j.

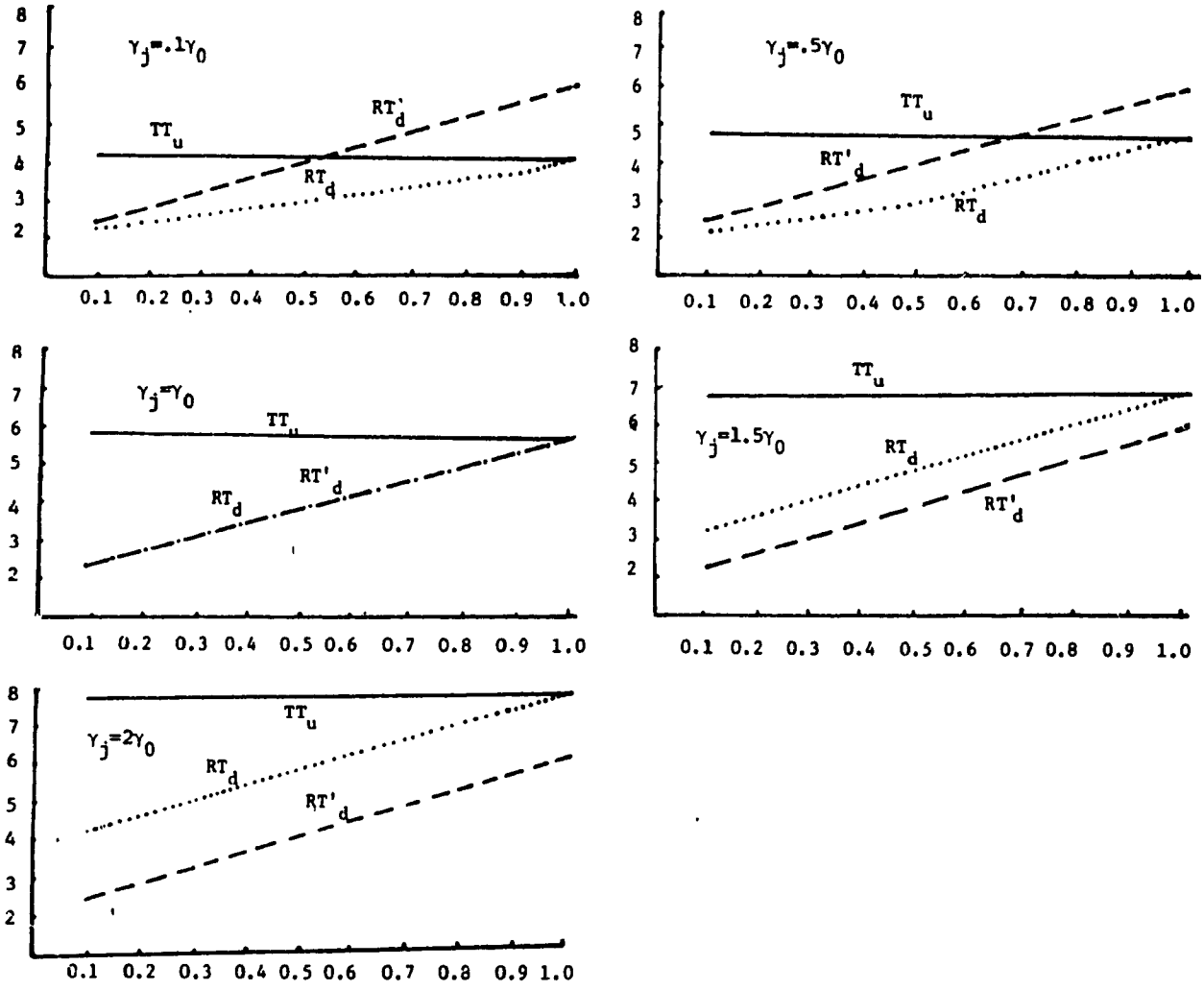


Figure 4.1 Response time versus δ

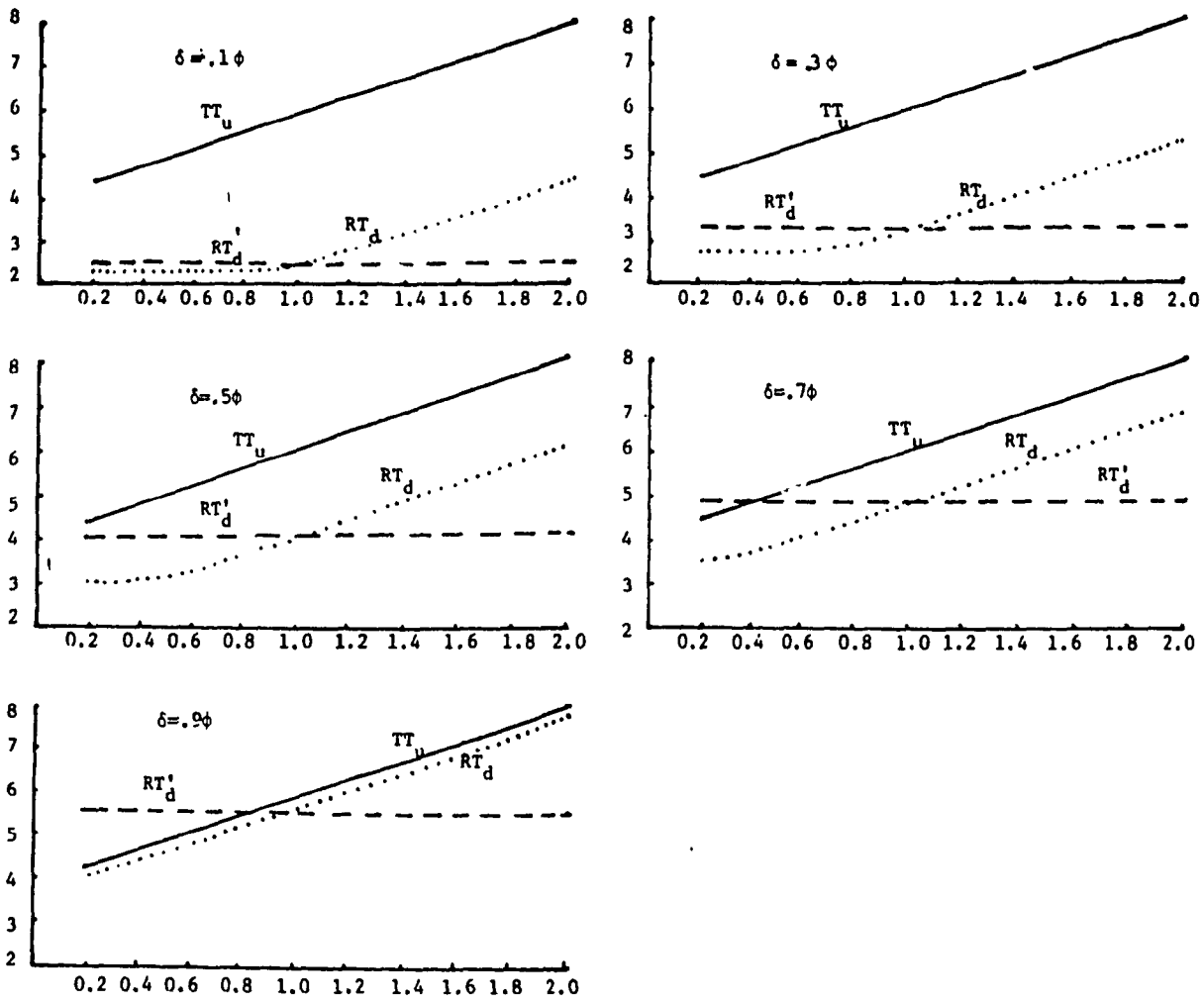


Figure 4.2 Response time versus $\frac{\gamma_j}{\gamma_0}$

Fig. 4.1 shows five different comparisons of TT_u, RT_d and RT'_d. Fixing the ratio $\frac{\gamma_j}{\gamma_0}$ at one of the values 0.1, 0.5, 1.0, 1.5, 2.0, the response time in kiloseconds is shown as a function of δ, the expected ratio of the relation size at a slave during an iteration to the initial relation size. The graph clearly demonstrates the savings in response time achieved by Program SG_1 over Program SG. Further, SG_2 can be seen superior to SG_1 if $\gamma_j < \gamma_0$. Fig. 4.2 also provides five different comparisons of TT_u, RT_d and RT'_d. Fixing δ at one of the values 0.1, 0.3, 0.5, 0.7, 0.9, the response time in kiloseconds is shown as a function of $\frac{\gamma_j}{\gamma_0}$, the expected ratio of the relation size at master during an iteration to the initial relation size. This graph also clearly demonstrates the savings in response time achieved by Programs SG_1 over Program SG. Further, SG_2 can be seen superior to SG_1 if $\gamma_j < \gamma_0$. Thus Program SG_1(Program D") is to be selected if the number of new tuples produced at the master during each iteration is less than the size of the recursive predicate before the iteration and Program SG_2 (program D") is to be chosen otherwise.

4.5 Performing Early Selections

In this section we address the issue of pushing selection ahead in the presence of bound variables. For the sake of simplicity, we consider Program SG. Recall that Program SG is query equivalent to Program SG_1 and Program SG_2. Since Program SG_1 contains the rule

$S_0(j, x, y)$:- $S_1(j, x, x_1)$, $S_0(j\text{-}1, x_1, y_1)$, $S_2(j, y, y_1)$.

to be computed at the master, the relation $S_0(j\text{-}1, x_1, y_1)$ has to be evaluated for all j. Therefore, selections cannot be pushed any

further. On the other hand in Program SG_2 there is no recursive rule at the master node. The only rule to be evaluated at the master is

$SS_0(j, x, y)$:- $SS_1(j, x, x_1)$, $I(x_1, y_1)$, $SS_2(j, y, y_1)$.

