University of Cape Town

Department of Computer Science

An Empirical Study of Negation in Datalog Programs

By

Tian Xiao Jun

A thesis
Prepared Under the Supervision of
Associate Professor P.T. Wood
In Fulfilment of the Requirements for the
Degree of Master of Science in
Computer Science

September 1996

Abstract

Datalog is the fusion of prolog and database technologies aimed at producing an efficient, logic-based, declarative language for databases. Since negation was added to Datalog, Datalog has become more expressive.

In this thesis, I focus my attention on adding negation to DatalogIC which is a language which has been implemented by Mark P. Wassell, a past MSc student in the Department of Computer Science at UCT. I analyse and compare stratified, well-founded and inflationary semantics for negation, each of which has been implemented on top of INFORMIX; we call the resulting system NDatalog. According to the test results, we find that some results are unexpected. For example, when we evaluate a recursive stratified program, the results show that $\mathrm{NDatalog}_{\mathbf{stra}}$ is slower than $\mathrm{NDatalog}_{\mathbf{wellf}}$ although $\mathrm{NDatalog}_{\mathbf{wellf}}$ is more complex. After further investigation, I find the problem is that the NDatalog system has to spend a lot of time imitating the MINUS function, which does not exist in INFORMIX-SQL. So the running time depends on what kind of database system is used as backend. When we consider the time spent on pure evaluation, excluding auxiliary functions, we find that the results support our expectations, namely, that $\mathrm{NDatalog}_{\mathbf{stra}}$ is faster than $\mathrm{NDatalog}_{\mathbf{wellf}}$ which is faster than $\mathrm{NDatalog}_{\mathbf{infl}}$.

Acknowledgements

Many thanks to my supervisor, Professor Peter Wood, for his assistance and excellent guidance through the various stages of this work. I would also like to thank Professor Pieter Kritzinger and all of the staff in the Computer Science Department here at UCT for their kindly help and wish them well for the future.

I wish to thank my wife Li Xiao-liang, my parents and my brother for their continual enthusiasm and interest in my work.

Contents

Chapter 1

Introduction

The fields of deductive databases and logic programming are intimately related. Datalog is a rule-based language that integrates logic programming and deductive databases. In order to develop extensions of Datalog, much research has been done during the last decade. One extension is the DatalogIC language which has been implemented by Wassell at the University of Cape Town [Was90]. As with other Datalog languages, DatalogIC programs allow one to define the transitive closure of a relation. For example, the transitive closure of *arc* can be computed as follows:

$$\begin{array}{ll} r_1: & path(X,Y) : -arc(X,Y). \\ r_2: & path(X,Y) : -path(X,Z), arc(Z,Y). \end{array}$$

This query cannot be expressed in relational algebra or in SQL which is the most popular relational query language. On the other hand, a simple relation like the complement of one relation with respect to another one cannot be expressed in DatalogIC, although this is definable by the relational algebra. For example, we may express *vegetarian* in relational algebra as:

$$vegetarian(X) = person(X) - eat\_meat(X)$$

which means that X is vegetarian if X is a person and X does not eat meat.

Unfortunately, there are frequent situations where we would like to use the negation of a predicate to help express a relationship by logical rules. In order to overcome this deficiency, it is necessary to introduce negation in logic programs and Datalog.

Early theories about negation in logic programs were proposed by Reiter (*the Closed World Assumption*) [Rei78], and by Clark (*Negation as Failure*) [Cla78]. A survey of treatments of negation in logic programming was done by Shepherdson [She88].

In the past few years, much more research has been devoted to incorporating negation in deductive databases and logic programs. For instance, Chandra and Harel first proposed a semantics for stratified logic programs with negation in [CH82]. Roughly speaking, a logic program is stratified if all mutually recursive predicates depend positively on one another. Let us see a program as follows:

$$\begin{array}{lll} r_1: & bachelor(X) & :-male(X),\ not\ husband(X). \\ r_2: & husband(X) & :-married(X,Y). \end{array}$$

Obviously, the program is stratified because there are no mutually recursive predicates dependent negatively on one another. However not every logic program with negation is stratified, let us consider the next program:

$$win(X) :-move(X,Y),\ not\ win(Y).$$

Since the recursive predicate *win* depends negatively on itself, the program is not stratified. So Gurevich and Shelah [GS86b] investigate an inflationary model semantics of logic programs with negation which leaves the programmer totally free to write any program he or she wants. But, for a given query, the answers given by the inflationary semantics and by the stratified semantics may differ. Fortunately, under these circumstances, a simple modification of the original program will ensure that the same answer is obtained under the two semantic structures. This will be discussed in greater detail in chapters 2 and 3. We also investigate another semantics which enables a programmer to write any program—the well-founded model semantics. This semantics, proposed in [GRS88], guarantees us the same answer as the stratified semantics when the logic programs are stratified. More detail will be discussed in chapters 2 and 4.

Other semantics that have been proposed include the perfect model [Prz88], stable model [GL88], and default model [BF87] semantics, but we do not consider them in this thesis. Bidoit [Bid91] surveys and compares different techniques to integrate negation in rule-based query languages, and surveys the problem of defining the declarative semantics of logic programs with negation.

This thesis is an attempt to survey and compare the major solutions of the current ideas on semantic models with negation in rule-based query languages, and then synthesize them into the DatalogIC system. Moreover, we explore which semantics with negation is suited for a Datalog system including negation from two aspects: efficiency and expressive power.

1.1 Organization of the thesis

The thesis consists of a further four chapters:

Chapter 2 first gives a brief presentation of the basic concepts and notation of first order logic [Llo87] and presents its syntax and semantics. Then Datalog is introduced through its proof theory and model theory [Ull88] [CGT89]. I describe the rationale behind the development of Datalog, and discuss why negation is needed. I give an introduction to the three major semantics: stratified, inflationary and well-founded ~~model~~ semantics which involve negation and show some of the problems that arise when negation is introduced [CH85] [GS86a] [GRS88] [Bid91].

Chapter 3 describes a new system, NDatalog which is based on DatalogIC [Was90]. The NDatalog system was written in the C programming language, on the UNIX[1] operating system, using the INFORMIX[2] database management system as the backend. I give an overview of the system and then discuss the user-interface module and the evaluation algorithms used. The latter convert a Datalog program into SQL[3] statements and interact with INFORMIX. I analyse those semantics which involve negation that I mentioned in Chapter 2 and describe their implementation in the NDatalog system. Some major algorithms, namely, Semi-naive, Stratified, Well-founded and Inflationary, are given in this chapter.

In Chapter 4, I focus on the efficiency of the different semantics. I briefly analyse and compare those semantics that I have implemented in the NDatalog system through some specific examples. In order to demonstrate that our system produces correct answers on logic programs, we compare our results with the XSB system[4]. We first divide logic programs into stratified and nonstratified logic programs, and then test those programs dealing with recursive and nonrecursive rules. The testing shows that the efficiency of the various semantics is different. For instance, when we compute a transitive closure program (Example 4.1), the speed of the well-founded semantics is faster than the stratified semantics on large databases and the stratified semantics is faster than the inflationary, but XSB is much faster than all of them. This is unexpected because the stratified semantics should be faster than the well-founded semantics in this case. The reasons for this are discussed in Chapter 4. However, the stratified semantics is much faster than others when we evaluate nonrecursive rules with negation (see Example 4.2). Finally, when we compute a nonstratified recursive program, our implementation of the well-founded semantics and inflationary semantics is much faster than XSB (see Example 4.5 for more details).

[1]UNIX is a trademark of AT&T.

[2]INFORMIX is a registered trademark of Informix Software, Inc.

[3]Structured Query Language.

[4]XSB is a logic programming system developed at the Department of Computer Science, SUNY at Stony Brook, USA.

Chapter 5 concludes the dissertation and mentions further work. The data structures and the standard algorithms used in the implementation are listed in Appendix A.

Chapter 2

Background

The aim of this chapter is to introduce some well-known concepts and recall recent developments in logic programs with negation. We will discuss various alternative semantics with negation that have been proposed.

2.1 Logic programs

We begin by reviewing some well-known concepts of *first order logic* (FOL) and *logic programming* (LP). The main notation used throughout the thesis is presented in this section.

FOL has two aspects: *syntax* and *semantics*. The *syntax* of FOL should be computable. That is, at least in theory, an automatic proof procedure should exist. In other words, it is concerned with well-formed formulas admitted by the grammar of a formal language, as well as deeper proof-theoretic issues. The *semantics* of FOL should be clear and easily intelligible. That is, the programmer should be able to understand the full meaning of what he or she writes. In other words, it is concerned with the meanings attached to the well-formed formulas and the symbols they contain. More details can be found in [Llo87] [Bid91].

2.1.1 Syntax

In what follows, I will be giving some definitions of a first order theory, such as the alphabet of FOL, a first order language, formulas, and Horn clauses.

Definition: 2.1 A first order logic *alphabet* consists of six classes of symbols:

1. *Variables:* words beginning with an uppercase letter, i.e., X, Y, Z.
2. *Constants:* words beginning with a lowercase letter, i.e., a, b, c.
3. *Predicate Symbols:* also words beginning with a lowercase letter, but we normally use a letter from the middle of the alphabet, i.e., p, q, r.
4. *Connectives:* the symbols are $\{\leftarrow, \wedge, \vee, \neg\}$.
5. *Quantifiers:* universal quantification, denoted ($\forall$), and existential quantification, denoted ($\exists$).
6. *Punctuation Symbols:* ( ) , □

Definition: 2.2 A logic programming *alphabet* consists of five classes of symbols:

1. *Variables:* words beginning with an uppercase letter, i.e., X, Y, Z.
2. *Constants:* words beginning with a lowercase letter, i.e., a, b, c.
3. *Predicate Symbols:* also words beginning with a lowercase letter, but we normally use a letter from the middle of the alphabet, i.e., p, q, r.
4. *Connectives:* the symbols are {:-, ",", not}.
5. *Punctuation Symbols:* ( ) , □

Definition: 2.3 A *term*[1] is either a variable or a constant. □

Definition: 2.4 A *formula* is defined inductively as follows:

1. If p is an n-ary predicate and $t_1, \ldots, t_n$ are terms, then $p(t_1, \ldots, t_n)$ is formula (called an *atomic formula* or, more simply, an *atom*).
2. If F and G are formulas, then so are $\neg F$, $F \wedge G$, $F \vee G$ and $G \leftarrow F$.
3. If F is a formula and X is a variable, then $(\forall X\ F)$ and $(\exists X\ F)$ are formulas.□

Definition: 2.5 The *first order language* given by a first order logic alphabet is the set of all formulas constructed from the symbols of the alphabet [Llo87]. □

[1] We are not concerned with function symbols in this thesis.

The following example describes a formula in a first order language and its meaning.

Example: 2.1 In the formula $\forall X(p(X) \leftarrow q(X) \wedge \neg r(X))$, X is a variable, and p, q and r are predicate symbols. □

Definition: 2.6 A *literal* is either an atom or the negation of an atom. A *positive literal* is an atom. A *negative literal* is the negation of an atom. A *clause* is a formula of the form $\forall X_1, \ldots, X_s(B_1 \vee \ldots \vee B_m)$ where each B_i is a literal and $X_1 \ldots X_s$ are all the variables occurring in $B_1 \vee \ldots \vee B_m$ [Llo87]. A *Horn clause* is a clause with at most one positive literal. We write a Horn clause as follows:

$$\neg L_1 \vee \ldots \vee \neg L_n \vee L_0 \tag{2.1}$$

which is logically equivalent to

$$L_0 \leftarrow L_1 \wedge \ldots \wedge L_n \tag{2.2}$$

A *rule* is of the form (2.2), where L_0 is the *head* of the rule (at the left-hand side of the symbol "$\leftarrow$"), and each L_i $(1 \leq i \leq n)$ is a *subgoal*. The subgoals together are called the *body* of the rule (at the right-hand side of the symbol "$\leftarrow$"). The body is possibly empty, a rule with an empty body being called a *fact*. If L_0 is the only positive literal, then it is called a *definite clause*; if L_0 is empty , $\leftarrow L_1, \ldots, L_n$, then it is called a *definite goal*.

We shall follow the logic programming style for expressing Horn clauses, using:

$$L_0 : -L_1, \ldots, L_n. \tag{2.3}$$

for the Horn clause in (2.2). □

Definition: 2.7 A literal, term, fact, clause, or rule containing no variable symbols is called *ground*. □

Definition: 2.8 A *substitution θ* is a finite set of the form $\{X_1/t_1, \ldots, X_n/t_n\}$, where each X_i is a distinct variable and each t_i is a term, such that $X_i \neq t_i$. □

As we know, in the real world, we cannot avoid comparisons. For instance, "John is older than Tom"; "6 is not equal to 5"; etc. In order to express these facts, we need to define some comparison operators. We call these operators *built-in predicates* (or, *evaluable predicates*):

$$\{>, \geq, =, \neq, \leq, <\}$$

We can use these operators to denote the above examples, i.e., $age(john) > age(tom)$, $6 \neq 5$. Useful properties of these operators are detailed in [Was90].

A *logic program* (or *logic database*) consists of a finite set of facts and rules. Facts are assertions about a relevant piece of the world, such as: "John is the brother of Tom". This is written $brother(john, tom)$. Rules are sentences which allow us to deduce facts from other facts. An example of a rule is "If X is a parent of Y and Y is a parent of Z, then X is a grandparent of Z". This is written

$$grandparent(X, Z) : -parent(X, Y), parent(Y, Z).$$

2.1.2 Semantics

The semantics of a logic program is usually defined by means of particular model of the first order logic notation of the program. We recall below some well-known notions used to define the semantics of first order logic. The presentation essentially concentrates on Herbrand interpretations [Bid91].

Definition: 2.9 Let L be a first order language. The *Herbrand universe* H_U (or *Herbrand domain*) for the language L is the set of all ground terms[2] which can be formed out of the constants in L. If there are no constants in L, add an arbitrary constant, say a.□

Definition: 2.10 The *Herbrand base* H_B for L is the set of all ground atoms (or atomic formulas) which can be formed by using predicates symbols from L with ground terms from the Herbrand universe H_U as terms (or arguments). □

Definition: 2.11 A *Herbrand interpretation* H_I for L is an assignment of values to the constants and predicates of L defined as follows:

1. Constants in L are assigned to themselves in H_U.
2. Each n-ary predicate symbol in L is assigned a mapping from $H_U \times \ldots \times H_U$ (n times) into { True, False }. □

Definition: 2.12 Let H_I be an interpretation of a first order language L. A *variable assignment* (with respect to H_I) is an assignment to each variable in L of an element in the H_U. □

[2]In our case, there are just constants.

Definition: 2.13 Let H_I be a Herbrand interpretation of a first order language L with no function symbols and let A be a variable assignment. The *term assignment* (with respect to H_I and A) of the terms in L is defined as follows:

1. Each variable is given its assignment according to A.
2. Each constant is given its assignment according to H_I. □

Definition: 2.14 Let H_I be a Herbrand interpretation of a first order language L with no function symbols and let A be a variable assignment. A formula in L can be given a *truth value*, true or false, (with respect to H_I and A) as follows:

1. If the formula is an atom $p(t_1, \ldots, t_n)$, then the truth value is obtained by calculating the value of $p'(t'_1, \ldots, t'_n)$, where p' is the mapping assigned to p by H_I and $t'_a, \ldots, t'_n$ are the term assignments of $t_1, \ldots, t_n$ with respect to H_I and A.
2. If the formula has the form $\neg F$, $F \wedge G$, $F \vee G$ or $G \leftarrow F$, then the truth value of the formula is given by the following Table 2.1:

F	G	$\neg$ F	F $\wedge$ G	F $\vee$ G	G $\leftarrow$ F
true	true	false	true	true	true
true	false	false	false	true	false
false	true	true	false	true	true
false	false	true	false	false	true

Table 2.1: The values of the formula.

3. If the formula has the form $\exists x\ F$, then the truth value of the formula is true if there exists $d \in H_U$ such that F has truth value true with respect to H_I and $A(x/d)$, where $A(x/d)$ is A except that x is assigned d; otherwise, its truth value is false.
4. If the formula has the form $\forall x\ F$, then the truth value of the formula is true if, for all $d \in H_U$, we have that F has truth value true with respect to H_I and $A(x/d)$; otherwise, its truth value is false. □

Example: 2.2 The informal semantics of $\forall X(p(X) \leftarrow q(X) \wedge \neg r(X))$ is " for every X, if $q(X)$ is true and $r(X)$ is false, then $p(X)$ is true". □

In a clause, every variable lies within the scope of an implicit universal quantifier. Therefore the truth value of a clause is independent of any variable assignment and we can speak unambiguously of the truth value of a clause with respect to a Herbrand interpretation.

Definition: 2.15 Let $\mathcal{F}$ be a set of formulas in L. A *Herbrand model* H_M for $\mathcal{F}$ is a Herbrand interpretation of L that makes all the clauses of $\mathcal{F}$ true. We say that a Herbrand model H is contained in another Herbrand model H_M if $H \subseteq H_M$. If a Herbrand model for $\mathcal{F}$ is contained in every other Herbrand model for $\mathcal{F}$, then it is called the *least Herbrand model* (H_m) of $\mathcal{F}$. □

Example: 2.3 In order to explain the above clearly, we give an example as follows. Consider a program P:

(a) $p(1)$.

(b) $q(2)$.

(c) $r(3)$.

(d) $p(X) : -q(X)$.

(e) $q(X) : -r(X)$.

The *Herbrand universe* H_U is $\{1,2,3\}$.
The *Herbrand base* H_B is $\{ p(1), p(2), p(3), q(1), q(2), q(3), r(1), r(2), r(3)\}$.
A *Herbrand model* H_M for P is $\{ p(1), p(2), p(3), q(1), q(2), q(3), r(2), r(3)\}$.
The *least Herbrand model* H_m for P is $\{ p(1), p(2), p(3), q(2), q(3), r(3)\}$.
□

2.2 Datalog

2.2.1 What is Datalog ?

The term "Datalog" was first used by Maier and Warren [MW88] in order to develop a more powerful query language for relational databases. In fact, Datalog uses a syntactic subset of logic programs which does not allow function symbols as arguments of predicates. While the meaning of Datalog programs and logic programs is the same, how such programs are evaluated is different. Logic programs are evaluated top-down (as in Prolog, for example), and Datalog programs are evaluated bottom-up. There are other differences; see [Ull88]

[CGT90] for more details.

Datalog is the fusion of first order logic syntax and database theory aimed at producing a new logic-based data model and a logic-based language in the deductive database field (Horn clauses with no function symbols). It is a database query language based on the logic programming paradigm.

From the above description of Datalog, we know Datalog is a kind of database query language containing first order logic language properties. So all the concepts we gave in the previous section are suited to Datalog. Next we consider some definitions not covered in the previous section.

Definition: 2.16 For the Datalog program P,

(a) $link(a, b)$.

(b) $link(a, c)$.

(c) $link(b, d)$.

(d) $link(d, e)$.

(e) $path(X, Y) : -link(X, Y)$.

(f) $path(X, Y) : -path(X, Z), link(Z, Y)$.

(g) $? - path(?X, ?Y)$.

(a)—(d) are the set of ground facts called the *Extensional Database* (EDB). Predicates appearing in the EDB are called EDB *predicates* , which may only occur in rule bodies. Rules (e)— (f) are the set of rules called the *Intensional Database* (IDB) and predicates appearing in the head of IDB rules are called IDB *predicates*. Of course, we should distinguish *built-in predicates* from IDB *predicates*. Rule (g) is a *goal*, a *query form* that is a conjunction of predicates written as:

$$? - q_1(l_{11}X_{11}, \ldots, l_{1m_1}X_{1m_1}), \ldots, q_n(l_{n1}X_{n1}, \ldots, l_{nm_n}X_{nm_n}).$$

where the label l_{ij} attached to the variable X_{ij} is empty, ? or !. This indicates whether the variable is an existential, answer or an input variable, respectively [CGT89] [Was90]. □

Intuitively, the above shows that a Datalog program consists of ground facts, Datalog rules and query forms. From here on, EDB predicates and IDB predicates will be distinguished.

2.2.2 Safety of Datalog

When we interpret a rule, moreover, we cannot ignore the fact that certain rules can produce infinite answer relations. For example, the rules:

$$biggerThan(X, Y) :- X > Y. \tag{2.4}$$

$$favourite\_sport(X, Y) :- jump(Y). \tag{2.5}$$

Rule (2.4) defines an infinite relation if X and Y are allowed to range over the integers or any infinite set. Assume *jump* denotes a set of sports involving jumping, for example, $jump(longjump), jump(highjump)$ and $jump(triplejump)$. Rule (2.5) also defines an infinite set of pairs $favourite\_sport(X, Y)$, because the rule means for all X, X's favourite sport is jumping, where X can range over an infinite set.

From the above examples, we know that there are two sources of infiniteness. One is a variable that appears only in a built-in predicate, as in (2.4). The other is a variable that appears only in the head of a rule, as in (2.5). In order to avoid rules that create infinite relations appearing in the program, we can insist that each rule is *safe*. One way of doing this is to ensure that each variable appearing in the rule is *limited* [Ull88].

Definition: 2.17 We say a rule is *safe* if all its variables are limited[3]. This means that the variables appearing in the head must appear in the body, and variables appearing in built-in predicates must appear in other non-built-in predicates or be connected by a chain of = predicates to a limited variable or to a constant. □

This is a very important definition and will be used throughout the thesis. When negation is introduced to the body of rules in the logic program, the definition of safety will be stronger. We will cover this in chapter 3.

In addition, rules having no multiple occurrences of a variable or constants in the head will ease the evaluation of multiple rules in a predicate definition. Ullman introduces a process to achieve this, called *rectification* [Ull88]. In outline, rectification involves the following:

- For every constant "a" in the head, we replace it in the head by a new and distinct variable "X_a" and add "$X_a = a$" to the body.
- For every repeated variable "X" in the head, we replace it's ith occurrence by "X_i" and add "$X = X_i$" to the body.

[3]There is a different definition of safety in [BR86], namely, that a program is safe when it is guaranteed not to produce infinite answer relations.

Definition: 2.18 A *fully rectified* rule is one with no multiple occurrences of variables, and no constants in the head or the non-evaluable predicates of the body. This implies that all multiple occurrences of variables and constants will appear only in the evaluable predicates.

From here on, the rules which we discuss will be treated as fully rectified rules. The rectification algorithm is given in Appendix A.2.1. In order to study Datalog programs, we will briefly discuss the proof theory and model theory of Datalog.

2.2.3 Proof theory of Datalog

Proof theory is concerned with the analysis of logical inference [CGT89]. That is, new Datalog facts can be proved using the Datalog rules in all possible ways from given Datalog facts. Consider a Datalog rule R of the form $L_0 : -L_1, \ldots, L_n$ and a list of ground facts $F_1, \ldots, F_n$. If a substitution θ exists such that, for each $1 \leq i \leq n$, $L_i\theta = F_i$, then, from rule R and from the facts $F_1, \ldots, F_n$, we can *infer* in one step the fact $L_0\theta$. The inferred fact may be either a new fact or it may be already known. What we have just described is a general inference rule, which produces new Datalog facts from given Datalog rules and facts. We refer to this rule as *Elementary Production* (EP). In some sense, EP can be considered a *meta-rule*, since it is independent of any particular Datalog rules, and it treats them just as syntactic entities.

Example: 2.4 Consider the Datalog rule $R : p(X,Y) : -p(X,Z), p(Z,Y)$ and the ground facts $\{p(a,b), p(b,c)\}$. Then, by EP, we can infer in one step the fact $\{p(a,c)\}$ using the substitution $\theta = \{X/a, Y/c, Z/b\}$. This is a new fact. If we consider the Datalog rule $R' : p(X,Y) : -p(Y,X)$ and the fact $\{p(a,a)\}$. We only can infer the fact itself by applying EP. □

Let us finally define the concept of *inferred ground fact*. Let C be a set of Datalog clauses. Informally, a ground fact F can be *inferred from* C, denoted by $C \vdash F$, iff either $F \in C$ or F can be obtained by applying the inference rule EP a finite number of times. The relationship "$\vdash$" is more precisely defined by the following recursive rules:

- $C \vdash F$ if $F \in C$.
- $C \vdash F$ if a rule $R \in C$ and ground facts $F_1, \ldots, F_n$ exist such that $\forall 1 \leq i \leq n$, $C \vdash F_i$ and F can be inferred in one step by the application of EP to R and $F_1, \ldots, F_n$.

The sequence of applications of EP which is used to infer a ground fact F from C is called a *proof* of F. Any proof can be represented as a *proof tree* with different levels and with the derived fact F at the top. The proof theory of Datalog is sound and complete [CGT90].

2.2.4 Model theory of Datalog

Some concepts of model theory were introduced earlier. Datalog can be described very easily in terms of model theory. A Datalog rule can be interpreted in several different ways. A rule may be true under a certain interpretation in which case we say the interpretation satisfies the rule and false under another one. A fact F follows logically from a set of clauses C iff each interpretation satisfying every clause of C also satisfies F. If F follows from C, we write $C \models F$.

