ON EFFICIENT FIXPOINT COMPUTATION OF DEDUCTIVE DATABASES WITH UNCERTAINTY

Zhi Hong Zheng

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Computer Science at

Concordia University

Montreal, Quebec, Canada

June 2004

Your file Votre référence
ISBN: 0-612-94763-7
Our file Notre référence
ISBN: 0-612-94763-7

ABSTRACT

On Efficient Fixpoint Computation of Deductive Databases with Uncertainty

Zhi Hong Zheng

Uncertainty management has been identified as an important challenge in AI and database research. Many frameworks of logic programming have been proposed to manage uncertain information in deductive databases and expert systems. These proposals address fundamental issues of modeling, semantics, query processing and optimization. However, there have been fewer reports on efficient implementation of such frameworks. In this research, we study this issue in the context of a fragment of the parametric framework [19] over the certainty domain of [0, 1]. It has been shown that the standard Semi-Naive fixpoint evaluation method does not have a counterpart when uncertainty is present [32]. We have refined a Semi-Naive method, originally proposed in [32], and developed a Semi-Naive evaluation engine that takes into account the multiplicity of derivations of the same atom. We also introduced a refinement of this evaluation, called Semi-Naive with Partition, which further improves the efficiency of the Semi-Naive method with uncertainty. Finally, we adopt "stratification" from Datalog, which is shown to make a significance efficiency for certain input programs with uncertainty. We have conducted numerous experiments to assess the benefits of the collection of optimizations proposed in this research. Our experiments and benchmarks indicate that the proposed techniques and tricks yield a useful, efficient evaluation engine for deductive databases with uncertainty.

Acknowledgments

I express deep gratitude to my supervisor, Professor Nematollaah Shiri for his extraordinary and continuous guidance, support, and valuable inside throughout my studies at Concordia University. His support and patience were invaluable in the preparation of this thesis. He was always gracious in giving me the time and strength even amidst the tight schedule. I consider myself blessed to be under his supervision.

I am grateful to the staff in the Department of Computer Science, especially Halina Monkiewicz and Veronica Jacobo-Gutierrez. They are always available for helping to solve the problems encountered and getting things done at the right time.

I am thankful to my friends Hong Bin Zhang and Tong Zhang for helping me develop the design and implementation of our deductive database with uncertainty system. I would also like to thank Wei Du, Ali Kiani and Wei Shen Lin for their generous support and encouragement.

I am deeply indebted to my wife Shu Hong in many ways. Without her I would not have been able to go through this wonderful stage of my life. My deep love goes to our sweet daughter Helena and little sun Patrick. Their childish smiling always makes me forget the fatigue.

Finally, I would like to thank my parents for encouraging me to purse the good education, from which everything else springs.

Contents

List of Figures viii

List of Tables x

1 Introduction 1

1.1 A Motivating Example..... 3

1.2 Contribution of the Thesis..... 5

1.3 Thesis Outline..... 7

2 Background and Related Work 9

2.1 Classification of Deductive Databases with Uncertainty..... 9

2.2 The Parametric Framework: A Review..... 11

2.2.1 The Basic Concepts..... 11

2.2.2 Fixpoint Evaluation of P-program..... 14

2.2.3 Classification of Disjunction Function..... 15

2.3 Implementations of DDB+Uncertainty..... 16

2.4 Stratified Evaluation of Programs in Datalog$^{\neg}$..... 17

3 Efficient Evaluation of LP+DDB with Uncertainty 20

3.1 Inapplicability of Classical SN Algorithms..... 20

3.2 A Multiset Based Semi-Naive Algorithm..... 24

3.3 The Semi-Naive with Partition Algorithm..... 29

3.4 Stratification and Efficiency..... 35

4 System Architecture **42**

4.1 Data Interpreter Component..... 43

4.2 Data Manager Component..... 44

4.3 Query Optimizer Component..... 45

4.4 Query Processor Component..... 46

4.5 Library Manager Component..... 48

5 System Implementation **49**

5.1 Data Representation..... 49

5.1.1 Representation of Dictionary..... 50

5.1.2 Representation of Atoms..... 51

5.1.3 Representation of Relations and Fact Table..... 53

5.1.4 Representation of Rules and Rule Table..... 56

5.1.5 Representation of Indices..... 60

5.2 Indices Creation..... 62

5.2.1 Index Plan Creation..... 62

5.2.2 Reordering Body Predicates..... 65

5.2.3 Index Containment and Index Creation..... 68

5.3 Query Optimization..... 70

5.3.1 Rules Rewriting Technique..... 71

5.3.2 Information Backtracking..... 75

5.3.3 Run-Time Decisions..... 81

5.3.4 Rule Reordering and Stratification..... 84

5.4 Query Evaluation..... 88

5.4.1 Materialization and Pipelining..... 89

5.4.2 Dynamic Variables Binding..... 90

5.4.3 Precision Controlling..... 93

5.4.4 Stratified Evaluation..... 94

6 Performance Analysis and Evaluation **96**

6.1 Experiment Environment..... 97

6.2 Test Programs Selection..... 97

6.3 Test Data Selection and Generation.....100

6.4 Index Performance Evaluation..... 105

6.5 Semi-Naive Technique: Performance Evaluation..... 112

7 Conclusion and Future Research **120**

References **123**

Appendix A **127**

Appendix B **130**

List of Figures

2.1 A Naive algorithm for evaluating p-programs..... 15

2.2 An example of Datalog$^{\neg}$ program and its dependency graph..... 17

3.1 A Semi-Naive algorithm for evaluating programs with uncertainty..... 27

3.2 The Semi-Naive with Partition algorithm..... 32

3.3 A p-program $P_{3.2}$..... 35

4.1 System architecture..... 42

4.2 Data transformation procedure..... 43

4.3 Index creation procedure..... 45

4.4 Program evaluation procedure..... 47

5.1 Internal representation of p(10,3): 0.5..... 50

5.2 Internal representation of program $P_{5.1}$..... 57

5.3 An example of Index plan..... 61

5.4 An algorithm for body predicate re-ordering..... 66

5.5 The procedure of locate_predicate..... 68

5.6 Procedure for Semi-Naive evaluation with partition and backtracking..... 80

5.7 The procedure for evaluation of rewritten rules..... 81

5.8 An example of predicate dependency graph..... 85

5.9 The stratification algorithm 87

6.1 Test programs in performance experiment..... 98

6.2 Data set for program *p1* and *p2*.....101

6.3 Data set for program *p3* and *p4*.....103

6.4 Running *p1* on CT with/without indexing.....107

6.5 Running *p2* on M with/without indexing.....108

6.6 Running *p3* on T with/without indexing.....109

6.7 Running *p3* on C with/without indexing.....111

6.8 SN and SNP performance: running *p1* on M.....113

6.9 SN and SNP performance: running *p2* on CT.....114

6.10 SN and SNP performance: running *p3* on A.....116

6.11 SN and SNP performance: running *p4* on S.....117

List of Tables

6.1 Running *p1* on CT_n with/without indexing.....106

6.2 Running *p2* on $M_{n,m}$ with/without indexing 108

6.3 Running *p3* on $T_{n,m}$ with/without indexing 109

6.4 Running *p3* on C_n with/without indexing 110

6.5 SN and SNP performance: running *p1* on $M_{n,m}$.....113

6.6 SN and SNP performance: running *p2* on CT_n.....114

6.7 SN and SNP performance: running *p3* on A_n.....115

6.8 SN and SNP performance: running *p4* on S_n.....117

Chapter 1

Introduction

The world is replete with uncertainty. Mortgage bankers are uncertain about the outcome of loans that they make. Military planners are uncertain about what the enemy might do. Airline passengers are uncertain about whether their plane will be on time or will be late. The messages sent from a satellite are uncertain about the received signal with noise. These and many other real-life applications require an ability to represent, manage, and reason with uncertain information. Even when the information gained is certain, answering some complex queries still requires associating certainties to the answers. For instance, in weather forecasting, certain information of the current temperature, humidity, wind speed, and cloud distribution are applied to answer the probabilistic query about tomorrow's weather.

Uncertainty is a form of imperfection in information, which arises when the truth of the information is not established definitely. More precisely, uncertainty is the "degree" of truth of information pieces as estimated by an individual or sensor device, which may be represented by associating with the information, a value coming from an appropriate domain.

Uncertainty management has been a challenging issue in AI and database systems for a long time. Numerous researches have been carried out mainly in the last two decades, resulting in a number of concepts being investigated, a number of problems being identified and a number of solutions being developed [19, 3, 2, 14, 36, 11, 17, 18]. Most

of this work is concerned with developing frameworks with uncertainty by extending the classical logic database programming with its advantages of modularity and its powerful top-down and bottom-up query processing techniques. In [19], these frameworks are classified into annotation based (AB), and implication based (IB). The Parametric framework [19] is a generic IB framework that unifies and generalizes all the IB frameworks, which is also the basis we applied for our query optimization study in this thesis. In this research, we study query processing and optimization in the context of a fragment of the parametric framework, namely over the certainty domain [0, 1].

There have been some works on logic frameworks with uncertainty. However, there is little work on handling their effective and efficient implementation. In [20], Leach and Lu investigated an implementation approach to top-down query processing in Annotated Logic Programs with set-based semantics. For IB frameworks, a so-called Problog evaluation system is introduced in [31] to evaluate parametric programs on top of the XSB system [29]. Another attempt for IB framework implementations is a version of CORAL [27], which supports some special annotation to compute the certainty associated with each atom. Even though XSB and CORAL systems both support multisets, their evaluation scheme is not useful in our context. The main problem is that they tend to consider and combine derivations across iterations. Query processing could be complicated when the semantics is based on multisets. This motivates our work here to design and implement a new system under which all parametric programs can be evaluated correctly and efficiently. This is illustrated next in the following examples.