Hence it is possible to apply a selection at slaves. For example, if the query is

$query_2(John, y)$:- $SS_0(j, John, y)$.

then selection pushing can be described as follows. First, the master node sends the binding "x = John" to the slave one. Upon receiving this information, instead of computing the entire relation $SS_1(j\text{-}1, x, x_1)$, only the relevant part, $SS_1(j\text{-}1, John, x_1)$, is computed. If we visualize $A(x, x_1)$ as a parent relation, then x_1 in $SS_1(j, John, x_1)$ can be viewed as j^{th} level ancestors of John. Consequently, the size of the relation $SS_1(j, John, x_1)$ is much smaller than the size of $A^j(x, x_1)$. In fact under uniform distribution, we have $|SS_1(j, John, x_1)| = \frac{1}{p}|SS_1(j, x, x_1)|$, where p is the number of distinct values of x appearing as the first attribute in A. This leads to a significant reduction in the following: (1) computation at the slave U_1, (2) communication cost at slave U_1, (3) computation cost at the master U_0. In fact, it is not unreasonable to assume that the response time on pushing selections is $\frac{1}{p}$ times the response time without pushing selections.

4.6 Strongly Partitionable Logic Programs

In this section, we introduce the concept of *strongly partitionable* logic programs which is a special case of partitionable logic programs.

A Datalog program is *strongly partitionable* if there exists a nontrivial partition P covering partition sets of all rules in the

program. Note that if a program is vertically partitionable, there exits a *minimum cover* $\mathcal{M}$ in the sense that $\mathcal{M}$ is a cover for all rules in the program and if $\mathcal{M}'$ is also a cover for all the rules in the program, then $\mathcal{M}'$ is a cover of $\mathcal{M}$. The main result of this section is stated next.

Theorem 4.2

Let VD be the following Datalog program rewritten according to the minimum cover.

$S(x_1, ..., x_n)$:- $P_1(x_1)$, ..., $P_n(x_n)$.
$S(x_1, ..., x_n)$:- $A_1(x_1, y_1)$, ..., $A_n(x_n, y_n)$, $S(y_1, ..., y_n)$

Then VD is equivalent to the Datalog Program VD′ given below:

Program VD′
$S_1(0, x_1)$:- $P(x_1)$.
. . .
$S_n(0, x_n)$:- $P(x_n)$.
$S_1(j, x_1)$:- $A_1(x_1, y_1)$, $S_1(j\text{-}1, y_1)$.
. . .
$S_n(j, x_n)$:- $A_n(x_n, y_n)$, $S_n(j\text{-}1, y_n)$.
$S(x_1, ..., x_n)$:- $S_1(j, x_1)$, ..., $S_n(j, x_n)$.

Proof Similar to that of Theorem 4.1. □

Lemmas 4.2 and 4.3 are valid. The expressions for TT_u, TT_d, TT_m, RT_d and RT_m can be similarly obtained. Since the arity of relations S_i′s got reduced by a factor of 2, the communication cost as well as the computation cost of program VD′ will be much less than

that of Program D′. However, we feel that this situation is very rare. For example, If the program Same-Generation is to be strongly partitionable, I has to be a cartesian product of two sets; for example {(adam, adam)} or {(adam, adam), (adam., eve), (eve, adam), (eve, eve)} are two possible values of I.

4.7 Conclusion

Two different transformation schemes for the parallel evaluation of logic programs are presented. Using a simple model, we have given expressions for the total time and response time. The analysis clearly indicates the superiority of the parallel algorithms over the serial algorithm. We show that the two schemes are complementary in the sense that if the derived relation is expected to produce more new tuples at each iteration compared to the initial value, then Program D″ is to be applied; Program D″ is definitely superior. In fact the cross over point occurs when the number of tuples produced is the same as the initial value of the derived predicate.

We discussed the effect of pushing selections when the query has bound variables. Program D″ is shown to be capable of pushing selections ahead of joins, which in turn further reduces the response time of a given query.

Chapter 5

PERMUTATION DEPENDENCY IN DATALOG PROGRAMS

5.1 Introduction

In this chapter a efficient scheme for maintaining minimal knowledge in the presence of a permutation dependency is presented. Since this scheme eliminates the redundancy in the initial value of the recursive predicate, cost of query processing is reduced by a constant factor. Thus this method improves over other conventional methods in both space and time.

As noted in [Naug87], a logic database must recognize different classes of programs and use algorithms tailored to those classes whenever possible. To achieve this end, a precise notion of classes is necessary. The emphasis of this chapter is on the definition and study of *base-invariant* Datalog programs. Postponing a detailed treatment for later sections, we comment that the following Datalog programs taken from [Banc86a] are base-invariant.

Program Same-Generation

r_0 : SG(x, x).

r_1 : SG(x, y) :- Par(x, x_1), SG(x_1, y_1), Par(y, y_1).

Program Co-Same-Generation

r_0 : CSG(x, x).

r_2 : CSG(x, y) :- Par(x, x_1), CSG(y_1, x_1), Par(y, y_1).

It may be noted that the Same-Generation program is a canonical example for linear Datalog programs. Further it is difficult to

comment on the utility of a class of recursions [Naug87]. However, the results of this chapter is applicable at four different levels of query optimization. First, a linear time algorithm is presented to isolate the minimum knowledge that has to be kept in the Knowledge Base System. This in turn will reduce the space requirement of the system. In the next level, the concept of base-invariance is used to remove the redundant rules from a Datalog program. It is interesting to note that the formalism presented in [Sagi87] is not capable of detecting the type of equivalence detected by the method presented in this chapter. Thus our method is complementary to the one in [Sagi87]. It has been shown that even if the program is a simple linear Datalog program, the magic set transformation [Banc86a] may produce a program with mutually recursive rules. We present a simple method to transform the original program so that the application of the magic set method results in program devoid of any mutually recursive predicates. Yet another level of optimization in both time and space is possible if the derived predicate satisfies a permutation dependency.

In Section 5.2, we introduce the concept of permutation dependency. We provide a linear time algorithm to compute the *generator* of a base predicate. In Section 5.3, *base-invariance of a rule* under a permutation is presented. The concept is then is used to detect and eliminate redundant rules from a Datalog program thus optimizing a given program. Further, the concept of base-invariance used to rewrite the original program so that the modified program will give raise to a better program under magic set transformation than the original program. We present a scheme for the efficient evaluation of derived predicate in Section 5.4. We conclude this chapter in Section 5.5 indicating some of the possible future research

directions suggested by this approach.

5.2 Permutation Dependency

In this section we introduce the concept of permutation dependency, a special case of tuple generating dependency.

Let $\sigma = \begin{pmatrix} 1 & 2 & \dots & n \\ i_1 & i_2 & \dots & i_n \end{pmatrix}$ be a permutation of order k; that is, $\sigma^j \neq I_n$, for $j = 1, \dots, k-1$, and $\sigma^k = I_n$, where I_n is the identity permutation. The *action* of σ on a tuple $t = \langle a_1, a_2, \dots, a_n \rangle$ is the tuple $\sigma(t) = \langle a_{i_1}, a_{i_2}, \dots, a_{i_n} \rangle$. For any relation R, the *action* of σ on R is the relation $\sigma(R) = \{\sigma(t) : t \in R\}$. We say that R satisfies permutation dependency under σ if for all tuples t in R, $\sigma(t)$ is a tuple in R.

Lemma 5.1

If a relation R satisfies a permutation dependency under σ then $\sigma^j(R) = R$, for all integer $j \geq 0$. □

If a base predicate R satisfies a permutation dependency under σ then any minimal set R′ with $R = \bigcup_{j=0}^{k-1} \sigma^j(R')$ is called a *generator* of R. It is clear that if R′ is a generator of R then for all j, $\sigma^j(R')$ are also generators of R.

Our aim is to obtain a generator R′ of the base predicate R under σ. Note that once an efficient algorithm for finding a generator is developed, it can be applied to store a generator instead of the entire relation. Since R′ must be minimal, it must contain

exactly one member t′ of the set $\Gamma_t = \{ \sigma^i(t) : i = 0, ..., k-1\}$, for every tuple $t \in R$. We call t′ the *certificate* of t under σ. Since R′ is a collection of certificates, our aim is to devise a simple method to identify the certificates of the tuples in R.

When the permutation σ has only one cycle then the *certificate* of a tuple t under σ is the tuple which is lexicographically the smallest in the sequence t, $\sigma(t)$, $\sigma^2(t)$, ..., $\sigma^{k-1}(t)$, where k is the length of the cycle. For example, the certificate of the tuple t = <3 6 3 5> under the cycle σ = (1 2 3 4) is <3 5 3 6> because, the lexicographical ordering of {t, $\sigma(t)$, $\sigma^2(t)$ $\sigma^3(t)$} is {<3 5 3 6>, <3 6 3 5>, <5 3 6 3>, <6 3 5 3>}.

Let us next consider a permutation with more than one cycle. It is easy to see that the sequence t, $\sigma(t)$, $\sigma^2(t)$, ..., $\sigma^{k-1}(t)$, cannot be obtained by cyclically rotating the given tuple. Hence we first define the *canonical* form of a permutation. Observe that the disjoint cycles of σ are pairwise commutative and hence they can be rearranged in non-deceasing order of their lengths. Further, members of each cycle can be permuted cyclically so that the least element becomes the leftmost member of the cycle. If two cycles c_1 and c_2 are of the same length then c_1 precedes c_2 provided the leftmost member of c_1 is smaller than the leftmost member of c_2. Since the cycles are disjoint it is always possible to obtain the unique representation for any permutation, and call this the *canonical representation* of σ. For example, the canonical representation of σ = (3 2 4)(6 1)(8 7) is (1 6)(7 8)(2 4 3).

In defining the certificate of t under σ, we assume that σ is canonical and the tuple t is represented as a sequence of *blocks* corresponding to the projections of t on the cycles of σ. Since σ is

canonical, this representation is unique and will be referred to as *canonical representation* of t. For example, the tuple t = < 11 2 13 1 5 7 17 3 9 10 16 2 3 14 5 6 17 > has the canonical representation

$$[t] = [2\ 1]\ [3\ 14]\ [7\ 9\ 16]\ [17\ 3\ 6\ 5]\ [11\ 13\ 5\ 10\ 2\ 17]$$

under σ = (2 4)(8 14)(6 9 11)(7 13 16 15)(1 3 5 10 12 17). Further, the canonical representations of $\sigma(t)$ and $\sigma^2(t)$ are the following:

$$[\sigma(t)] = [1\ 2]\ [14\ 3]\ [16\ 7\ 9]\ [5\ 17\ 3\ 6]\ [17\ 11\ 13\ 5\ 10\ 2]$$

$$[\sigma^2(t)] = [2\ 1]\ [3\ 14]\ [9\ 16\ 7]\ [6\ 5\ 17\ 3]\ [2\ 17\ 11\ 13\ 5\ 10]$$

Notice that i successive cyclic rotations of each block of [t] produce $[\sigma^i(t)]$, $i \geq 1$. Let $[b]^i$ denote the block obtained by i successive cyclic rotations of [b]. We define the *period* of a block [b] as the least positive integer i such that $[b]^i = [b]$. It is easy to see that the period of a block divides the length of the block. For example, the blocks [5 5 5] and [2 4 7 2 4 7] have periods one and three respectively.