Example: 2.5 Consider a set C consisting of the clauses $C_1 : p(X, Y) : -p(Y, X)$ and $C_2 : \{p(a, b)\}$ and a fact $F : \{p(b, a)\}$. Clearly, for each possible interpretation of our constant and predicate symbols, whenever C_1 and C_2 are satisfied then F is also satisfied, hence, $C \models F$. □

The set $cons(C)$ of all consequence facts of a set C of Datalog clauses can thus be characterized as follows: The set $cons(C)$ is the set of all ground facts in the Herbrand base which satisfy each Herbrand model of C. $cons(C)$ is a Herbrand model of C for each set C of Datalog clauses. If $cons(C)$ is a subset of every other Herbrand model of C, we call $cons(C)$ the least Herbrand model (H_m) of C [CGT90].

2.2.5 The least fixpoint semantics

Associated with every definite program is a monotonic mapping which plays a role in the theory. This section introduces some concepts that are useful. Datalog is viewed as a set of rules and facts together with some basic operations for applying rules to facts in order to generate new facts. Thus we can associate a mapping T with a Datalog program. The fixed point semantics of Datalog is given by means of the facts obtained by iterative application of the rules of the program to the facts, starting with an empty set of facts (as extensively discussed in [Llo87] [Ull88] [Bid91]).

Definition: 2.19 A relation R on a set O is a *partial order* if the following conditions are satisfied:

1. xRx, for all $x \in O$.
2. xRy and yRx imply $x = y$, for all $x, y \in O$.
3. xRy and yRz imply xRz, for all $x, y, z \in O$. □

Definition: 2.20 Let S be a complete lattice and $T : S \rightarrow S$ be a mapping. There is a partial order on S, denoted by "$\leq$". We say that the mapping T is *monotonic* iff $s \leq s'$ entails $T(s) \leq T(s')$ for each pair s and s' in S. Let $s \in S$, we say s is a fixpoint of T iff $T(s) = s$ [Llo87].□

Definition: 2.21 A *fixed point* of a Datalog program with respect to a set of relations for the EDB predicates, say $R_1, \ldots, R_n$, is a *solution* for the relations corresponding to the IDB predicates. □

A fixed point with respect to $R_1, \ldots, R_n$, together with those relations, forms a model M of the Datalog program. We know the model M is not unique, so we shall continue to be interested primarily in fixed points and models that are *minimal.*

Definition: 2.22 A fixed point s of T is a *minimal fixed point* iff there is no other fixed point $s' \in S$ such that $s' \leq s$. □

Definition: 2.23 A fixed point s of T is a *least fixed point*, denoted $lfp(T)$, iff for any fixed point $s' \in S$, we have $s \leq s'$. □

Notice that if there is a *least fixed point*, then that is the only *minimal fixed point.* An alternative semantics, known as the *least fixed point semantics* of a Datalog query, can also be given by defining a fixed point operator and taking the least fixed point as the result of the query.

Up to now we have introduced a number of concepts about Datalog. During the last decade, much research has been done in order to develop extensions of Datalog, especially, negation has been introduced into the bodies of Datalog rules. Technically, rules with negated subgoals are not Horn clauses, but we shall see that the use of negated subgoals will increase the expressive power of Datalog programming. Negation in Datalog programming is the most important part of this thesis.

2.3 Evaluation Methods

In this section, we present methods for evaluating a Datalog program, namely, for generating the actual set of tuples which satisfy a given user's goal for a given set of Datalog rules.

2.3.1 Rule/Goal Graphs

As in many areas of computer science and other disciplines, graph theoretic tools play an importa. role also in databases [Yan90]. In this section, we focus our attention on some definitions which are useful to our system. As with all disciplines where objects are studied, it helps to classify the Datalog$^\neg$ programs[4]. With Datalog$^\neg$, the principle division is into recursive and non-recursive programs, while the primary investigation tool is the graph. The benefit of dividing programs into classes has been discussed in [BR86] [Ull88] [Was90]. The most common graphs used are the *rule/goal graph* and *reduced rule/goal graph* [BR86].

Definition: 2.24 A *rule/goal graph* is a graph that has two sets of nodes: one consists of square nodes which are associated with predicates, the other consists of oval nodes which are associated with rules. There is an arc from a predicate node to a rule node if the predicate appears in the body of the rule and we label the arc "$\neg$" if the predicate appears negatively in the body of the rule. There is also an arc from a rule node to a predicate node if the name of the head of the rule is the same as the predicate node's. □

Example: 2.6 Consider a Datalog$^\neg$ program P, where $c(X)$, $s(X)$, and $t(X,Y)$ are EDB predicates.

$$\begin{array}{lll}
r_1: & a(X) & :- \ c(X),\ not\ b(X). \\
r_2: & b(X) & :- \ not\ a(X). \\
r_3: & p(X) & :- \ q(X),\ not\ r(X). \\
r_4: & p(X) & :- \ r(X),\ not\ s(X). \\
r_5: & p(X) & :- \ t(X,Y). \\
r_6: & q(X) & :- \ p(X). \\
r_7: & r(X) & :- \ q(X). \\
r_8: & r(X) & :- \ not\ c(X).\square
\end{array}$$

From Definition 2.24 we get the rule/goal graph shown in Figure 2.1.

Definition: 2.25 Let $G = (V, E)$, where V are called vertices, and E are called edges, be a digraph (short for "directed graph"). A digraph G is *strongly connected* if, for each pair of vertices A and B, there is a path from A to B (and hence by interchanging the roles of A and B in the definition, there is a path from B to A as well). A *strongly connected component* (SCC) of a digraph is a maximal strongly connected subgraph.□

[4]We will introduce negation in Section 2.4

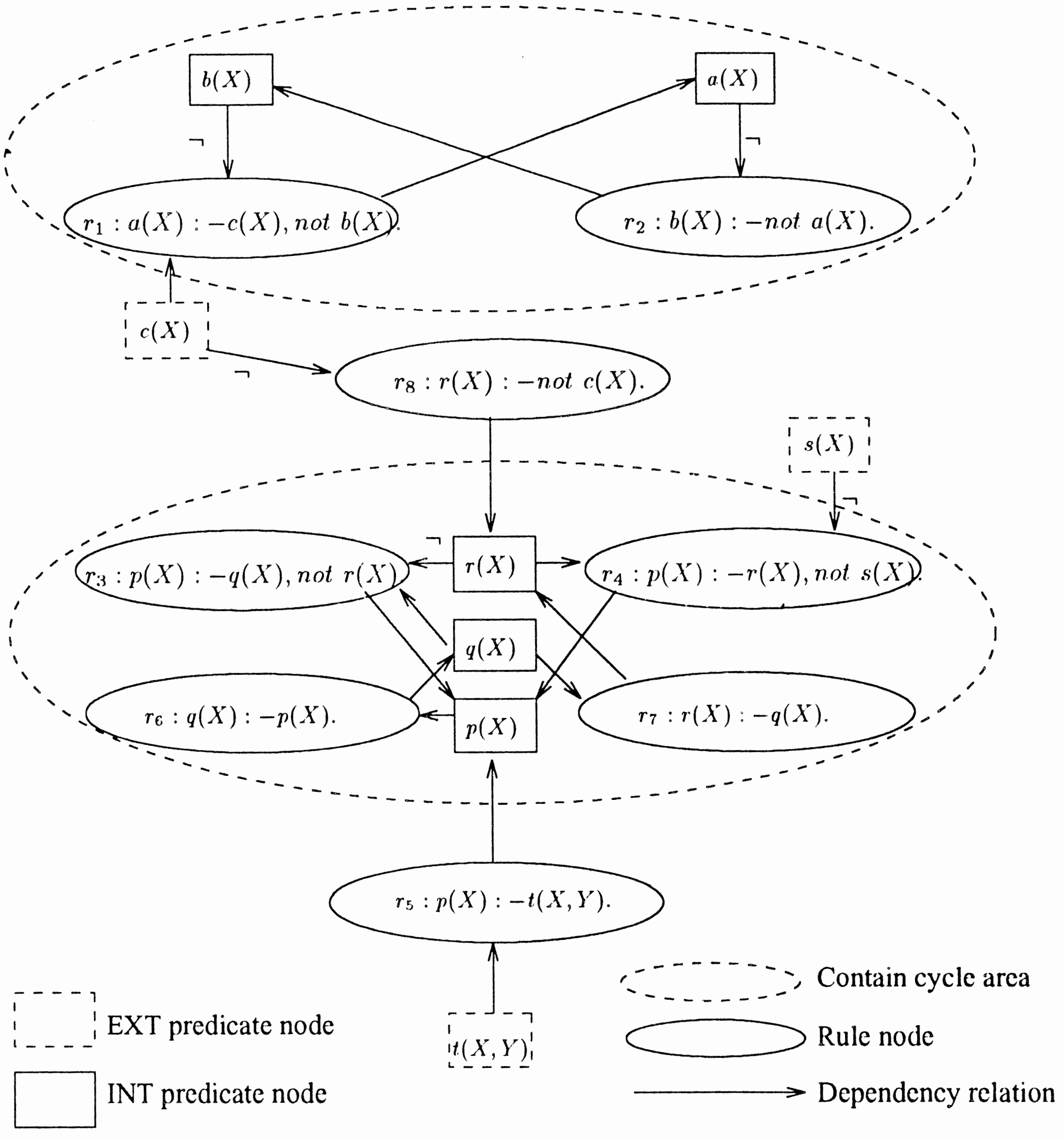


Figure 2.1: Rule/Goal Graph

Definition: 2.26 A *reduced rule/goal graph* is a variety of rule/goal graph which groups all nodes which are mutually recursive, the parts in dotted ovals in Figure 2.1, into a single node. In graph-theoretic terms we identify the *strongly connected components* of the rule/goal graph and form the acyclic condensation of the graph. □

According to Definition 2.26, we can easily draw up a *reduced rule/goal graph* in terms of Example 2.6 from Figure 2.1. This is shown in Figure 2.2.

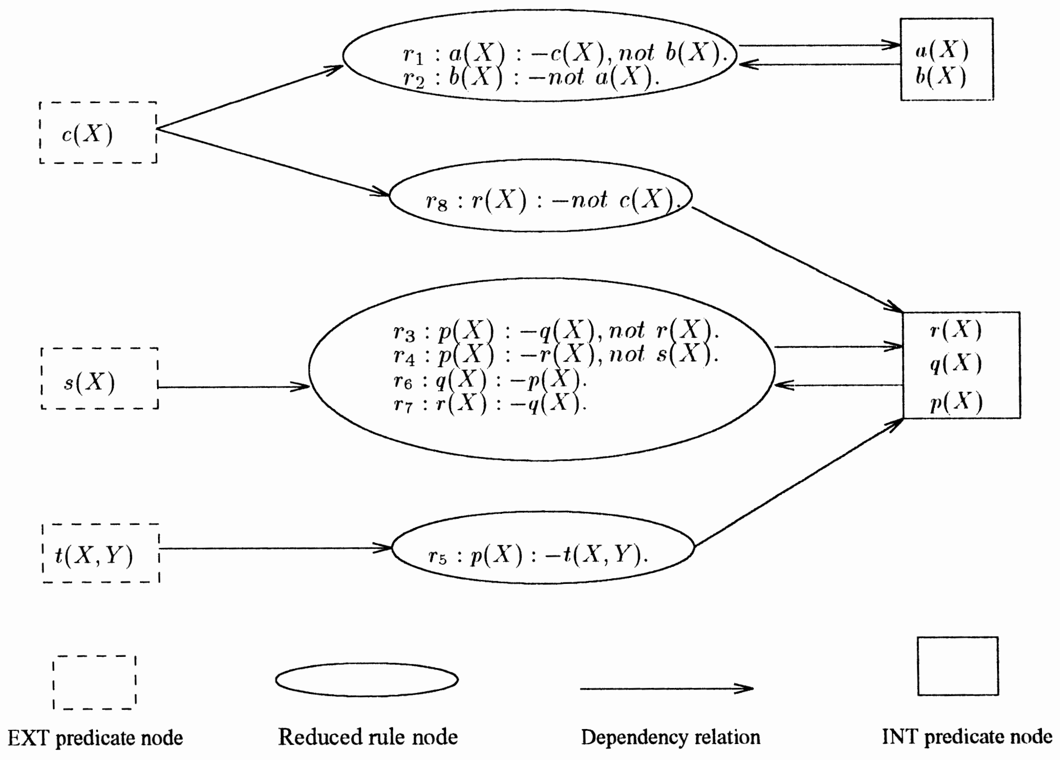
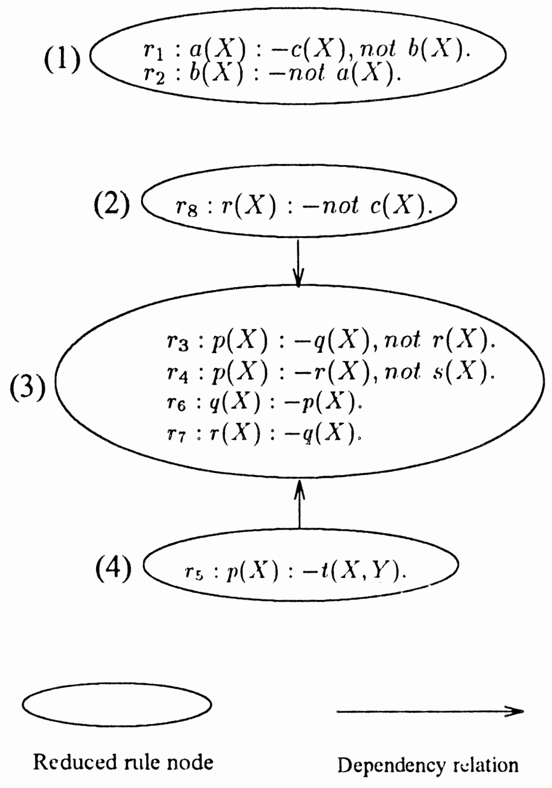


Figure 2.2: Reduced Rule/Goal Graph

The most important aspect of the rule/goal graph is the grouping of predicate and rule nodes. Intuitively, this grouping provides an order of computing. We can group definition (or goal) and rule predicate nodes in many ways, but different ways will directly influence the efficiency of the computation.
We were concerned with SCCs in Definition 2.26. In Figure 2.2, the ovals are SCCs[5]. According to the definition, SCCs give us a method to group the rule nodes. The algorithm then evaluates the SCCs in *topological order*, so that, if an SCC s_1 contains a rule which is dependent on some rule in an SCC s_2, the s_2 SCC will be evaluated first.

[5]The algorithm is shown in Appendix A.2

Example: 2.7 Consider Example 2.6 from Figure 2.1. Figure 2.3 illustrates the dependencies amongst the SCCs. Clearly, any ordering which has (3) as the last SCC is a correct topological ordering.



(1)
$r_1 : a(X) : -c(X), not\ b(X).$
$r_2 : b(X) : -not\ a(X).$
(2)
$r_8 : r(X) : -not\ c(X).$
(3)
$r_3 : p(X) : -q(X), not\ r(X).$
$r_4 : p(X) : -r(X), not\ s(X).$
$r_6 : q(X) : -p(X).$
$r_7 : r(X) : -q(X).$
(4)
$r_5 : p(X) : -t(X,Y).$
Reduced rule node
Dependency relation

Figure 2.3: SCC dependence graph

2.3.2 Methods

As we aim to have a relational database system as a backend to our Datalog system, we should be able to convert rules into relational algebra expressions. Normally, we deal with *bottom-up* and *top-down* evaluation methods which have been introduced in [BR86] [CGT89]. A bottom-up evaluation will start with the EDB predicates and generate relations for IDB predicates, using the query at the end to select the tuples required. A top-down evaluation will work from the query down generating joins of terminal symbols. Top-down is usually the more efficient, but more complex, while bottom-up is simpler, but less efficient since it does not use the query being presented until the end. The Prolog algorithm is an example of a top-down algorithm, while *naive* and *semi-naive* are two bottom-up methods. The naive evaluation can be improved, to give the so-called semi-naive evaluation. Before we give the naive and semi-naive algorithms, let us describe how to construct an expression of relational algebra that computes the relation for a positive rule body.

Algorithm: 2.1 Computing the Relation for a Positive Rule Body, Using Relational Algebra Operations (from [Ull88]).

INPUT: The body of a rule of Datalog program, which we assume consists of subgoals $Q_1, \ldots, Q_n$ involving variables $X_1, \ldots, X_k$. For each $Q_i = q_i(A_{i1}, \ldots, A_{ik_i})$ with an ordinary predicate, there is a relation R_{q_i} already computed, where the A's are arguments, either variables or constants.

OUTPUT: An expression of relational algebra, which we call

$$EVAL\_RULE(R_p, R_{q_1}, \ldots, R_{q_n})$$

that computes from the relations $R_{q_1}, \ldots, R_{q_n}$[6], a relation $R(X_1, \ldots, X_k)$ with all and only the tuples $(a_1, \ldots, a_k)$ such that, when we substitute a_j for X_j, $1 \le j \le k$, all the subgoals $Q_1, \ldots, Q_n$ are made true.

METHOD: The expression is constructed by the following steps.

1. For each ordinary Q_i, let R_i be the expression $\Pi_{V_i}(\sigma_{F_i}(R_{q_i}))$. Here, V_i is a set of components including, for each variable X that appears among the arguments of Q_i, exactly one component where X appears. Also, F_i is a list of built-in predicates which are linked together with "AND" of the following form:

   - If position k of Q_i has a constant a, then F_i has the term $k = a$.
   - If positions k and l of Q_i both contain the same variable, then F_i has the term $k = l$.

   If Q_i is a subgoal such that there are no terms in F_i, e.g., $Q_i = q(X, Y)$, then take F_i to be the identically true condition, so $R_i = R_{q_i}$.

2. For each variable X not found among the ordinary subgoals, compute an expression D_X that produces a unary relation containing all the values that X could possibly have in an assignment that satisfies all the subgoals of rule p. Since p is safe, there is some variable Y to which X is equated through a sequence of one or more = subgoals, and Y is limited either by being equated to some constant a in a subgoal or by being an argument of an ordinary subgoal.

   - If $Y = a$ is a subgoal, then let D_X be the constant expression { a }.

[6]Technically, not all relations may be present as arguments, because some of them may have built-in predicates and thus not have corresponding relations.

   - If Y appears as the jth argument of ordinary subgoal Q_i, let D_X be $\Pi_j(R_{q_i})$.

3. Let E be the natural join ($\bowtie$) of all the R_i's defined in (1) and the D_X's defined in (2). In this join, we regard R_i as a relation whose attributes are the variables appearing in Q_i, and we regard D_X as a relation with attribute X[7].

4. Let $EVAL\_RULE(R_p, R_{q_1}, \ldots, R_{q_n})$ be $\sigma_F(E)$, where F is the list of built-in predicates appearing among $q_1, \ldots, q_n$, and E is the expression constructed in (3). If there are no built-in subgoals, then the desired expression is just E.

□

As we see from Figure 2.2, the reduced rule/goal graph has two types of nodes: the nodes for nonrecursive definitions and the nodes for the recursive components. Each type of node is treated differently.

2.3.3 Evaluating Nonrecursive Rules

Because the rules are rectified, we have only to project the relation for each rule body onto the variables of the head and, for each predicate, take the union of the relations produced from each of its rules.

Algorithm: 2.2 (Nonrec) Evaluating Nonrecursive Rules Using Relational Algebra Operations (from [Ull88]).

INPUT: All IDB predicates and EDB predicates appearing in a nonrecursive Datalog program.

OUTPUT: An expression of relational algebra for each IDB predicate p in terms of the relation $R_1, \ldots, R_m$ for the EDB predicates.

METHOD: Begin by constructing the dependency graph for all input rules, and order the predicates $p_1, \ldots, p_n$, so that if the dependency graph for these rules has an arc from p_i to p_j, then $i < j$. We can find such an order because the rules are nonrecursive, and therefore the dependency graph has no cycles. Then for $i = 1, 2, \ldots, n$, form the expression for relation P_i for p_i as follows.

[7]Since any X for which D_X is constructed cannot be an attribute of any R_i, the natural join really involves the Cartesian product of all the D_X's, if any. See [Ull88] for more details.

1. For each rule r having p_i as its head, use Algorithm 2.1 to find an expression E_r that computes the relation R_r for the body of rule r, in terms of relations for the predicates appearing in r's body. If the predicate q appearing in r's body is an EDB predicate, let Q be the given relation for q.

2. Since all rules are nonrecursive, all the predicates appearing in the body of r already have expressions for their relations in terms of the EDB relations. Substitute the appropriate expression for each occurrence of an IDB relation in the expression E_r to get a new expression F_r.

3. We may assume that the head of each rule for p_i is $p_i(X_1, \ldots, X_k)$ because the rules are rectified. Then take the expression for P_i to be the union over all rules r for p_i, of $\Pi_{X_1,\ldots,X_k}(F_r)$. □

Example: 2.8 In order to illustrate Algorithm 2.2, let us consider an example ([Ull88]) which has three rules:

$$\begin{array}{lll} r_1: & p(a,Y) & :-\ r(X,Y). \\ r_2: & p(X,Y) & :-\ s(X,Z),\ r(Z,Y). \\ r_3: & q(X,Y) & :-\ p(X,Z),\ s(Z,Y). \end{array}$$

where r and s are EDB predicates with relations R and S, p and q are IDB predicates, for which we want to compute relations P and Q. We begin by rectifying the rules:

$$\begin{array}{lll} r_1: & p(X,Y) & :-\ r(Z,Y),\ X = a. \\ r_2: & p(X,Y) & :-\ s(X,Z),\ r(Z,Y). \\ r_3: & q(X,Y) & :-\ p(X,Z),\ s(Z,Y). \end{array}$$

From the dependency graph we know q depends on p, so the proper order is to work on p first, then q. We can get the relations for the body of r_1 and r_2 by Algorithm 2.1, $R(Z,Y) \bowtie D_X(X)$ and $S(X,Z) \bowtie R(Z,Y)$, where $D_X = \{ a \}$. We project these expressions onto the list of attributes X, Y before the union is taken. The expression for P is

$$P(X,Y) = \Pi_{X,Y}(R(Z,Y) \bowtie \{a\}(X)) \cup \Pi_{X,Y}(S(X,Z) \bowtie R(Z,Y))$$

Next, we consider q. By Algorithm 2.1, the expression for the body of r_3 is $P(X,Z) \bowtie S(Z,Y)$ so the expression for Q is

$$Q(X,Y) = \Pi_{X,Y}(P(X,Z) \bowtie S(Z,Y))$$

□

2.3.4 Evaluating Recursive Rules

In the last subsection we gave an algorithm which does not apply to recursive rules, because for recursive rules there must be cycles in the dependence graph and we cannot find an order as we did in Algorithm 2.2. So when we try to evaluate a predicate on the cycle, there will be a rule with a subgoal whose expression is not yet available.

Bottom-up evaluation is the simplest of the evaluation methods when recursive rules are involved. Let us express the set of provable facts for the predicate p_i (corresponding to IDB relation $P_i, 1 \leq i \leq m$) by the assignment

$$P_i := EVAL(p_i, R_1, \ldots, R_k, P_1, \ldots, P_m)$$

where $EVAL$ is the union of $EVAL\_RULE$ which was defined in Algorithm 2.1 for each of the rules for p_i, projected onto the variables of the head. $R_1, \ldots, R_k$ are EDB relations and $P_1, \ldots, P_m$ are the IDB relations which are to be computed.

Naive Evaluation

Algorithm: 2.3 (Naive$_{scc}$) Naive Evaluation of an SCC in a Simple Datalog Program.
INPUT: All rules for an SCC with recursive predicates $p_1, \ldots, p_n$ and relations $R_1, \ldots, R_k$ for all other predicates in bodies of these rules.

OUTPUT: Relations $P_1, \ldots, P_n$ for $p_1, \ldots, p_n$ in the current SCC.

METHOD: We initialize each P_i to the empty set, and the R_i's are given. Then evaluating each rule as: $P_i = EVAL(p_i, R_1, \ldots, R_k, P_1, \ldots, P_m)$ and repeatedly apply $EVAL$ to obtain new values for the P_i. When at some point, no more new facts can be added to IDB relation P_i, output P_i.

```
for i := 1 to m do begin /*m is the number of IDB predicates*/
        $P_i := \emptyset$;
end
repeat
        for i := 1 to m do begin
            $P_i.old := P_i$;/*save old $P$'s*/
        end
        for i := 1 to m do begin
            $P_i := EVAL(p_i, R_1, \ldots, R_k, P_1.old, \ldots, P_m.old)$;
            /*$R_i$ is a relation which has been computed or an EDB relation*/
        end
until $P_i = P_i.old$ for all $i$;
output $P_i$'s
```

□

Algorithm: 2.4 (Naive) Naive Evaluation of a Sample Datalog Program.
INPUT: Datalog program with IDB relations $P_1, \ldots, P_m$, EDB relations $R_1, \ldots, R_k$.

OUTPUT: Relations $P_1, \ldots, P_n$ for $p_1, \ldots, p_n$.