1.1 A Motivating Example

A run-time optimization technique for bottom-up evaluation of classical logic programs and deductive databases is the Semi-Naive fixpoint evaluation method, which is proposed as an alternative to the Naive method. Semi-Naive method tries to avoid or minimize repeated applications of rules at every iteration step. Obviously, it is also desired to use the Semi-Naive method for efficient evaluation of the programs in parametric framework.

However, a "straight" extension of the standard Semi-Naive method to take into account the presence of certainty values does not work in general, simply because the results may not always coincide with the results obtained by the corresponding Naive method with uncertainty. Program $P_{1.1}$ shown in Figure 1.1 illustrates this point.

$$r1: B \xleftarrow{0.5} .$$
$$r2: C \xleftarrow{0.8} .$$
$$r3: A \xleftarrow{1} C; \langle ind, *, - \rangle .$$
$$r4: A \xleftarrow{0.6} B, A; \langle ind, *, * \rangle .$$

Figure 1.1: A logic program with uncertainty: $P_{1.1}$

In program $P_{1.1}$, rules *r1* and *r2* define the fact-certainty pairs *B:0.5* and *C:0.8* respectively to indicate that *B's* certainty is *0.5* and *C's* certainty is *0.8*. The value on $\leftarrow$ in each rule represents the truth degree that the body of the rule implies the rule head. In rule *r3*, the truth degree that C implies A is 1 and that of rule *r4* is 0.6. The triple $\langle f_d, f_p, f_c \rangle$ associated with each rule indicates the "combination functions" used to compute with certainties. f_c is a conjunction function derives the certainty of body, given the certainties of each subgoal in the body. For instance, * in *r4* indicates the certainties of B are multiplied to yield the certainty of the body of r4. It is clear that the choice of f_c

is immaterial in rule *r3*, any reasonable f_c would return the same value. We use "–" to indicate this situation. The propagation function f_p, such as * in *r4*, is used to combine the certainty of the rule body with the certainty of the rule, to obtain the certainty of the head atom. In case there is more than one derivation that infers the same atom, the disjunction function f_d is applied to combine them into one single value. In this program, $f_d = ind$ is associated with A, where $ind(\alpha, \beta) = \alpha + \beta - \alpha * \beta$.

Let us first consider the Naive fixpoint evaluation of $P_{1.1}$, by which every rule is evaluated at every iteration. We use I_j to denote the collection of fact-certainty pairs obtained at iteration *j*. At iteration 1, we get $I_1 = \{B:0.5, C:0.8\}$ from *r1* and *r2*. At iteration 2, rule *r3* generates A:0.8 through $f_p = 1 * 0.8$, and hence $I_2 = \{A:0.8, B:0.5, C:0.8\}$. At iteration 3, two derivations of A is produced; one from *r4* with certainty 0.6*(0.5*0.8)=0.24 and another from *r3* with certainty 0.8, which are then combined through $ind(0.24,0.8) = 0.24 + 0.8 - 0.24 * 0.8 = 0.848$, therefore, $I_3 = \{A:0.848, B:0.5, C:0.8\}$. Since *A's* certainty is improved in I_3 compared to I_2, the fixpoint evaluation goes on to the next step in which we get $I_4 = \{A:0.85088, B:0.5, C:0.8\}$. The evaluation continues until it reaches the limit. By adopting a recurrence relation based technique, we get the final result $I_\omega = \{A:0.85106, B:0.5, C:0.8\}$ in the limit.

Next we consider the evaluation of the same program with a "straight" extension of the standard Semi-Naive method. The basic idea of the standard Semi-Naive method is to apply at each iteration *i*, only those rules for which "something new" was obtained for the

rule body at iteration *i-1*. Extending this idea, we derive $I_1 = \{B : 0.5, C : 0.8\}$ at iteration 1. At iteration 2, only r3 is applied since *r1* and *r2* have nothing "new" in the body. Thus, $I_2 = \{A : 0.8, B : 0.5, C : 0.8\}$. Similarly, only *r4* is applied at iteration 3 and which yields *A:0.24* from *r4*. This certainty of A is then combined with the best certainty 0.8 of A known previously in I_2. This gives, $I_3 = \{A : 0.848, B : 0.5, C : 0.8\}$. At iteration 4, again, only r4 is applied and derives *A:0.2544*. Through *ind*(0.848, 0.2544), we obtain $I_4 = \{A : 0.886688, B : 0.5, C : 0.8\}$, which is incorrect and continues to yield accumulative wrong result in the limit. By adopting a recurrence relation based technique, we get the final result $I_\omega = \{A : 1, B : 0.5, C : 0.8\}$ at iteration ω.

Existing powerful and efficient system such as CORAL and XSB evaluate logic programs in this manner, thus disallowing us in general to take advantage of them in our context of evaluating programs with uncertainty. Careful extension of the standard Semi-Naive method is thus required [31].

1.2 Contributions of the Thesis

In this section, we will highlight our main contributions in this research. We will also indicate the chapter or section in which the contribution is discussed.

A background review of inapplicability of the standard Semi-Naive (SN) method (section 3.1) for uncertainty computation is explored. We also recall earlier results that identify classes of programs in the parametric framework for which the standard Semi-Naive evaluation method may be used.

A careful extension of standard SN algorithm for logic framework with uncertainty is proposed. This extended SN method is sensitive to duplicates of the derived atom-certainty values and thus called multiset-based Semi-Naive technique (section 3.2).

Moreover, to further increase the efficiency caused by the SN evaluation, we introduce a refined version of multiset-based SN method, called Semi-Naive with Partition (SNP). This is done by enabling enough derivation sources tracking during the evaluation (section 3.3).

Stratification is a concept introduced in standard logic programs and deductive databases with uncertainty to influence the run time environment to compute a desired model, called perfect model [35]. While considering the efficient evaluation beyond the bottom-up approach, we claim that, unlike the classical logic program, the stratification of a parametric program may provide a considerable efficiency for query processing. A desired stratification of a parametric program may maximally reduce the intermediate certainty computation compared to the other stratification (section 3.4). We present a method of finding desired stratification of a parametric program (section 5.3.4).

To show that the ideas in this thesis lend themselves to an efficient environment for deduction with uncertainty, we have built a prototype deductive database system with uncertainty (DDBS+Uncertainty), which implements the proposed optimizations introduced in this research. This prototype is implemented in C++ (chapter 4 and 5). To reduce the indexing cost, we introduce a technique, based on subgoal reordering, to reduce the number of indices while supporting efficient evaluations of programs (section 5.2).

In order to measure the efficiency of the proposed techniques, we conduct a number of experiments of evaluating a variety of classes of parametric programs and compare the execution time of different techniques (chapter 6). To this end, we also developed programs to generate different classes of suitable data of different structures, facts to be more precise.

1.3 Thesis Outline

The rest of this thesis is organized as follows. In Chapter 2, we give some background knowledge about logic frameworks with uncertainty. This includes the classification of the logic framework with uncertainty and a brief review of the parametric framework [19]. Some existing implementations of the logic framework with uncertainty are also discussed in this chapter. The stratified evaluation of Datalog with negation programs is presented in section 2.4. We assume that the reader is familiar with the concepts and techniques of logic programming, and deductive databases. The introduction of the preliminaries of them is ignored in this thesis. [21] is an excellent source for logic programming. Moreover, [7] surveys what you always want to know about Datalog.

In chapter 3, we study efficient evaluation techniques for programs with uncertainty. The multiset-based Semi-Naive method, Semi-Naive with Partition method, and stratified evaluation are also presented in this chapter.

The architecture of our system prototype is provided in chapter 4. The components of the system and the interactions among them are discussed in detail.

The system implementation issues are discussed in chapter 5. We highlight several implementation decisions that allow us integrate diverse evaluation techniques and

optimization into an efficient evaluation scheme. Specially, we consider the issues of: *(1)* data representation, *(2)* relation representation, *(3)* index structure, and *(4)* evaluation techniques.

In chapter 6, a number of experiments are conducted to demonstrate the efficiency gained from different query processing techniques. We report the experimental results together with some analysis which provide insight to the evaluation scheme developed.

Finally, we give a retrospective discussion of the efficient evaluation with uncertainty and outline some future research directions in chapter 7.

Chapter 2

Background and Related Work

Before discussing the efficiency issue of the query processing, and our system design and implementation, we quickly review the frameworks proposed for deductive databases with uncertainty from [19]. We especially review the basic concepts and development of the *parametric framework* [19], which is a generic IB framework that allows us to address the problem of query optimization in a "framework independent" manner. Furthermore, some existing query optimization techniques in classical deductive databases are also discussed in this chapter to help understand the proposals of the query optimization techniques for deductive databases with uncertainty.

We assume the reader is familiar with the basic concepts and techniques of logic programming and deductive databases, such as rules, EDB and IDB predicates, subgoals, least fixpoint, etc. For more details, please refer to [9, 34].

2.1 Classification of Deductive Databases with Uncertainty

In the past 20 years, numerous frameworks have been proposed for uncertainty by extending the standard logic database programming with its advantages of modularity and its powerful top-down and bottom-up query processing techniques. There are a number of basis on which these frameworks may differ. On the basis of their underlying

mathematical foundation of certainty, these frameworks may vary and include probability theory [16, 17, 23, 24], fuzzy set theory [36], multi-valued logic [14], possibilistic logic [11], and so on. These frameworks differ in *(1)* their underlying notion of uncertainty; *(2)* the way in which uncertainties are manipulated; and *(3)* the way in which uncertainty is associated with the facts and rules in a program. On the basis of *(3)*, these frameworks can be classified into two categories [19]: annotation based (**AB**, for short) [3, 2, 14] and implication based (**IB**) [36, 11, 17, 18]. The Comprehensive comparison of these approaches can be found in [19].