Towards giving a formal definition of the certificate of a tuple under σ, we define an order $\prec$ on the set $[\Gamma_t] = \{ [t], [\sigma(t)], [\sigma^2(t)], \ldots, [\sigma^{k-1}(t)] \}$. If $[u_1]\ [u_2]\ \ldots\ [u_j]$ and $[v_1]\ [v_2]\ \ldots\ [v_j] \in [\Gamma_t]$ then $[u_1]\ [u_2]\ \ldots\ [u_j] \prec [v_1]\ [v_2]\ \ldots\ [v_j]$, if there exists an $i < j$ such that $[u_1] = [v_1], \ldots, [u_i] = [v_i]$ and $[u_{i+1}]$ precedes $[v_{i+1}]$ in the lexicographical ordering. It is clear that $\prec$ is a total ordering on the set of canonical forms. Further, this induces a total ordering on the set $\Gamma_t = \{ \sigma^i(t) : i = 0, \ldots, k-1 \}$: $\sigma^i(t) \prec \sigma^p(t)$ whenever $[\sigma^i(t)] \prec$

$[\sigma^p(t)]$. Therefore, the totally ordered set $(\Gamma_t, \prec)$ has a unique least element, which will be called the *certificate* of t under σ.

Example 5.1

This example is to illustrate the process involved in finding the certificate of the tuple t = < 11 2 13 1 5 7 17 3 9 10 16 2 3 14 5 6 17 > under the permutation σ = (2 4)(8 14)(6 9 11)(7 13 16 15)(1 3 5 10 12 17). The certificate of t is obtained from the canonical representation of the certificate of t, which can be constructed by first finding the certificate of the leftmost block in [t] and extending it to the other blocks of [t]. The extension is carried out in such a way that the previously found certificates do not change under the extension procedure. The certificate of the first block in [t] under σ is [1 2]. Notice that this is the first block of $[\sigma(t)]$. Since the period of [1 2] is 2, the first blocks of $[\sigma(t)]$, $[\sigma^2(\sigma(t))]$, $[\sigma^4(\sigma(t))]$, ..., $[\sigma^{2j}(\sigma(t))]$, ... are the same. Note that the period of the block [14 3] is the same as the period of the block [1 2]. Therefore the period of [1 2] [14 3] is also 2, the least common multiple of individual periods. So, once the first block is fixed, the second block also gets fixed. That is, the certificate of the first two blocks of [t] is [1 2] [14 3]. Moreover, these two blocks also appear as the first two blocks of $[\sigma(t)]$, $[\sigma^2(\sigma(t))]$, $[\sigma^4(\sigma(t))]$, ..., $[\sigma^{2j}(\sigma(t))]$, ... The period of the third block of [t] being 3, the period of the first three blocks of [t] is 6. Further, the period of the first two blocks of [t] being 2, there are three possible candidates for the third block of the certificate of [t]. So, we consider the third blocks of $[\sigma(t)]$, $[\sigma^2(\sigma(t))]$ and $[\sigma^4(\sigma(t))]$. Since [7 9 16], the third block of $[\sigma^2(\sigma(t))]$ is lexicographically smallest, [7 9 16] is the certificate of the third block. Thus the first three blocks of $[\sigma^3(t)]$ are in the certificate of [t]. Further, the period of

the first three blocks being 6, the first three blocks of $[\sigma^3(t)]$, $[\sigma^6(\sigma^3(t))]$, ... are identical. The period of the fourth block of [t] being 4, the period of the first four blocks of [t] is 12. Since the period of first three blocks of [t] is 6, there are two candidates for the fourth block of the certificate of [t]. Hence we consider the fourth blocks of $[\sigma^3(t)]$ and $[\sigma^9(t)]$. Since the fourth block of $[\sigma^3(t)]$ is lexicographically smaller than the fourth block of $[\sigma^9(t)]$, the first four blocks of $[\sigma^3(t)]$ should be identically the same as the first four blocks of the certificate of [t]. Moreover, the period of the first four blocks of [t] = the period of [t] = 12. Hence all the five blocks of $[\sigma^3(t)]$ will be identical to all the five blocks of the certificate of [t] under σ. Hence, $[\sigma^3(t)]$ is the canonical representation of the certificate of t under σ. □

To determine whether or not t is a certificate it is sufficient to incrementally examine each block in [t]. In fact, if t is the certificate, then each block of [t] is already its own certificate. This observation is made use of in Algorithm G described below, which will find the generator R′ of a relation R under σ.

Algorithm G

INPUT A permutation $\sigma = \sigma_1 \ldots \sigma_r$, the canonical form of σ. and the base relation R.

OUTPUT A generator R′ of R under σ.

1. $R' \leftarrow \emptyset$

2. *for* each tuple $t \in R$ *do* the following steps.

2.1. Derive the canonical representation of t under σ :

$[t] = [t_1]\ [t_2]\ \ldots\ [t_r]$;

$i \leftarrow 1$;
$p_i \leftarrow$ period of the block $[t_i]$;
$q_i \leftarrow p_i$;

2.2. $w_i \leftarrow \min_{0 \leq j < p_i} \{[t_i]^j\}$.

2.3. certificate $\leftarrow ([w_i] = [t_i])$;

2.4. (* $[t_1]$ $[t_2]$ ... $[t_i]$ is in the certificate of t *)
while $i < r$ *and* certificate *do*
(* examine $[t_{i+1}]$ to see whether $[t_{i+1}]$ is in the certificate of [t] *)
begin
$i \leftarrow i+1$;
$p_i \leftarrow$ period of the block $[t_i]$;
$q_i \leftarrow \text{lcm}(q_{i-1}, p_i)$;
(* lcm denotes the least common multiple *)

$$s_i \leftarrow \frac{q_i}{q_{i-1}};$$

if $s_i > 1$ *then*
begin
$w_i \leftarrow \min_{0 \leq j < s_i} \{ [t_i]^u : u = jq_{i-1} \}$;
certificate $\leftarrow ([w_i] = [t_i])$;
end
end

2.5. *if* certificate *then* $R' \leftarrow R' \cup \{t\}$;
end.

Any algorithm to compute the generator of a relation should consult each tuple of the relation at least once. Since Algorithm G examines each tuple of R just once, the algorithm is linear and is optimal.

We conclude this section by observing the fact that if k is the order of σ then the size of a generator of R under σ is $\frac{|R|}{k}$.

Thus the space requirements are reduced at least by a factor of k.

5.3 Base-Invariance of a Rule

In this section we define the concept of *base-invariance of a rule* under a permutation. The results of this section can be used to eliminate redundant rules from a Datalog program. Further, the concept of base-invariance can be used to rewrite Program D into Program D′ so that the magic set transformation for Program D′ is better than the magic set transformation for Program D.

A permutation σ is compatible with a linear recursive rule r having $S(x_1, ..., x_n)$ as the consequent and $S(y_1, ..., y_n)$ as its antecedent if $(x_k = y_k) \Leftrightarrow (x_{x_{i_k}} = y_{i_k}])$, $i_k = \sigma(k)$. If σ is compatible with r, the *action* of σ on r is defined and is the rule $\sigma(r)$ obtained from r by replacing each occurrence of $x_k(y_k)$ with $x_{i_k}(y_{i_k})$. Note that if σ is not compatible with rule then the action of σ on r is undefined. A linear recursive rule r is *base-invariant* under σ if σ is compatible with r and there exists an extension of σ to local variables of r such that both r and $\sigma(r)$ have identical base predicate invocations upto a renaming of local variables.

Example 5.2

Consider the following rules:

r_1 : A(x, y) :- P(x, z), A(z, y).
r_2 : S(x, y) :- P(x, xp), S(xp, yp), P(y, yp).
r_3 : S(x, y) :- P(x, xp), S(yp, xp), P(y, yp).

All three rules r_1, r_2 and r_3 are compatible under the permutation σ

= (1 2). Let us assume that σ maps z in r_1 to z_1. Therefore,

$\sigma(r_1)$: A(y, x) :- P(y, z_1), A(z_1, x).

Note that even if renaming of the local variable z_1 to z is performed, r and $\sigma(r_1)$ still fail to have identical base predicate invocations. So, r_1 is not base-invariant under σ. On the other hand, by a consistent extension of σ it is possible to map xp(yp) to yp(xp). Hence,

$\sigma(r_2)$: S(y, x) :- P(y, yp), S(yp, xp), P(x, xp).
$\sigma(r_3)$: S(y, x) :- P(y, yp), S(xp, yp), P(x, xp).

Since r_2 (r_3) and $\sigma(r_2)$ ($\sigma(r_3)$) have identical base predicate invocations, r_2 (r_3) is base-invariant under σ. □

For the rest of this chapter, Program D shall mean the following Datalog program.

Program D

e : S(x_1, x_2, ..., x_n) :- E(x_1, x_2, ..., x_n).
r : S(x_1, x_2, ..., x_n) :- S(y_1, y_2, ..., y_n), R.
q : S(x_1, x_2, ..., x_n).

In program D, e is the exit rule with base predicate E, r is the recursive rule with base predicates designated by R, and q is the query.

Theorem 5.1

Let the Datalog program D be such that the predicate E satisfies a permutation dependency under σ. If the rule r is base-invariant under σ then Program D is equivalent to the following Datalog

program D_1.

Program D_1

e : $S(x_1, ..., x_n)$:- $E(x_1, ..., x_n)$.

r : $S(x_1, ..., x_n)$:- $S(\sigma(y_1, ..., y_n))$, R.

q : $S(x_1, ..., x_n)$.

Proof

Since at each iteration, the presence of the tuple $<y_1, ..., y_n>$ in S guarantees and is guaranteed by the presence of $\sigma(<y_1, ..., y_n>)$ in S, the theorem follows. □

5.3.1 Removal of redundant rules

Theorem 5.1 is a strong tool for the detection and removal of redundant rules from a Datalog programs.