METHOD: Compute SCCs and for each SCC$_i$ (i= 1 to n) call Algorithm 2.3,

$$Naive_{scc}(SCC_i, P_1, \ldots, P_l, R_1, \ldots, R_k)$$

where $P_1, \ldots, P_l (1 \leq l \leq m)$ are IDB relations in lower SCCs. □

Example: 2.9 Consider the logic program P:

$$\begin{array}{ll} r_1: & ancestor(X,Y) \ :- \ ancestor(X,Z), \ parent(Z,Y). \\ r_2: & ancestor(X,Y) \ :- \ parent(X,Y). \end{array}$$

assume PAR and ANC are the relations of *parent* and *ancestor*, respectively, where

$$PAR = \{(b,a), \ (b,g), \ (a,d), \ (a,e), \ (d,f), \ (c,h)\}.$$

We translate the program into the relational algebra expression:

$$ANC = \Pi_{X,Y}(ANC \bowtie PAR) \bigcup PAR$$

Let us follow Algorithm 2.3. Initially, we set $ANC^0 = \emptyset$. We then enter the **repeat** loop. We initialize $ANC.old = ANC = \emptyset$. Now we compute the first value of ANC. Since the join with an empty relation is empty:

$$ANC^1 = PAR = \{(b,a), \ (b,g), \ (a,d), \ (a,e), \ (d,f), \ (c,h)\}.$$

$ANC^1 \neq ANC.old$, so we enter the second iteration. After saving ANC^1 to $ANC.old$, we compute the next value of ANC as follows:

$$ANC^2 = \{(b,a), \ (b,g), \ (a,d), \ (a,e), \ (d,f), \ (c,h), \ (b,d), \ (b,e), \ (a,f)\}.$$

Note here that $ANC^2 = ANC^1 \bigcup \{(b,d), \ (b,e), \ (a,f)\}$. $ANC^2 \neq ANC.old$, so we enter the third iteration. After setting $ANC.old = ANC^2$, we obtain:

$$ANC^3 = \{(b,a), \ (b,g), \ (a,d), \ (a,e), \ (d,f), \ (c,h), \ (b,d), \ (b,e), \ (a,f), \ (b,f)\}.$$

Here $ANC^3 = ANC^2 \bigcup \{(b,f)\}$. ANC^3 is different from $ANC.old$, so we enter the fourth iteration. Let $ANC.old = ANC^3$, then:

$$ANC^4 = \{(b,a), \ (b,g), \ (a,d), \ (a,e), \ (d,f), \ (c,h), \ (b,d), \ (b,e), \ (a,f), \ (b,f)\}.$$

Finally, $ANC^4 = ANC.old = ANC^3$ and the loop terminates. □

We see from the example, however, inefficiencies arise since if a tuple is proved in one iteration then, as its antecedent tuples are still in P_i, it will be proved again in each subsequent iteration. In order to avoid this, we just pay attention to the new tuples that are produced by each rule which can be found if we substitute the full relation for all but one of the subgoals and substitute only the incremental tuples for the others. There is a well-known evaluation method which takes advantage of incremental relations, called "semi-naive" evaluation [Ull88], which we will describe next.

Semi-naive Evaluation

The naive evaluation method is simple, but it may repeat a lot of computation. A semi-naive evaluation is a bottom-up technique designed for eliminating redundancy in the evaluation of tuples at different iterations [CGT89]. Semi-naive evaluation aims to overcome redundancies in the looping mechanisms by only calculating the increment between the old value for a relation and the new, thus the old tuples will not be recalculated. We must do this substitution using each subgoal in turn as the subgoal with the incremental relation, and then take the union of the resulting relations. Since we cannot get any incremental tuples for EDB relations, we may just take the union over the IDB predicates. Let us define more formally the operation of incremental evaluation of the relations associated with rules and predicates. Let r be a rule with ordinary subgoals $S_1, \ldots, S_n$; we exclude from this list any subgoals with built-in predicates. Let $R_1, \ldots, R_n$ be the current relations associated with subgoals $S_1, \ldots, S_n$, respectively, let $\triangle R_1, \ldots, \triangle R_n$ be the list of corresponding incremental relations which is the difference between the old R_i and the new one. Recall that $EVAL\_RULE()$ is defined by Algorithm 2.1. Then the incremental relation for rule r is the union of the n relations

$$EVAL\_RULE(r, R_1, \ldots, R_{i-1}, \triangle R_i, R_{i+1}, \ldots, R_n)$$

for $1 \leq i \leq n$. That is one incremental relation $\triangle R_i$ is substituted for the full relation R_i in each term. Formally, we define:

$$EVAL\_RULE\_INCR(r, R_1, \ldots, R_n, \triangle R_1, \ldots, \triangle R_n)$$

$$= \bigcup_{1 \leq i \leq n} EVAL\_RULE(r, R_1, \ldots, R_{i-1}, \triangle R_i, R_{i+1}, \ldots, R_n)$$

Now, suppose we are given relations $R_1, \ldots, R_k$ for the EDB predicates $r_1, \ldots, r_k$. For the IDB predicates $p_1, \ldots, p_m$ we are given associated relations $P_1, \ldots, P_m$ and associated incremental relations $\triangle P_1, \ldots, \triangle P_m$. Let p_i be an IDB predicate. Define:

$$P_i := EVAL\_INCR(p_i, R_1, \ldots, R_k, P_1, \ldots, P_m, \triangle P_1, \ldots, \triangle P_m)$$

to be the union of what $EVAL\_RULE\_INCR()$ produces for each rule for p_i. Since the incremental relations for the EDB predicates are $\emptyset$ in each application of $EVAL\_RULE\_INCR()$, so the terms for those subgoals that are EDB predicates do not have to appear in the union for *EVAL_RULE_INCR*(). We shall next give the complete semi-naive algorithm for linear programs that has been implemented in the Datalog system. Restricting programs to linear recursive is not a problem as most "real life" recursive situations are linearly recursive [BR86]. More details have been given in [Ull88] [Was90].

The Datalog system actually uses a slightly more sophisticated algorithm—it analyses the rule dependence graph and finds the SCCs (strongly-connected components) in the graph (see Section 2.3.1).

Algorithm: 2.5 (Semi-naive$_{scc}$) Semi-Naive Evaluation of an SCC in Simple Datalog Program.
INPUT: Same as Algorithm 2.3.

OUTPUT: Same as Algorithm 2.3.

METHOD: The first stage is to set up the temporary tables for each IDB predicate p_i:

- P_i.*delta*—the difference between the current value of the relation and the value of the relation from the previous iteration.
- P_i—the current value of the relation.

Compute the rules that involve no IDB predicates in the body once, and let P_i be the relation for predicate p_i. Then use *EVAL_INCR*() repeatedly on incremental IDB relations. For each IDB predicate p_i, there is an associated relation P_i that holds all the tuples, and there is an incremental relation P_i.*delta* that holds only the tuples added on the previous round. The computation is shown in Figure 2.4. □

Algorithm: 2.6 (Semi-naive) Semi-Naive Evaluation of a Sample Datalog Program.
INPUT: Datalog program with IDB relations $P_1, \ldots, P_m$, EDB relations $R_1, \ldots, R_k$.

OUTPUT: Relations $P_1, \ldots, P_n$ for $p_1, \ldots, p_n$.

METHOD: Compute SCCs and for each SCC$_i$ (i= 1 to n) call Algorithm 2.5,

$$\text{Semi-naive}_{scc}(SCC_i, P_1, \ldots, P_l, R_1, \ldots, R_k)$$

where $P_1, \ldots, P_l (1 \leq l \leq m)$ are IDB relations in lower SCCs. □

```
for i := 1 to m do begin /*m is the number of all IDB predicates*/
          $P_i.delta := EVAL(p_i, R_1, \ldots, R_k, \emptyset, \ldots, \emptyset)$;
          $P_i := P_i.delta$;
end;
repeat
          for i := 1 to m do begin
               $P_i.old := P_i.delta$;/*save old $P.delta$'s*/
          end;
          for i := 1 to m do begin
               $P_i.delta := EVAL\_INCR(p_i, R_1, \ldots, R_k, P_1, \ldots, P_m, P_1.old, \ldots, P_m.old)$;
               $P_i.delta := P_i.delta - P_i$ /*remove "new" tuples that actually appeared before*/
          end;
          for i := 1 to m do begin
               $P_i := P_i \cup P_i.delta$;
          end
until $P_i.delta = \emptyset$ for all $i$;
output $P_i$'s
```

Figure 2.4: Semi-naive$_{scc}$ evaluation of Datalog programs.

Example: 2.10 Recall Example 2.9. Let us follow Algorithm 2.5 on the same example. Initially, we obtain:

$$\begin{aligned} ANC^1.delta &= PAR \\ &= \{(b,a),\ (b,g),\ (a,d),\ (a,e),\ (d,f),\ (c,h)\}. \end{aligned}$$

$$\begin{aligned} ANC^1 &= ANC^1.delta \\ &= \{(b,a),\ (b,g),\ (a,d),\ (a,e),\ (d,f),\ (c,h)\}. \end{aligned}$$

$ANC.old = ANC^1.delta = \{(b,a),\ (b,g),\ (a,d),\ (a,e),\ (d,f),\ (c,h)\}$.

$$\begin{aligned} ANC^2.delta &= \{(b,d),\ (b,e),\ (a,f)\} - ANC^1 \\ &= \{(b,d),\ (b,e),\ (a,f)\} \end{aligned}$$

$$\begin{aligned} ANC^2 &= ANC^1 \bigcup ANC^2.delta \\ &= \{(b,a),\ (b,g),\ (a,d),\ (a,e),\ (d,f),\ (c,h),\ (b,d),\ (b,e),\ (a,f)\}. \end{aligned}$$

$ANC.old = ANC^2.delta = \{(b,d),\ (b,e),\ (a,f)\}$.

$$ANC^3.delta = \{(b,f)\} - ANC^2 = \{(b,f)\}$$

$$\begin{aligned} ANC^3 &= ANC^2 \bigcup ANC^3.delta \\ &= \{(b,a),\ (b,g),\ (a,d),\ (a,e),\ (d,f),\ (c,h),\ (b,d),\ (b,e),\ (a,f),\ (b,f)\}. \end{aligned}$$

$ANC.old = ANC^3.delta = \{(b,f)\}$.

$$\begin{aligned} ANC^4.delta &= \{\emptyset\} - ANC^3 \\ &= \emptyset \end{aligned}$$

$$\begin{aligned} ANC^4 &= ANC^3 \bigcup ANC^4.delta \\ &= \{(b,a),\ (b,g),\ (a,d),\ (a,e),\ (d,f),\ (c,h),\ (b,d),\ (b,e),\ (a,f),\ (b,f)\}. \end{aligned}$$

The computation terminates because $ANC^4.delta = \emptyset$. ANC^4 is the final result. □

Obviously, the results of two computations, naive and semi-naive, are the same, but the latter is more efficient. In Example 2.10 only $ANC^i.delta$ has been involved in the joins, while in Example 2.9 we had to compute joins for each of the temporary ANC^i, which always have at least as many tuples than the corresponding values of $ANC^i.delta$.

2.4 Negation in logic programs and Datalog

In this section, the main topic of discussion is why negation is needed. We first look at our real world. Roughly speaking, there are two aspects to objects in the world[8], for instance, {positive, negative}, {true, false}, {good, bad}, or {black, white}. Under some premises, we may illustrate one aspect of some thing using its opposite, for example, in mathematical logic, we can say not(not(A)) is A. We will replace "not" with "$\neg$", as follows:

$$A \equiv \neg(\ \neg A\ )$$

Example: 2.11 Let us consider the rule: If X is a person and X doesn't eat meat, then X is a vegetarian. In Datalog without negation, there is no way to represent such a rule. □

As we have demonstrated above the necessity of negation in our real world, we will next discuss the feasibility of negation in our logical world.

In recent years, various semantics for negation in logic programming have been studied. There are three well-known semantics in our survey. One is the *Closed World Assumption* (CWA) model [Rei78], another is the *Negation as Failure* (NF) model [Cla78], and the third is the *Completed Data Base* (COMP) (or *Completion*) model [Llo87]. In fact, the last one has been introduced by Clark [Cla78] for justifying the use of the negation as failure rule in Prolog and developed by Lloyd [Llo87]. Roughly speaking, the CWA is that a ground atom is taken to be false if it is not a logical consequence of the program; NF states that a ground atom can be inferred to be false if every possible proof of the ground atom fails. The CWA is in fact made implicitly when evaluating queries on relational databases. Since we are concerned in this thesis with evaluating Datalog programs rather than logic programs, we will not discuss these semantics further.

Definition: 2.27 A Datalog with negation clause, denoted by Datalog$^\neg$, is either a positive (ground) fact or a rule where negative literals are allowed to appear in the body. Datalog$^\neg$ is not a Horn clause, but is an extended form of Horn clause which has the form:

$$\neg P_1, \ldots, \neg P_n, P_0, P_{n+1}, \ldots, P_{n+m}. \qquad (2.6)$$

either as

$$P_0 :- P_1, \ldots, P_n,\ not\ P_{n+1}, \ldots,\ not\ P_{n+m}. \qquad (2.7)$$

or as

$$P_0 \leftarrow P_1, \ldots, P_n, \neg P_{n+1}, \ldots, \neg P_{n+m}. \qquad (2.8)$$

[8]For now, we are only concerned with a 2-valued world.

$P_1, \ldots, P_n$ (from Form(2.7)) are the positive literals, $\neg P_{n+1}, \ldots, \neg P_{n+m}$ are the negated literals, and a conjunction of them is called the *body* of the clause. The unnegated literal P_0 is called the *head* of the clause. We have a regulation that the head comprises at most one positive literal. Also, for safety reasons we require that each variable occurring in a negative literal of a rule body also occurs in a positive literal of the same rule body. We also use Datalog$^\neg$ to denote the language in which Datalog$^\neg$ rules are allowed. The Datalog$^\neg$ language has been described in [Ull88] [CGT89]. □

Example: 2.12 Let us recall Example 2.11. Assume that the predicate symbols *person*, *eat_meat* and *vegetarian*, represent the properties of being a person, a person who eats meat, and a vegetarian, respectively. Now we can represent the example as a Datalog$^\neg$ clause as follows:

$$vegetarian(X) : -person(X),\ not\ eat\_meat(X).$$

Note that in relational algebra an expression corresponding to the above rule can be formulated with ease by use of the *minus* operator "–". Assume the one-column relation *Person* contains the name of all persons and another one-column relation *Eat_meat* contains the names of all people who eat meat. Then we obtain the relation *Vegetarian*, that is, the names of all *Vegetarians*, simply by subtracting *Eat_meat* from *Person*, thus

$$Vegetarian = Person - Eat\_meat.$$

Obviously, the negative subgoal that was introduced into the Datalog rule body increased the expressive power of Datalog. □

Just as we mentioned before, there are two aspects of all things. We allow the use of negative information to increase the expressive power of Datalog programs. However here are some of the problems we face if we allow a program P to contain rules with negated predicates:

1. We shall encounter the first problem in what rules with negation mean. Sometimes for a rule with negation there is an apparent divergence between what we intuitively expect the rule should mean and the answer we would get.

2. When rules with negation are allowed, there might not be a least fixed point, but several minimal fixed points.

More details are discussed in [Ull88].

2.5 Different semantics for Datalog$^\neg$

In the previous section, we have related why negation is needed intuitively. In this section, we will specify some previously proposed semantics for Datalog$^\neg$. These semantics have been implemented in our system described in Chapter 3. Different techniques to integrate negation in logic programs are surveyed by Bidoit [Bid91].

2.5.1 Stratified semantics

Stratified negation was first proposed by Chandra and Harel [CH82] and later, independently, rediscovered by [Gel88] [Naq86] [ABW86] [Prz88]. Stratified negation in Datalog$^\neg$ help deal with the problem of many minimal fixed points.

Definition: 2.28 A Datalog$^\neg$ program P is *stratified* if we can group all predicates into *strata*, that is, disjoint sets $S_1, \ldots, S_n$, which are the largest sets of predicates such that

1. All predicates with the same predicate name belong to the same stratum.
2. If a predicate p is the head of a rule with a subgoal that is a negative predicate q, then q is in a lower stratum than p. In other words, $S_p > S_q$ if there is a rule of the form
$$p \ :- \ \ldots, not\ q, \ldots$$
3. If predicate p is the head of a rule with a subgoal that is a positive q, then the stratum of p is at least as high as the stratum of q. In other words, $S_p \geq S_q$ if there exists a rule of the form
$$p \ :- \ \ldots, q, \ldots$$

Any grouping S_i of P satisfying the above conditions is called a *stratification* of P. □

In order to discuss *stratified* Datalog$^\neg$ more intuitively, let us recall the definition of a dependency graph from [CGT89].

Definition: 2.29 The *dependency graph* of a Datalog$^\neg$ program P, denoted by G_P, is defined as follows. The nodes of G_P consist of the IDB predicate symbols occurring in P. There is an edge $< q, p >$ from q to p in G_P iff the predicate symbol q occurs positively or negatively in a body of a rule whose head predicate is p. We will mark the edge $< q, p >$ with "$\neg$" iff there exists a rule with head predicate p and with negative predicate q in the body. □

Example: 2.13 Consider the following Datalog$^\neg$ program P with EDB predicate d:

$$\begin{array}{lll} p & :- & not\ q, s. \\ q & :- & q. \\ q & :- & d,\ not\ r. \\ r & :- & d. \\ s & :- & q. \end{array}$$

The *dependency graph* G_P of the program P is depicted in Figure 2.5. □

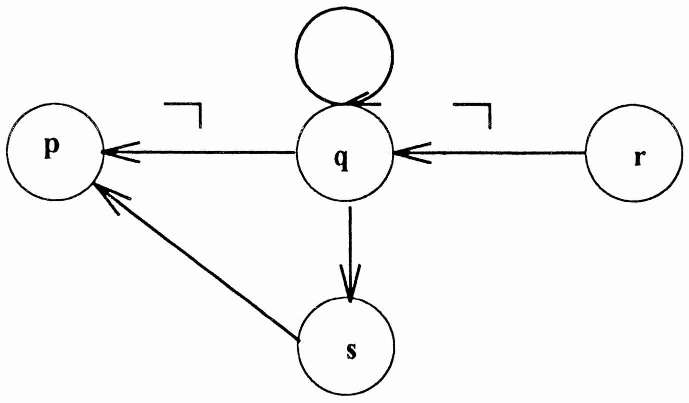


Figure 2.5: Dependency Graph G_P

An alternative definition of a stratified program in terms of a dependency graph is as follows.

Definition: 2.30 A Datalog$^\neg$ program P is *stratified* iff its dependency graph G_P does not contain any cycles involving a negative edge, labeled with "¬". See [ABW86] for more details. □

Some definitions which extend the class of stratified Datalog$^\neg$ have also been proposed. For example, Przymusinski proposed a class of logic programs called *locally stratified* logic programs in [Prz88]. Another class of stratified programs, called *modular stratified* programs, was proposed by Ross in [Ros90]. This generalizes both stratification and local stratification. A program is *modularly stratified* if and only if its mutually recursive components are *locally stratified* once all instantiated rules with a false subgoal that is defined in a "lower" component are removed. However, we did not implement these extensions of stratified semantics in this thesis and we will leave them for future extensions.

2.5.2 Inflationary semantics

In this section we will introduce an approach which leaves the programmer totally free to write any Datalog$^\neg$ program he or she wants. A semantics for Datalog$^\neg$ programs, the *inflationary semantics*, was first proposed by Gurevich and Shelah in [GS86b] and later, studied further in [KP91] [AV91].

Definition: 2.31 The term *inflationary operator* was coined in [GS86b] where an operator H mapping k-ary relations to k-ary relations is said to be *inflationary* if $S \subseteq H(S)$ for every k-ary relation S. In other words, if M is a mapping from k-ary relations to k-ary relations, then the operator $inf\_M(S) = S \cup M(S)$ is *inflationary.*

Definition: 2.32 [KP91] Consider a Datalog$^\neg$ program P over an arbitrary but fixed finite vocabulary σ[9] having a single IDB relation S. Let k be the arity of S and suppose that $D = (\psi, R_1, \ldots, R_n)$ is a database over σ having universe ψ. If we have a mapping Θ, the *inflationary semantics* of the Datalog$^\neg$ program P on D is defined by iterating the mapping Θ in the following way: we define first the sequence Θ^n, $n \geq 1$, of k-ary relations on ψ by the equations

$$\Theta^1 = \Theta(\emptyset), \quad \Theta^{n+1} = \Theta^n \cup \Theta(\Theta^n)$$

and then we put

$$\Theta^\infty = \bigcup_{n=1}^{\infty} \Theta^n.$$

The *inflationary semantics* of the Datalog$^\neg$ program P on the database D is the k-ary relation Θ^∞. For Datalog$^\neg$ programs with more than one IDB relation, the *inflationary semantics* is defined in a similar way by simultaneous induction in the defining equations. □

In what follows we will illustrate how the inflationary semantics of a Datalog$^\neg$ program can be computed. Let P be a Datalog$^\neg$ program and E an EDB. The inflationary evaluation of P on E is performed iteratively so that all rules of P are processed in parallel at each step. From the EDB and the facts already derived, new facts are derived by applying the rules of P. These new facts are added to the result at the end of each step. At each step, the CWA is made temporarily during the evaluation of the rule bodies–it is assumed that the negation of all facts not yet derived is valid. The procedure terminates when no more additional facts can be derived. Let us see an example as follows:

[9] i.e., we have a fixed sequence $\sigma = (R_1, \ldots, R_n)$ of database relational symbols such that each R_i is of arity m_i.

Example: 2.14 Consider a Datalog$^\neg$ program P where d is the only EDB *predicate*.

$$\begin{array}{ll} r_1: & s(X) : -p(X), q(X), \ not\ r(X). \\ r_2: & p(X) : -d(X), \ not\ q(X). \\ r_3: & q(X) : -d(X), \ not\ p(X). \\ r_4: & r(X) : -d(X), d(0). \end{array}$$

We ask that this program be evaluated on an inflationary basis on an EDB $E = \{d(1)\}$. Let M be the mapping associated with P. Initially, $M^0 = M(\emptyset) = \{d(1)\}$. After the first iteration, we get two new facts: one is $p(1)$ from r_2 and the other is $q(1)$ from r_3. We can write $M^1 = M^0 \cup M(M^0) = \{d(1), p(1), q(1)\}$. After the second iteration, rule r_1 produces the new fact $s(1)$, so $M^2 = M^1 \cup M(M^1) = \{d(1), p(1), q(1), s(1)\}$. Since no further facts are derivable, the procedure stops with the result $\{d(1), p(1), q(1), s(1)\}$. Obviously, the result is a least fixpoint of the Datalog$^\neg$ program P. This is also a Herbrand model of $P \cup E$ but this model is not minimal since $\{d(1), p(1)\}$ and $\{d(1), q(1)\}$ are smaller models of $P \cup E$. □

In [Bid91] [AV91], they give two different ways of "evaluating" the inflationary model semantics. One is the *deterministic* fixpoint semantics which corresponds to "apply ALL rules" at once, the other is the *non-deterministic* semantics which corresponds rather to "apply ONE rule" at a time. We adopt the first approach in our system which is described in more detail in the next chapter.

2.5.3 Well-founded semantics

In this section we will investigate another semantics which enables a programmer to write any Datalog$^\neg$ program—the *well-founded* (partial) model. It was proposed first by Gelder, Ross, and Schlipf in [GRS88]. Their method nicely extends the stratified approach to arbitrary logic programs with negation. Later a fixpoint method for computing the well-founded partial model was given in [Gel88], while resolution-based procedural semantics for well-founded negation is provided in [Ros89]. Further important papers related to well-founded model semantics are [Prz89] and [Bid91] where the relationship to logical constructivism is investigated. First we introduce some definitions.

Definition: 2.33 Let S be a set of literals. We denote the set formed by taking the complement of each literal in S by $\neg \cdot S$.

- If $q \in \neg \cdot S$ then we say the literal q is *inconsistent* with S.

- If some literal in a set of literals R is inconsistent with the set of literals S then we say R and S are *inconsistent.*
- If a set of literals S is inconsistent with itself, then we say S is *inconsistent*; otherwise it is *consistent.*

□

Definition: 2.34 Let P be a Datalog$^{\neg}$ program, H_B its associated Herbrand base, and I be a set of literals. We say that $U_p(I) \subseteq H_B$ is an *unfounded set of P with respect to I* if each atom $p \in U_p(I)$ satisfies the following condition: For each instantiated rule r of P whose head is p, (at least) one of the following holds:

1. The complement of some subgoal literal in rule r is in I.
2. Some positive subgoal literal of rule r is in $U_p(I)$.

A literal that makes either of the above conditions true is called a *witness of unusability* for rule r with respect to I. The union of all unfounded sets with respect to a given I is also unfounded, and is called the *greatest unfounded set*, denoted by $U_P(I)$. □

Example: 2.15 Consider the following program from [GRS88], where the atoms are abbreviated to single letters.

$$
\begin{array}{lll}
r_1: & a & :- \ c,\ not\ b. \\
r_2: & b & :- \ not\ a. \\
r_3: & c. & \\
r_4: & p & :- \ q,\ not\ r. \\
r_5: & p & :- \ r,\ not\ s. \\
r_6: & q & :- \ p. \\
r_7: & r & :- \ q.
\end{array}
$$

From the definition above, we say the atoms $\{p, q, r, s\}$ form an unfounded set and the pair $\{a, b\}$ do not form an unfounded set. The difference between $\{p, q, r, s\}$ and $\{a, b\}$ is that: declaring any of p, q or r false does not resurrect a proof for any other element of the set. Clearly s can never be proven because it has no rules in the program. However, as soon as one of a or b is declared false, it becomes possible to prove the other is true (only if c is true, in the case of a). And if both are declared false at once, we have an inconsistency. □

Definition: 2.35 For a set of literals I:

- $T_P(I)$ is the usual *immediate consequence* transformation defined by: $p \in T_P(I)$ if and only if there is some instantiated rule r of P such that r has head p, and each subgoal literal in rule r occurs in I.
- $U_P(I)$ is the greatest unfounded set of P with respect to I.
- $W_P(I) = T_P(I) \cup \neg \cdot U_P(I)$.