A typical rule r in an **AB** framework is an expression of the form:

$$H : f(\beta_1,...,\beta_n) \leftarrow B_1 : \beta_1,...,B_n : \beta_n.$$

where H and B_is are facts, β_i is an annotation constant or variable, and f is an n-ary function to compute the certainty of the rule head by "combining" the certainties of the subgoals in the rule body. This rule asserts "the certainty of H is at least (or is in) $f(\beta_1,...,\beta_n)$, whenever the certainty of atom B_i is at least (or is in) β_i, $1 \le i \le n$." Alternate derivations of the same atom from the program are "combined" using a user-defined disjunction function.

For **IB** framework, a rule r is an expression of the form:

$$H \xleftarrow{\alpha} B_1,...,B_n.$$

where H and B_is are facts, α is a certainty value, which means that the certainty that the rule body implies the head is α. Given a certainty valuation υ of B_is, the certainty of H is computed by taking the "conjunction" of $\upsilon(B_i)$ and then "propagating" it to the rule head. Aside from the syntactic difference between the AB and IB approaches, there are

other important differences. For AB framework, the annotation functions are unconstrained, or at least not discussed. However, the computation in **IB** frameworks are constrained by some principle making sure the certainty computation makes intuitive sense. Details of certainty computation will be discussed later. In our study, we focus on the programs in IB frameworks.

2.2 The Parametric Framework: A Review

The parametric framework [19] is a generic IB framework, which unifies and /or generalizes all the **IB** frameworks. Every query programs in an **IB** framework may be converted to a parametric program by selecting different parameters.

2.2.1 The Basic Concepts

Let $\boldsymbol{L}$ be an arbitrary first order language that contains infinitely many variable symbols, finitely many constants and predicates, but no function symbols. While $\boldsymbol{L}$ does not contain function symbols, it contains symbols for families of *propagation* (F_p), *conjunction* (F_c), and *disjunction functions* (F_d). These functions are collectively called as combination functions $F = F_c \cup F_p \cup F_d$.

Definition 1. *A parametric program (p-program) P is a 5-tuple* $\langle T, R, D, P, C \rangle$, *whose components are defined as follows:*

1) $L = \langle T, \preceq, \otimes, \oplus \rangle$ *is a complete lattice, where T is a set of certainty values, partially ordered by* $\preceq$, $\otimes$ *is the meet operator, and* $\oplus$ *is the join. The least element of the lattice is denoted by* $\perp$ *and the greatest element by* T.

2) *R is a finite set of parametric rules (p-rules), each of which is an expression of the form:* $H \xleftarrow{\alpha} B_1,...,B_n;\langle f_d, f_p, f_c\rangle.;$, *where* H, $B_1,...,B_n$ *are atomic formulas, and* α *is a certainty value in* $T-\{\perp\}$.

3) $f_d \in D$ *is the disjunction function associated with the head predicate.*

4) $f_p \in P$ *is the propagation function associated with the rule.*

5) $f_c \in C$ *is the conjunction function associated with each p-rule in P.*

We use $\pi(A)$ to denote the predicate symbol of a form A and $Disj(\pi(A))$ to denote the disjunction function associated with $\pi(A)$. We remark that not every rule will be explicitly associated with these three functions. In case the rule represents an EDB fact, only disjunction function is needed. We use "–" when the choice of conjunction and propagation functions is immaterial, for example $\langle f_d,-,-\rangle$. This is when the rule body has single subgoal.

It is important to be aware that the semantics of the parametric framework is based on multisets since some combination functions, especially some disjunction functions, might be sensitive to duplicates. $\{|...|\}$ is used to represent multisets.

In general, the following properties that different kinds of combination functions must posses:

1) Monotonicity: $f(\alpha_1,\alpha_2) \preceq f(\beta_1,\beta_2)$,whenever $\alpha_i \preceq \beta_i$, for $i \in \{1,2\}$;

2) Continuity: f is continuous w.r.t. each one of its arguments.

3) Bounded-Above: $f(\alpha_1,\alpha_2) \preceq \alpha_i$, for i =1,2, $\forall \alpha_i \in T$;

4) Bounded-Below: $f(\alpha_1,\alpha_2) \succeq \alpha_i$, for i =1,2, $\forall \alpha_i \in T$;

5) Commutativity: $f(\alpha_1,\alpha_2) = f(\alpha_2,\alpha_1),\ \forall\alpha_1,\alpha_2 \in T$;

6) Associativity: $f(\alpha_1, f(\alpha_2,\alpha_3)) = f(f(\alpha_2,\alpha_3),\alpha_1),\ \forall\alpha_1,\alpha_2,\alpha_3 \in T$;

7) $f(\{|\alpha|\}) = \alpha,\ \forall\alpha \in T$;

8) $f(\phi) = \perp$;

9) $f(\phi) = \mathsf{T}$;

10) $f(\alpha,\mathsf{T}) = \alpha,\ \forall\alpha \in T$;

The above properties help to define different kinds of combination function.

Postulate 1. *Let T be the certainty domain and B(T) be a finite multiset of T. A disjunction function in the parametric framework is a mapping from B(T) to T and satisfies properties 1, 2, 4, 5, 6, 7, and 8 mentioned above.*

Postulate 2. *Let T be the certainty domain and B(T) be a finite multiset of T. A conjunction function in the parametric framework is a mapping from B(T) to T and satisfies properties 1, 2, 3, 5, 6, 7, 9, and 10 mentioned above.*

Postulate 3. *Let T be the certainty domain. A propagation function in the parametric framework is a mapping from TxT to T and satisfies properties 1, 2, 3, and 10 mentioned above.*

Postulate 4. *Let P be a p-program, B_P be the Herbrand base of P, and T be the certainty domain. A valuation υ of P is a mapping from B_P to T, which assigns to every ground atom in B_P, a certainty value in T.*

Postulate 5. *Let r be a p-rule in a p-program P. A ground instance of r is a rule obtained from r by replacing all occurrences of each variable in r with an element of the Herbrand Universe.*

Postulate 6. *The Herbrand instantiation of a p-program P, denoted as* P^*, *is a collection of all ground instances of all p-rules in P.*

2.2.2 Fixpoint Evaluation of P-program

The fixpoint theory developed for p-programs is based on the notion of the immediate consequence operator T_P, defined as a mapping from a set of valuation γ_P to γ_P, such that for every $\upsilon \in \gamma_P$ and every atom $A \in B_P$: $T_P(\upsilon)(A) = f_d(X)$, where $f_d = Disj(\pi(A))$, and X is a multiset of certainties such that $X = \{| f_p(\alpha_r, f_c(\{| \upsilon(B_1),...,\upsilon(B_n) |\})) | (A \xleftarrow{\alpha_r} B_1,...B_n; \langle f_d, f_p, f_c \rangle) \in P^* |\}$. The bottom-up evaluation of T_P is then defined as in the standard case. It is shown that T_P is both monotone and continuous and for any p-program P, the least fixpoint of T_P, denoted $lfp(T_P)$, is equivalent to the declarative semantics of P [19].

The basic bottom-up fixpoint evaluation is called Naive evaluation. For a p-program, the Naive evaluation is similar to the Naive evaluation in the Datalog, the standard deductive database. At every iteration, all p-rules/facts are applied and defined. The difference is that in the Naive evaluation for a p-program we need to aggregate (or combine) the derived certainties of a fact into one, at every iteration. The fixpoint is reached when not only all facts are generated but also that the certainties of them can not be further improved. The Naive algorithm for p-programs is shown in Figure 2.1.

Procedure: Naive (P, D; $lfp(T_{P \cup D})$)
  forall $A \in B_P$:
1 $\upsilon_0(A) := \perp$;
2 $M_1(A) := \{| \alpha | (A : \alpha) \in D |\}$;
3 $\upsilon_1(A) := f_d(M_1(A))$, where $f_d := Disj(\pi(A))$;
  end forall

4 $New_1 := \{A \mid (A:\alpha) \in D\}; i := 1;$

while $(New_i \neq \phi)$

5 $i := i+1;$

forall $\forall (r : A \xleftarrow{\alpha_r} B_1,...,B_n; \langle f_d, f_p, f_c \rangle) \in P^*:$

6 $M_i(A) := \{| \ f_p(\alpha_r, f_c(\{| \ \upsilon_{i-1}(B_1),..., \upsilon_{i-1}(B_n) \ |\})) \ |\};$

7 $\upsilon_i := f_d(M_i(A))$, where $f_d := Disj(\pi(A));$

end forall;

8 $New_1 := \{A \mid A \in B_P, \upsilon_i(A) \succ \upsilon_{i-1}(A)\};$

end while

9 $lfp(T_{P \cup D}) := \upsilon_i;$

end procedure

Figure 2.1: A Naive algorithm for evaluating p-programs

2.2.3 Classification of Disjunction Functions

In [19], the family F_d of disjunction functions is classified into three types, called *types 1-3,* defined as follows.

Definition 2. *Let f_d be a disjunction function in the parametric framework. Then we say:*

1) *f_d is of type 1 provided,* $f_d = \oplus$, i.e., when *f_d coincides with the lattice join.*

2) *f_d is of type 2 provided,* $\oplus(\alpha,\beta) \prec f_d(\alpha,\beta) \prec \mathsf{T}, \ \forall \alpha,\beta \in T - \{\bot, \mathsf{T}\}.$

3) *f_d is of type 3 provided,* $\oplus(\alpha,\beta) \prec f_d(\alpha,\beta) \preceq \mathsf{T}, \ \forall \alpha,\beta \in T - \{\bot, \mathsf{T}\}.$

Intuitively, a type 1 disjunction function means that it coincides with the join $\oplus$ operator in the underlying certainty lattice T, while type 2 functions are strictly greater than $\oplus$, whenever its arguments are different from the bottom and top elements of T. A type 3 disjunction function behaves as the join operator at some input arguments and is strictly greater than join at other points. The difference between type 2 and 3 is that the value returned by a type 2 function always keeps increasing when supplied with "better"

argument values, while a type 3 function may return T for some such values. These two types of disjunction functions may cause a fixpoint evaluation of p-programs not to terminate. In our context of query optimization, the distinction between type 2 and 3 is not necessary. We consider them to be in the same category and be different to type 1.