Example 5.3

Consider the following Datalog program.

Program D

r_0 : S(x, x).

r_1 : S(x, y) :- P(x, x_1), S(x_1, y_1), P(y, y_1)

r_2 : S(x, y) :- P(x, x_1), S(y_1, x_1), P(y, y_1)

Note that Program D will remain unaffected by methods suggested in [Sagi87]. However, it is easy to observe that all the rules, r_0, r_1 and r_2 are base-invariant under the transposition σ = (1, 2). Hence, by Theorem 5.1, we can rewrite rule r_2 as

r'_2 : S(x, y) :- P(x, x_1), S(σ(y_1, x_1)), P(y, y_1)

That is,

r'_2 : S(x, y) :- P(x, x_1), S(x_1, y_1), P(y, y_1)

which is the same as r_1 . So program D is equivalent to the Datalog program with only two rules r_0 and r_1 . □

In fact we have the following general result.

Lemma 5.2

Let the Datalog program D be such that the predicate E satisfies a permutation dependency under σ. If the rule r is base-invariant under σ then Program D is equivalent to Program D_2 given below:

Program D_2

e : S(x_1, ..., x_n) :- E(x_1, ..., x_n).

r : S(x_1, ..., x_n) :- S(y_1, ..., y_n), R.

r′ : S(x_1, ..., x_n) :- S(σ(y_1, ..., y_n)), R.

q : Query(x_1, ..., x_n) :- S(x_1, ..., x_n). □

5.3.2 Rule rewriting for a better magic set program

One of the problems associated with a magic set transformation is that the transformed program may contain mutually recursive rules even when the original program is free from mutually recursive rules. However, using the concept of base-invariance, it may be possible to transform Program P into an equivalent program P′ so that the magic set transformation will not introduce any mutually recursive predicate. We illustrate this point through the following example.

Example 5.4

Consider the following Datalog program.

Program DD

r_0 : S(x, x).

r_2 : S(x, y) :- P(x, x_1), S(y_1, x_1), P(y, y_1)

r_3 : query(y) :- S(C, y).

Observe that this program is devoid of mutually recursive predicates. The magic set transformation [Banc86a] will produce the following Datalog program

Program Magic-DD

magic.Sg^{fb}(x_p) :- P(x, x_p), magic.Sg^{bf}(x).

magic.Sg^{bf}(y_p) :- P(y, y_p), magic.Sg^{fb}(y).

magic.Sg^{bf}(C).

Sg^{bf}(x, y) :- magic.Sg^{bf}(x), P(x, x_p), P(y, y_p), Sg^{fb}(y_p, x_p).

Sg^{fb}(x, y) :- magic.Sg^{fb}(y), P(x, x_p), P(y, y_p), Sg^{bf}(y_p, x_p).

Sg^{bf}(x, x) :- magic.Sg^{bf}(x).

Sg^{fb}(x, x) :- magic.Sg^{fb}(x).

$query^{f}$(x) :- Sg^{bf}(C, x).

Note that the above program contains four mutually recursive predicates and four mutually recursive rules. However, by Theorem 5.1, program DD is equivalent to the following Datalog Program DD′.

Program DD′

r_0 : S(x, x).

r_2 : S(x, y) :- P(x, x_1), S(x_1, y_1), P(y, y_1)

r_3 : query(y) :- S(C, y).

and the magic set transformation [Banc86a] will produce the following

Datalog program which is devoid of any mutually recursive predicate.

Program Magic-DD′

magic(C).

magic(u) :- magic(v), P(v, u).

Sg(x, x) :- magic(x).

Sg(x, x) :- magic(x), P(x, x_p), P(y, y_p), Sg(x_p, y_p).

query(y) :- Sg(C, y).

This program is definitely superior. Note that in the worst case both predicates magic.Sg^{bf}(x) and magic.Sg^{fb}(y) can be the same. Similarly, Sg^{bf}(x, y) = Sg^{fb}(x, y). For example, if the relation P = {<C, B>, <B, A>, <A, C>, <H, G>, <G, F>, <F, E>, <E, A>} then relations magic.Sg^{bf}(x) and magic.Sg^{fb}(y) are the same and equal to {A, B, C}. Similarly, Sg^{bf}(x, y) = Sg^{fb}(x, y) = {<A, A>, <B, B>, <C, C>} ∪ {<X, Y> | X = A, B, C and Y = E, F, G, H}. On the other hand, if Program Magic-DD′ is used, then the magic set is {A, B, C} and predicate Sg is {<A, A>, <B, B>, <C, C>} ∪ {<X, Y> | X = A, B, C and Y = E, F, G, H}. Thus using Program Magic-DD′ reduces the number of tuples generated by a factor of 2. In general, the cost of the transformed program is $\frac{1}{k}$ times the cost of the original program. □

5.4 An Efficient Scheme for Evaluating Derived Predicates

In this section we present an efficient scheme for evaluating a derived predicate satisfying a permutation dependency.

Lemma 5.3

Let S(E) denote the answer to the query in program D. Assume $E = E_1 \cup E_2$, and let D_i, i = 1, 2 be Datalog programs such that the recursive part in each of them is identical to the recursive part in D and the base predicate in the exit rule of D_i is E_i. Then $S(E) = S(E_1) \cup S(E_2)$.

Proof

From rule r, it can be seen that to establish the presence of a tuple in $S(E_i)$ we require exactly one tuple from the predicate E_i, for i = 1, 2. □

As a consequence of Lemma 5.3, the result of a query in program D can be realized as the union of results of Datalog programs which differ only in their exit rules. In particular, there is no duplication of work due to recursive iterations if the input instances in the exit rules are disjoint.

Theorem 5.2

Let σ be a permutation on {1, 2, ..., n} and D_1 be the program

$e_1 : S_1(\sigma(x_1, ..., x_n)) :- E(x_1, ..., x_n).$
$r_1 : S_1(\sigma(x_1, ..., x_n)) :- S_1(\sigma(y_1, ..., y_n)), R.$
$q_1 : S_1(x_1, ..., x_n).$

Then the tuple $<a_1, ...,a_n>$ is in the predicate S defined by program D if and only if the tuple $\sigma(<a_1, ...,a_n>)$ is in the predicate S_1 defined by program D_1.

Proof

The proof is by induction on the iteration level. At level 0, S is assinged the value of E, and S_1 is assigned the value of σ(E). So the result is true for level 0. Assume that result is true for level m. Let $<b_1, ..., b_n>$ be a tuple of S generated at level m and let $<a_1, ..., a_n>$ be a tuple generated at level m+1 by the application of the recursive rule r on $<b_1, ..., b_n>$. Now $\sigma(<b_1, ..., b_n>)$ is a tuple in S_1 (generated at level m) by induction hypothesis. Since "R" is the same for both D and D_1, it follows that, in the body of the recursive rule x_1 will be instantiated to a_1, x_2 will be instantiated to a_2, ..., and x_n will be instantiated to a_n during the application of r_1 with $y_1 = b_1, ..., y_n = b_n$. Therefore, the tuple $\sigma(<a_1, ..., a_n>)$ will be generated at level m+1 in the case of program D_1. □

The next theorem combines the above results and provides an algorithm for speeding up the process of logic query evaluations.

Theorem 5.3

Let the Datalog program D be such that the predicate E satisfies a permutation dependency under σ of order k. If the rule r is base-invariant under σ then

a) $\sigma^j(S(E')) = S(\sigma^j(E')), j = 0, 1, ..., k-1$

b) $S(E) = \bigcup_{j=0}^{k-1} S(\sigma^j(E')) = \bigcup_{j=0}^{k-1} \sigma^j(S(E'))$

c) S(E) is invariant under σ,

where E′ is a generator of E under σ.

Proof

Proof of (a) is by induction on j. Since E′ is a generator of E, both E′ and σ(E′) are subsets of E. Therefore, by Theorem 5.2, $<a_1, a_2, ..., a_n>$ is a tuple of E′ if and only if $\sigma(<a_1, a_2, ..., a_n>)$ is a tuple of σ(E′). Hence $\sigma(S(E')) = S(\sigma(E'))$. So the result is true j = 1. By induction hypothesis, $\sigma^j(S(E')) = S(\sigma^j(E'))$. Now $\sigma^{j+1}(S(E')) = \sigma^j(\sigma(S(E'))$ $= \sigma^j(S(\sigma(E'))) = S(\sigma^{j+1}(E'))$. This completes the proof of (a). To prove (b) observe that $E = \bigcup_{j=0}^{k-1} \sigma^j(E')$. Further, E′ and $\sigma^j(E')$ (j = 1, ..., k−1) are subsets of E. Therefore, by Lemma 5.3 and (a) we have $S(E) = \bigcup_{j=0}^{k-1} S(\sigma^j(E')) = \bigcup_{j=0}^{k-1} \sigma^j(S(E'))$. Clearly $\sigma(S(E)) = \bigcup_{j=0}^{k-1} \sigma^{j+1}(S(E')) = \bigcup_{j=0}^{k-1}$ $\sigma^j(S(E')) = S(E)$. □

Theorem 5.3 gives an important tool for the optimization of Datalog programs. By applying the recursive rule to E′, a generator of E, we can find S(E′). Once this computation is over, S(E) can be computed by using the result (b). If the order of σ is k, it can be seen that $|E'| = \frac{1}{k}|E|$ and further if $S^i(E')$ and $S^i(E)$ denote the tuples generated at the i^{th} iteration then $|S^i(E')| = \frac{1}{k}|S^i(E)|$. Therefore, we can speed up the evaluation of predicate S by a factor of k.

Secondly, by result (c) of Theorem 5.3, S(E) is also invariant under σ. Therefore, Theorem 5.3 can be extended to any Datalog program in which the initial predicate as well as the recursive rules are invariant under σ. Moreover, if the Datalog program as well as the initial predicate are invariant under several permutations σ_1, σ_2, ..., σ_j then choose the permutation σ of maximum order in the group generated by $\{\sigma_1, \sigma_2, ..., \sigma_j\}$ for optimization purposes.

Example 5.5

Consider the following Datalog program D.