□

Immediately from the definitions we can prove that $T_P(I)$, $U_P(I)$, and $W_P(I)$ are monotonic transformations.

Definition: 2.36 Let P be a Datalog$^\neg$ program. The *well-founded model* of P is the least fixpoint of the operator $W_P(I)$ associated with P. Every negative literal denotes that its atom is false, every positive literal denotes that its atom is true, and missing atoms have undefined truth value. So the well-founded model semantics is based on three-valued logic.□

Example: 2.16 Recall Example 2.15, a well-founded model is $\{c, \neg p, \neg q, \neg r, \neg s\}$. □

2.6 Expressive power

From the previous sections, we know that a Datalog program is a Datalog$^\neg$ program with the additional condition that no negation occurs in the body of the rules. So intuitively speaking, Datalog$^\neg$ programs have greater expressive power than Datalog programs. In this thesis, we introduce stratified (Datalog$^\neg_{\mathrm{stra}}$), inflationary (Datalog$^\neg_{\mathrm{infl}}$) and well-founded (Datalog$^\neg_{\mathrm{wellf}}$) semantics. In [Bid91], there is a comparison of Datalog$^\neg_{\mathrm{stra}}$ with FOL and FP, Datalog$^\neg_{\mathrm{infl}}$ with FP and Datalog$^\neg_{\mathrm{wellf}}$ with FP. The following result holds on finite databases.

Theorem: 2.1 ([Bid91] [AV91] [KP91])
Datalog is a strict subclass of Datalog$^\neg_{\mathrm{stra}}$, Datalog$^\neg_{\mathrm{wellf}}$ and Datalog$^\neg_{\mathrm{infl}}$.

However, it has been discussed above that Datalog$^\neg_{\mathrm{stra}}$ cannot express all Datalog$^\neg$ programs. In particular, it cannot express the Datalog$^\neg$ programs involving an edge labeled with "$\neg$" in a cycle of its dependency graph. Thus:

Theorem: 2.2 ([Bid91] [AV91] [KP91])
$\text{Datalog}^{\neg}_{\text{stra}}$ is a strict subclass of $\text{Datalog}^{\neg}_{\text{wellf}}$ and $\text{Datalog}^{\neg}_{\text{infl}}$, that is, $\text{Datalog}^{\neg}_{\text{wellf}}$ and $\text{Datalog}^{\neg}_{\text{infl}}$ are more expressive than $\text{Datalog}^{\neg}_{\text{stra}}$.

The next question, of course, is which semantics is more expressive: $\text{Datalog}^{\neg}_{\text{wellf}}$ or $\text{Datalog}^{\neg}_{\text{infl}}$. We know that there are no syntactic restrictions on programs for $\text{Datalog}^{\neg}_{\text{infl}}$ and $\text{Datalog}^{\neg}_{\text{wellf}}$. But which one has more expressive power? In [KP91], they give a program P which with inflationary semantics is used to compute what they call the distance query. With well-founded semantics P computes only the transitive closure. It is an open problem (see [Kol89]) whether one can always find a well-founded program to compute the same query as an inflationary program. On the other hand, there must exist a program which can make the inflationary semantics correspond to the well-founded semantics. We can rewrite a program P to a program P' such that inflationary(P')=well-founded(P) (see chapters 3 and 4 for more details).

The same problem is discussed in [Bid91] [AV91] [KP91] and a direct proof is provided by [AV91]. Thus

Lemma: 2.1 $\text{Datalog}^{\neg}_{\text{infl}}$ has at least the same expressive power as $\text{Datalog}^{\neg}_{\text{wellf}}$.

For the expressive power of various semantics for negation which we have chosen, the following summaries known results on finite databases ($\subset$ denotes proper inclusion in terms of expressive power).

$$\text{Datalog} \subset \text{Datalog}^{\neg}_{\text{stra}} \subset \text{Datalog}^{\neg}_{\text{wellf}} \subseteq \text{Datalog}^{\neg}_{\text{infl}}$$

Chapter 3

Implementation of NDatalog

Our implementation of Datalog$^\neg$, which is called NDatalog, evaluates programs containing negation. NDatalog extends the DatalogIC [Was90] language which was devised to serve as a vehicle for semantic optimization and was implemented on top of an Oracle relational database system. Since the DatalogIC system did not include any form of negation, so I had to devise ways to implement the three forms of negation by translating to SQL (see Section 3.6). I had extended the DatalogIC system to include the negation syntax, as well as an evaluation algorithm for each negation semantics. In addition, NDatalog system uses the INFORMIX database system instead of Oracle.

In this chapter, I draw together the ideas introduced in the previous chapter on Datalog$^\neg$. I first outline the syntax of the language NDatalog. And then, we focus our attention on how the various semantics for negation which have been described are implemented in the NDatalog system.

3.1 The NDatalog Language

The NDatalog language is based on the DatalogIC language, but can in addition express negation semantics in Horn clause form as a part of the program.

Similar to DatalogIC, and as I mentioned in the previous chapter, there are two types of predicate definition in the NDatalog language: one for intensional predicates, the other for extensional predicates. This is intended to introduce some modularisation into NDatalog programs. An intensional predicate is given a definition that starts with the "INT" reserved word, which is followed by the name of the predicate and its argument list (which must be variables). This is the *head* of the definition. An INT predicate leads to one or more *rules*.

Any rule in the definition which has a head matching the definition (same predicate and arguments) can be written with the head omitted. A predicate which is in the rule's *body* can be negated. The definition for extensional predicates starts with the "EXT" reserved word, followed by the definition head. The head of extensional definitions must contain the name of the relation being described and its attributes as they appear in the database. If this was not done, we could not convert the rules into SQL expressions. The name of the relation and the arguments in an intensional definition are also used as the name for the SQL table definition for the predicate. Sections 3.5 and 3.6 will give more details on this.

As well as a set of predicate definitions, a program contains a set of query forms, as we discussed in Definition 2.16. There we specify what queries the program will accept. Variables in a query are preceded by a label. A "?" indicates that the variable is an input variable, while a "!" indicates that it is an output variable. At runtime, users are given a list of query forms from which they select a query to be evaluated. The system collects bindings for the input variables in the query selected, processes the query and returns bindings for the output variables. The query forms are important as the system is made aware of the type of queries that are going to be asked and can therefore optimise and compile the program accordingly. Ideally the user should be allowed to present arbitrary queries but this has not been implemented.

Example: 3.1 Now let us look back at Example 2.12. To change the example a little, we suppose that $person(X)$ is an EDB relation, and $eat\_meat(X, Y)$ is an EDB relation with the meaning that X eats meat Y (Y may be baby chicken, etc.). So we can write a program in NDatalog to define a set of vegetarians as follows:

EXT $person(X)\{\}$ /* The variable type has been defined when we create the table in INFORMIX. */
EXT $eat\_meat(X, Y)\{\}$
INT $vegetarian(X)\{$

$$:-person(X), not\ eat\_meat(X, Y).$$

$\}$
$?\text{-}vegetarian(?X).$
□

Converting the relation $eat\_meat(X)$ into $eat\_meat(X, Y)$, results in some new problems. We will discuss them more in Section 3.4.

3.1.1 The BNF of NDatalog

The Backus-Naur specification of NDatalog is presented below. The items which are in DatalogIC but not in NDatalog are in *italics*.

Program	::= { Statement }
Statement	::= Ext_Definition | Int_Definition | Rule | Query | *Integrity_Constr*
Int_Definition	::= 'INT' Head_Predicate '{' { Inside_statement } '}'
Ext_Definition	::= 'EXT' Head_Predicate '{' { *Integrity_Constr* } '}'
Inside_statement	::= Headless_Rule | Rule | *Integrity_Constr*
Headless_Rule	::= ':-' Horn_Cl_Body '.'
Rule	::= Predicate ':-' Horn_Cl_Body '.'
Query	::= [Predicate] '?-' Horn_Cl_Body '.'
Integrity_Constr	::= 'IC' Horn_Cl_Body '→' '.' | 'IC' '→' Predicate '.'
	| 'IC' Horn_Cl_Body '→' Predicate '.'
Horn_Cl_Body	::= { Predicate | Comparison_Op }
Predicate	::= Positive_Predicate | Negation_Predicate
Positive_Predicate	::= PRED_ID '(' Argument { ',' Argument } ')'
Negation_Predicate	::= 'not' PRED_ID '(' Argument { ',' Argument } ')'
Head_Predicate	::= PRED_ID '(' Variable { ',' Variable } ')'
Comparison_Op	::= Argument OPERATOR Argument
Argument	::= Variable | Constant
Variable	::= [Label] VAR_ID
Label	::= '!' | '?'
Constant	::= STRING | INTEGER

The lexical symbols are as follows:

- PRED_ID is an alphanumeric string beginning with a lowercase letter.
- VAR_ID is an alphanumeric string beginning with a capital letter.
- STRING is an alphanumeric string delimited with single quotes.
- INTEGER is a string of digits.
- OPERATOR is one of $\{>, >=, <, <=, !=, =\}$.
- Comments are delimited with /* and */.
- // indicates that the rest of the line is a comment.

Note that only query variables use the labels ! and ?.

Definition: 3.1 A NDatalog program is a *complete* NDatalog *program* if it includes EXT, INT and query define clauses. In the NDatalog program, we shall allow a negated subgoal to appear only in the body of an INT definition (rule), forbid the name of an INT definition to be the same as an EXT definition, and require that at least one EXT relation be used by the program. □

3.2 System View

In this section, I will describe the structure of NDatalog system as shown in Figure 3.1.

3.2.1 User Interface

Just as with other systems, we need a program which handles all the interaction between the user and the system. A parser, written using the UNIX tools, Lex and YACC, parses NDatalog programs. The commands available to the user are as follows:

- **parse** -[*option*][*filename*]

[*option*]	
s	Stratified model.
w	Well-founded model.
i	Inflationary model.

  [*filename*] A NDatalog program's name.
  Parses the given NDatalog program, building a rule/goal graph; it also checks safety and reports errors. The program then becomes the current program.

- **compile** Depending on the option selected in the parse command, compiles the current program into SQL expressions.

- **list** [*option*]

[*option*]	
defn { *defn name* }{ *root* }	Lists definition "name".
query	Lists all queries.
all	Gives a list of all definitions.
prog	List current program.

- **query** [*queryname*] Executes the named query.

- **quit** Exit the NDatalog system.

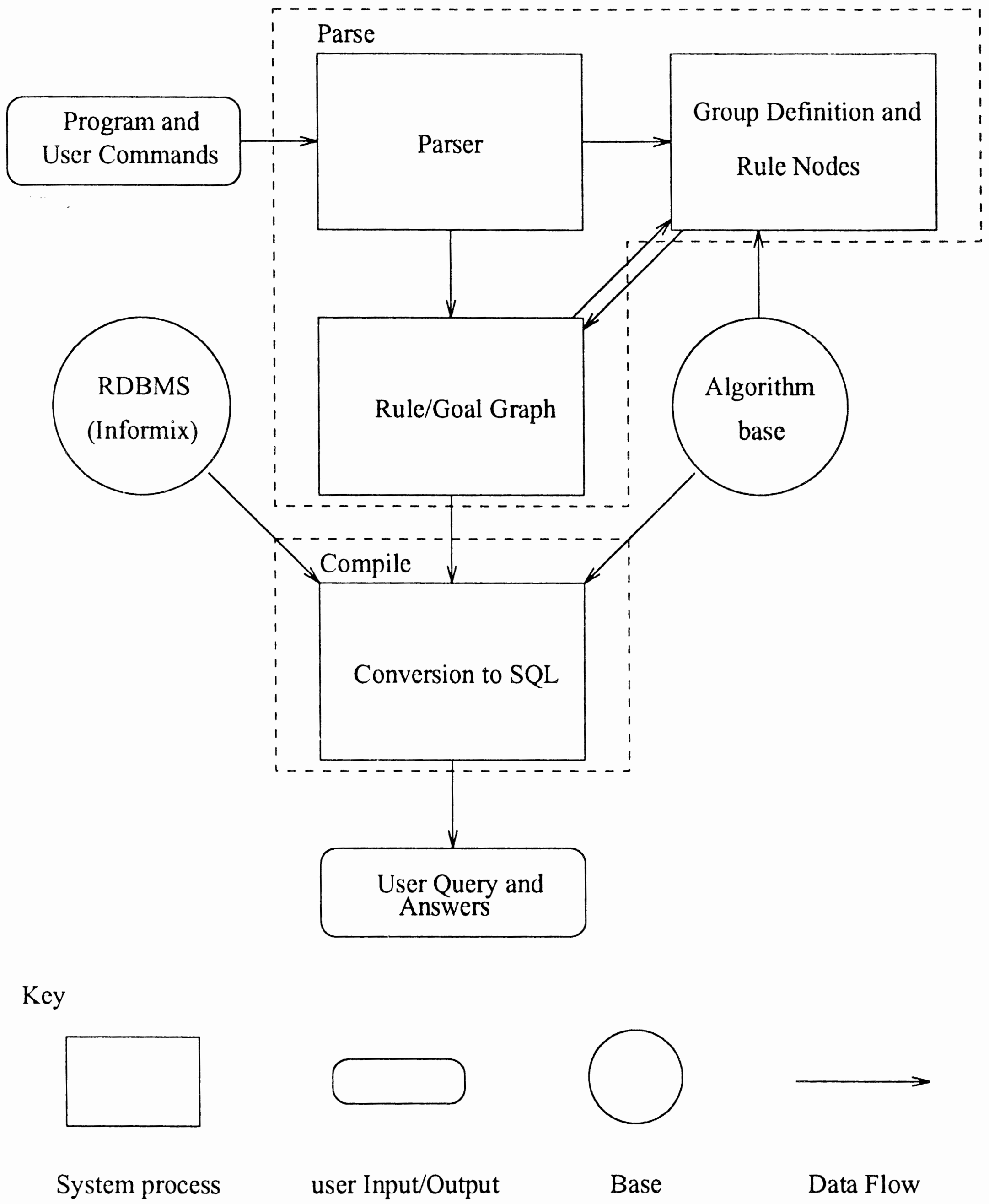


Figure 3.1: Overview of the NDatalog system

Example: 3.2 Assume EDB relation *male*(X) = { tom, tony, david, park, sam, bob, bell } and *married*(X,Y) = { (tony, mary), (sam, tina), (bob, susan) }. Consider the program:

```
EXT married ( X, Y) { }
EXT male ( X ) { }

INT female ( X ) {
        :- married ( Y, X ), not male ( X ).
}

?-femle(?X).
```

Assuming the program is stored in file "test", let us see how the program is parsed, compiled and queried. User commands are in `typewriter font`, while system input and output is in *italics*.

Starting at the UNIX prompt $, the user enters the NDatalog system.

```
$dlog

Welcome to NDatalog. Enter '?' for help.

dlog>parse -s test

Checking stratified: Program is stratified.
Note: in order to keep the program running, we overlook the safety checking here
Program test has been parsed. (0 errors 0 warnings)
Log file is test.lis

dlog>list program

Program type = No specific type
EXT married ( X00, Y01) {
        File name : married Index : None
}

INT female ( X00) {
        female ( X00) :- married ( Y10, X11 ), not male ( X20 ), X11 = X20, X00 = X11.
}
```

```
EXT male ( X00) {
        File name : male Index : None
}
```

Note: the program is fully rectified.

```
dlog>compile
Compiling Queries. SQL statements sent to test.sql file
Connected to INFORMIX user: Tian
comm=···
It will convert the NDatalog program into SQL.
comm= An available SQL sentence (described in the next sections)
Stratified took 0 :0 :3 (3 )
```

Activity	Time h:m:s (s)
Stratified	0 : 0 : 3(3)
Total Time	0 : 0 : 3(3)

```
dlog>query Query0
Query output sent to test.sqlout
unloaded 3 rows
Query0 took 0 :0 :1 (1 )
```

Activity	Time h:m:s (s)
Stratified Query0	0 : 0 : 3(3) 0 : 0 : 1(1)
Total Time	0 : 0 : 4(4)

```
dlog>quit
----===<<NDatalog Session Terminated>>===----
Disconnected from INFORMIX user: Tian
```

```
> cat test.sqlout
Result of query: Query0
______________________
mary
susan
tina
```

□

3.3 Relation DOM and Negative Facts

In Chapter 2 we introduced the closed world assumption. We know that a negated subgoal $\neg q(X_1, \ldots, X_n)$ may be interpreted as a finite relation. We define a relation DOM of arity one which is the union of the constants appearing in the EDB relations and in the rules themselves. As Ullman [Ull88] discussed , since we assume the rules are safe (Definition 2.17), no symbol not in the EDB or the rules can appear in a substitution that makes the body of rule true. Therefore, we lose nothing by restricting the relation for a negated subgoal to consist only of tuples whose values are chosen from DOM.

Thus, let Q be an arity n relation on q which has already been computed for or is an EDB predicate. Let the relation $\overline{Q}$ for subgoal $\neg q(X_1, \ldots, X_n)$ be expressed as:

$$\overline{Q} = \underbrace{DOM \times \ldots \times DOM}_{n\ \ times} - Q \tag{3.1}$$

In this thesis, I propose another way of expressing the relation $\overline{Q}$. It also uses the DOM concept, but divides DOM into DOM_{X_i} $(1 \leq i \leq n)$. Each DOM_{X_i} (X_i is the name of an attribute) is the union of the symbols appearing in the same attribute type in the EDB relations and $DOM_{X_i} = DOM_{X_j}$ if attribute X_i and X_j have the same type. So the form above can be written:

$$\overline{Q'} = DOM_{X_1} \times \ldots \times DOM_{X_i} \times \ldots \times DOM_{X_n} - Q \tag{3.2}$$

The benefit of this is shown in the next example.

Example: 3.3 Let R and S be the two relations of Figure 3.2. From the DOM definition in [Ull88], we can get a one arity relation DOM which includes all components in R and S (no duplicates, see Figure 3.3(a)). If we want to indicate the domain of a relation of arity two, we just take the Cartesian product of DOM with itself (Figure 3.3(d)). On the other hand, Figure 3.3(b) and (c) show us two relations DOM_{ID} and DOM_{Name}. We can also get a

ID	Name
1	a
2	b
1	b

(a) Relation R

ID
1
3

(b) Relation S

Figure 3.2: Two Relations

domain of a relation of arity two by taking the Cartesian product of DOM_{ID} and DOM_{Name} (Figure 3.3(e)). There will be a difference between Figure 3.3(d) and Figure 3.3(e) if the attributes of R and S are of different types. Now let us express a negative relation $\overline{R}$ using form 3.1 and form 3.2, respectively. The results are shown in Figure 3.4(a) and (b).
□

Obviously, the relation $\overline{R}$ which is created by form 3.2 (Figure 3.4(b)) is more useful, because form 3.1 creates a relation $\overline{R}$ which includes numerous tuples which contain values of the wrong type. For instance, in Figure 3.4(a), some values are of different types but appear in an attribute together. In other words, we want to create a relation $\overline{Q'}$ that adopts form 3.2, where the Cartesian product will give $(N_1 * \cdots * N_i * \cdots * N_n)$ tuples in the relation $\overline{Q'}$, where N_i is number of tuples in DOM_{X_i}, and n is number of attributes of the relation Q'. On the other hand, form 3.1 can produce N^n tuples (where N is the number of values in DOM). From the definition above, we know that $N_i \leq N$, so relation $\overline{Q'}$ may be much smaller than $\overline{Q}$. $N_i = N$ and $\overline{Q'} = \overline{Q}$, if and only if all attributes of the relation Q are of the same type.

Using $\overline{Q'}$ rather than $\overline{Q}$ does not change the answers to queries, because the rules that we discuss must be safe. This means that the relation $\overline{Q}$ could not appear in a rule's body alone and all attributes of the relation $\overline{Q}$ must be restricted by a non-negative relation R. Thus, all values of the wrong type will be deleted by the join of relation R and $\overline{Q}$, denoted $R \bowtie \overline{Q}$. For example, for the safe rule:

$$p(X,Y) : -t(X,Y), not\ r(X,Y).$$

given relation T for t and R for r, P is $T(X,Y) \bowtie \overline{R}(X,Y)$. Assume the relation $T(X,Y) = \{(1,a),(1,b),(2,a),(2,b),(3,a),(3,b)\}$, and the relation R is in Figure 3.2(a). Let us compute the rule by using the two negative relations which are shown in Figure 3.4. We get the same result that $P(X,Y) = \{(2,a),(3,a),(3,b)\}$.

X
1
2
3
a
b

(a) Relation DOM

ID
1
2
3

(b) Relation DOM_{ID}

Name
a
b

(c) Relation DOM_{Name}

X	X
1	1
1	2
1	3
1	a
1	b
2	1
2	2
2	3
2	a
2	b
3	1
3	2
3	3
3	a
3	b
a	1
a	2
a	3
a	a
a	b
b	1
b	2
b	3
b	a
b	b

(d) $DOM \times DOM$

ID	Name
1	a
1	b
2	a
2	b
3	a
3	b

(e) $DOM_{ID} \times DOM_{Name}$

Figure 3.3: Domains and Cartesian Products of Domains

ID	Name
1	1
1	2
1	3
2	1
2	2
2	3
2	a
3	1
3	2
3	3
3	a
3	b
a	1
a	2
a	3
a	a
a	b
b	1
b	2
b	3
b	a
b	b

(a) $\overline{R} = DOM \times DOM - R$

ID	Name
2	a
3	a
3	b

(b) $\overline{R} = DOM_{ID} \times DOM_{Name} - R$

Figure 3.4: Two Negative Relations

3.4 Safety in NDatalog System

Recall Definition 2.17 where we defined rules to be "safe" if all their variables were limited, either by being an argument of a nonnegated, ordinary subgoal, or by being equated to a constant or to a limited variable, perhaps through a chain of equalities. Unfortunately, when we allow the rule to have some negated subgoals, the definition is sometimes not strong enough. Consider the example below.

Example: 3.4 Recall Example 3.1 from Chapter 3:

EXT $person(X)\{\}$
EXT $eat\_meat(X,Y)\{\}$
INT $vegetarian(X)\{$
$\qquad :-person(X), not\ eat\_meat(X,Y).$
$\}$
?-$vegetarian(?X)$.
when we compute the answer using Algorithm 2.2, we find that the answer is not the one we expect, which is $vegetarian(X) = \{$ mary,peter $\}$ (See Figure 3.5).

The reason that the answer produced is wrong is that some variables appear in a negated subgoal but not in a nonnegated subgoal. □

To avoid this wrong result, we need a stronger safety definition when we have negated subgoals in the rule: we are not allowed to use negated subgoals to help prove variables to be limited.

We say the rule of Example 3.4 is not safe, since Y appears in a negated subgoal but not in a nonnegated subgoal, so it could not be limited. However, as we will see in what follows, if we meet the problem above, we can convert such a rule to a pair of safe rules that intuitively mean the same thing (see [Ull88]).

Example: 3.5 Let us rewrite the rules in Example 3.4, by creating a new rule that can project out Y from *eat_meat*, giving a definition for nonvegetarians:

EXT $person(X)\{\}$
EXT $eat\_meat(X,Y)\{\}$
INT $nonvegetarian(X)\{$
$\qquad :-eat\_meat(X,Y).$
$\}$
INT $vegetarian(X)\{$
$\qquad :-person(X), not\ nonvegetarian(X).$

person(X)

X
mary
tom
john
peter
joy

(a) Relation R

eat_meat(X,Y)

X	Y
tom	beef
john	chicken
joy	lamb

(b) Relation S

DOM_X

X
mary
tom
john
peter
joy

DOM_Y

Y
beef
chicken
lamb

¬ eat_meat(X,Y)

X	Y	X	Y
tom	chicken	mary	beef
tom	lamb	mary	chicken
john	beef	mary	lamb
john	lamb	peter	beef
joy	beef	peter	chicken
joy	chicken	peter	lamb

(c) Relation $\overline{S} = DOM_X \times DOM_Y - S$

vegetarian(X)

X
mary
tom
john
peter
joy

(d) Relation $R \wedge \overline{S}$

Figure 3.5: A Computed Result

}
?-*vegetarian*(?*X*).
□

I should mention here that our system does not perform this rewriting automatically at the moment and this will be left for further work.