2.3 Implementations of DDB+Uncertainty

Even though there have been numerous proposals for AB/IB frameworks, there has been little progress in their effective and efficient implementation. In [20], Leach and Lu discuss implementation issues in the context of an AB framework with set-based semantics. They develop a top-down query processing in annotated logic programs that uses constraint solving.

For IB frameworks, Shiri [31] developed Problog, a system prototype to evaluate p-programs. It takes the top-down approach to evaluate p-programs, on top of the XSB system [29], which is a powerful logic programming system. Even though Problog evaluate p-programs on different certainty lattices, provided the disjunction function associated with recursive predicates is of type 1. This is because XSB is a system designed for classical deductive databases and is set-based, in general. Its support of multisets is not suitable in our context, as discussed later.

Another attempt to implement the parametric framework is a version of CORAL, which supports an annotation, called @aggsel_per_iteration, which can apply the disjunction functions. Similar to Problog, it only works on disjunction functions of type 1 unless it uses the Naive method to evaluate p-programs having disjunction functions of type 2 or 3

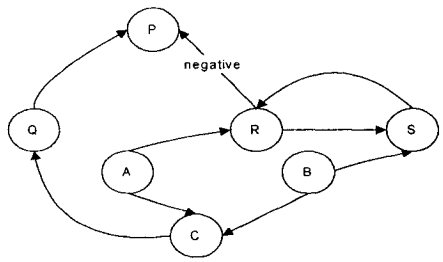
[32]. Our objective in this research is thus to design and implement a system for evaluation of all p-programs correctly and efficiently.

2.4 Stratified Evaluation of Programs in Datalog¬

To extend the expression power of pure Datalog, negation has been introduced into the body of rules. The result language is called Datalog¬. However, allowing negation may result a Datalog¬ program to have more than one *minimal Herbrand model*, while one of these minimal Herbrand models, called *perfect model*, is selected to define the semantics of the program. A normal way of doing such selection under closed world assumption (CWA) is the stratified evaluation of Datalog¬ programs, which stratifies a program based on its predicate dependency graph and evaluates it stratum-by-stratum in the order of lower stratum first. Figure 2.2 shows an example program $P_{2.1}$ in Datalog¬ and its predicate dependency graph.

$r1 : P \leftarrow Q, \neg R.$
$r2 : R \leftarrow S, A.$
$r3 : S \leftarrow R, B.$
$r4 : C \leftarrow A, B.$
$r5 : Q \leftarrow C.$

a) Program $P_{2.1}$



b) Dependency graph of $P_{2.1}$

Figure 2.2: An example of Datalog¬ program and its dependency graph

Definition 3. *Let P be a logic program, the predicate dependency graph pdg(P) is a directed graph, in which*

Vertices are predicates of P and edges are defined as follows:

- *a positive edge (p, q) if there is a rule in P in which q is the predicate of a positive subgoal and p is the head predicate*
- *a negative edge (p, q) if there is a rule in P in which q is the predicate of a negative subgoal and p is the head predicate*

Notice that not all Datalog$^{\neg}$ programs are stratified. A Datalog$^{\neg}$ program is stratified if no cycle in its predicate dependency graph contains a negative edge.

Definition 4. *A strongly connected component (SCC) of a directed graph is a maximal subgraph such that for every pair of vertices u, v in the subgraph, there is a directed path from u to v and a directed path from v to u.*

In Figure 2.2 (b), the subgraph containing vertices *R* and *S* forms SCC since *R* can reach *S* and *S* can reach *R*. It is not hard to be convinced that, given a logic program, all predicates in SCC are mutually recursive and each relies on others for its definition. In the situation that a program is to be evaluated stratum-by-stratum, the predicates in a particular SCC must be in the same stratum and therefore evaluated together. This is very important when we try to stratify a program and evaluate it sequentially.

Definition 5. *The definition of predicate H in a Datalog program P is a set of rules* $\{r_1,.., r_n\}$ *in P with H as the head predicate.*

Definition 6. *A stratification of a Datalog$^{\neg}$ program P partitions P into a set of disjoint rules. It is a mapping m from the set of rules in P to a set of nonnegative integers such that:*

1. *if a positive edge (p, q) is in pdg(P), then* $m(r_p) \geq m(r_q)$
2. *if a negative edge (p,q) is in pdg(P), then* $m(r_p) > m(r_q)$,

where r_p is the rule having predicate p as the head.

In other words, a valid stratified evaluation always defines the negative subgoal of a predicate H before defining H. Obviously, a stratified program may have several different stratifications. Program $P_{2.1}$, for example, has the following stratification: $P_1 = \{r2, r3\}$ $P_2 = \{r1, r4, r5\}$, where $P_{2.1} = P_1 \cup P_2$. An alternative stratification of $P_{2.1}$ is: $P_1 = \{r4\}$, $P_2 = \{r2, r3, r5\}$, $P_3 = \{r1\}$, where $P_{2.1} = P_1 \cup P_2 \cup P_3$. To find a stratification of a given program, many algorithms have been proposed [8, 34]. Stratification algorithms attempt to put the definition of an IDB predicate and its positive subgoal in the same stratum while defining its negative subgoal in some lower stratum. This helps to minimize the number of strata. For example, the first stratification mentioned above for the program $P_{2.1}$ is the one selected in this case.

Chapter 3

Efficient Evaluation of LP+DDB with Uncertainty

From the discussion of the Naive evaluation with uncertainty in section 2.2, it is clear that Naive evaluation is inefficient since a fact derived at a particular iteration continues to be derived at every subsequent iteration, without improving its certainty. Therefore, we need alternative efficient techniques that avoid such redundant computation. Even though there are many efficient techniques proposed for the standard deductive databases, however, as mentioned in chapter 1, a "straight" adaptation of them may not be suitable in our context. We need to develop such techniques for logic programming and deductive databases (LP+DDB) with uncertainty. In this chapter, we start by studying why the standard Semi-Naive (SN) algorithm is not applicable to uncertainty evaluation (section 3.1). We remark that the uncertainty evaluation must be based on multisets rather than sets. According to this, two multiset-based Semi-Naive algorithms are introduced in sections 3.2 and 3.3. In section 3.4, we show how "stratification" may further improve the efficiency, even though we do not consider negation in our study.

3.1 Inapplicability of Classical SN Algorithms

In chapter 1, we have demonstrated that a "straight" extension of classical SN algorithms does not give a correct evaluation with uncertainty. The main reason for this is that the

evaluation in the standard deductive databases is actually set-based. Even though some standard deductive database systems, such as Coral [27] and XSB [29], support multiset, they do not compute as desired. For efficiency purposes, a standard deductive database system generates "incomplete" multisets as long as this does not affect the final query result [28]. This is not a problem when uncertainty is not presented. If we look at the so-called basic Semi-Naive (BSN) algorithm presented in [1], we find that BSN actually generates an incomplete multiset during the evaluation. However, for logic programs with uncertainty, the "completed" multiset is required when there are disjunction functions of type 2, or of type 3 [19].

Moreover, many classical SN algorithms may not respect the border between evaluation iterations. Since there is no aggregation operation between two consecutive iterations, it is not mandatory to evaluate each iteration in isolation of others. To be more precise, the only disjunction function in the standard case is *max*, implicitly, which is also the lattice join. Based on this observation, many improved SN algorithms mix (or overlap) the evaluation of two or more iterations to achieve higher evaluation efficiency [26, 13, 1]. For instance, [26] introduces two algorithms which apply a rule to produce new facts, and then immediately makes these facts available to the subsequent rules applications at the same iteration. The purpose of this technique is to reduce the number of rule applications and iterations for an evaluation. Unfortunately, this method does not apply in our context, in general. Since disjunction functions are applied at the end of each iteration, all derivations must be obtained and aggregated iteration by iteration to ensure the correct evaluation. Violating this may cause an incorrect result when uncertainty is presented.

Even though the standard SN method is inapplicable to the evaluation with uncertainty in general, we find that, however, the existing inference systems may be used to evaluate the logic program with uncertainty in some particular cases. The implementation of the Problog [31], which implements the parametric framework using a meta-programming in XSB, is an example. Another implementation is a special version of the CORAL, which provides an annotation to perform the disjunction function.

A question comes at this stage: In which cases are the existing efficient engines applicable? In other words, we want to know in which cases the multiset-based evaluation is equivalent to the set-based evaluation. Since some existing deductive systems support multisets at each iteration but do *set operation* between iterations, it is obvious that if the input program is non-recursive, only one evaluation iteration is needed; the existing systems will perform correctly. In case the input program is recursive, things get more complicated because many iterations are needed. In this situation, whether existing systems are applicable depends strongly on the type of disjunction functions used, since the requirement of the multiset-based evaluation arises from the aggregation operation (through the disjunction functions used). If the disjunction functions used are set-based execution, the input program can then be evaluated using existing inference systems. For a recursive rule in a given p-program, if the associated disjunction function is of type 1, which means $f_d(\alpha,\beta) = \oplus(\alpha,\beta)$, set-based operation has the same effect as multiset-based operation since the number of copies of α or β does not affect the result of f_d. For example, suppose fact A has a multiset of certainties {| 0.1, 0.2, 0.2, 0.3 |}, generated from different derivations, and A has an associated disjunction function $f_d = max$. The result of the disjunction function is 0.3, no matter the number of

copies certainties is in a set or multiset. Moreover, the mixture of derivations over different iterations will not affect the final result in this case.