Program P

$S(x, y, z, w, v)$:- $A(x, y, z, w, v)$.
$S(x, y, z, w, v)$:- $C(x, z_1)$, $C(y, x_1)$, $C(z, y_1)$, $D(w, v_1)$,
$D(v, w_1)$, $S(x_1, y_1, z_1, w_1, z_1)$.

query(x, y, z, w,v) :- S(x, y, z, w, v).
where A instantiated as below:

a_{18}	a_{19}	a_{20}	b_{12}	b_{13}
a_{18}	a_{19}	a_{20}	b_{13}	b_{12}
a_{20}	a_{18}	a_{19}	b_{12}	b_{13}
a_{20}	a_{18}	a_{19}	b_{13}	b_{12}
a_{19}	a_{20}	a_{18}	b_{12}	b_{13}
a_{19}	a_{20}	a_{18}	b_{13}	b_{12}

Let relation C consist of all tuples of the form $<a_i, a_{i+3}>$ for i = 0, ..., 20 and relation D consist of all tuples of the form $<b_i, b_{i+2}>$, for i = 0, ..., 13.

The tuples generated by semi-naive evaluation are shown below.

Iteration Level	Tuples Generated				
	a_{18}	a_{19}	a_{20}	b_{12}	b_{13}
	a_{18}	a_{19}	a_{20}	b_{13}	b_{12}
0	a_{20}	a_{18}	a_{19}	b_{12}	b_{13}
	a_{20}	a_{18}	a_{19}	b_{13}	b_{12}
	a_{19}	a_{20}	a_{18}	b_{12}	b_{13}
	a_{19}	a_{20}	a_{18}	b_{13}	b_{12}
	a_{15}	a_{16}	a_{17}	b_{10}	b_{11}
	a_{15}	a_{16}	a_{17}	b_{11}	b_{10}
1	a_{17}	a_{15}	a_{16}	b_{10}	b_{11}
	a_{17}	a_{15}	a_{16}	b_{11}	b_{10}
	a_{16}	a_{17}	a_{15}	b_{10}	b_{11}
	a_{16}	a_{17}	a_{15}	b_{11}	b_{10}
.			.		
.			.		
.			.		
	a_0	a_1	a_2	b_0	b_1
	a_0	a_1	a_2	b_1	b_0
6	a_2	a_0	a_1	b_0	b_1
	a_2	a_0	a_1	b_1	b_0
	a_1	a_2	a_0	b_0	b_1
	a_1	a_2	a_0	b_1	b_0

It may be noted that at each level of iteration six new tuples are generated and they are used to produce new tuples in the next iteration. However, notice that relation A satisfies a permutation

dependency under σ = (2, 3, 1, 5, 4) and hence A′ = {(a_{18}, a_{19}, a_{20}, b_{12}, b_{13})} is a generator of A. Let P′ be the program obtained by replacing A in P by A′. Since the recursive rule is base-invariant, the output of Program P can be obtained in two steps as follows:

Step 1. Evaluation of Program P′

The output of Program P′ is

Iteration Level	tuples generated				
0	a_{18}	a_{19}	a_{20}	b_{12}	b_{13}
.	.	.			
.	.	.			
.	.	.			
6	a_0	a_1	a_2	b_0	b_1

During each iteration, only one new tuple is produced and so the number of new tuples necessary to produce the next iteration level tuples has been reduced by a factor of $\frac{1}{6}$. Thus the cost of evaluating P′ is only $\frac{1}{6}$ of the cost of evaluating P in both time and space.

Step 2. Applying Permutations

The output of Program P′ being a generator of Program P, the output of Program P can be obtained by the action of σ^j for j = 1, 2, ..., j-1 and then unioning the result. Once a tuple is produced, $\sigma^j(t)$, j = 1, 2, ···, j-1 can be obtained without any disk access. Thus the cost of our method is approximately $\frac{1}{6}$ times the cost of Program P. □

5.5 Conclusion

In this chapter we introduced the concept of permutation dependency. Optimization techniques were presented for four different levels: (1) in storing a base predicate, (2) removing redundant rules from a Datalog program, (3) rewriting a Datalog program so that the resultant program is more suitable for magic set optimization and (4) efficient computation of a derived predicate satisfying a permutation dependency.

Throughout our discussion, we kept the "famous" same-generation program as an example, which many researchers treat as a canonical example for linear Datalog programs. A future research direction would be to consider other types of tuple generating dependencies to optimize both space and time requirements of a Knowledge Base System.

Chapter 6

COMPLEXITY OF EVALUATING RECURSIVE QUERIES IN LOGIC DATABASE

6.1 Introduction

In this chapter complexity measures are obtained for the evaluation of recursive queries in logic database. In particular, we show that the same generation program is computationally more difficult than the transitive closure program. We classify chain rule programs into regular and non-regular programs. We prove that all regular chain programs have the same complexity as the transitive closure program and all non-regular chain programs have the same complexity as the same generation program. A simple algorithm to identify linear binary rule programs that can be transformed to an equivalent binary chain rule program is presented.

In spite of several attempts to optimize recursive queries it is known that the extent of achievable efficiency in general logic programs is limited. The first paper to address the fundamental complexity issues relating to the evaluation of logic queries is [Szym77]. More recently, several researchers, notably [Marc87, Sacc87] have attempted complexity analyses on some selected methods such as magic set, counting and magic counting methods; however no attempt had been made to obtain worst case complexity measures for general Datalog programs. In this chapter upper and lower bounds for linear chain rules are obtained. These results provide new insight into the complexity surrounding query evaluation.

6.2 Notations and Basic Results

We follow the general notations from [Banc86a, Banc86b] and make the usual assumption that the rules are range restricted. Let M = <e, r, q> denote a Datalog program where e is the unique exit rule, q is the query and r is a recursive rule. That is, a typical Datalog Program M is

Program M

e : P :- I.

r : P :- P, R_1, R_2, ···, R_t.

q : Query :- P.

The cost (also referred to as the time complexity) of Program M is dominated by the cost of evaluating the recursive rule r and hence the cost of the program will be identified with the cost of evaluation of r. Since we are interested only in the worst case time complexity, the initial value assigned to P by rule e does not play a role in our cost computation. Further, if a program has more than one recursive rule then its cost is the sum of the costs of the programs, each with one recursive rule. Since the query depends on only one recursive predicate, linear mutual recursion can be replaced by a linear recursive rule and a finite number of non-recursive rules. According to our convention, the cost of evaluation of non-recursive rules are ignored. Thus we need only consider typical Datalog programs such as M. Following [Szym77], assume that all attribute values belong to a single domain D of size n.

We use the notation O, Θ, Ω for describing the asymptotic time complexities. If the time complexity of algorithm is f(n) where n is the domain size and g(n) is another function of n, then

(1) $f(n) = O(g(n))$, if there exits constants c and n_0 such that $f(n) \leq cg(n)$ for all $n \geq n_0$.

(2) $f(n) = \Omega(g(n))$, if there exits constants c and n_0 such that $f(n) \geq cg(n)$, for all $n \geq n_0$.

(3) $f(n) = \Theta(g(n))$, if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

6.2.1 The model of computation

Our model of computation is very similar to the one described in [Szym77]. In particular, the only operations allowed are union ($\cup$), composition ($\circ$), reflexive transitive closure ($*$) and inverse ($^{-1}$). The operands are binary relation symbols chosen from an alphabet D (the dense relations). There is a major difference between the model considered in this chapter and the one considered in [Szym77]. While expressions involving only a finite number of operators are considered in [Szym77], in our model there is no bound on the number of operators.

As in [Szym77], the unit cost is the time taken to access a tuple in a relation. As observed in [Szym77] both join and transitive closure have the same complexity equal to $O(n^m)$; the cost of multiplying two $n \times n$ matrices. Observe that union of any two relations has time complexity less than $O(n^m)$. For M = <e, r, q>, the cost of Program M is given by $cost_\alpha(r)$ which is the time complexity of evaluating the recursive predicate in r using Algorithm α. Let *lcost(r)* denote the minimum computational cost involved in evaluating Program M irrespective of the algorithm in the worst case and let *ucost(r)* denote the least upper bound of $cost_\alpha(r)$ in the worst case.

6.3 Complexity of Linear Chain Rules

We start our discussion on complexity by considering chain rules. For us a linear chain rule has the following form:

$P(x_0, x_n)$:- $A_1(x_0, x_1)$, ···, $A_n(x_{n-1}, x_n)$.

where $x_i \neq x_j$ for $i \neq j$ and A_i is the same as P for at most one i. The predicate A_i immediately precedes A_{i+1} if the second argument of A_i is the first argument of A_{i+1}. Thus there is an inherent ordering of predicates in the body of a chain rule.

Example 6.1

Consider the following three chain rules:

r_1 : P(x, y) :- A(x, w), B(w, z), C(z, y).

r_2 : P(x, y) :- A(x, w), P(w, z), C(z, y).

r_3 : P(x, y) :- P(x, w), A(w, z), B(z, y).

Observe that rules r_2 and r_3 are recursive and linear. However, rule r_1 is not recursive. □

6.3.1 Language of a linear chain program

Consider the following chain rule program M_1:

Program M_1

Ancestor(x, y) :- Parent(x, y).

Ancestor(x, y) :- Parent(x, z), Ancestor(z, y).

query(x) :- Ancestor(John, x).

We can view each of these rules as productions in a grammar. In this context, the database predicates (Parent in this example) appear as terminal symbols, and the derived predicates (Ancestor in this example) as the non-terminal symbols. Finally, to pursue the analogy,

we shall take the distinguished symbol to be query(x). Although there is no ordering of the predicates in the body of an arbitrary rule, in the case of chain rules we have imposed a unique order as dictated by the appearance of variables. Thus the analogy can be strengthened further. In fact we can talk about the language generated by the grammar and discuss concepts such as regular grammar, context-free grammar, etc. in the context of chain rule programs. The language generated by the "grammar" of Program M_1 is

{Parent(John, x); Parent(John, x), Parent(x, x_1);
Parent(John, x), Parent(x, x_1), Parent(x_1, x_2); ···; }

This language has two interesting properties: (i) it consists of first order sentences involving only base predicates; that is, each word of this language can be directly evaluated against the database, and (ii) if we evaluate each word of this language against the database and take the union of all these results, we get the answer to the query.

We begin our discussion considering the following two chain programs.

Program TC (* Transitive Closure *)

e_1 : TC(x, y) :- A(x, y).
r_1 : TC(x, y) :- B(x, z), TC(z, y).
q_1 : Query(x, y) :- TC(x, y).