3.5 Evaluating Positive Rules with SQL

The principal part of this section is a description of how to convert a fully rectified rule into SQL statements. We shall deal with only safe Datalog rules that have no negation in this section. As we aim to have a relational database system as a back-end to our NDatalog system, we should be able to convert rules into SQL expressions. We focus our attention on how to convert a rule into SQL. We first briefly introduce the most typical statement of SQL language, the so-call *query block*. An SQL block has the form:

```
SELECT   < attribute_list >
FROM     < relations >
WHERE    < predicates >
```

The SQL block has a simple interpretation in relational algebra:

- It is equivalent to performing a selection operation using the predicate of the WHERE clause.

- On the Cartesian product of the relations specified by the FROM clause.

- Projecting the result on the attributes of the SELECT clause.

It is worth noting that the Cartesian product reduces to a join if the predicate of the WHERE clause includes the join condition and no Cartesian product is required if the FROM clause contains just one relation.

Consider the Datalog rule

$$p(\overline{X_0}) : -q_1(\overline{X_1}), \ldots, q_n(\overline{X_n}), \delta \tag{3.3}$$

where δ is a list of built-in predicates of the form x_1 *op* x_2, where *op* is one of $>, \geq, <, \leq, =$ or $\neq$, and x_i is a variable or a constant (no functions are allowed).

We convert the rule into SQL expressions by calling the function SQL(). Consider the form 3.3, assume the relations $R_{q_i}(1 \leq i \leq n)$ are already computed, we obtain the SQL expression

$$\begin{array}{ll} \text{SELECT} & alias_{m_1}.Y_{m_1}, \ldots, alias_{m_k}.Y_{m_k} \\ \text{FROM} & R_{q_1}\ alias_1, \ldots, R_{q_n}\ alias_n \\ \text{WHERE} & \delta'' \end{array}$$

where

- Each variable in a predicate occurrence is mapped onto its *underlying attribute* taken from the predicate's definition. This is the attribute that appears in the same position in the head of the definition as the variable does in the predicate occurrence (see the end of Example 3.7 for an example of this). Each variable in the rule has a unique underlying attribute because the rule is fully rectified.
- $alias_i$ is a unique alias for q_i. We have to use aliases as there may be more than one occurrence of a predicate in the body of the rule.
- Y_{m_i} is the underlying attribute of the first body variable that is equated to the head variable X_{m_i} while $alias_{m_i}$ is the alias of the predicate occurrence (i.e. q_{m_i}) in which the body variable appears.
- δ'' is the same as δ except that every variable, X_i is replaced with $alias_i.X_i$, where $alias_i$ is the alias of the predicate occurrence containing x_i, and δ'' is a list of comparisons involving only the variables mentioned in the non-built-in predicates and the comparisons in δ'' are linked together with "AND"s.

Example: 3.6 Consider the rule

$$manager(Ename, Mname) : -emp(Ename, Dept), dept(Mname, Dept).$$

We convert the rule into the SQL expression

$$\begin{array}{ll} \text{SELECT} & emp_1.Ename,\ dept_2.Mname \\ \text{FROM} & emp\ emp_1,\ dept\ dept_2 \\ \text{WHERE} & emp_1.Dept = dept_2.Dept \end{array}$$

□

3.5.1 Evaluating Nonrecursive Rules with SQL

Because the rules are rectified, we project each expression onto the same variables of the rule's head and, for each predicate, take the union of the expression produced from each of its rules. On encountering a nonrecursive intensional definition R_p, the NDatalog system generates an SQL "CREATE TABLE" statement. If R_p has attributes $X_1, \ldots, X_k$, and is defined by rules $R_{p_1}, \ldots, R_{p_n}$, then SQL(R_p) is the expression:

```
CREATE TABLE  $R_p$ ($X_1$ char(20), ..., $X_k$ char(20))[1];
INSERT INTO   $R_p$
              SQL($R_{p_1}$)
              UNION
...
              UNION
              SQL($R_{p_n}$);
```

where, for each $i = 1, \ldots, n$, SQL(R_{p_i}) is

```
SELECT  $R_{p_i}.X_1$, ..., $R_{p_i}.X_k$
FROM    $R_{p_i}$
WHERE   $R_{p_i}.X_1 = R_p.X_1$ AND ... AND $R_{p_i}.X_k = R_p.X_k$
```

Example: 3.7 Consider a Datalog program

```
EXT  $manager(Name, Dept)\{\}$
EXT  $dept(Enum, Dept)\{\}$
EXT  $emp(Ename, Enum)\{\}$

INT  $job(Name, Dept)\{$
     $:-emp(Name, Enum), manager(Name, Dept), Enum <' 20'.$
     $:-emp(Name, Enum), dept(Enum, Dept).$
}

?-   $job(?Name, ?Dept).$
```

We convert the program into the SQL expressions

[1]In order to simplify the system, we only consider the character type. The focus of the work was on evaluating rules with negation.

```
CREATE TABLE job  (Name char(20), Dept char(20));
INSERT INTO job   SELECT empl.Ename, manager2.Dept
                  FROM emp empl, manager manager2
                  WHERE empl.Ename = manager2.Name
                  AND empl.Enum < '20'
                  UNION
                  SELECT empl.Ename, dept3.Dept
                  FROM emp empl, dept dept3
                  WHERE empl.Enum = dept3.Enum;

CREATE TABLE      Query0 (Name char(20), Dept char(20));
INSERT INTO       Query0 SELECT DISTINCT jobl.Name, jobl.Dept
                  FROM job jobl;
```

□

Consider a Datalog program with given EDB relations $R_1, \ldots, R_k$ and with positive IDB relations $P_1, \ldots, P_m$ to be computed. For each i, $(1 \le i \le m)$, we could have an evaluation statement of the form

$$P_i := EVAL(p_i, R_1, \ldots, R_k, P_1, \ldots, P_m)$$

where $EVAL$ is the union of SQL() for each of the rules for p_i. This form indicates an SQL expression as discussed above.

3.5.2 Evaluating Recursive Rules with SQL

On encountering a recursive node in the reduced dependency tree, Section 2.3.4 described two methods: naive and semi-naive. Corresponding algorithms were given also. In this subsection, the main difference is that we evaluate recursive rules with SQL rather than relational algebra. Let us see how to evaluate a recursive rule by the semi-naive method.

Example: 3.8 Recall the *ancestor* example (Example 2.9). The base is the following SQL expression: we use the ordinary $EVAL$ operation to compute the rules without IDB predicates in the body (see Figure 2.4).

```
DROP TABLE      DELTAancestor;
CREATE TABLE    DELTAancestor (X char(20), Y char(20)) ;
INSERT INTO     DELTAancestor (X, Y)
                SELECT parent1.X, parent1.Y
                FROM parent parent1;
DROP TABLE      ancestor;
CREATE TABLE    ancestor (X char(20), Y char(20)) ;
INSERT INTO     ancestor
                SELECT DISTINCT * FROM DELTAancestor ;
```

Next we enter the repeat loop. Here we call the *EVAL_SQL*() function which evaluates an SQL expression that is equivalent to the *EVAL_INCR*() function given in Section 2.3.4.

```
DROP TABLE      OLDancestor;
CREATE TABLE    OLDancestor (X char(20), Y char(20)) ;
INSERT INTO     OLDancestor
                SELECT DISTINCT * FROM DELTAancestor;
INSERT INTO     DELTAancestor (X, Y)
                SELECT DISTINCT ancestor1.X, parent2.Y
                FROM OLDancestor ancestor1, parent parent2
                WHERE ancestor1.Y = parent2.X;
DELETE FROM     DELTAancestor WHERE EXISTS (
                SELECT *
                FROM ancestor
                WHERE DELTAancestor.X = ancestor.X
                AND DELTAancestor.Y = ancestor.Y );
INSERT INTO     ancestor
                SELECT * FROM ancestor
                UNION
                SELECT * FROM DELTAancestor;
```

We repeat this until *DELTAancestor* is empty, after which we output *ancestor*. Because no MINUS function exists in INFORMIX-SQL, we replace MINUS with the DELETE function. We also can use the SELECT function instead of the DELETE function to implement a MINUS evaluation, the SQL expression is

```
INSERT INTO  DELTAancestor (X, Y)
             SELECT DISTINCT ancestor1.X, parent2.Y
             FROM ancestor ancestor1, parent parent2
             WHERE ancestor1.Y = parent2.X;
             AND NOT EXISTS (
             SELECT *
             FROM ancestor
             WHERE DELTAancestor.X = ancestor.X
             AND DELTAancestor.Y = ancestor.Y );
```

We prefer to use the DELETE function because it is more intuitive and the running times of the two methods are similar. □

3.6 Evaluating Negative Rules with SQL

It is time to discuss how to evaluate rules with negation. First I give a method to create DOM using SQL sentences. Assume $R_1(X_1, \ldots, X_m), \ldots, R_k(X_1, \ldots, X_n)$ are EDB relations, then

```
CREATE TABLE  DOM(X char(20));
INSERT INTO   DOM(X)
SELECT        R1alias.X1
FROM          R1 R1alias
UNION
...
UNION
SELECT        R1alias.Xm
FROM          R1 R1alias
UNION
...
UNION
SELECT        Rkalias.Xn
FROM          Rk Rkalias
```

Let $Q(X_1, \ldots, X_n)$ be an IDB relation for q, then DOM$_q$ is

```
CREATE TABLE  DOMq(X1 char(20),...,Xn char(20));
INSERT INTO   DOMq(X1,...,Xn)
SELECT        DOM1.X,...,DOMn.X
FROM          DOM DOM1,...,DOM DOMn
```

Now, let us put negative subgoals into the form (3.3) as

$$p(\overline{X_0}) : -q_1(\overline{X_1}), \ldots, q_i(\overline{X_i}), not\ q_{i+1}(\overline{X_{i+1}}), \ldots, not\ q_n(\overline{X_n}), \delta$$

We derive the $SQL^{\neg}$ expression

```
SELECT   $alias_{m_1}.Y_{m_1}, \ldots, alias_{m_k}.Y_{m_k}$
FROM     $q_1\ alias_1,\ \ldots,\ q_i\ alias_i,\ not\ q_{i+1}\ alias_{i+1},\ \ldots,\ not\ q_n\ alias_n$
WHERE    $\delta''$
```

where *not* $q_j(\overline{X_j})$ $(i+1 \leq j \leq n)$ can be obtained by:

1. creating DOM_{q_j}, and
2. subtracting q_j from DOM_{q_j}.

We have discussed above how to create a DOM relation using SQL. The subtraction is done as follows:

```
DELETE FROM     $DOM_{q_j}$
WHERE EXISTS (
SELECT          $q_j1.Y_1, \ldots, q_j1.Y_k$
FROM            $q_j\ q_j1$
WHERE           $DOM_{q_j}.Y_1 = q_j1.Y_1$ AND ... AND $DOM_{q_j}.Y_k = q_j1.Y_k$ );
```

The only difference to the previous expression is that the negated subgoals are involved and we must create the relation for the negated subgoals. It can easily be solved by creating a relation DOM as we discussed in Section 3.3.

Let us recall Algorithm 2.1 and modify it for a rule body with negation, so we get an algorithm as follows.

Algorithm: 3.1 Computing the Relation for a Rule Body with Negation.

INPUT: The body of a rule.

OUTPUT: An expression of SQL, which we call

$$SQL^{\neg}(R_p, R_{q_1}, \ldots, R_{q_j}, \overline{R_{q_{j+1}}}, \ldots, \overline{R_{q_n}})$$

METHOD: The same as Algorithm 2.1, except that if there is a subgoal *not* $q(X_1, \ldots, X_n)$, we use a negative relation expression in SQL as shown above. □

We can get an algorithm, Nonrec$^\neg$, by modifying Algorithm 2.2 for the nonrecursive rules with negation.

Algorithm: 3.2 (Nonrec$^\neg$) Evaluating Nonrecursive Rules with Negation Using SQL.

INPUT: A nonrecursive NDatalog program and a relation for each EDB predicate appearing in the program.

OUTPUT: An expression for each IDB predicate p.

METHOD: Same as Algorithm 2.2, but each P_i is given by

$$P_i := EVAL^\neg(p_i, R_1, \ldots, R_k, P_1, \ldots, P_j, \overline{P_{j+1}}, \ldots, \overline{P_m})$$

where $EVAL^\neg$ is the union of $SQL^\neg$ for each of the rules for p_i (Algorithm 3.1). □

For recursive rules with negation, we can modify Algorithms 2.4 (Naive) and 2.6 (Semi-naive) to Naive$^\neg$ and Semi-naive$^\neg$, respectively.

Algorithm: 3.3 (Naive$^\neg_{scc}$) Naive Evaluation of an SCC with Negation.

INPUT: Rules for an SCC, the current values for the IDB predicates, the true EDB values, and the not true EDB values. We use four input arguments because they are needed in the algorithm for the well-founded semantics presented in Section 3.6.3. In this case, the IDB predicates are each initialized to the empty set (as in Algorithm 2.3) and the not-true facts in the EDB relations are just the complement of the true facts in the EDB relations with respect to DOM. That is, we call

$$Naive^\neg_{scc}(SCC, \emptyset, \{R_1, \ldots, R_k\}, \{\overline{R_1}, \ldots, \overline{R_k}\})$$

METHOD: Same as Algorithm 2.3, but it must call Algorithm 3.2 instead of Algorithm 2.2. We replace $EVAL()$ by $EVAL^\neg()$. □

Algorithm: 3.4 (Naive$^\neg$) Naive Evaluation of a Datalog Program with Negation.

METHOD: Same as Algorithm 2.4, but it must call Algorithm 3.3 for each SCC. □

Algorithm: 3.5 (Semi-naive$^\neg_{scc}$) Semi-Naive Evaluation of an SCC with Negation.

METHOD: Same as Algorithm 2.5, but it must call Algorithm 3.2 not Algorithm 2.2. Here we replace $EVAL()$ and $EVAL\_INCR()$ by $EVAL^\neg()$ and $EVAL\_SQL^\neg()$, where $EVAL\_SQL^\neg()$ is $EVAL\_INCR()$ evaluation allowing negative predicates in a rule's body. □

Algorithm: 3.6 (Semi-naive$^\neg$) Semi-Naive Evaluation of a Datalog Program with Negation.

METHOD: Same as Algorithm 2.6, but it must call Algorithm 3.5 for each SCC. □

Example: 3.9 Consider a simple stratified NDatalog program:

$$\begin{array}{lll} r_1: & \mathrm{p(X)} & :-\ \mathrm{r(X)}. \\ r_2: & \mathrm{q(X)} & :-\ \mathrm{s(X),\ not\ p(X)}. \end{array}$$

Let EDB predicates r and s have corresponding relations R and S, and let IDB predicates p and q have relations P and Q. Suppose $R = \{1\}$ and $S = \{1,2\}$. Obviously, this is a nonrecursive program with negation. Therefore we use Algorithm 3.2, starting with predicate p because the subgoal *not p* appears in the body of r_2. We use the ordinary $EVAL^\neg$ operation to compute r_1, so relation P gets tuple $\{1\}$, and at same time, we get $\overline{P} = DOM - P = \{2\}$ because $DOM = \{1,2\}$. The next stage of the evaluation is similar, so that relation Q contains tuple $\{2\}$ from r_2. Repeating the above evaluation, the program can no longer yield new tuples so we reach the least fixpoint. □

In Chapter 2, we surveyed different techniques to integrate negation in logic programs. We chose three semantics, namely, stratified, well-founded and inflationary to compute logic programs in order to compare them. In the next subsections, our main aim is to describe the implementation of the different semantics in the NDatalog system.

3.6.1 Evaluating stratified semantics

In the previous chapter, we have discussed when a program is stratified from a theoretical point of view. An algorithm that tests for and finds a stratification which is implemented in NDatalog is in Appendix A. Here we introduce a way to test for and find a stratification from graph theory.

Definition: 3.2 We say a logic program P is stratified iff there are no cycles which contain a negative edge in its definition dependency graph. We can get a stratification from the acyclic graph as follows:

- All definition nodes are assigned to stratum 1 at the beginning.
- Assume there is an edge from p to q ($p \longrightarrow q$) and let p and q currently be assigned to strata i and j respectively. If the edge is positive and $j < i$, then reassign q to stratum

i (no change for $j \geq i$). If the edge is negative and $j \leq i$, then reassign q to stratum $i+1$ (no change for $j > i$).

Recall Example 2.13, where the definition dependency graph is shown in Figure 3.6. A stratification is given by the numbers labeling the nodes. □

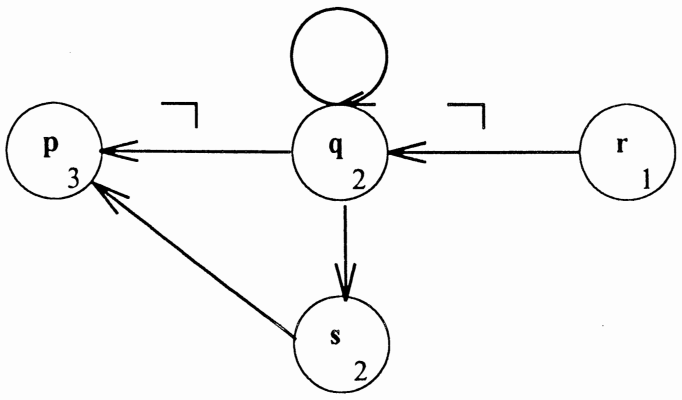


Figure 3.6: Testing For And Finding A Stratification Dependency Graph

The main point mentioned before is that once we get an order for the program, we can evaluate the program. We now focus our attention on how to evaluate a stratified program in the NDatalog system. The algorithm is as follows:

Algorithm: 3.7 Evaluation of Stratified Semantics. (NDatalog$_{\text{stra}}$)
INPUT: An NDatalog program.
OUTPUT: The solution to the relational equations obtained from the program.
METHOD: Let the function TFS() be testing for and finding a stratification which decides whether the program is stratified and if it is, produces a stratification which groups the predicates into strata. Let function max-strata() return the maximum number of strata. The TFS function also orders rules within a stratum on the basis of the rule dependence graph, so that evaluating rules in the given order yields the correct result. The strata give us an order in which the relations for the IDB predicates may be computed. The useful property of this order is that following it, we may treat any negated subgoals as if they were EDB relations. For a set of recursive rules, we use the semi-naive evaluation approach in Algorithm 3.5, otherwise we evaluate rules directly.

```
begin
    TFS(program);
    if non-stratified program
```

```
            print error information;
        else
            for i:= 1 to max-strata(program) do begin
                    if (nonrecursive)
                        Call Algorithm 3.2 to evaluate the rules
                        in the order generated by TFS;
                    else
                        Call Algorithm 3.5 to evaluate the rules using Semi-naive¬
                        evaluation in the order generated by SCC;
            endfor
            output all IDB predicates computed results;
        endif
end
```

□

3.6.2 Evaluating inflationary semantics

[GS86b] [KP91] [AV91] proposed and analyzed another evaluation approach which can compute programs that involve unstratified negation, named *inflationary* semantics. [Bid91] described two different evaluation possibilities for inflationary semantics: deterministic and non-deterministic. Roughly speaking, deterministic semantics corresponds to "apply ALL rules" at once; non-deterministic semantics corresponds to "apply ONE rule" at a time. Intuitively, the result of non-deterministic semantics depends on the position of rules in the program. Therefore, we implemented the deterministic semantics.

The way that deterministic evaluation of the inflationary semantics proceeds is: consider first the empty Herbrand interpretation and apply all rules whose premises are satisfied by the empty interpretation at once. Then repeatedly apply all rules which are satisfied by the last interpretation until no new interpretation appears. A concrete algorithm for deterministic inflationary semantics implemented in NDatalog is shown below.

Algorithm: 3.8 Evaluation of Inflationary Model Semantics. (NDatalog$_{\text{infl}}$)
INPUT: An NDatalog program with IDB predicates $p_1, \ldots, p_m$.
OUTPUT: A solution to the relational equations obtained from the program.
METHOD: Similar to the Algorithm 3.3 (Naive$^{\neg}_{scc}$).

```
begin
        for i := 1 to m do begin
        $P_i := \emptyset$;
    endfor
    repeat
        for i := 1 to m do begin
            $OLD\_P_i = P_i$;
            if ($P_i$ is used in a negative subgoal)
                $\overline{P_i} := DOM_{P_i} - P_i$;
        endfor
        for i := 1 to m do begin
            $TEMP\_P_i := EVAL^{\neg}(p_i, R_1, \ldots, R_k, P_1, \ldots, P_j,$
                        $\overline{P_{j+1}}, \ldots, \overline{P_m})$;
        endfor
        for i := 1 to m do begin
            $P_i := P_i \cup TEMP\_P_i$;
        endfor
    until($OLD\_P_i = P_i$ for all $(1 \leq i \leq m)$);
output computed results for all IDB predicates;
end
```

□

Example: 3.10 Consider the logic program P which we used in chapter 2. Assume that we have an EDB relation $LINK = \{(a,b),(a,c),(b,d),(d,e)\}$.

$$\begin{array}{ll} r_1\text{:} & path(X,Y) :- link(X,Y). \\ r_2\text{:} & path(X,Y) :- path(X,Z), link(Z,Y). \end{array}$$

Initially, the relation $PATH$ is empty as we discussed above. On the first round, the deterministic evaluation leads to the relation $TEMP\_PATH$ which is $\{(a,b),(a,c),(b,d),(d,e)\}$, so the relation

$$PATH = TEMP\_PATH = \{(a,b),(a,c),(b,d),(d,e)\}.$$

On the second round,

$$TEMP\_PATH = (PATH \bowtie LINK) \cup LINK = \{(a,b),(a,c),(b,d),(d,e),(a,d),(b,e)\}.$$

At the end of this round,

$$PATH = \{(a,b),(a,c),(b,d),(d,e),(a,d),(b,e)\}.$$

On the third round, only one new tuple (a, e) is produced from r_2 so that

$$PATH = \{(a,b),(a,c),(b,d),(d,e),(a,d),(b,e),(a,e)\}$$

at the end of this round. On the fourth round, no more new tuples be produced from the program and the repeat loop terminates. □

Example: 3.11 Let us consider another example which we used in chapter 2, Example 2.14, and an EDB $D = \{d(1)\}$.

$$\begin{array}{ll}
r_1: & s(X) : -p(X), q(X),\ \mathit{not}\ r(X). \\
r_2: & p(X) : -d(X),\ \mathit{not}\ q(X). \\
r_3: & q(X) : -d(X),\ \mathit{not}\ p(X). \\
r_4: & r(X) : -d(X), d(0).
\end{array}$$

Initially, the relations S, P, Q, R are empty and $\overline{S} = \{1\}$, $\overline{P} = \{1\}$, $\overline{Q} = \{1\}$, $\overline{R} = \{1\}$. On the first round, we get two new tuples from r_2 and r_3, so the relations $P = \{1\}$, $Q = \{1\}$ and $\overline{P} = \{\emptyset\}$, $\overline{Q} = \{\emptyset\}$. On the second round, a new tuple, $s(1)$ is produced from r_1, so $S = \{1\}$ and $\overline{S} = \{\emptyset\}$. On the third round, no more new tuples are produced from the program and the procedure stops with the result $\{p(1), q(1), s(1)\}$. □

It is worth noting that when we use Algorithm 3.8 to compute some NDatalog programs, the inflationary semantics of the program does not correspond to the intuitive meaning of the program. However, the programs can be easily modified so that their inflationary semantics corresponds to the intuitive definition. We will discuss more about this in the next chapter.

3.6.3 Evaluating well-founded semantics

Some basic principles of the well-founded semantics have been introduced in Chapter 2. As we know, not all logic programs are stratified and we sometimes need recursion through negative predicate occurrences.

Example: 3.12 Let us consider an example discussed in [Bid91]. There is a NDatalog program P that defines an *even* number for a finite subset of the natural numbers, say for the natural numbers from 0 to i. The order on natural numbers is represented by means of an EDB relation $SUC = \{(0,1),(1,2),\ldots,(i-1,i)\}$ (instead of using a function), and the other EDB relation is $EVEN0 = \{0\}$. The NDatalog program is then written:

$$\begin{array}{lll}
r_1: & even(X) & : -even0(X). \\
r_2: & even(X) & : -suc(Y,X), \mathit{not}\ even(Y).
\end{array}$$

Obviously, this is a nonstratified program. Let us follow Algorithm 3.5 (Semi-naive$^{\neg}_{scc}$) when computing the program. At first, relation $EVEN$ gets only the tuple $\{0\}$ from r_1 and

$$\overline{EVEN} = \overline{EVEN}.delta = DOM - EVEN = \{1, 2, \ldots, i\}$$

on the first round. Then we will through the repeat-loop for the first time, where we are going to contribute some tuples to $EVEN.delta$. These tuples then find their way into $EVEN$ at the end of repeat-loop. That is, on second round we compute:

$$\begin{aligned} EVEN.delta &= EVAL\_SQL^{\neg}(even, EVEN0, SUC, EVEN.delta, \overline{EVEN}.delta) \\ &= \Pi_X(SUC(Y,X) \bowtie \overline{EVEN}(Y).delta) \end{aligned}$$

After removing those tuples that actually appeared before, the relation $EVEN.delta$ contains $\{2, 3, \ldots, i\}$. At the end of this round, the relation $EVEN = EVEN \cup EVEN.delta = \{0, 2, \ldots, i\}$ and $\overline{EVEN} = \overline{EVEN}.delta = DOM - EVEN = \{1\}$. On the third round, $EVEN.delta = EVEN.delta - EVEN = \{2\} - \{0, 2, \ldots, i\} = \emptyset$, and this stops the repeat loop. Finally, the relation $EVEN$ is $\{0, 2, 3, \ldots, i\}$ and this is not what we we intuitively expect the rules should mean. □

This result shows that Algorithm 3.5 is not correct for all kinds of NDatalog programs. In fact, Algorithm 3.5 does not handle the case in which the dependency graph contains a cycle in which an edge is labeled with "$\neg$". Fortunately, [GRS88] proposed the well-founded model which is suitable for every program with negation, not just stratified programs. In [KSS91], they describe an algorithm to evaluate the well-founded models. The evaluation method for computing the well-founded model semantics is based on the *doubled program*.