As identified in [32], a correctness requirement should obeyed during the evaluation p-program: *if the disjunction functions presented in a p-program are of type 2 or 3, a correct evaluation procedure should combine the certainties of atoms derived at the same iteration, and not combine newly derived certainties with prior certainties of the same atom from the same rule.*

There are two factors deciding the applicability of the existing systems for computing with uncertainty: (1) whether the input program is recursive or not, and (2) the recursive rules in the program are associated with type 1 disjunctions f_d or not. If the program is not recursive, or the disjunction associated with every recursive predicate is of type 1, then existing system inference systems can be used.

Proposition 1. *For any p-program* ***P*** *in the parametric framework, the fixpoint evaluation of* ***P*** *using the set-based Semi-Naive evaluation method generates the same result as the multiset-based Naive method if the disjunction functions associated with the recursive predicates in* ***P*** *is of type 1.*

Proof:

It is obvious that the result returned by a disjunction function of type 1 is not changed if there are duplicates in the input argument. That is, the input multiset argument can be turned into a set. Hence, the set-based Semi-Naive method yields the same result as the multiset-based Naive method.□

Notice that, as mentioned before, the existing systems such as CORAL and SXB are applicable for evaluation with uncertainty in case the input program is not recursive, no matter what kind of disjunction function it has. This does not violate Proposition 1. It is because they support multiset within the iteration (but not inter-iterations). Therefore, for evaluation of non-recursive program, the definition of every IDB predicate is done only at one iteration, whereas the iteration at which different predicates are defined may be different. In this situation, both CORAL and XSB produce the same result as the Naive method of the evaluation with uncertainty.

3.2 A Multiset Based Semi-Naive Algorithm

From the discussion in section 3.1, it is clear that existing systems cannot be used in general to evaluate all logic programs with uncertainty (the p-programs in our context). We need to develop efficient, multiset-based algorithms and techniques for deductive databases with uncertainty. We developed a bottom-up Semi-Naive method for evaluating query programs with uncertainty and established its equivalence with the corresponding Naive method [32]. Here, we review the proposed method, and then introduce ways to further improve the efficiency.

The Naive evaluation method of p-programs was discussed in section 2.2, and its procedure is shown in Figure 2.1. Similar to the Naive method for standard deductive databases, this method is inefficient since it repeats many redundant computations. The inefficiency is due to the redundant computation in lines 6 and 7 in Figure 2.1. An atom A derived at iteration i, continues to be derived at every subsequent iteration, even if its associated certainty is not changed. To improve this situation, we should identify and

apply only those rules that have something new (either a new fact or an old fact with a higher certainty) in the body at the subsequent iteration. Other rules would not have further contribution to the evaluation. Based on this observation, we suggest to bookkeep the derivations of those rules that have no "new" subgoal and evaluate only those rules that have something "new" in the subgoals. In this case, some computations of those unchanged derivations are saved, and have results in increased efficiency.

To achieve this, we associate with every ground atom A in the given p-program, a pair $\langle M_i, \sigma_i \rangle$, where M_i is a multiset containing all certainties derived so far from different paths, and σ_i is the certainty of A, obtained by applying to M_i the computation of the disjunction function associated with the predicate of A. That is, $\sigma_i = f_d(M_i)$, where $f_d = Disj(\pi(A))$. Every element in M_i is of the form (r: α), indicating that a derivation of A with certainty α is obtained by rule r. This information is used to identify the certainties of A derived by rule r, a subgoal of which was identified as "new" at the last iteration. This is important when we want to replace these certainties with the better certainties derived by the improved subgoal(s). The use of multiset for storing certainties is also crucial in our context. It guarantees that the number of copies of the same certainty, which may be sensitive for some disjunction function, is respected.

Figure 3.1 presents the multiset-based Semi-Naive algorithm [32]. We will use SN to refer to this algorithm. As in the Naive algorithm, SN takes a collection D of fact-certainty pairs, (basically, the extensional database EDB), and a collection P of p-rules, and produces the least model of the program, when it terminates, i.e., when the fixpoint

of $T_{P\cup D}$ is reached. In the algorithm, we use symbols $\cup$ and $-$ to denote the multiset union and difference operations, respectively.

If we compare the SN and Naive algorithms, we will see that the source of efficiency of the SN algorithm is due to lines 8 and 9 in Figure 3.1. At iteration i, only rule r, which has at least one improved subgoal, will be evaluated. If fact A is derived by r through some improved subgoal B, we remove from the multiset $M_i(A)$ all elements (r: α), for every α associating with r. Then the derivations that derive A by r are done again by applying better certainty of subgoal B. The new results (r: β) for A are added into $M_i(A)$ to form $M_{i+1}(A)$ for the further evaluation. It is important to note that this replacing in $M_i(A)$ makes the result at each iteration to be identical to the Naive method. It ensures that no derivation of A from the same rule r is mixed with the derivations of A by r at earlier iterations. When this replacement is complete, a new certainty of A is obtained in line 10, using the disjunction $f_d = Disj(\pi(A))$, the disjunction function, associated with the predicate symbol of A.

Procedure: $Semi-Naive\ (P,\ D,\ lfp(T_{P\cup D}))$

forall $A \in B_P$:

1 $\quad\upsilon_0(A) := \perp$;

2 $\quad M_1(A) := \{|\ \alpha\ |\ (A:\alpha) \in D\ |\}$;

3 $\quad\upsilon_1(A) := f_d(M_1(A))$, where $f_d := Disj(\pi(A))$;

end forall

4 $New_1 := \{A\ |\ (A:\alpha) \in D\}$; $i := 1$;

while $(New_i \neq \phi)$

5 $\quad i := i+1$;

forall $A \in B_P$:

if $\exists(r : A \xleftarrow{\alpha_r} B_1,...,B_n;\langle f_d, f_p, f_c\rangle) \in P^*$

such that $\exists B_j \in New_i$, for some $j \in \{1,...,n\}$:

then begin

7 $M_i(A) := M_{i-1}(A)$;

8 **forall** $(r : A \xleftarrow{\alpha_r} B_1,...,B_n; \langle f_d, f_p, f_c \rangle) \in P^*$
such that $\exists B_j \in New_i$, for some $j \in \{1,...,n\}$:

9 $M_i(A) := M_i(A) - \{|\sigma^r_{i-1}(A)|\} \cup \{|\sigma^r_i(A)|\}$, where
$\sigma^r_i(A) := f_p(\alpha_r, f_c(\{|\upsilon_{i-1}(B_1),...,\upsilon_{i-1}(B_n)|\}))$;
end forall;

10 $\upsilon_i := f_d(M_i(A))$, where $f_d := Disj(\pi(A))$;
end if;

11 **else** $\upsilon_i(A) := \upsilon_{i-1}(A)$;
end forall;

12 $New_i := \{A \mid A \in B_P, \upsilon_i(A) \succ \upsilon_{i-1}(A)\}$;

13 **end while**

14 $lfp(T_{P \cup D}) := \upsilon_i$;

end procedure

Figure 3.1: A Semi-Naive algorithm for evaluating programs with uncertainty

After applications of all possible rules at a particular iteration, we are able to re-define *New* in line 12, which is a set that includes every fact whose certainty is improved at this iteration, compared to the previous iteration. If *New* is empty, it means the certainty of no fact is improved and hence the evaluation terminates and the result v_i is returned (line 14). Otherwise, the evaluation continues to the next iteration.

Theorem 1 below establishes the correctness of the SN algorithm proposed above.

Theorem 1. *Let P be any p-program in the parametric framework and D be a collection of facts. A fixpoint computation of P on D using the Semi-Naive algorithm above produces the same result as the Naive method.*

Proof:

It is obvious that the application of a rule having no "new" subgoal will generate the same derivation(s) as that of the previous iteration. Even though the SN method does not evaluate this kind of rules, it records the result of them in M_i. For those rule(s) that have

"new" subgoal, SN removes their previous derivation result from M_i first, and recomputed them. The new result is then inserted into M_i. So, *at every iteration, the derivation result in* M_i *is the same as the result derived by Naive method.* By applying the same disjunction function, the certainty of every fact A is also identical to that in Naive method. When the set *New* is empty, no rule can generate the "new" fact-certainty pair. Hence, two consecutive iterations generate the same result, and SN terminates at the same situation as the Naive method. Therefore, the Semi-Naive method yields the same result as the Naive method. □

Let us consider again the program $P_{1.1}$ presented in chapter 1 to illustrate the SN algorithm. Initially, every fact-certainty pair is assigned certainty 0. At iteration 1, only two fact-certainty pairs are improved: $I_1 = \{B : 0.5, C : 0.8\}$. Rules r_3 and r_4 do not produce any such pair. After the evaluation of iteration 1, we find that r_1 and r_2 need not to be evaluated again, since there is no subgoal for these two rules. At iteration 2, since r_3 and r_4 have some improved subgoals, both are applied. As a result, r_3 produces A: 0.8 and r_4 produces nothing. Hence, $I_2 = \{B : 0.5, C : 0.8, A : 0.8\}$. Since r_3 has only one subgoal, C, and C will not be further improved, r_3 will not be evaluated at the subsequent iterations. At iteration 3, r4 is applied, which generates fact A with certainty 0.6*0.5*0.5=0.24. Now, the multiset $M_3(A)$ contains two elements: $\{|\, r_3 : 0.8, r_4 : 0.24 \,|\}$. Using the disjunction function f_d associated with A, we obtain $f_d = ind(0.8,0.24) = 0.848$, and hence $I_3 = \{B : 0.5, C : 0.8, A : 0.848\}$. At iteration 4, r_4 is applied again since its subgoal A is improved at iteration 3. In this situation, element $r_4 : 0.24$ in $M_4(A)$ is removed and replaced by $r_4 : 0.2544$, produced by r_4. Applying the

disjunction function $f_d = ind(0.8,0.2544) = 0.85088$, we get $I_4 = \{B : 0.5, C : 0.8, A : 0.85088\}$. Since the certainty of A improves at every iteration, this computation will terminate only in the limit. This example demonstrates that the evaluation result obtained by SN is identical with that obtained by the Naive method.