Program GS (* Generalized Same generation *)

e_2 : GS(x, y) :- P(x, y).
r_2 : GS(x, y) :- L(x, z), GS(z, w), R(w, y).
q_2 : Query(x, y) :- CT(x, y).

It may be noted that in rule r_1 the recursive predicate appears as the last one among the predicates appearing in the body whereas in the case of rule r_2 the recursive predicate appears neither as the first nor as the last predicate. In the terminology of language theory, the grammar associated with program TC is a regular grammar and the one associated with Program GS is not a regular one. It is well known from the language theory that a grammar is regular if and only if the production rules are either left linear or right linear. Therefore, we define a linear chain program to be *regular* if the recursive predicate appears either as the first or as the last one in the recursive rule.

It may be recalled that a language is regular if and only if the it has a regular grammar. Hence we state the following lemma without any proof.

Lemma 6.1

The language associated with a linear chain rule program is regular if and only if the recursive predicate appears either as the first or as the last one in the body. □

Recall that a regular expression determines a regular language and conversely a regular language can be expressed through a regular expression involving only a finite number of operators +, $^{-1}$, · and *. Hence it follows that the answer to the query in a linear chain rule program can be expressed as a regular expression involving a finite number of operators $\cup$, $^{-1}$, $\circ$ and * if and only if the program is regular. This result we state in the next lemma.

Lemma 6.2

The answer to the query in a linear chain program has a finite relational expression involving the operators $\cup$, $^{-1}$, $\circ$ and * if and only if the recursive predicate in the body appears either as the first or as the last one. □

Corollary 6.1

The answer to the query "$query_1$" in Program TC has a finite relational expression involving the operators $\cup$, $^{-1}$, $\circ$ and *. □

Corollary 6.2

The answer to the query "$query_2$" in Program SG does not have a finite expression involving the operators $\cup$, $^{-1}$, $\circ$ and *. □

Note that Corollary 6.1 only shows the existence of a finite expression involving operators $\cup$, $^{-1}$, $\circ$ and *. In fact the expression for $query_1$ is B^*A. Thus we have the lower and upper bounds for r_1.

Lemma 6.3

$$lcost(r_1) = \Omega(n^2) \text{ and } ucost(r_1) = O(n^m).$$ □

On the other hand Corollary 6.2 claims the non-existence of a finite expression for the answer to the $query_2$ involving $\cup$, $^{-1}$, $\circ$ and *. However, it may be noted that $query_2 = \sum_{i=0}^{\infty} L^i P R^i$. Although there is no regular expression equivalent to the above expression, for a given database, one need to compute only a finite number of terms. In fact, since the domain size is n, treating the domain as the vertex set and the relations as edge set, it is clear that there can not be

any cycle in L or R having more than n nodes. Consequently, $query_2 = \sum_{i=0}^{n^2} L^iPR^i$. Since L^iPR^i can be computed from $L^{i-1}PR^{i-1}$ by two compositions, the entire expression may be computed through $O(n^2)$ compositions. Thus $ucost(r_2) = O(n^{m+2})$.

Lemma 6.4

$$lcost(r_2) = \Omega(n^3). \qquad \square$$

Proof.

Since R is a binary relation, R can be represented as an n by n matrix. Since R is of rank n in the worst case, the matrices I (= the identity matrix), R, R^2, ..., R^{n-1} are algebraically independent. Therefore, any polynomial in R of degree n-1 requires at least n-1 "multiplications". Since there can not be a finite expression to compute $query_2$, any expression that computes $query_2$ should compute $\sum_{i=0}^{n-1} L^iPR^i$ explicitly or implicitly. Consequently, the computation of $query_2$ involves at least n-1 compositions. Since each composition needs at least $\Omega(n^2)$ time, the result follows.

Since $m < 2.81$, we state the main result as

Theorem 6.1

$$lcost(r_2) > ucost(r_1). \qquad \square$$

Thus there is an inherent order of magnitude difference in the computational complexity of regular versus non-regular query evaluation costs.

This raises an important question: what class of queries are equivalent to regular queries under some transformation scheme? A simple algorithm to address this issue is presented in the next section.

6.4 Valid Transformations

In this section we discuss five transformations on binary logic programs such that any finite sequence of them may be applied in transforming a binary logic program to a chain rule program. This transformed program can then be tested for regularity. These transformations possess two important properties: (1) The result of applying the transformation is a binary logic program; (2) The cost of the transformation is at most the cost of performing a transitive closure.

Transformation : Inverse

Let R(x, y) be a binary base relation. Then $R^{-1}(x, y)$ is defined as the binary relation obtained by interchanging the attributes x and y. Since this can be done in $\Theta(n^2)$ time, this is a valid transformation. We shall first illustrate the transformation through a simple example.

Example 6.2

Consider the following three rules:

rr_1 : P(x, y) :- A(x, w), B(z, w), C(z, y).
rr_2 : P(x, y) :- A(x, w), P(w, z), C(y, z).
rr_3 : P(x, y) :- P(x, w), A(z, w), B(z, y).
Observe that Rule rr_1 can be replaced by
e_1 : D(w, z) :- B(z, w).

r_1 : P(x, y) :- A(x, w), D(w, z), C(z, y).
Rule rr_2 can be replaced by
e_2 : E(z, y) :- C(y, z).
r_2 : P(x, y) :- A(x, w), P(w, z), E(z, y).
and Rule rr_3 can be replaced by
e_3 : F(w, z) :- A(z, w).
rr_3 : P(x, y) :- P(x, w), F(w, z), B(z, y).

Since the rules r_i are regular, the time complexities of rr_i are the same as the time complexities of r_i, for i = 1, 2 and 3. □

We make the above observation formal through the following lemma.

Lemma 6.5
Let r be a binary rule and let r′ be the rule obtained by replacing one or more base predicates in the body of r by their inverses. Then *lcost(r) = lcost(r′)* and *ucost(r) = ucost(r′)*. In other words, the time complexity is invariant under an inverse operation. □

Transformation : Composition

Let two base predicates A_i and A_j share a common variable which they do not share with any other base predicate in the body of the recursive rule. We replace A_i and A_j by the predicate obtained by taking their composition after performing any necessary inverse operation. Since composition and transitive closure operation have the same complexity, we have the next lemma.

Lemma 6.6

Let r be a binary rule and let r′ be the rule obtained by replacing two base predicates in the body of r by their composition. Then *lcost(r)* = *lcost(r′)* and *ucost(r)* = *ucost(r′)*. In other words, the time complexity is invariant under the composition transformation. □

Transformation : Intersection

Another simple transformation that has cost less than $O(n^m)$ is the intersection operator. We illustrate this transformation through the following example.

Example 6.3

Consider the following chain program:

Program One

T(x, y) :- E(x, y).
T(x, y) :- A(x, z), B(x, z), T(z, y).
$Query_3$:- C(x, y).

We claim that Program One is query equivalent to Program Two defined below:

Program Two

C(x, y) :- A(x, y), B(x, y).
T(x, y) :- E(x, y).
T(x, y) :- C(x, z), T(z, y).
$Query_4$:- T(x, y).

Since the predicate C can be computed using one intersection, the cost of computing $Query_4$ is the same as the cost of computing $Query_3$.

Thus the cost of Program One is the same as the cost of Program Two, which is the transitive closure program. □

Lemma 6.7

Let r be a binary rule and let r′ be the rule obtained by replacing one or more base predicates in the body of r by their intersections. Then *lcost(r) = lcost(r′)* and *ucost(r) = ucost(r′)*. In other words, the time complexity is invariant under the intersection transformation. □

Transformation : separation of local components

For the definition of local components see Section 4.2.1

Lemma 6.8

The Datalog program D is equivalent to the program D′ if C is non-empty, and is equivalent to D″ if C is empty.

Program D: $S(x, z)$:- $P(x, z)$.

$S(x, z)$:- $B(x, x_1)$, $S(x_1, z)$, $C(y, y_1)$.

Program D′: $S(x, z)$:- $P(x, z)$.

$S(x, z)$:- $B(x, x_1)$, $S(x_1, z)$.

Program D″: $S(x, z)$:- $P(x, z)$. □

It may be noted that whether or not C is empty, the resulting program is regular. Therefore, the cost of the original program is the same as that of finding the transitive closure.

Transformation : separation of independent components

Lemma 6.9

The Datalog program M is equivalent to the program M′.

Program M: $S(x, y) :- P(x, y).$

$S(x, y) :- B(x, x_1), C(x_1, z), S(z, y), D(y, w).$

Program M′: $S(x, y) :- P(x, y).$
$S_1(x, y) :- P(x, y), D(y, w).$

Proof. See Section 4.2.1 □

Two similar results are stated below:

Lemma 6.10

The Datalog program N is equivalent to the program N′.

Program N: $S(x, y) :- P(x, y).$

$S(x, y) :- B(x, x_1), C(x_1, z), S(z, y), D(x_1, w).$

Program N′: $S(x, y) :- P(x, y).$
$C_1(x_1, z) :- D(x_1, w), C(x_1, z).$
$S(x, y) :- B(x, x_1), C_1(x_1, z), S(z, y).$ □

Lemma 6.11

The Datalog program L is equivalent to the program L′.

Program L: $A(x, y)$:- $B(x, x_1)$, $C(x_1, z)$, $D(x_1, w)$.

Program L′: $C_1(x_1, z)$:- $D(x_1, w)$, $C(x_1, z)$.
$A(x, y)$:- $B(x, x_1)$, $C_1(x_1, z)$. □

6.5 The Graph Model

In this section we present a simple algorithm to detect whether or not a binary rule can be transformed to a finite number of chain rules, through a finite sequence of transformations presented in the last section. We consider linear binary recursive rules with no constants. Further no variable appears more than once in the recursive predicate.

Define a graph, called B-graph, as follows: Variables occurring in the rule form the vertex set V of the graph. If A(u, v) is a predicate appearing in the body, then there is a directed labled edge in the B-graph from u to v with label A. Define a *trail* from u to v as a directed path from vertex u to vertex v without any loops. Then we have the following:

Lemma 6.12
A binary rule with head predicate P(x, y) is a chain rule if and only if the B-graph of the rule is a trail from x to y. Further, the rule is regular if and only if the edge labeled P is the first or as the last edge in the trail. □

6.5.1 Graphical interpretation of valid transformations

Recall that the effect of an inverse transformation on a binary rule is to interchange the attributes appearing in one of the base predicates in the rule body. Hence, performing an inverse

transformation on an edge (u, v) labeled A is equivalent to removing the edge (u, v) and adding a new edge (v, u) labeled A^{-1} to the edge set. Since inverse is a valid transformation, by an *undirected trail* we mean a set of edges of B-graph which may form a trail under a sequence of inverse operations.