Definition: 3.3 Consider a clause C:

$$P_0 \leftarrow P_1, \ldots, P_n, \neg P_{n+1}, \ldots, \neg P_{n+m}.$$

in any NDatalog program P. In the *doubled program* $D(P)$, C is represented by precisely two clauses:

1. an undashed clause:

$$P_0 : -P_1, \ldots, P_n, \neg P'_{n+1}, \ldots, \neg P'_{n+m}.$$

2. and a dashed clause:

$$P'_0 : -P'_1, \ldots, P'_n, \neg P_{n+1}, \ldots, \neg P_{n+m}.$$

The set of undashed clauses is termed the "undashed half" of the doubled program, while the rest are termed its "dashed half". □

The intuition behind this procedure is to compute the well-founded model of P using the two sets of clauses. One computes the true facts, the other computes the complement of the false facts. Each clause is positive if we consider the negated predicates to be fixed, so we can compute the fixpoint of each one using standard bottom-up techniques for programs without negation. Next we present a bottom-up operational procedure for computing the well-founded model in our NDatalog system. This procedure provides a practical method of handling all kinds of programs.

Algorithm: 3.9 Evaluation of Well-Founded Model Semantics. (NDatalog$_{\text{wellf}}$)
INPUT: An NDatalog program.
OUTPUT: A solution to the relational equations obtained from the program.
METHOD: First, compute the order of definition nodes for the program by the Strongly Connected Components (SCC) algorithm (see Appendix A.2). Group the rule nodes into four groups below, so we also get an order of rule nodes. Use EDB relations to compute the DOM relation. Assume

- E^+ represents the true facts in lower strata.
- E^- represents the not false facts in lower strata.
- I^+ represents the true facts in current stratum (Strongly Connected Component).
- I^- represents the not false facts in current stratum.

I^+ and I^- refer to the results of evaluations each time around the loop, and the current evaluation (assume I^+) is based on the previous results (assume I^-), and *vice versa*. We repeat the doubled computing until there is no change in I^+.

We divide the rules of each SCC in the NDatalog program into four parts:

rule1:	Rules with no IDB predicates in the body.
rule2:	Rules with some IDB predicates in the body.

```
begin
    $E^+ := E^- :=$ all EDB relations;
    SCC(program);
    n:=max-number-by-SCC(program);
    for i:= 1 to n do begin
        $I^+ := I^- := I^+.old := \emptyset$;
```

```
            $I^+$ :=$Naive^{\neg}_{scc}$(rule1, $\emptyset$, $E^+$, $E^-$);
            **repeat**
                    $I^+.old := I^+$;
                    $I^-$:=$Naive^{\neg}_{scc}(rule2, I^+, E^+, E^-) \cup I^+.old$;
                    $I^+$:=$Naive^{\neg}_{scc}(rule2, I^-, E^+, E^-) \cup I^+.old$;
            **until**($I^+ = I^+.old$);
            $E^+$:=$E^+ \cup I^+$;
            $E^-$:=$E^- \cup I^-$;
    **endfor**
    output computed results of all IDB predicates in $E^+$;
**end**
```

$Naive^{\neg}_{scc}()$ is described in Algorithm 3.3[2].
□

Example: 3.13 Now let us use Algorithm 3.9 to evaluate Example 3.12 again. As a matter of convenience, we use the natural numbers from 0 to 9. Figure 3.7 illustrates the evaluation process. For this example, the function SCC() only finds one strongly connected component. Hence, the algorithm executes the for loop once. The repeat loop executes five times as shown in the figure. □

[2]In [KSS91], they used semi-naive evaluation.

	$EVEN^{+}.old$	$EVEN^{+} = EVEN^{+}$ $\cup EVEN^{+}.old$	$\neg EVEN^{+}$	$EVEN^{-} = EVEN^{-}$ $\cup EVEN^{+}.old$	$\neg EVEN^{-}$
initial	—	0	1,2,...,9	—	—
1	0	—	—	0,2,3,...,9	1
	0	0,2	1,3,4,...,9	—	—
2	0,2	—	—	0,2,4,5,...,9	1,3
	0,2	0,2,4	1,3,5,6,...,9	—	—
3	0,2,4	—	—	0,2,4,6,7,...,9	1,3,5
	0,2,4	0,2,4,6	1,3,5,7,8,9	—	—
4	0,2,4,6	—	—	0,2,4,6,8,9	1,3,5,7
	0,2,4,6	0,2,4,6,8	1,3,5,7,9	—	—
5	0,2,4,6,8	—	—	0,2,4,6,8	1,3,5,7,9
	0,2,4,6,8	0,2,4,6,8			

Figure 3.7: Calculating the well-founded model

Chapter 4

Testing and Results

In the last chapter we described the evaluation algorithms for three semantics of negation. In this chapter, we will test the various semantics embodied in NDatalog based on various programs and analyse the results. From a theoretical point of view some comparisons have been done [Bid91] [AV91] [KP91] [CGT89] that we mentioned in chapter 2. We will compare the efficiency of the various semantics through an empirical study in this chapter. For the sake of further discussion, we first divide the logic programs into two groups: restricted logic programs (stratified programs) and nonrestricted logic programs (nonstratified programs). For each test, I give the program used and a table in which the times for each column are the averages obtained from 10 test runs.

4.1 The XSB system

The XSB system (version 1.4.0) is a top-down evaluation system for Prolog programs which has been developed by the Department of Computer Science, SUNY at Stony Brook, USA. There is a meta-interpreter to compute the well-founded semantics in the XSB system; it is based on the XOLDTNF algorithm[1], which can be exponential. In order to check the correctness of our $\text{NDatalog}_{\text{wellf}}$ evaluation, we chose the XSB system as our comparison tool. We found the results of $\text{NDatalog}_{\text{wellf}}$ and XSB are the same when we tested using various $\text{Datalog}^{\neg}$ programs. Meanwhile, there are also some time comparisons shown in the tables which follow. XSB is not coupled to an external database and evaluates programs containing negation in a top-down manner. As a result, we expected it to be faster than our system. It is used so that we may check the correctness of results produced by our system, as well as compare the speed of evaluation to determine the feasibility of our system as an

[1]This algorithm is an extension of the OLDT algorithm and it computes, under certain conditions, the well-founded semantics of general logic programs. See [War91] for more details.

efficient alternative.

4.2 Testing restricted programs on NDatalog

In this section, I present a few tests I made on the system in order to compare the efficiency of NDatalog$_{\mathbf{stra}}$ with the others. First, I discuss the tests for recursive and non-recursive programs, then I give test results, and finally I discuss the query processing time required using different semantics in the NDatalog system.

4.2.1 Restricted programs without negation

In order to compare more clearly, let us first consider the program below.

Example: 4.1 This is the common problem of computing the transitive closure (TC) of a directed graph. Define an EDB predicate $arc(X, Y)$ which states that there is an arc from node X to node Y. Then we can express the paths in the finite graph by the rules:

$$\begin{array}{ll} r_1: & path(X,Y) :- arc(X,Y). \\ r_2: & path(X,Y) :- path(X,Z), arc(Z,Y). \end{array}$$

Assume the arcs relation is given by $ARC(X,Y) = \{(0,1),(1,2),\ldots,(i-1,i)\}$, where i is a finite positive integer. Then what will happen when we choose different semantics to

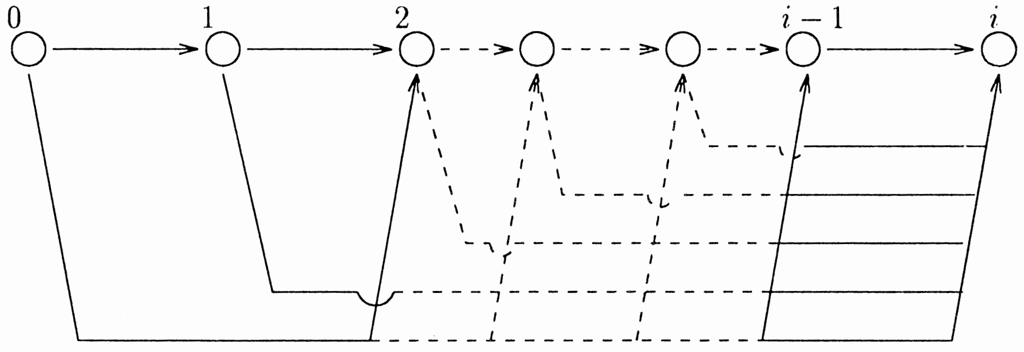


Figure 4.1: A path graph

compute the path relation? The result is that we get the same answer no matter what kind of semantics are chosen. The result is as shown in Figure 4.1.

$$\begin{aligned} path(X,Y) &= \{(0,1),(0,2),\ldots,(0,i),(1,2),(1,3),\ldots,(1,i), \\ &\quad \ldots,(i-1,i)\} \end{aligned}$$

In this case, we can say that NDatalog$_{\text{stra}}$ has the same expressive power with NDatalog$_{\text{wellf}}$ and NDatalog$_{\text{infl}}$ for the Datalog programs without negation. The times for testing on the NDatalog system using the various semantics, as well as XSB, are shown in Table 4.1 and Figure 4.2. □

T(s)	$NDatalog_{stra}$	$NDatalog_{wellf}$	$NDatalog_{infl}$	XSB	Result tuples
					path
i=10	6.33	5.00	7.00	1.20	55
i=20	18.70	15.30	22.67	3.50	210
i=30	72.14	54.70	77.22	7.66	465
i=40	172.46	109.28	153.09	12.50	820
i=50	309.75	194.10	280.07	20.20	1275

Table 4.1: The times (in seconds) for a recursive program without negation

From Table 4.1 we can see that in this case XSB is much faster than NDatalog$_{\text{stra}}$, NDatalog$_{\text{wellf}}$ and NDatalog$_{\text{infl}}$ when the databases grow in size. At the same time, NDatalog$_{\text{stra}}$ and NDatalog$_{\text{infl}}$ take longer than NDatalog$_{\text{wellf}}$. It should be pointed out that the times spent on executing the stratified program, NDatalog$_{\text{stra}}$ should be faster than NDatalog$_{\text{wellf}}$, but we get the opposite result. For NDatalog$_{\text{stra}}$, we use the semi-naive algorithm which includes a MINUS operation to remove "new" tuples that appeared before in the repeat-loop (see Figure 2.4):

$$\Delta P(X) := \Delta P(X) - P(X)$$

But the problem is that there is no MINUS function in the INFORMIX-SQL language, so we have to use another function instead. For example, we can use the DELETE or SELECT functions as follows:

```
DELETE FROM    ΔP
WHERE EXISTS   ( SELECT   P.X
                 FROM     P
                 WHERE    P.X = ΔP.X )
```

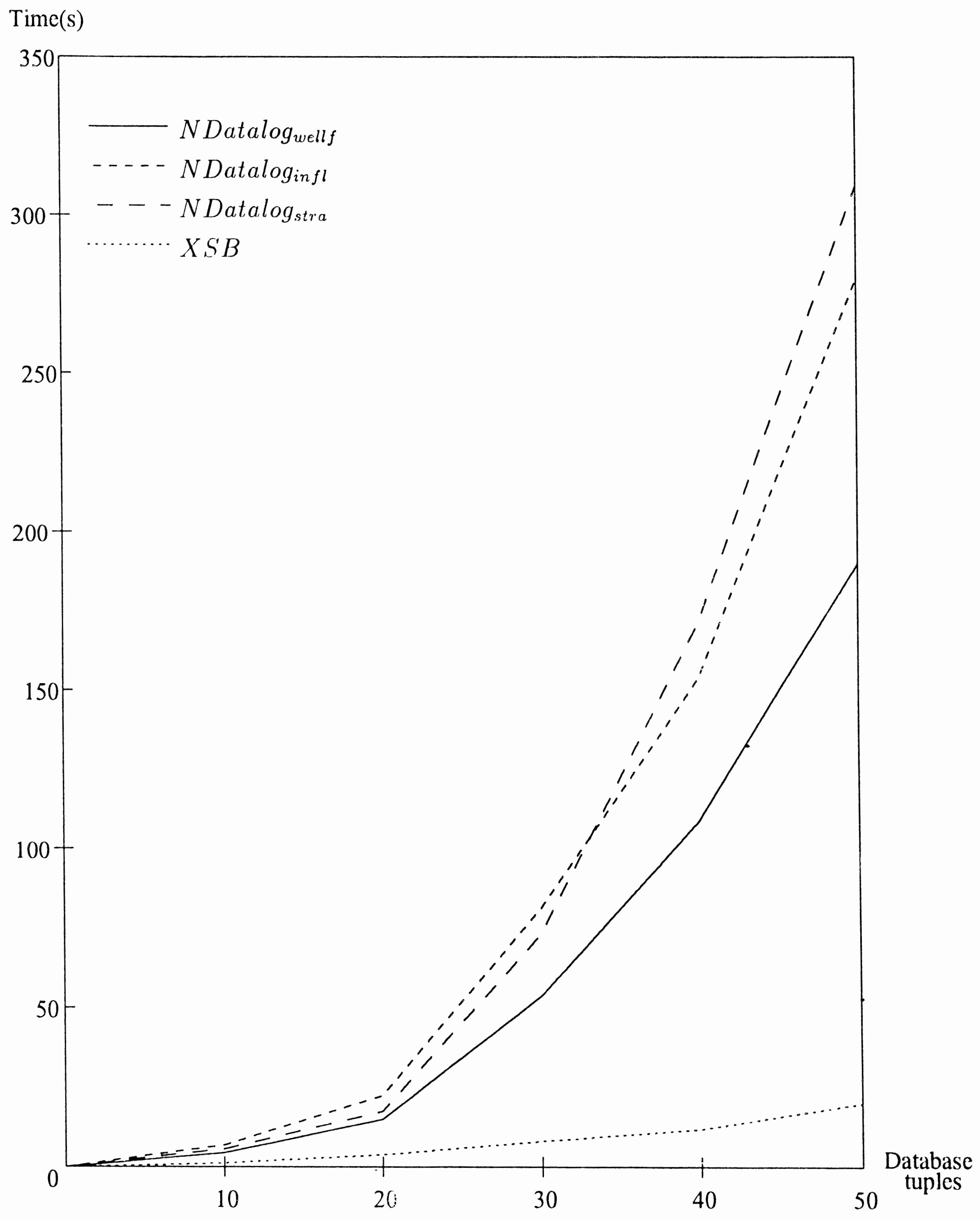


Figure 4.2: A times comparison graph for a recursive program without negation

or

```
INSERT INTO       ΔP
SELECT X FROM     ΔP
WHERE             ΔP.X = P.X
                  AND NOT EXISTS (
                  SELECT          X
                  FROM            P
                  WHERE           P.X = ΔP.X )
```

It turns out that these functions are very time consuming. In our system, we use the DELETE function as we mentioned in chapter 3. The times we obtained when using NOT EXISTS were similar to those with DELETE.

For $\text{NDatalog}_{\text{infl}}$, the problem is that the algorithm duplicates evaluation of parts of the program. For example, if a rule only involves EDB predicates in its body, we should evaluate it only one time because we cannot get more new facts from the rule. However, $\text{NDatalog}_{\text{infl}}$ will evaluate it at least twice. The check for equality of relations is also expensive in the repeat-loop and the number of $\text{NDatalog}_{\text{infl}}$ iterations is more than for $\text{NDatalog}_{\text{wellf}}$. We should also point out that for this example the MINUS operation does not improve efficiency since only new tuples are generated on each iteration.

There are time proportion graphs shown in Figures 4.3, 4.4 and 4.5 for stratified, well-founded and inflationary semantics, respectively. These graphs represent the percentage of time spent on various stages of the evaluation. From Figure 4.3, it is easy to see how expensive it is to imitate the MINUS function in INFORMIX-SQL. If we only consider the time of rule evaluation in Figures 4.3, 4.4 and 4.5, the result is given in Table 4.2[2]. For $\text{NDatalog}_{\text{stra}}$ and $\text{NDatalog}_{\text{infl}}$ the proportion of time is given by *eval* in Figure 4.3 and Figure 4.5, respectively. For $\text{NDatalog}_{\text{wellf}}$, the proportion of time is given by both *eval* P^- and *eval* P^+ in Figure 4.4. There is a time comparison graph shown in Figure 4.6. The relative efficiency of the three semantics is now more in line with our expectations.

As another experiment, I also wrote another program using naive evaluation with stratified semantics ($\text{NDatalog}^{\text{naive}}_{\text{stra}}$) to remove the problem of the MINUS operation. We would now expect the times for $\text{NDatalog}_{\text{wellf}}$ and $\text{NDatalog}^{\text{naive}}_{\text{stra}}$ to be similar. The times for testing are shown in Table 4.3 and Figure 4.7. The reason $\text{NDatalog}_{\text{wellf}}$ is still somewhat faster than $\text{NDatalog}^{\text{naive}}_{\text{stra}}$ is that the termination condition ($I^+ = I^+.old$) in $\text{NDatalog}_{\text{wellf}}$ (see Algorithm 3.9) is tested only half as often as in $\text{NDatalog}^{\text{naive}}_{\text{stra}}$.

[2]Since XSB does not use an external database system, it is not possible to determine how much time is spent on different operations.

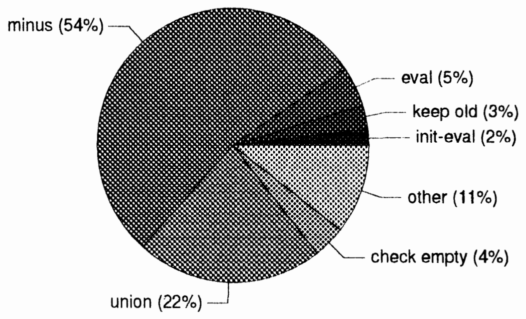


Figure 4.3: A time proportion graph for stratified semantics.

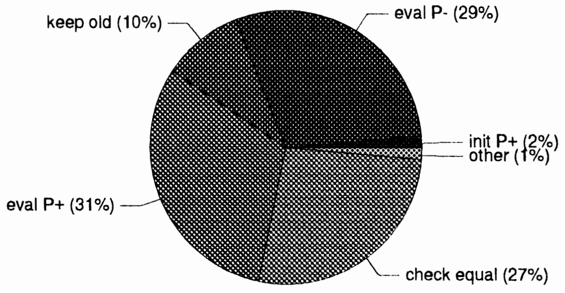


Figure 4.4: A time proportion graph for well-founded semantics.

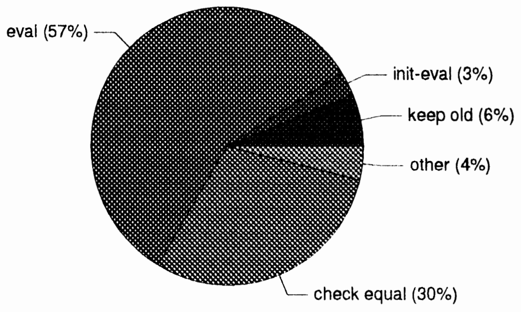


Figure 4.5: A time proportion graph for inflationary semantics.

T(s)	$NDatalog_{stra}$	$NDatalog_{wellf}$	$NDatalog_{infl}$	XSB	Result tuples path
$i = 10$	0.25	2.61	3.97	——	55
$i = 20$	0.92	8.69	15.01	——	210
$i = 30$	2.77	23.52	37.47	——	465
$i = 40$	6.93	61.55	87.34	——	820
$i = 50$	15.07	103.99	185.43	——	1275

Table 4.2: The times (in seconds) for a recursive program without negation (only rule evaluation)

T(s)	$NDatalog_{stra}$	$NDatalog_{wellf}$	$NDatalog_{infl}$	XSB	$NDatalog_{stra}^{naive}$	Result tuples path
i=10	6.33	5.00	7.00	1.20	5.33	55
i=20	18.70	15.30	22.67	3.50	16.67	210
i=30	72.14	54.70	77.22	7.66	62.00	465
i=40	172.46	109.28	153.09	12.50	125.00	820
i=50	309.75	194.10	280.07	20.20	220.00	1275

Table 4.3: The times for a recursive program without negation

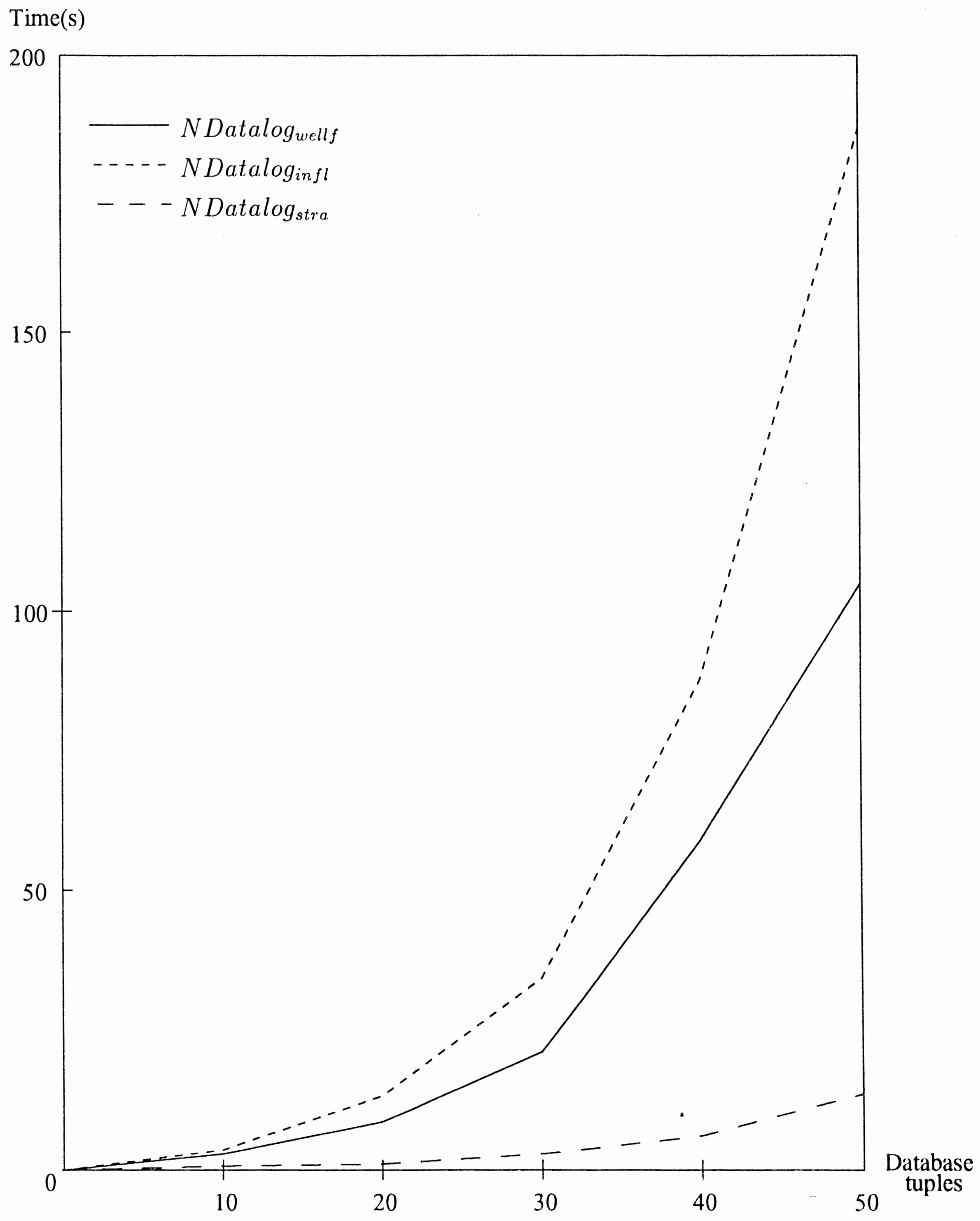


Figure 4.6: A times comparison graph for a recursive program without negation (only rule evaluation)