3.3 The Semi-Naive with Partition Algorithm

As mentioned in section 3.2, the SN method proposed provides an efficient evaluation by avoiding the computations of some derivations that do not yield improved certainties. This is done by identifying and recording, at every iteration, the results of non-improved derivations. In case some subgoals of a fact A derived by rule r gain improved certainties in the last iteration, all the derivations of A derived from r are removed from the records and redone by taking better certainties of their subgoals obtained at the last iteration. This operation is done at line 9 in Figure 3.1. However, not all redundant computations could be avoided by the SN method. For instance, if more than one derivations of rule r generate fact A, there may exist a case in which only one of these derivations may have some improved subgoal(s), while the others do not. According to the SN method, all derivations of a rule r that generate fact A, including those which do not have improved subgoal(s), should be removed from the bookkeeping and redone. Therefore, some redundant computations are repeated. Let us use the following program ($P_{3.1}$) to demonstrate this situation:

$r1 : a(1) \xleftarrow{0.5}.$
$r2 : a(2) \xleftarrow{0.8}.$
$r3 : b(1) \xleftarrow{0.6}.$
$r4 : c(2) \xleftarrow{0.7}.$

$r5: q(X) \xleftarrow{1} a(X); < ind, *, \_ >.$

$r6: q(X) \xleftarrow{1} c(X), q(X); < ind, *, * >.$

$r7: p(X) \xleftarrow{1} b(X), q(Y); < ind, *, * >.$

At the first iteration, *r1, r2, r3*, and *r4* generate the following fact-certainty pairs:

$$M_1 = I_1 = \{a(1): 0.5, a(2): 0.8, b(1): 0.6, c(2): 0.7\}$$

These rules will not be further applied since they have no subgoals (that may probably increase). At iteration 2, *r5* generates $q(1)$:0.5, and $q(2)$:0.8, while *r6* and *r7* generate nothing because relation "*q*" is still empty. We thus have:

$$M_2 = I_2 = \{a(1): 0.5, a(2): 0.8, b(1): 0.6, c(2): 0.7, q(1): 0.5, q(2): 0.8\}$$

At iteration 3, *r5* will not be further evaluated since its subgoal a(X) will not be improved. However, *r6* is applied and yields $q(2)$:0.56 through $c(2)$ and $q(2)$ with certainties $(0.7 * 0.8) * 1 = 0.56$, and *r7* yields $p(1)$:0.3 by joining $b(1)$ and $q(1)$, and yields p(1):0.48 by joining $b(1)$ and $q(2)$. Now, $M_3(q(2)) = \{| r5: 0.8, r6: 0.56 |\}$, and $M_3(p(1)) = \{| r7: 0.3, r7: 0.48 |\}$. Each of these multisets is then aggregated using the disjunction function *ind*. This yields $q(2)$:0.912, and $p(1)$:0.636. The result of iteration 3 is thus:

$$I_3 = \{a(1): 0.5, a(2): 0.8, b(1): 0.6, c(2): 0.7, q(1): 0.5, q(2): 0.912, p(1): 0.636\}$$

in which, $q(2)$:0.912, and $p(1)$:0.636 are in New_3. Notice that there are two derivations of *r7*, which generate $q(1)$ at iteration 3. One is $p(1) \xleftarrow{1} b(1), q(1); < ind, *, * >$, and the other is $p(1) \xleftarrow{1} b(1), q(2); < ind, *, * >$. Since $q(2)$ is a subgoal of the derivations of $p(1)$ by r7, both of these two derivations are removed from $M_4(p(1))$ and redone at next iteration, even though derivation $p(1) \xleftarrow{1} b(1), q(1); < ind, *, * >$ has no improved

subgoal: $M_4(p(1)) = \{| r7:0.3, r7:0.5472 |\}$. Similarly, *r6*: 0.56 is also removed from $M_4(q(2))$ and replaced by *r6*: 0.6384 at iteration 4. That is,

$$M_4(q(2)) = \{| r5:0.8, r6:0.6384 |\}.$$

The interpretation I_4 is thus:

$I_4 = \{a(1):0.5, a(2):0.8, b(1):0.6, c(2):0.7, q(1):0.5, q(2):0.92768, p(1):0.68304\}$
This computation goes on and terminates only at ω.

The above example suggests that the proposed SN method leaves some opportunities for further improvement of the efficiency. This is also our motivation to further refine and improve our SN method. Let us refer to the refined SN method as Semi-Naive with Partition (SNP) since it actually partitions every IDB relation into two parts: improved part and non-improved part. The improved part contains all fact-certainty pairs that are generated or improved at the last iteration, while non-improved part contains the rest of the fact-certainty pairs. Using SNP, the method focuses on derivations in which at least one of their subgoals is improved at the last iteration. The technique for partitioning will be introduced in section 5.3.1. The procedure of SNP is shown in Figure 3.2.

Procedure: Semi-Naive-with-Partition (*P*, *D*; $lfp(T_{P\cup D})$)
  forall $A \in B_P$:
1 $C_1(A) := \{| (\alpha:\phi) | (A:\alpha) \in D |\}$;
2 $\upsilon_1(A) := f_d(C_1(A)(\alpha))$, where $f_d := Disj(\pi(A))$;
  end forall
3 $New_1 := \{A | (A:\alpha) \in D\}; i := 1;$
  while $(New_i \neq \phi)$
5 $i := i+1;$
6 **forall** $A \in B_P$:
7 $C_i(A) := C_{i-1}(A);$
  forall $B \in New_{i-1} \wedge (\alpha, S_B) \in C_{i-1}(A) \wedge B \in S_B$:
8 $C_i(A) := C_i(A) - \{| (\alpha, S_B) |\}$
  end forall

forall $(r: A \xleftarrow{\alpha_r} B_1,...B_n; < f_d, f_p, f_c >) \in P^* \wedge \exists B_j \in New_i$,
where $j \in \{1,...n\}$

9 $C_i(A) := C_i(A) \cup \{|\ (\sigma_i^r(A), S_B)\ |\}$,

10 where $\sigma_i^r(A) := f_p(\alpha_r, f_c(\{|\upsilon_{i-1}(B_1),...,\upsilon_{i-1}(B_n)|\}))$, and

11 $S_B := \{B_j \mid B_j \in IDB \wedge j \in \{1,...n\}\}$

end forall;

12 $\upsilon_i := f_d(C_i(A)(\alpha))$, where $f_d := Disj(\pi(A))$;

end forall;

13 $New_i := \{A \mid A \in B_P, \upsilon_i(A) \succ \upsilon_{i-1}(A)\}$;

end while

14 $lfp(T_{P \cup D}) := \upsilon_i$;

end procedure

Figure 3.2: The Semi-Naive with Partition algorithm

There are two main problems to be solved in the SNP method. *One is how to identify the certainties whose derivations associated with of some improved subgoal(s), from the bookkeeping and remove them before starting the next iteration. Another problem is how to restrict the evaluation to consider only derivations that may generate something new.* To solve the first problem, we associate, with every ground fact A, a pair $< C_i, \sigma_i >$, where C_i is a multiset containing all certainties of A derived so far, and σ_i is A's certainty obtained by applying the disjunction function f_d associated with A. That is, $\sigma_i = f_d(C_i(A))$. If we look back at the SN method, we may see that, $M_i(A)$, which is of the form $(r:\alpha)$, is very similar to $C_i(A)$ in the SNP method. Every certainty in $M_i(A)$ is annotated with a rule *id* indicating the rule which was used to derive the certainty α. This annotation can help identify the certainties derived by a rule that has some improved subgoal(s), and then removing them from $M_i(A)$. Clearly, the certainties in $M_i(A)$ are classified by their associated rule *id*. Unlike the SN method, the SNP method replaces $M_i(A)$ with $C_i(A)$, which is of the form $(\alpha : S_B)$, where α is a derived certainty, and

S_B is a set containing all IDB subgoals in this derivation. There is no information about which rule derives which certainty in $C_i(A)$ while S_B somehow records the information about the "source" of a certainty derived. In other words, it helps to track the derivation source. In case a fact B is improved at iteration i, we may check $C_i(A)$ to see whether some element of $C_i(A)$ has S_B containing B. If so, it implies that the certainty of this element depends on the certainty of B. We then remove this element from $C_i(A)$. This is shown in lines 7 and 8 in Figure 3.2. Notice that only IDB subgoals are included in S_B. This is based on the observation that *EDB facts will never be improved after the first iteration.* So, it is not necessary to record the EDB subgoals in the corresponding S_B (see line 11). Furthermore, defining S_B as multiset is not necessary because the number of the duplicated subgoals has nothing to do with the decision that which elements should be removed from $C_i(A)$.

Addressing the second problem of SNP is straightforward, once we remove related records properly from the bookkeeping. As shown in line 9, we simply have to evaluate the ground rules containing some improved subgoal(s) and add their results to the bookkeeping. We then have the following result.