The composition transformation can be seen as a vertex removing operation. If v is a vertex in the graph with degree two, it is possible to apply inverse transformation so that the vertex v has both indegree and outdegree equal to one. Let the incoming edge to v be (x, v) with label A_i and the outgoing edge from v be (v, y) with label A_j. If both A_i and A_j are base predicates then the composition transformation can be performed. The new B-graph is obtained by removing edges labeled A_i, A_j and the vertex v and adding a new edge from x to y with label $A_i \circ A_j$ to the edge set.

The intersection transformation can be interpreted as follows: Let A_i and A_j be two parallel edges from u to v. Since we have included inverse transformation, without any loss of generality, we may assume that both the edges have the same orientation. Then the intersection transformation is equivalent to removing the edges labeled A_i and A_j, and adding a new edge (u, v) with label $A_i \cap A_j$.

It is important to note that a new edge added is treated similar to any other edge corresponding to a base predicate for all future transformations.

The graphical interpretation of Lemma 6.8 is that if the B-graph consists of more than one connected component then the cost of the rule is the maximum of the costs involved in computing each component seperately. Therefore, by Lemma 6.8, transformations may be applied to each of the components separately. It all the

components can be reduced to trails then the orginal rule is equivalent to a set of chain rules.

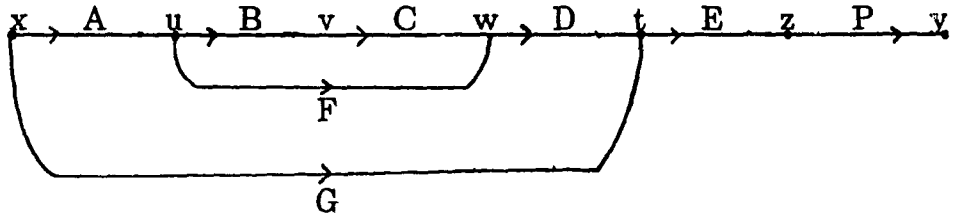
Now we proceed to give a graphical interpretation to Lemmas 6.9, 6.10 and 6.11. Notice that above three lemmas are similar in the sense that they replace a given rule by two rules through the introduction of a new predicate. Further, body predicates of the given rule body has been partitioned into two sets $\mathcal{B}_1$ and $\mathcal{B}_2$ so that $\mathcal{B}_i$ is the set body predicates of new rule i (i = 1, 2). Let $\mathcal{V}(\mathcal{B}_i)$ denote the set of variables in the orginal rule appearing in one of the predicates of $\mathcal{B}_i$. Let P(x, y) be the head predicate of a recursive rule. By Lemmas 6.9, 6.10 and 6.11, it is possible to replace a rule by two rules if $\mathcal{V}(\mathcal{B}_1) \cap \mathcal{V}(\mathcal{B}_2)$ is a singleton set and either $\{x, y\} \subseteq \mathcal{V}(\mathcal{B}_1)$ or $\{x, y\} \subseteq \mathcal{V}(\mathcal{B}_2)$. Applying the transformations to the B-graph of a rule is equivalent to splitting a component of B-graph into two components by "cutting" a vertex into two. Recall that a if s and t are two vetices of an undirrected graph such that any path from s to t involves a vertex r, then r is a cut vertex of the graph. Therefore, we define a *cut vertex* of a B-graph as the cut vertex of the underlying undirected graph. From the above discussion it follows that a component of a B-graph can be replaced by two components by duplicating any one of its cut vertices and partitioning the edges of the given component into two sets provided all trails from x to y are preserved. Hence we may call any of these transformations a *cut rule*. Further, we may apply transformations to each of the components and the resulting rule is a chain rule if and only the resulting graph reduces to one or more disjoint undirected trails. Further, the rule with head predicate P(x, y) is regular if either the first edge or the last edge in the undirected trail from x to y is labeled P.

We now present a simple example to illustrate the intersection transformation.

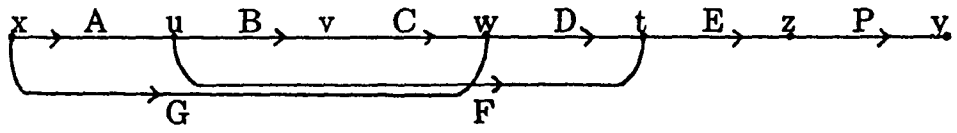
Consider the following two recursive rules and their corresponding B-graphs.

(a) P(x, y) :- A(x, u), B(u, v), C(v, w), D(w, t),
E(t, z), P(z, y), F(u, w), G(x, t).

(b) P(x, y) :- A(x, u), B(u, v), C(v, w), D(w, t),
E(t, z), P(z, y), F(u, t), G(x, w).



(a)



(b)

Figure 6.1 B-graphs

Notice that in Fig. 6.1(a), it is possible to compose B and C and create an edge (u, w) labeled B∘C parallel to the edge labeled F. Now, the intersection transformation can be applied. Thus edges B∘C and F are removed and a new edge (B∘C)∩F is added to the graph. This modified graph is a trail with a subtrail from x to y. Further, P appears as the label of the last edge. Therefore, we conclude that the rule (a) can be replaced by a finite number of chain rules together with at most one regular rule.

On the other hand rule (b) cannot be replaced by a finite number of chain rules. Notice that the vertex u and w can not be removed through the composition and hence it is not possible to remove either edge G or F from the B-graph. We call edges G and F *a pair of conflicting edges.*

We summarize our discussion as a general result in the next theorem. The proof is straight forward and hence omitted.

Theorem 6.2

A binary rule with head predicate P(x, y) can be transformed into a regular rule if the underlying undirected graph of the B-graph do not have a pair of conflicting edges. □

6.6 Conclusion

We conclude this section with a simple algorithm to identify all binary logic programs that can be transformed using a finite sequence of valid transformations into an equivalent chain rule program. The input to the algorithm is the B-graph of a binary recursive rule and if there exists a chain rule equivalent to the given program, the algorithm outputs the modified B-graph; otherwise, the algorithm outputs "NO".

Algorithm Find-Equivalent-B-graph

INPUT : A B-graph of a binary rule with head predicate P(x, y).

OUTPUT : An equivalent B-graph under valid transformations if chain rule program equivalent to the original exists; and returns "NO" otherwise.

1. Let S be the set of connected components in B-graph.
   2.1 Set Okay true;
   2.2 *repeat*
      2.2.1 Set C to a component in S.
      2.2.2 *repeat*
         Set Changes to *false*;
         2.2.2.1 {Apply cut rules}
         While permitted, cut C through a vertex to form two components C_1 and C_2 and set Changes to *true*;Put C_2 into S and set C to be C_1.

2.2.2.2 {Apply intersection}
Remove all parallel edges from C through intersection transformation (and inverse transformation, if necessary) on C; If at least one parallel edge was removed set Changes to *true*.

2.2.2.3 {Apply composition}
Remove all vertices having degree two from C through composition transformation (and inverse transformation, if necessary) on C. If at least one composition is performed set Changes to *true*.

until *not*Changes.

2.2.3 *if* C is an undirected trail *then*
output C and remove C from S *else*
Output "NO" and set Okay *false*;

until S is empty *or* Okay is false.

3. end.

When the algorithm outputs the modified B-graph, it is trivial to determine whether or not the transformed program is regular. If a "NO" is output the complexity cannot be determined.

Chapter 7

CONCLUDING REMARKS

The focus of research reported in this thesis is on the efficient realization of inference engines that can process queries expressed using Horn Clause rules. Recently numerous strategies have been proposed addressing the issue of efficient query processing. The strategies based on rule rewriting attracted more researchers and among the rule rewriting methods both magic set (*MS*) and magic counting (*MC*) methods are the most significant. Agrawal-Devanbu (*AD*) method, which is a generalization of Aho and Ullman's algorithm of commuting selections with (Least Fixpoint) LFP operator works very well for a limited class of recursive rules. The integrated magic set (*IMS*) method described in Chapter 3 inherits the best of these two methods and improves the processing efficiency of Datalog programs by a significant factor. An important advantage of the *IMS*-method is its ability to do some global optimization independent of the given query. Further, if the architecture permits, one can perform this global optimization in parallel with magic set computation. This method can be viewed either as an improvement on the *MS*-method or as an integrated method involving *MS* and *AD* methods. A by-product is a classification of Datalog programs based solely on the nature of the rules and query form as applied to these processing strategies.

One of the means of achieving efficiency is to exploit the possible parallelism. Two parallel algorithms for efficient query evaluation of Datalog programs have been discussed in Chapter 4 and can be

implemented in any distributed architecture. We have also given expressions for the total time and response time using a simple model. This analysis clearly indicates the superiority of the parallel algorithms over the semi-naive serial algorithm. We have shown that the two parallel schemes are complementary in the sense that if the derived relation is expected to produce more new tuples at each iteration compared to the initial value, then the second algorithm is to be applied. In fact the cross over point occurs when the number of tuples produced is the same as the initial value of the derived predicate. We discussed the effect of pushing selections when the query has bound variables. The second algorithm is shown to be capable of pushing selections ahead of joins, which in turn further reduces the response time of a given query.

We have also presented a data dependent optimization strategy. It is worth noticing that a large class of Datalog programs including the "same generation program", can be evaluated under the proposed parallel processing scheme. A future research direction is to consider situations where the set of recursive rules as such cannot be vertically partitioned but it is possible to combine recursive rules into groups, say two, such that each group of rules can be vertically partitioned.

In Chapter 5 we introduced the concept of permutation dependency. Optimization techniques were presented for four different levels: (1) for storing a base predicate, (2) for removing redundant rules from a Datalog program, (3) for rewriting a Datalog program so that the resultant program is more suitable for magic set optimization and (4) for efficient computation of a derived predicate satisfying a permutation dependency.