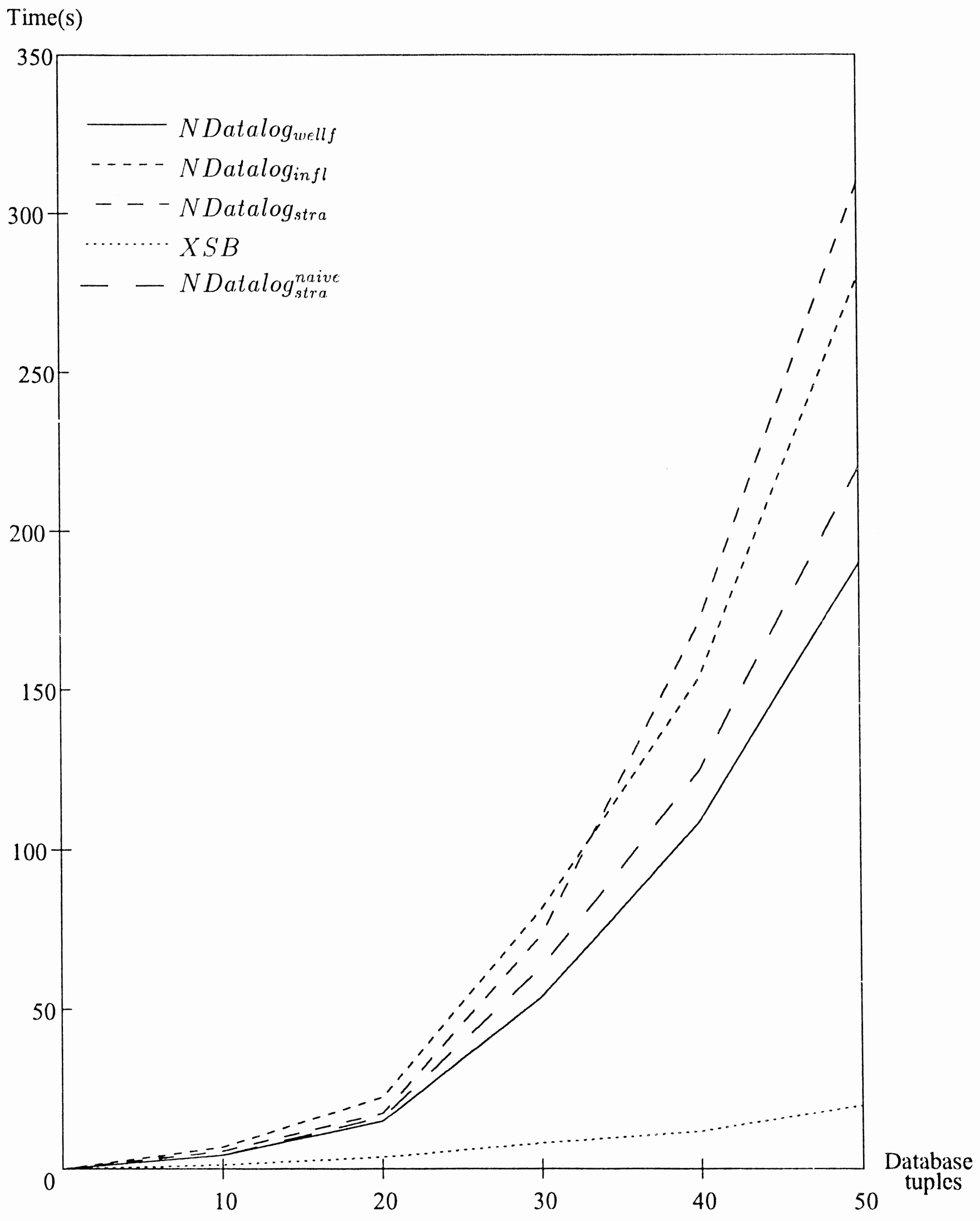


Figure 4.7: A times comparison graph for a recursive program without negation

4.2.2 Restricted programs with negation

Example: 4.2 Consider a logic program that defines a bachelor relation. Let $Male(X)$ be an EDB relation with the obvious meaning and $Married(X, Y)$ be an EDB relation with the meaning that X is the husband of Y. Then a bachelor relation can be defined as follows:

$$\begin{array}{lll} r_1: & bachelor(X) & :-male(X), not\ husband(X). \\ r_2: & husband(X) & :-married(X, Y). \end{array}$$

The results are shown in Figure 4.8 after computing the program with various semantics. Assume the relation $Male$ contains 50, 100 and 200 tuples, while the relation $Married$ contains 25 tuples. A table of execution times is shown in Table 4.4. From Figure 4.8,

male

X
tom
tony
david
park
bob
sam
bell

married

X	Y
tom	marry
sam	tina
bob	susan

bachelor

X
tony
david
park
bell

$NDatalog_{stra}$

$NDatalog_{wellf}$

bachelor

X
tom
tony
david
park
bob
sam
bell

$NDatalog_{infl}$

Figure 4.8: Answers to the bachelor program

obviously, the inflationary semantics produces a result which is not wanted by us. The reason is that the inflationary semantics fires all rules at once. From [Rei78] [Cla78], we know that the relation

$$\overline{Husband} = DOM - Husband$$

and the relation $Husband$ is empty at the beginning. So r_1 will get all the tuples of male after the first iteration. In order to avoid this apparent divergence between what we intuitively expect a rule should mean and what answer we would get if the programs contain some negative predicates, sometimes we have to modify the rules which contain only negative IDB predicates and EDB predicates in their bodies by defining a new IDB predicate and putting it into the rule's body. The new predicate is used to delay the derivation of new tuples in

T(s)	$NDatalog_{stra}$	$NDatalog_{wellf}$	$NDatalog_{infl}$	XSB	Result tuples
					bachelor
male= 50	1.6	4	——	2.0	25
male= 100	1.8	4.5	——	6.0	75
male= 200	2.7	6.8	——	22.5	175

Table 4.4: Times for the bachelor program

bachelor until the new predicate and *husband* have been computed in the current iteration. Finally, the result which is evaluated by NDatalog$_{\mathrm{infl}}$ should be same as the other semantics. This rewriting can be done automatically so that the programmer is relieved of the task. A generalisation of the method for all restricted programs is described below in Example 4.3. For example, we can change the above program by making a new IDB predicate *men* and putting it into r_1, and then define *men* to be the same as the EDB predicate *male*. We rewrite the program as:

$$\begin{array}{lll} r_1: & bachelor(X) & :-male(X), not\ husband(X), men(X). \\ r_2: & husband(X) & :-married(X,Y). \\ r_3: & men(X) & :-male(X). \end{array}$$

For this new program, NDatalog$_{\mathrm{infl}}$ provides the expected answer. A running times comparison is shown in Table 4.5 and Figure 4.9. □

T(s)	$NDatalog_{stra}$	$NDatalog_{wellf}$	$NDatalog_{infl}$	XSB	Result tuples
					bachelor
male= 50	1.8	5.5	6.2	2.5	25
male= 100	2.4	7.5	8.8	9.0	75
male= 200	3.3	9.7	12.2	35.5	175

Table 4.5: Times for the modified bachelor program

From this example we know that we should be careful in choosing the inflationary semantics when the programs involve some negative subgoals in a rule's body. But we still can

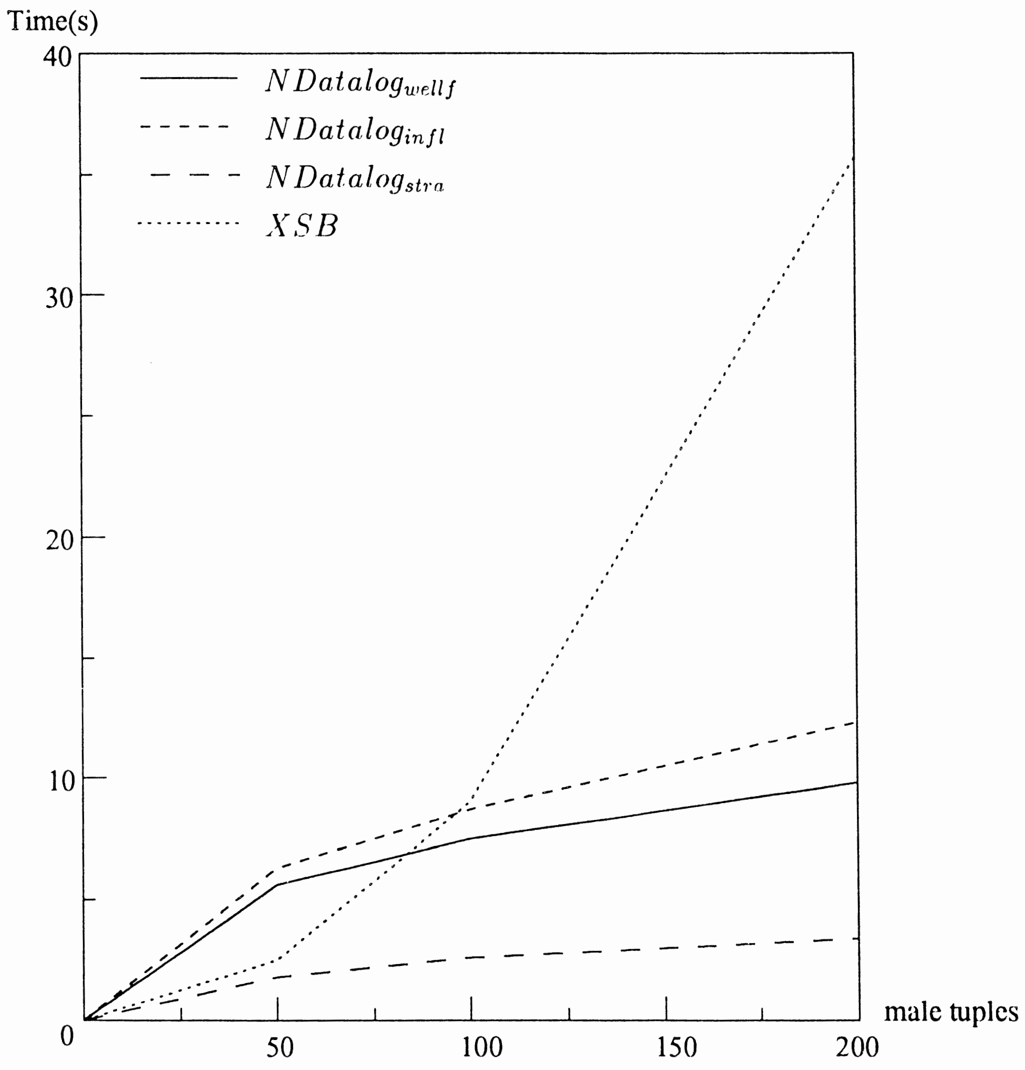


Figure 4.9: A times comparison graph for the modified bachelor program

say that NDatalog$_{\text{stra}}$, NDatalog$_{\text{infl}}$ and NDatalog$_{\text{wellf}}$ have the same expressive power using restricted programs with negation. Now let us compare Table 4.5 against Table 4.4. The times for the modified program are about 1.5 times slower than the original program. It is interesting that XSB becomes slower when negation is involved and there is a big database. The reason is that, for the well-founded semantics in XSB, a very simple meta-interpreter is used. This interpreter simulates an algorithm, called XOLDTNF, which can be exponential as mentioned in section 5.1. We also see that stratified is faster than well-founded here, because the program is not recursive and therefore removal of tuples is not needed.

Let us use an example to illustrate a standard way to modify the restricted programs in order to ensure that the inflationary semantics has the same meaning as the stratified and well-founded semantics when some negative subgoals are involved.

Example: 4.3 Let us continue to consider Example 4.2. As we know, this is a stratified program with *bachelor* in stratum 2 and *husband* in stratum 1. According to the number of the stratum, let us define a new IDB predicate $stratum1(X)$. Assuming there is an EDB predicate $stratum0(1)$, the program will be converted to

$$\begin{array}{lll} r_1: & bachelor(X) & :-male(X), not\ husband(X), stratum1(Z). \\ r_2: & husband(X) & :-married(X,Y), stratum0(Z). \\ r_3: & stratum1(X) & :-stratum0(X). \end{array}$$

□

As seen from the example, for the inflationary semantics, we just use some "temporary" relations (the number of "temporary" relations depends on the number of strata in the program) to delay the firing of certain rules. We add an extra stratum subgoal to each rule according to its stratum number. In addition, if there are n strata, we first assume $stratum_0(1)$ is an EDB fact, and then add $n-1$ rules to the program where each rule has the form $stratum_i(X) : -stratum_{i-1}(X)$. $(1 \leq i \leq n-1)$. In this way we can get the same meaning with the stratified and well-founded semantics if the programs are stratified. Let us consider another example.

Example: 4.4 Assume $people(Person) = \{a_1, a_2, \ldots, a_i\}$, $seen(Person, Film) = \{(a_1, f_1), (a_1, f_2), \ldots, (a_1, f_{10}), (a_2, f_1), \ldots, (a_2, f_{10}), (a_4, f_1), (a_5, f_5), (a_6, f_6), (a_7, f_7), (a_8, f_8), (a_9, f_9), (a_{10}, f_{10})\}$ and $showing(Film) = \{f_1, f_2, \ldots, f_{10}\}$ are EDB predicates, and in order to comparing easily, we only increase the number of people. Let $seen\_all\_films(Person)$ be an IDB predicate which represents people who have seen every film that is currently showing. Then $seen\_all\_films(Person)$ can be defined by the following program:

r_1: $seen\_all\_films(Person)$:– $people(Person), not\ not\_seen\_some\_film(Person).$
r_2: $not\_seen\_some\_film(Person)$:– $people(Person), showing(Film),$
$not\ seen(Person, Film).$

Clearly, the program is a stratified program and there are 3 strata. If we wish to get the same results as the stratified and well-founded semantics from the inflationary semantics, we need to rewrite the program by using the way which we have just mentioned above. The program will be converted to

r_1: $seen\_all\_films(Person)$:– $people(Person), not\ not\_seen\_some\_film(Person),$
$stratum2(X).$
r_2: $not\_seen\_some\_film(Person)$:– $people(Person), showing(Film),$
$not\ seen(Person, Film), stratum1(X).$
r_3: $stratum2(X)$:– $stratum1(X).$
r_4: $stratum1(X)$:– $stratum0(X).$

A comparison of running times is shown in Table 4.6[3] and Figure 4.10. □

T(s)	$NDatalog_{stra}$	$NDatalog_{wellf}$	$NDatalog_{infl}$	XSB
$i = 10$	5.7	10.2	11.4	5.1
$i = 20$	9.3	14.9	15.5	———
$i = 30$	15.7	22.7	22.8	———
$i = 40$	24.5	32.4	30.3	———
$i = 50$	35.3	42.9	39.9	———

Table 4.6: Times for the modified program for $seen\_all\_films$

As we see, $NDatalog_{stra}$ is faster than $NDatalog_{infl}$ and $NDatalog_{wellf}$. The reason is that both examples in this section are nonrecursive, so $NDatalog_{stra}$ never uses the MINUS operation.

[3]XSB could not run for $i > 10$ in this example.

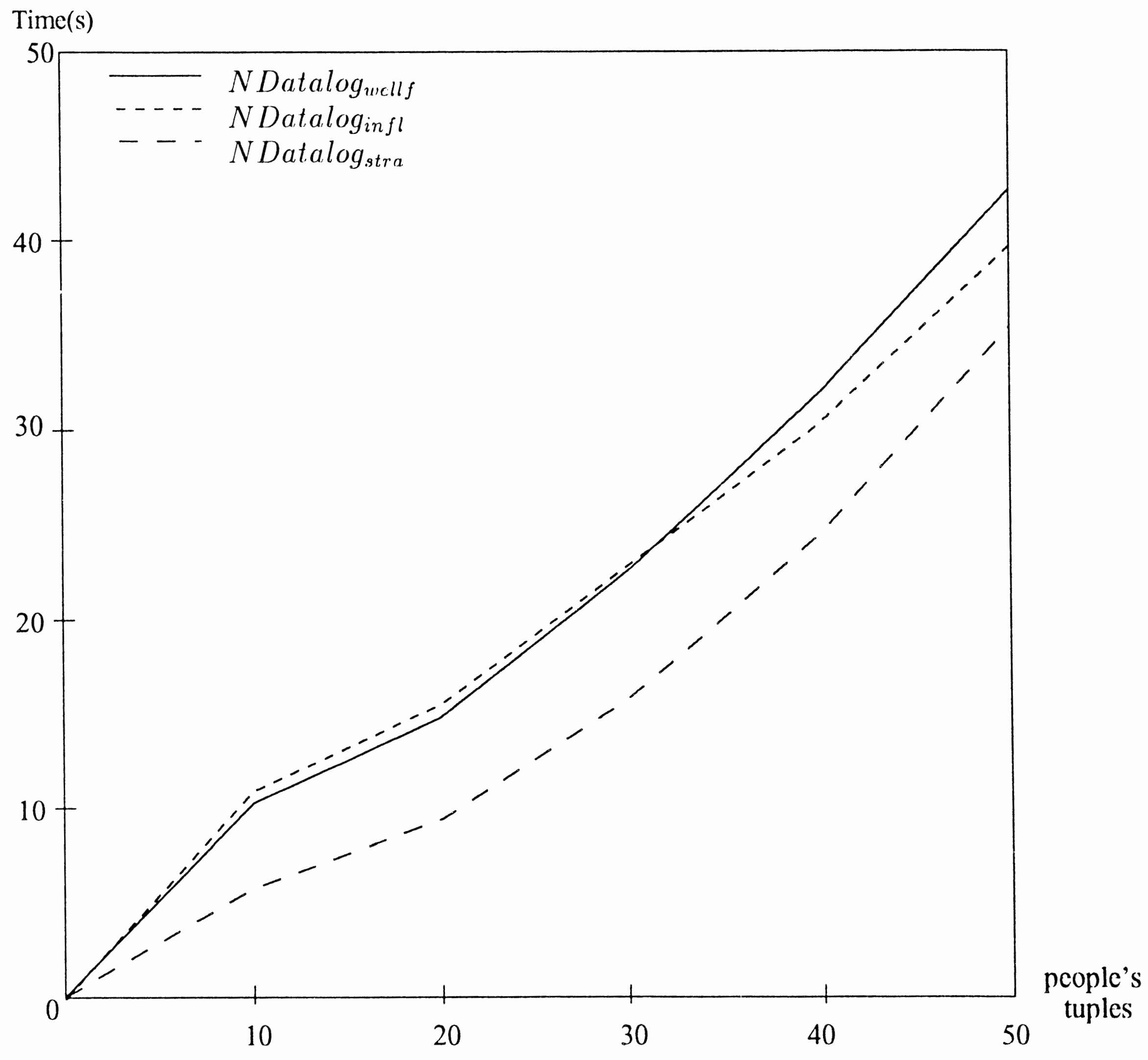


Figure 4.10: Running times for the modified program for *seen_all_films*

4.3 Testing nonrestricted programs on NDatalog

In the last section we have dealt with stratified programs. Recall that some queries are not expressible by stratified programs. So in this section, we will focus our attention on all kinds of Datalog$^\neg$ programs and not just stratified programs. First let us quote an example which is discussed by [Bid91].

Example: 4.5 Consider a logic program that defines even numbers for a finite subset of natural numbers. There is an EDB predicate $suc(X,Y)$ which represents the finite natural numbers from 0 to i. Suppose there exist facts $\{ suc(0,1), \cdots, suc(i-1,i), even0(0) \}$. The program should be written:

$$\begin{array}{lll} r_1: & even(X) & :- \; even0(X). \\ r_2: & even(X) & :- \; suc(Y,X), not\ even(Y). \end{array}$$

Obviously, the program is not stratifiable because of the negation appearing in the recursive rule r_2. How about the other semantics? After testing on the NDatalog system, we find that NDatalog$_{wellf}$ gives us even numbers as we would expect. Times are shown in Table 4.7, with the corresponding graphs in Figure 4.11.

T(s)	$NDatalog_{stra}$	$NDatalog_{wellf}$	$NDatalog_{infl}$	XSB	Result tuples
					even
$i=5$	——	3.4	——	0.5	3
$i=10$	——	5.8	——	2.5	6
$i=20$	——	11.9	——	30.3	11
$i=30$	——	19.3	——	150.5	16
$i=40$	——	30.3	——	498.0	21
$i=50$	——	39.2	——	1458.0	26

Table 4.7: The times for the even program

For NDatalog$_{infl}$, all natural numbers are given except $even(1)$. The problem is, as we mentioned in Example 4.2, that there is not any positive IDB predicate to restrict the negative predicate $not\ even(Y)$ in r_2 in the program. Can we find a logic program which

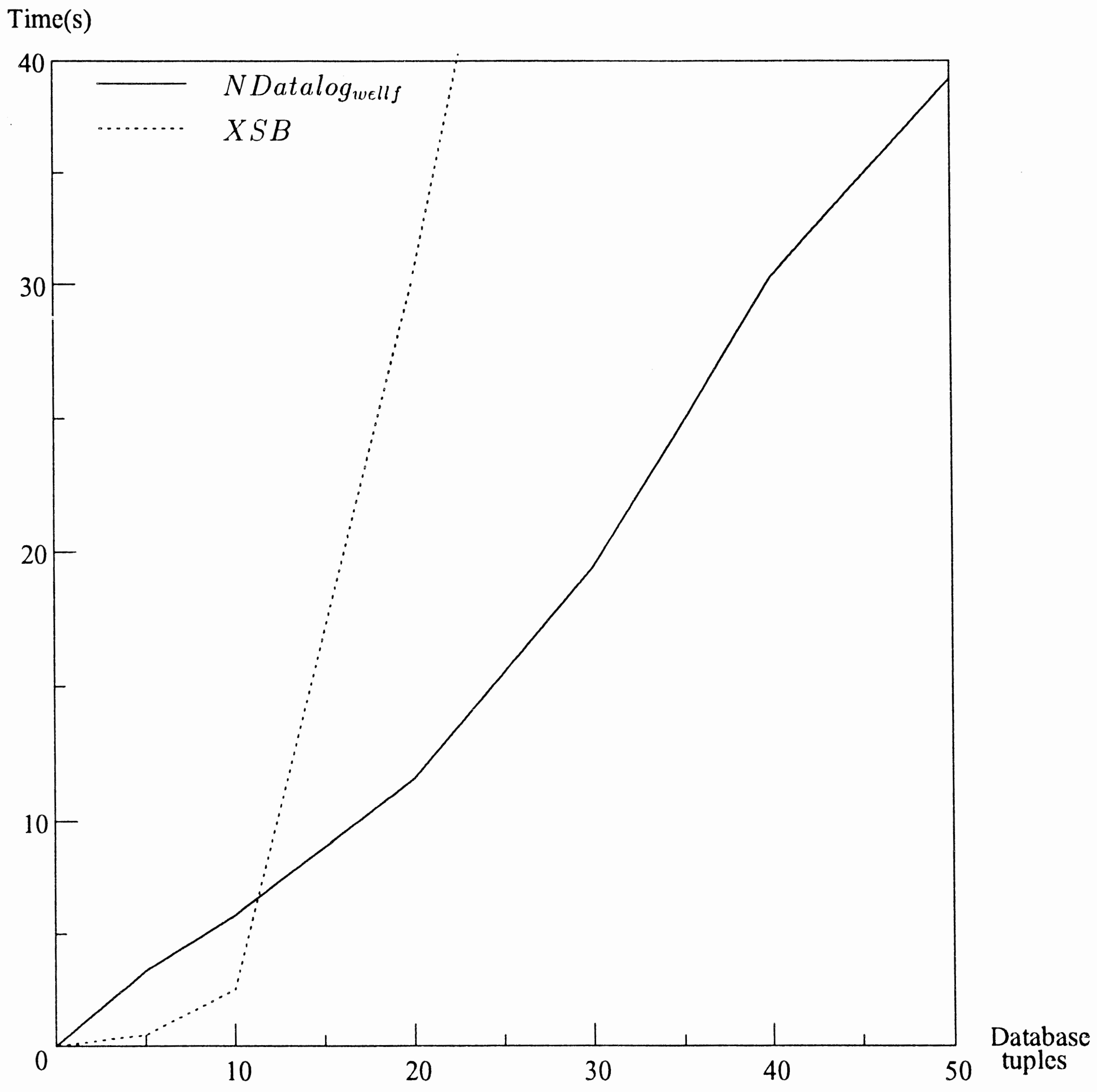


Figure 4.11: XSB vs $\text{NDatalog}_{\text{wellf}}$ for the even program

computes even numbers using NDatalog$_{\mathrm{infl}}$? Fortunately, the program can be easily modified in a similar way to Example 4.3. We need to introduce an IDB predicate *reached* into r_2 that will intuitively be used to delay the production of certain facts and achieve the goal of restricting the negative predicate *not even*(Y). The modified program follows:

$$\begin{array}{lll} r_1\text{:} & even(X) & :-\ even0(X). \\ r_2\text{:} & even(X) & :-\ suc(Y,X), not\ even(Y), reached(Y). \\ r_3\text{:} & reached(X) & :-\ even0(X). \\ r_4\text{:} & reached(X) & :-\ suc(Y,X), reached(Y). \end{array}$$

Now the inflationary semantics produces the same answer as the well-founded semantics, that is, it corresponds to the intended definition of even numbers. A table of running times is shown in Table 4.8, with the corresponding graphs in Figure 4.12.