Theorem 2. *Let P be any p-program in the parametric framework and D be a collection of facts. A fixpoint computation of P on D using the Semi-Naive with Partition method produces the same result as the Naive method.*

Proof:

The basic idea of the proof is the argument that both SN and SNP methods produce the same result at every iteration, though with different efficiency, in general. From Figure

3.1 and Figure 3.2, we can see that even though, at every iteration, SN and SNP methods remove a different number of certainties from their bookkeepings at every evaluation iteration, replacing them is the same in each method. Furthermore, the certainties that are replaced in both SN and SNP methods are recomputed in the same way in each method. The rest of the certainties removed by the SN method are inserted back without any change. Therefore, SNP produces the same result as the SN method, which in turn produces the same result as the Naive method (Theorem 1), at every iteration. This establishes the equivalence of SN and SNP methods. □

Let us consider the program $P_{3.1}$ again and apply the SNP method to demonstrate how SNP improves the efficiency of the evaluation. Obviously, SNP performs the same operation as the SN method does in the first 2 iterations. This is because all facts are newly generated and not improved yet at the first 2 iterations. At iteration 3, we have $C_3(q(2)) = \{|\, 0.8 : \phi,\ 0.56 : \{q(2)\}\,|\}$, and $C_3(p(1)) = \{|\, 0.3 : \{q(1)\},\ 0.48 : \{q(2)\}\,|\}$. The interpretation I_3 would thus be:

$$I_3 = \{a(1) : 0.5,\ a(2) : 0.8,\ b(1) : 0.6,\ c(2) : 0.7,\ q(1) : 0.5,\ q(2) : 0.912,\ p(1) : 0.636\}$$

Since $q(2)$ and $p(1)$ are improved at iteration 3, $0.56 : \{q(2)\}$ is removed from $C_3(q(2))$. However, unlike in the SN method, we only remove $0.48 : \{q(2)\}$ from $C_3(p(1))$; $0.3 : \{q(1)\}$ remains since the certainty of $q(1)$ is not improved at the last iteration. Hence, only two derivations, $q(2) \xleftarrow{1} c(2), q(2); < ind, *, * >$, and $p(1) \xleftarrow{1} b(1), q(2); < ind, *, * >$ are redone for I_3. At iteration 4, we get $C_4(q(2)) = \{|\, (0.8 : \phi), (0.6384 : \{q(2)\})\,|\}$, $C_4(p(1)) = \{|\, (0.3 : \{q(1)\}), (0.5472 : \{q(2)\})\,|\}$, and thus

$I_4 = \{a(1):0.5, a(2):0.8, b(1):0.6, c(2):0.7, q(1):0.5, q(2):0.92768, p(1):0.68304\}$

At this step, it is enough to show that the SNP method produces same result as the SN, but with less computation.

3.4 Stratification and Efficiency

As discussed in section 2.4, stratification is a syntactic restriction to programs in Datalog with negation. It helps to compute the "intended" model called *perfect model* of programs with negation. Stratification has nothing to do with the efficiency. That is, Stratification was introduced in Datalog$^{\neg}$ as a way of model preference, and may or may not affect the efficiency of evaluating such programs. However, in our context, we note that a "desired" stratification could result in significant efficiency in evaluation of deductive databases with uncertainty.

Let us consider $P_{3.2}$ shown in Figure 3.3 to illustrate this point.

$r1: A \xleftarrow{0.3}.$

$r2: B \xleftarrow{0.6}.$

$r3: C \xleftarrow{0.5} B; \langle \max, \min, - \rangle.$

$r4: D \xleftarrow{1} A; \langle \max, *, * \rangle.$

$r5: D \xleftarrow{0.8} C; \langle \max, *, - \rangle.$

$r6: E \xleftarrow{1} D, A; \langle \max, *, * \rangle.$

Figure 3.3: A p-program $P_{3.2}$

The p-program $P_{3.2}$ has a stratification: $P_1 = \{r1, r2, \}$, $P_2 = \{r3\}$, $P_3 = \{r4, r5\}$, and $P_4 = \{r6\}$, where $P_{3.2} = P_1 \cup P_2 \cup P_3 \cup P_4$. We first evaluate $P_{3.2}$ in a regular way using Naive method. The result of $P_{3.2}$ at each iteration is as follows:

$I_1 = \{A:0.3, B:0.6, \}$

$I_2 = \{A:0.3, B:0.6, C:0.5, D:0.3\}$

$$I_3 = \{A : 0.3, B : 0.6, C : 0.5, D : 0.4, E : 0.09\}$$

$$lfp(P_{3.2}) = I_4 = \{A : 0.3, B : 0.6, C : 0.5, D : 0.4, E : 0.12\}$$

It is easy to see that the certainty of E is derived based on the certainty of D. When the certainty of D is improved, the certainty of E improves accordingly. At iteration 3, rule *r6* is evaluated using the certainty of D, which is less than D's value in I_4. Obviously, the computation of rule *r6,* using the intermediate certainty of subgoal D, is redundant. If we can delay the application of rule *r6* until D is done, which happens at iteration 3 for this example, we may in theory save one computation of *r6*. This efficiency can be achieved by evaluating program $P_{3.2}$ stratum-by-stratum. The evaluation result of each iteration under the stratification mentioned above is shown as follows, in which we use I^{Pi} to denote the interpretation of P restricted to rules in partition P_i.

$$I^{P1} = I_1^{P1} = \{A : 0.3, B : 0.6\}$$

$$I^{P2} = I_1^{P2} = \{C : 0.5\}$$

$$I^{P3} = I_1^{P3} = \{D : 0.4\}$$

$$I^{P4} = I_1^{P4} = \{E : 0.12\}$$

$$lfp(P_{3.2}) = I^{P1} \cup I^{P2} \cup I^{P3} \cup I^{P4}$$
$$= \{A : 0.3, B : 0.6, C : 0.5, D : 0.4, E : 0.12\}$$

We first evaluate P_1 at iteration 1 and bookkeep the result. The bookkeeping is then used as the input for evaluation of P_2. This sequence of fixpoint operations is conducted until all four strata are evaluated. The least fixpoint of the program $P_{3.2}$ is the union of all the interpretations of each stratum. Through this example, we can see that the stratified evaluation may result in increased efficiency by avoiding the computation of some intermediate certainties as well as the computations of the evaluation of lower strata, while computing a stratum at a higher level. It is clear that if some lower strata need more

iterations to produce their final results, the stratified evaluation may avoid more computations of intermediate certainties.

Our next result establishes that the stratified evaluation of parametric programs is equivalent to the Naive evaluation.

Theorem 3. *Let P be a p-program in the parametric framework and D be any collection of facts. A fixpoint computation of P on D using the stratified evaluation produces the same result as the Naive method.*

Proof:

The difference between stratified evaluation and the Naive evaluation of a p-program P is that the former allows only the final results defined in the lower strata in P to contribute to the evaluation of the rules in higher strata, while the latter takes the intermediate as well as the final values of subgoals to evaluate every rule in P. Obviously, the stratified evaluation generates the same result as the Naive evaluation if both apply the final result of subgoals to derive the final result of rule head for each rule in the given program. What we need to prove is that the intermediate values of subgoals in the Naive method do not generate the final result, or they generate the final result that is the same as what the final values of subgoals generate.

Let $r: A \xleftarrow{\alpha_r} B_1,...,B_n;\langle f_d, f_p, f_c\rangle$ be any rule in the given p-program, $\upsilon(B_j)$ is the final valuation of B_j for $j \in \{1,...,n\}$. The valuation of A derived by $\upsilon(B_j)$ is:

$$\upsilon(A) = f_d\{|\, f_p(\alpha_r, f_c(\{|\, \upsilon(B_1),...,\upsilon(B_n)\,|\}))\,|\} \qquad (1)$$

For stratified evaluation, $\upsilon(A)$ is the final result of A. Assume that the final result of A, $\upsilon'(A)$, under the Naive method is derived by some intermediate value of B_j, $\upsilon'(B_j)$, where $\upsilon'(B_j) \preceq \upsilon(B_j)$ according to property of monotonicity of the evaluation:

$$\upsilon'(A) = f_d\{|\, f_p(\alpha_r, f_c(\{|\, \upsilon'(B_1),...,\upsilon'(B_n)\,|\}))\,|\}$$

Recall that all combination functions in p-programs are assumed to be monotone (see section 2.2.1) and $\upsilon(B_j)$ is also evaluated under the Naive method. Therefore, we have that $f_c(\{|\, \upsilon'(B_1),...,\upsilon'(B_n)\,|\}) \preceq f_c(\{|\, \upsilon(B_1),...,\upsilon(B_n)\,|\})$, since $\upsilon'(B_j) \preceq \upsilon(B_j)$, and hence $\upsilon'(A) \preceq \upsilon(A)$. Since we assume that some intermediate result, $\upsilon'(A)$, is the final result of A, we have that $\upsilon'(A) \succeq \upsilon(A)$. Therefore, we can conclude that $\upsilon'(A) = \upsilon(A)$, which establishes the result. □

As mentioned in section 2.4, the concept of stratification is introduced for standard programs with negation, and is applicable if the dependency graph of the program does not have a cycle with a negative edge. Since negation is not allowed in p-programs, the stratification is universally applicable for any p-program. However, not every stratification may increase the efficiency of an evaluation. On the other hand, different stratifications may provide the same efficiency for an evaluation. Clearly, stratified evaluation does not help in case there is only one stratum in a given program. In this case, all the rules are evaluated concurrently, exactly the same as the Naive evaluation. Unfortunately, the existing stratification finding algorithms [8, 34] always stratify p-programs into single stratum since they like to put the definitions of positive subgoals in the same stratum with that of the goal. To gain the efficiency from the stratified evaluation, a *desired stratification* needs to be defined and constructed.

Definition 7. *A desired stratification of a parametric program (p-program) P is a stratification of P that:*

1) *includes at least two strata in addition to the lowest stratum that contains all the EDB tuples*

2) *If predicate B is used to defined predicate A in P, and B and A are not mutually recursive, then the definition of B is in a lower stratum than the one that includes the definition of A*

If we look at the p-program $P_{3.2}$ again, the stratification: $P_1 = \{r1, r2\}$, $P_2 = \{r3\}$, $P_3 = \{r4, r5\}$, and $P_4 = \{r6\}$ is a desired stratification for the program. Notice that not every p-programs has a desired stratification. For instance, when all the IDB predicates in a p-program are defined only by EDB predicate(s) or all of them are mutually recursive, the condition (1) in Definition 7 will not be satisfied, and therefore there is no desired stratification.