Throughout our discussion, the well known same-generation program served as an example, which many researchers treat as a canonical example for linear Datalog programs. The classification of logic queries based on complexity measures is a difficult research issue. The well known same generation program is shown to be computationally more difficult than the transitive closure program and a simple algorithm to transform a binary logic program to a chain rule program is presented. These and other preliminary results provided in Chapter 6 are restrictive; yet they are significant. Their extensions to the full class of linear logic programs and non-linear logic programs seems to be promising future research topics.

References

[Afra86] F. Afrati, C. Papadimitriou, G. Papageorgiou, A. Roussou, Y. Sagiv and J. D. Ullman, *"Convergence of Sideways Query Evaluation"*, Proc. of the Fifth ACM SIGMOD-SIGACT Symp. on Principles of Database Systems, (March 1986).

[Agra88] R. Agrawal and P. Devanbu, *Moving Selections into Linear Least Fixpoint Queries*, Proceedings of the Fourth Int'l Conf. on Data Engineering, (Feb. 1988):452-461.

[AhoU79] A. Aho and J. D. Ullman, *"Universality of Data Retrieval Languages"*, Proceedings of the Sixth ACM Symposium on Programming Languages, San Antonio, Texas, Jan. 1979.

[Alag88] V. S. Alagar, P. Goyal, P. S. Nair and F. Sadri, *"Integrated Magic Set Method: A Rule Rewrite Scheme for Optimizing Linear Datalog Programs"*, to appear in Computer Journal.

[Alag89a] V. S. Alagar, P. Goyal, P. S. Nair and F. Sadri, *"Parallel Evaluation of Datalog Programs"*, submitted for publication, 1989.

[Alag89b] V. S. Alagar, P. Goyal, P. S. Nair and F. Sadri, *"Permutation Dependency in Datalog Programs"*, submitted for publication, 1989.

[Alag89c] V. S. Alagar, P. S. Nair and F. Sadri, *"Complexity of Datalog Programs"*, manuscript, 1989.

[Banc85] F. Bancilhon, *"Naive Evaluation of Recursively Defined Relations"*, in On Knowledge Base Management Systems - Integrating Database and AI Systems, Brodie and Mylopoulos, EDs., Springer-Verlag, 1985.

[Banc86a] F. Bancilhon, D. Maier, Y. Sagiv and J. Ullman, *"magic Sets and Other Strange Ways to Implement Logic Programs"*, Proc. of the Fifth ACM SIGMOD-SIGACT Symp. on Principles of Database Systems, (March 1986):1-15.

[Banc86b] F. Bancilhon and R. Ramakrishnan, *"An Amateur's Introduction to Recursive Query Processing Strategies"*, Proceedings of the 1986 ACM SIGMOD Int'l. Conference on Management of Data, (May 1986):16-52.

[Baro81] A. Baroody and D. DeWitt, *"An Object-Oriented Approach to Database System Implementation"*, ACM Transactions on Database Systems 6(4), Dec. 1981.

[Beer86] C. Beeri and R. Ramakrishnan, *"On the Power of Magic"*, Proc. of the Sixth ACM SIGART-SIGMOD-SIGACT Symp. on Principles of Database Systems, (March 1986):1-15.

[Bord84] M. Brodie, J. Mylopoulos and J. Schmidit (Eds), *"On Conceptual Modeling"*, Springer-Verlag, 1984.

[Brat84] K. Bratbergsengen, *"Hashing Methods and Relational Algebra Operators"*, Int'l. Conf. on Very Large Data Bases, Singapore, Aug. 1984.

[Care85] M. Carey and D. Dewitt, *"Extensible Database Systems"*, Proceedings of the Islamorada Workshop on Large Scale Knowledge Base and Reasoning Systems, Islamorada, Florida, Feb. 1985.

[Chan80] A. Chandra and D. Harel, *"Computable Queries for Relational Databases"* JCSS, Vol. 21, No.2, (Oct. 1980):156-178.

[Chan81] C. Chang, *"On the evaluation of queries containing derived relations in relational databases"*, in Advances in Data Base Theory, Vol. 1, H. Gallaire, J. Minker and J. M. Nicolas, Plenum Press, New York, (1981):235-260.

[Codd72] E. F. Codd, *"Relational Completeness of Database Sublanguages,"* in Database Systems, R. Rustin, ed., Prentice-Hall (1972):65-98.

[Cope84] G. Copeland and D. Maier, *"Making Smalltalk a Database System"*, Proceedings of the 1984 ACM SIGMOD Int'l. Conference on Management of Data, (May 1984).

[Fros86] R. Frost, *"Introduction to Knowledge Base Systems"*, Macmillan, Publishing Company, New York, 1986.

[Gall84] H. Gallaire, J. Minker and J. Nicolas, *Logic and Databases : A deductive Approach"*, Computing Survey 16(2), 1984.

[Gold83] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation"*, Addison-Wesley, 1983.

[HanH87] J. Han and J. L. Hanschen, *"Handling redundancy in the processing of recursive database queries"*, Proceedings of the 1987 ACM SIGMOD Int'l. Conference on Management of Data, (May 1986):73-81.

[Hens84] L. J. Henschen and S. Naqvi, *"On Compiling Queries in Recursive First-order Databases"*, JACM 31(1),(January 1984):47-85.

[Imme82] N. Immerman, *"Relational queries computable in polynomial time"*, Fourteenth ACM SIGACT Symposium, (May 1982):147-152.

[Ioan88] Y. E. Ioannidis and R. Ramakrishnan, *"Efficient Transitive Closure Algorithms"*, Proceedings of the Thirteenth Int'l Conf. on Very Large Data Bases; Los Angeles, 1988.

[Jaga87] H. V. Jagadish and R. Agrawal and L. Ness, *"A Study of Transitive Closure as a Recursion Mechanism"*, Proceedings of the 1987 ACM SIGMOD Int'l. Conference on Management of Data, (May 1987).

[Jark84] M. Jarke, J. Clifford and Y. Vassiliou, *"An Optimizing PROLOG Front-End to Relational Query System"*, Proceedings of the 1984 ACM SIGMOD Int'l. Conference on Management of Data, (May 1984).

[Kife88] M. Kifer and E. L. Lozinskii, *"Implementing Logic Programs as a Database System"*, Proceedings of the third int'l conf. on Data Engineering, (Feb. 1988):452-461.

[Kola88] P. G. Kolaitis and C. H. Papadimitriou, [A*"Why Not Negation by Fixpoint?"*, Proc. of the Sixth ACM SIGART-SIGMOD-SIGACT Symp. on Principles of Database Systems, (March 1988).

[Kowa87] R. Kowalski, F. Sadri and P. Soper, *"Integrity Checking in Deductive Database"*, Proceedings of the Thirteenth Int'l Conf. on Very Large Data Bases; Brighton, 1987.

[LeeH88] S. Lee and J. Han, *"Semantic Query Optimization in Recursive Databases"*, Proceedings of the third int'l conf. on Data Engineering, (Feb. 1988):452-461.

[Lozi86] E. L. Lozinski, *"A problem oriented inferential database systems"*, ACM Transactions on Database Systems 11(3), (September 1986):323-356.

[Marc87] A. Marchetti-Spaccamela, A. Pelaggi and D. Sacca, *"Worst-case complexity analysis of methods for logic query implementation"*, Proc. of the ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, (March 1987):294-301.

[Naug87] J.F. Naughton, *"One sided recursions"*, ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, (March 1987):340-348.

[Nejd87] W. Nejdl, *"Recursive Strategies for Answering Recursive Queries - The RQA/FQI Strategy"*, Proceedings of the Thirteenth Int'l Conf. on Very Large Data Bases; Brighton, 1987.

[Rama88] R. Ramakrishnan, C. Beeri and R. Krishnamurthy, *"Optimizing existential Datalog queries"*, Proc. of the ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, (March 1988):89-102.

[Reit78] R. Reiter, *Deductive Question Answering on Relational Data Base"*, in Logic and Data Bases, H. Gallaire and Minker, Eds., Plenum Press, New York, 1978, pp 149-177.

[Sacc87] D. Sacca and C. Zaniolo, *"Magic Counting Methods"*, Proceedings of the 1987 ACM SIGMOD Int'l. Conference on Management of Data, (May 1987):49-59.

[Sadr87] F. Sadri, *"Data Dependencies in Deductive Databases"* Concordia Univ. Tech. Rep. CSD-87-02, 1987.

[Sagi87] J. Sagiv, *"Optimizing Datalog Programs"*, Proc. of the ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, (March 1987).

[Shap80] S. Shapiro and D. McKay, *"Inference with recursive rules"*, Proc. first Annual National Conference on Artificial Intelligence, (August 1980):151-153.

[Smit84] J. Smith, *Expert Database Systems : A Database Perspective"*, Proceedings of First International Workshop on Expert Database Systems, Kiawah Islands, South Carolina, 1984.

[Ullm85] J. D. Ullman, *"Implementation of Logical Query Languages for Databases"*, ACM Trans. on Database Systems 10:3, pp. 289-321.

[Ullm89] J. D. Ullman, *"Bottom-up beats Top-Down for Datalog"*, Proc. of the ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, (March 1989).

[Vard82] M. Vardi, *"Complexity of relational query languages"*, Fourteenth ACM SIGACT Symposium on theory of computation, (May 1982):137-146.

[Vald88] P. Valduriez and S. Khoshafian, *"Parallel Evaluation of the Transitive Closure of a Database Relation,"* Int'l. J. of Parallel Programming, 17(1), (January 1988):19-42.

[Warr84] D. Warren and E. Sciore, *"Towards an Integrated Database-PROLOG System"*, Proceedings of First International Workshop on Expert Database Systems, Kiawah Islands, South Carolina, 1984.

[Wolf88] O. Wolfson and A. Silberschartz, *"Distributed Processing of Logic Programs"*, Proceedings of the 1988 ACM SIGMOD Int'l. Conference on Management of Data, (June 1988):329-336.

[Zani84] C. Zaniolo, *"Prolog : A Database Query Language For All Seasons"*, Proceedings of First International Workshop on Expert Database Systems, Kiawah Islands, South Carolina, 1984.

[Zani86] C. Zaniolo, *"Safety and Compilation of Non-Recursive Horn Clauses"*, Proceedings of First International Conference on Expert Database Systems, Charleston, 1986.

[Szym77] T. G. Szymanski and J. D. Ullman, *"Evaluating Relational Expressions with Dense and Sparse Arguments"*, SIAM J. COMPUT., Vol. 6, No. 1, (March 1977),109-122.