Another standard method which modifies nonrestricted programs in order to make the inflationary semantics have the same meaning as the well-founded semantics when some negative subgoal are involved is discussed in [AV91] and [Bid91]. □

T(s)	$NDatalog_{stra}$	$NDatalog_{wellf}$	$NDatalog_{infl}$	XSB	Result tuples	
					reached	even
$i = 5$	—	6.7	7.1	1.5	6	3
$i = 10$	—	11.0	12.5	142	11	6
$i = 20$	—	17.8	20.8	over 3 hours	21	11
$i = 30$	—	30.4	37.2	over 3 hours	31	16
$i = 40$	—	43.0	50.8	over 3 hours	41	21
$i = 50$	—	65.6	67.1	over 3 hours	51	26

Table 4.8: The times for the modified even program

It is time to compare Table 4.8 and Table 4.7. Obviously, XSB is good for small samples in this case, namely, nonstratified programs. When the databases grow, the times of XSB have a sudden change as $i = 20$ in Table 4.7 and $i = 10$ in Table 4.8. This would seem to indicate exponential behavior. So using a meta-interpreter in XSB is just not a very good way to compute the well-founded semantics [War94]. In order to make the inflationary semantics have the same meaning as the well-founded semantics, we may have to modify the program. As Table 4.8 shows, the times for the modified program is about 1.9 times longer than before

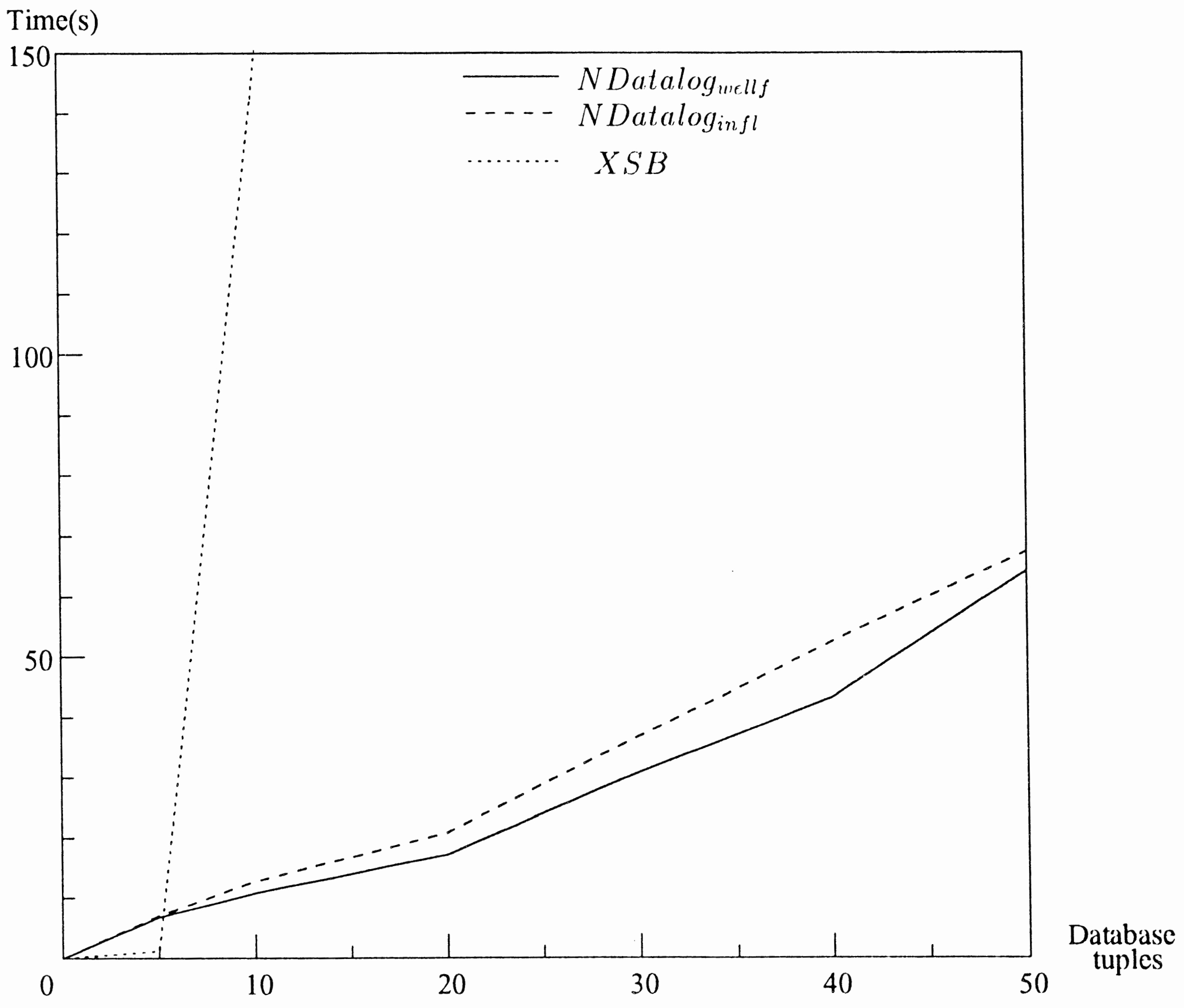


Figure 4.12: A times comparison graph for the modified even program

using NDatalog$_{\text{wellf}}$. But the change may bring about a 57 times increase for XSB.

We also tested an example [Ros90] which concerns the operation of a complex mechanism that is constructed from a number of components each of which may itself have smaller components.

Example: 4.6 We express that the mechanism is known to be *working* either if it has been (successfully) *tested*, or if all its components (assuming it has at least one component) are known to be working. The program can be written as follows:

$$
\begin{array}{lll}
r_1\text{: } working(X) & :- & tested(X). \\
r_2\text{: } working(X) & :- & part(X,Y), not\ has\_suspect\_part(X). \\
r_3\text{: } has\_suspect\_part(X) & :- & part(X,Y), not\ working(Y).
\end{array}
$$

Here, *tested* and *part* are EDB predicates. A table of *part* for $i = 10$ is shown in Table 4.9. The *tested* components are a and all even numbers. Evaluating the program gives *working*

Y \ X	1	2	3	4	5	6	7	8	9	10
a	*		*		*		*		*	
b		*		*		*		*		*
c	*	*	*	*	*	*	*	*	*	*

Table 4.9: Table for $part(X, Y)$ for $i = 10$.

components which are a, b and all even numbers. The times for getting these results from NDatalog$_{\text{wellf}}$ and XSB are shown in Table 4.10 for 5 different values for $part(X, Y)$. Once again, we see that XSB is very slow. □

T(s)	$NDatalog_{stra}$	$NDatalog_{wellf}$	$NDatalog_{infl}$	XSB
$i = 10$	———	5.7	———	12.6
$i = 20$	———	6.3	———	210.7
$i = 30$	———	6.7	———	1282.9
$i = 40$	———	7.0	———	4430.0
$i = 50$	———	8.7	———	12396.0

Table 4.10: The times for getting working parts

Chapter 5

Conclusion and Further Work

5.1 Conclusion

Since theories about negation in logic programming were proposed by Reiter [Rei78], and by Clark [Cla78], more general theories have been nicely developed in Datalog during the last ten years. In particular, the interaction of negation and recursion has been extensively studied.

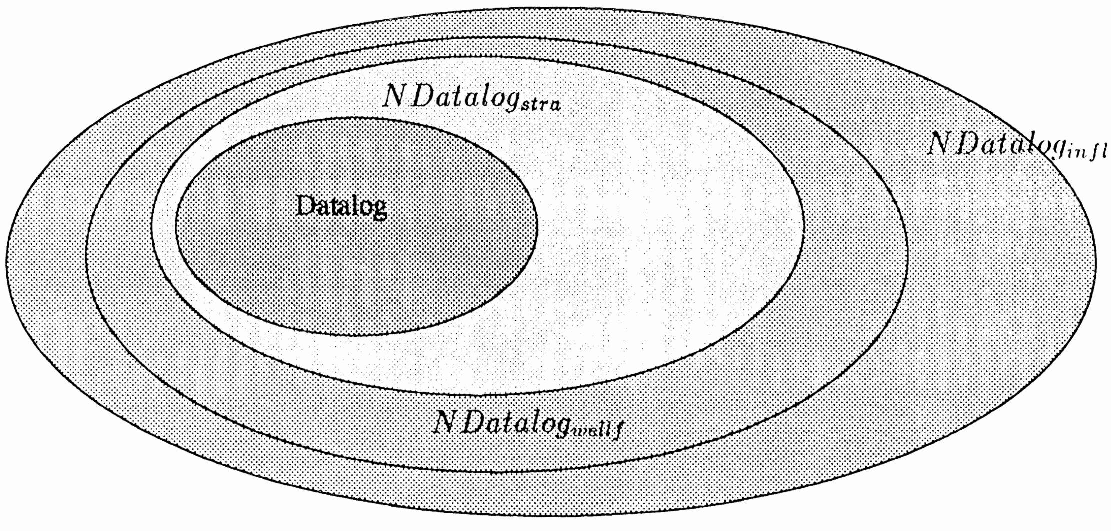
This thesis can be considered as being comprised of three phases: introduce negation into Datalog, analyse current semantics models with negation, and compare the efficiency of the various semantics. In the introduction phase, Chapter 2 and 3, I introduced Datalog which is a declarative language for deductive databases that has a Prolog-like syntax but whose programs are evaluated using database operations. Some concepts of negation, such as the Closed World Assumption and Negation As Failure, and semantics with negation, such as the Stratified, Well-founded and Inflationary semantics, were introduced in Chapter 2.

An overview of the NDatalog System and an evaluable NDatalog language which is based on the DatalogIC language was introduced in Chapter 3. In addition, I described how NDatalog programs are converted into SQL statements and evaluated. I further discussed three different semantics with negation, Stratified, Inflationary and Well-founded, and I gave corresponding implementation algorithms.

In the comparison phase, Chapter 4, I presented a performance analysis and comparison of the various semantics with negation on the NDatalog system. Firstly, the results indicate that, for a recursive program without negation, XSB is much faster than NDatalog$_{\mathrm{wellf}}$, NDatalog$_{\mathrm{stra}}$ and NDatalog$_{\mathrm{infl}}$ (see Figure 4.2). NDatalog$_{\mathrm{wellf}}$ is faster than NDatalog$_{\mathrm{stra}}$, but when we further analyse the operation time, we find most time is spent on removing "new"

"new" tuples in NDatalog$_{\text{stra}}$, the problem being that removing tuples using INFORMIX-SQL is slow; if we only consider pure evaluation time, the result is that NDatalog$_{\text{stra}}$ is much faster than NDatalog$_{\text{wellf}}$ and NDatalog$_{\text{infl}}$ (see Figure 4.6). For nonrecursive programs with negation, NDatalog$_{\text{stra}}$ is much faster than NDatalog$_{\text{wellf}}$, NDatalog$_{\text{infl}}$ and XSB (see Figure 4.9). NDatalog$_{\text{wellf}}$ is faster than NDatalog$_{\text{infl}}$ and XSB when we evaluate nonstratified programs on big databases (see Figure 4.12). Furthermore, results are presented to indicate that XSB is much faster than NDatalog$_{\text{stra}}$, NDatalog$_{\text{infl}}$ and NDatalog$_{\text{wellf}}$ when no negative subgoals are involved (see Figure 4.2). Finally, performance comparisons indicate that XSB is not always about *an order of magnitude faster* than current deductive databases systems as claimed in [War91]. I hope these results may provide information of use in designing and implementing the next generation of logic programming and deductive database systems.

Figure 5.1 shows the hierarchy of expressiveness of different semantics models of NDatalog in this thesis.



▨	░	▨
Programs without negation	Stratified programs	Non-stratified programs

Figure 5.1: Hierarchy of expressiveness of different versions of NDatalog

5.2 Further work

We have expanded the power of the original Datalog system [Was90], including the ability to handle rules which have negation subgoals, but the ability to express disjunctions and functions have still not been included. If the last of these can be achieved we can use Skolem functions to model existential variables and thus solve the subset constraint problem mentioned in [Was90].

Even though we have described three main semantics with negation, there are several aspects of semantics of negation in logic programming that are not dealt with in this thesis which we leave for further work, for example, the perfect model semantics [Prz88], the stable model semantics [GL88] and the default model semantics [BF87].

So far I do not consider extensions of the NDatalog language which allow for negative literals in heads of rules as discussed by [AV91]. These negative literals are interpreted as tuple deletions and are interesting with respect to expressive power.

It is also important to improve the NDatalog system so that it can create EDB relations directly and not in a DBMS such as INFORMIX. However, as evaluation speed is also important especially if we wish to evaluate the programs with negation, we should be careful in computing the DOM relation when large amounts of data are involved. If we are to expand the system to handle the DOM relation optimisation, as we discussed in Chapter 3, a corresponding expansion of the data types will have to be implemented.

Bibliography

[ABW86] K. Apt, H. Blair, and A. Walker, Towards a Theory of Declarative Knowledge. In *Proceedings, Workshop on Foundations of Deductive Databases and Logic Programming*, Washington DC, pp. 546–629, 1986.

[AV91] S. Abiteboul and V. Vianu, Datalog Extensions for Database Queries and Updates. In *Journal of Computer and System Sciences*, 43:62–124, 1991.

[BF87] N. Bidoit and C. Froidevaux, Minimalism subsumes default logic and circumscription in stratified logic programming, In *Proc. Logic In Computer Science*, (IEEE, New York 1987) pp. 89–97.

[Bid91] N. Bidoit, Negation in Rule-based Database Languages:a Survey. In *Theoretical Computer Science*, 78 (1991) pp. 3–83, North-Holland.

[BR86] F. Bancilhon and R. Ramakrishnan, An Amateur's Introduction to Recursive Query Processing Strategies. In *Proc. ACM-SIGMOD Conference on the Management of Data*, pages 16–52, 1986.

[CGT89] Stefano Ceri, Georg Gottlob, and Letizia Tanca, What You Always Wanted to Know About Datalog(And Never Dared to Ask). In *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166,March 1989.

[CGT90] Stefano Ceri, Georg Gottlob, and Letizia Tanca, Logic Programming and Databases. Springer-Verlag 1990.

[CH82] A. Chandra and D. Harel, Structure and Complexity of Relational Queries. In *Comput. System Sci.*, 25(1) pages:99–128, 1982.

[CH85] A. Chandra and D. Harel, Horn Clause Queries and Generalizations. In *J. Logic Programming*, 1(1985), pp. 1–15.

[Cla78] K. L. Clark, Negation as Failure. In *Logic and Data Bases* H.A. Gallaire and J. Minker, Eds., pp. 293–322, Plenum, New York, 1978.

April 1994.

[Gel88] A. Van Gelder, Negation as Failure Using Tight Derivations for General Logic Programs. In *Foundations of Deductive Databases and Logic Programming*, pages 149–176, 1988.

[GL88] Michael Gelfond and Vladimir Lifschitz, The Stable Model Semantics for Logic Programming. In *Proc. 5th International Conference on Logic Programming,1988*, pp. 1070–1080.

[GRS88] A. V. Gelder, A. Ross, and J. S. Schlipf, The Well-founded Semantics for General Logic Programs. In *7th ACM Symp. Principles Database Syst.(PODS)*. pp. 221–230, Mar 1988.

[GS86a] D. M. Gabbay and M. J. Sergot, Negation as Inconsistency. In *Journal of Logic Programming*, 3(1):1–36, 1986.

[GS86b] Gurevich and Shelah, Fixed Point Extensions of First Order Logic. In *Proc. 26th Symp. on Foundations of Computer Science*, 346–353 , 1986.

[Kol89] PH. G. Kolaitis, On the expressive power of stratified logic programs. In *Inform. and Comput*, Report CRL 89–14, Univ. of California, Santa Cruz, August 1989.

[KP91] Phokion G. Kolatts and Christos H. Papadimitriou, Why Not Negation by Fixpoint? In *Journal of Computer and System Sciences*, 43:125–144, 1991.

[KSS91] David B. Kemp and Peter J. Stuckey and Divesh Srivastava, Magic Sets and Bottom-up Evaluation of Well-Founded Models. In *Logic Programming*, Proceedings of the 1991 International Symposium, edited by Vijay Saraswat and Kazunoti Ueda, pp. 337–351, 1991.

[Llo87] J. W. Lloyd, Foundations of Logic Programming. 2nd extended ed. New York: Springer-Verlag, 1987.

[MW88] David Maier and David S. Warren, Computing with Logic–logic programming with prolog. The Benjamin/Cummings Publishing Company, Inc. 1988.

[Naq86] S.A. Naqvi, A Logic for Negation in Database Systems. In *Proc. Workshop on Foundations of Deductive Databases and Logic Programming*, pages 378–387, 1986.

[Prz88] T.C. Przymusinska, Perfect Model Semantics. In *Proc. 5th International Conference on Logic Programming,1988*, pp. 1081–1096.

[Prz89] T.C. Przymusinska, Every Logic Program Has a Natural Stratification and an Iterated Fixed Point Model. In *ACM Symposium on Principles of Database Systems*, 1989.

[Rei78] Raymond Reiter, The Closed World Assumption. In *Logic and Data Bases* H.A. Gallaire and J. Minker, Eds., pp. 55–76, Plenum, New York, 1978.

[Ros89] K.A. Ross, A Procedural Semantics for Well-Founded Negation in Logic Programs. In *Proc. of the 8th ACM Symposium on Principles of Database Systems*, 22–33, 1989.

[Ros90] K.A. Ross, Modular Stratification and Magic Sets for DATALOG Programs with Negation. In *Proc. of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 161–171, April 1990.

[SB88] Sara Baase, Computer Algorithms–Introduction to Design and Analysis. second edition, page 191–197, 1988.

[She88] John C. Shepherdson, Negation in Logic Programming. In *Foundations of Deductive Databases and Logic Programming*, (Jack Minker, Ed.), Morgan Kaufmann Publishers, Los Altos, CA, 19–88, 1988.

[Ull88] Jeffrey D. Ullman, Database and Knowledge-Base Systems. volume 1, Computer Science Press, Potomac, Md., 1988.

[War91] David S. Warren, Computing the Well-Founded Semantics of Logic Programs. In *Technical Report 91/12* Computer Science Department, SUNY at Stony Brook.

[War94] In private communication with Prof. David S. Warren in 1994.

[Was90] Mark P. Wassell, Semantic Optimisation in Datalog Programs. MSc Thesis, Department of Computer Science, University of Cape Town, S.A., 1990.

[Yan90] Mihalis Yannakakis, Graph-theoretic Methods in Database Theory. In *Proc. 9th ACM SIGMOD Symp. on Principles of Database Systems*, pp. 230–242. Nashville, Tennessee. Apr. 1990.

Appendix A

Data Structures and Algorithms

This appendix details some of the data structures which are based on the DatalogIC system and are somewhat different. Some principle algorithms used in the NDatalog system are given here also.

A.1 Data Structures

In this section, the main data structures are outlined as follows.

Definition Nodes

The **DEFNNODE** structure holds the information for both the IDB and EDB predicates. For an IDB predicate, the structure has the following major fields:

- Name of the definition.
- Argument list.
- Negation–A label is TRUE if the definition includes negation in the tree.
- Stratum number.
- Non-recursive rule list.
- Recursive rule list.
- Child list–List of the other definitions in this definition's recursive component.

- Recursive component–List of all definitions which depend on this one.
- Pointers to left and right children in definition tree.

EDB predicate definitions will have only the name, argument list and negation label fields.

Rule Nodes

The **RULENODE** structure is used for the rules and queries. The following is a list of the major fields in the structure.

- Name of the rule.
- Argument list for the head of rule.
- Negation–A label is TRUE if the rule have negation in the body.
- Body predicate list.
- Reloplist of rules.

Predicate Nodes

The **PREDNODE** structure can be of two types: Predicate and Relop. The former is for non-evaluable predicates, while the latter is for evaluable predicates. These are linked together to form the body of the rules. For database predicates the structure is as follows:

- Name of the predicate.
- Argument list–List of arguments in this occurrence.
- Negation–A label is TRUE if the predicate is negated.
- Predicate number–A number indicating the predicate's position in the body of the rule.
- Pointer to definition for this predicate[1].

There is one occurrence of the structure for each occurrence of a predicate in the program. Evaluable predicates are held in a structure that has fields for:

- Type of operator ($``=", ``\neq", ``>", ``\geq", ``<" or ``\leq"$).

[1]This is the definiton to rule arc in the rule/goal graph.

- Pointer to first operand.
- Pointer to second operand.

Argument Node

An argument that appears in a definition head, the head of a rule or a predicate occurrence is held in the **ARGNODE** structure. An argument can be one of three types: VARIABLE, CONST_STRING and CONST_NUM. The union type structure is not used since there is some sharing of fields.

- Type of argument.
- String–This is the text version of the argument (as it appeared to the lexical analyser).
- Number–For variables this is the argument's position in an argument list; for CONST_NUM arguments this is the number itself.
- Adornment–Used in production of adorned program (possible values are BOUND or FREE).
- Label–Used in queries (possible values are BOUND, FREE, or EXISTENTIAL).

A.2 Algorithms

A.2.1 Rectification

Algorithm: A.1 Fully Rectify a Rule
Input: A rule.
Output: Fully Rectify a Rule.

```
begin {rectification}
    for every predicate, $p_i$, in the rule (head and body)
        for every argument $a$ in arglist of $p_i$
            if $a$ is a variable, $X$
                Convert $X$ to $X_i$ form;
                if there is a variable $v$ with same name as $X$ in $T$
                        Add "$X_j^i = v$" to the reloplist for the rule;
                        /*$T$ is lookup table for variables*/
                Add $X_j^i$ to $T$ (replacing $v$ if necessary);
                else
                    Copy and convert $a$ to a variable $X_a$;
                    Add "$X_a = a$" to reloplist;
                endif
            endfor
        endfor
end{rectification}
```

A.2.2 Testing for and Finding a Stratification

Algorithm: A.2 Testing for and Finding a Stratification (TFS) [Ull88]
INPUT: A set of NDatalog rules.
OUTPUT: A decision whether the rules are stratified. If so, produce a stratification.
METHOD: Start with every predicate assigned to stratum 1. Repeatedly examine the rules. If a rule with head predicate p has a negated subgoal with predicate q, let p and q currently be assigned to strata i and j respectively. If $i \leq j$, reassign p to stratum $j+1$. Furthermore, if a rule with head p has a nonnegated subgoal with predicate q of stratum j, and $i < j$, reassign p to stratum j. These laws are formalized as follows:

```
begin
    for each predicate p do
        stratum[p] := 1;
    endfor
    repeat
            for each rule r with head predicate p do begin
                for each negated subgoal of r with predicate q do
                    stratum[p] := max(stratum[p],1+stratum[q]);
                for each nonnegated subgoal of r with predicate q do
                    stratum[p] := max(stratum[p],stratum[q])
            endfor
    untilthere are no changes to any stratum
          or some stratum exceeds the number of predicates
    if some stratum exceeds the number of predicates
       reture("no")
    endif
    output an order of definitions from stratum
    return("yes")
end
```

A.2.3 Finding Strong Connected Components

Algorithm: A.3 Strong Connected Components (SCC) [SB88]
Input: $G = (V, E)$, a digraph represented by linked adjacency lists.
Output: An order of vertices (definition nodes) in each strong component.

```
Procedure StrongComponents(adjacencyList: HeaderList; n: integer);
var
            dfsNumber: array[VertexType] of integer;
            low: array[VertexType] of integer;
            dfn: integer;
            v: VertexType;
            order: the order number of vertex;
            SC: Stack;
            We define that TOP is a function that returns the top item on a stack
            removed: array[VertexType] of boolean;

Procedure SCompDFS(v: VertexType);
Var
            w: VertexType;
            ptr: Nodepointer;

begin { SCompDFS }
      { Process vertex when first encountered. }
      dfn:=dfn+1;
      dfsNumber[v]:=dfn; low[v]:=dfn;
      remove[v]:=FALSE;
      ptr:=adjacencyList[v];
      while ptr ≠ nil do
            w:=ptr↑.vertex;
            if dfsNumber[w]==0 {unmarked} then
              SCompDFS(w);
              { Now backing up from w to v }
              low[v]:=min(low[v],low[w])
            else{ w was already encountered }
              if not removed[w] then
                    low[v]:=min(dfsNumber[w],low[v])
              endif{ if w is still in the tree }
            endif{ of processing w };
            ptr:=ptr↑.link;
      endwhile;
```

```
        { Now backing up from v }
        if low[v]==dfsNumber[v] then
           output an order number i of the definition node for a new strong component;
           removed[v]:=TRUE;
           output order;
           while SC is nonempty and dfsNumber[TOP(SC)] > dfsNumber[v] do
                  output order (of TOP(SC));
                  removed[TOP(SC)]:=TRUE;
                  pop SC
           endwhile{ while vertices from SC are in current strong component }
           order:=order+1;
        else{ haven't found a new strong component }
           push v onto SC
        endif{ backing up from v }
endbegin{ ScompDFS }

begin { StrongComponents }
        for v:= 1 to n do dfsNumber[v]:=0 endfor;
        dfn:=0;
        for v:= 1 to n do
            order:=1;
            if G contain recursive then
               if dfsNumber[v]==0 then SCompDFS(v) endif
            else
               output order;
            endif
        endfor
endbegin{ StrongComponents }
```

□