On the other hand, some p-programs may have more than one desired stratification. For instance, the following program may have three desired stratifications:

$r1: A \xleftarrow{\alpha_A} ; \langle f_d, f_p f_c \rangle$
$r2: B \xleftarrow{\alpha_B} ; \langle f_d, f_p f_c \rangle$
$r3: C \xleftarrow{\alpha_C} A; \langle f_d, f_p f_c \rangle$
$r4: D \xleftarrow{\alpha_D} B; \langle f_d, f_p f_c \rangle$
$r5: E \xleftarrow{\alpha_E} C, D; \langle f_d, f_p f_c \rangle$

one of the desired stratifications is $P_1^1 = \{r1, r2\}$, $P_2^1 = \{r3, r4\}$, and $P_3^1 = \{r5\}$. Another one is $P_1^2 = \{r1, r2\}$, $P_2^2 = \{r3\}$, $P_3^2 = \{r4\}$, and $P_4^2 = \{r5\}$. The third desired stratification is $P_1^3 = \{r1, r2\}$, $P_2^3 = \{r4\}$, $P_3^3 = \{r3\}$, and $P_4^3 = \{r5\}$. Even though a p-

program may have more then one desired stratification, these desired stratifications provide the same efficiency in terms of reducing intermediate certainty computations.

Theorem 4. *Let P be a p-program in the parametric framework that has more than one desired stratifications. The alternative stratified evaluations of P, would result the same efficiency in terms of reducing intermediate certainty computations.*

Proof:

Assume that *P* has two alternative desired stratifications, S_1 and S_2. Also assume the stratified evaluation over S_1 can avoid an intermediate certainty computation A: *d,* that cannot be avoided in evaluation over S_2. Since A: *d* is inevitable for the stratified evaluation over S_2, there must exist some mutually recursive predicate(s), whose definitions involve A: d. On the other hand, since S_1 can avoid the computation A: *d*, it means there is no mutually recursive predicate involving A: *d*, which is a contradiction. Thus, no such intermediate computation A: *d* exists, and hence S_1 and S_2 yield the same efficiency by avoiding identical intermediate certainty computations. □

An important consequence of Theorem 4 is that, to evaluate a p-program, we do not need to compare all its desired stratifications to find the best one; they all result in the same increased efficiency. Our next result illustrates that a desired stratification results in the maximum increased efficiency in term of reducing the intermediate certainty computations.

Theorem 5. *Let P be a p-program in the parametric framework. The stratified evaluation of P under any desired stratifications yields the maximal increased efficiency in terms of reducing intermediate certainty computations, compared to other stratified evaluations.*

Proof:

The basic requirement of the stratification of P is: always include mutually recursive predicates in P in the same stratum to ensure the correct definition of predicates.

Assume that *P* has two alternative stratifications S_1 and S_2, where S_1 is a desired stratification and S_2 is not. Also assume that the stratified evaluation of P over S_2 can avoid an intermediate certainty computation A: *d,* which is inevitable for evaluation over S_1. Since A: *d* is inevitable for a desired stratification, A: *d* must involve some mutually recursive predicate(s). Moreover, since S_2 can avoid A: *d*, the definition of mutually recursive predicates are stratified into different strata in S_2. Such stratification violates the basic requirement mentioned above. Therefore, S_2 does not exist. □

Now, it becomes clear that to maximize the efficiency gained from stratified evaluation, we simply need to pick any one of the desired stratifications of the given p-program and evaluate it stratum-by-stratum in order, starting with the lowest stratum. In section 5.3.4. we will discusses how to identify a desired stratification.

Chapter 4

System Architecture

Our prototype of a deductive databases with uncertainty is a single-user, in-memory system. All data, including IDB and EDB facts, rules, functions library and queries are organized and stored in main memory.

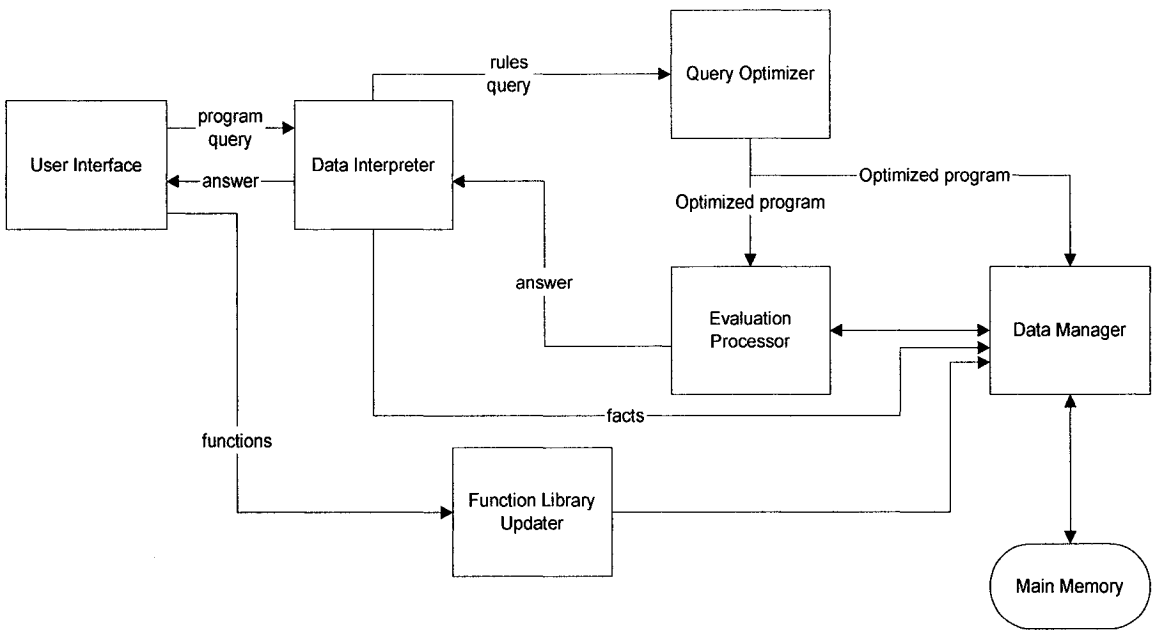


Figure 4.1: System architecture

Our system prototype includes five main components: data transformer, data manager, query processor, query optimizer, and library manager. These components allow users to input their parametric programs over the certainty domain [0, 1] and evaluate them efficiently. In addition to some well-known combination functions supported by the system, users can also add new functions to the functions library in a convenient way

provided by the library manager. All such functions are assumed to satisfy the properties introduced in [19]. In the rest of this chapter, we will explain these components and their interaction in more details.

4.1 Data Interpreter Component

Figure 4.2 shows the model diagram of Data Transformer (DT) component. Before submitting a p-program to the system, the user may store it in the secondary memory as a text file. Once it is submitted, the corresponding text file is read by the parser module in DT. The program is then checked for its syntax. If there is no syntax error, the input is separated into three parts: facts, rules and the query. Each of these parts is passed to the corresponding converter which transforms the input into the internal system representation. There is no query optimization in this process but it simply transforms the input expression to the internal representation, which is recognized by every component in the system and has the property of efficient manipulation in the system. The output of DT, which includes the transformed facts, rules, and query, is then passed to the Data Manager, Query Optimizer and Query Processor components, respectively.

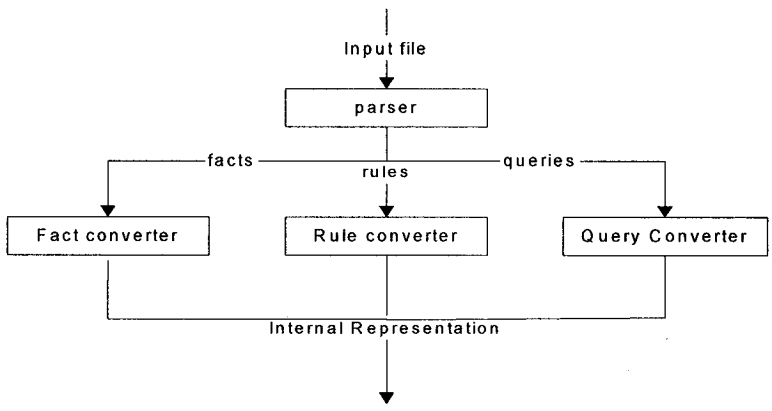


Figure 4.2: Data transformation procedure

4.2 Data Manager Component

The main task of the Data Manager (DM) in our system is to create suitable in-memory data structures for various input data/programs to support efficient search and/or access. DM takes the transformed facts generated by the Fact Converter module in DT and creates the EDB table. Also, the "optimized" program generated by the Query Optimizer, which will be described in section 4.3, is stored in the *rule tables*, and the user defined combination functions are added to the system library. These tables and the library are then stored in the main memory and accessed by DM. Moreover, DM also stores and manages the IDB facts during the evaluation.

To avoid duplicate copies, DM is used to build and store the "vocabulary dictionary", which contains all predicates and terms in the input program. Each predicate in the rules or each term in the facts has a pointer to the corresponding item in the dictionary.

Indexing is another task of DM to support efficient query evaluation. We apply indexing in conjunction with every optimization technique we developed in our system. To define a "reasonable" set of indices for an input program, we proposed a technique to reduce the number of necessary indices, presented in Figure 4.3. In this technique, we scan all predicates in the body of the rules and reorder them to match the current index plans. In case no index plan can be matched, a new plan is created by this module. Once all the index plans are created, index plans minimizer will take care of the plans and remove redundant index plans. Finally, an index table will be generated according to the minimized index plans. We will discuss details of this indexing mechanism in section 5.2.

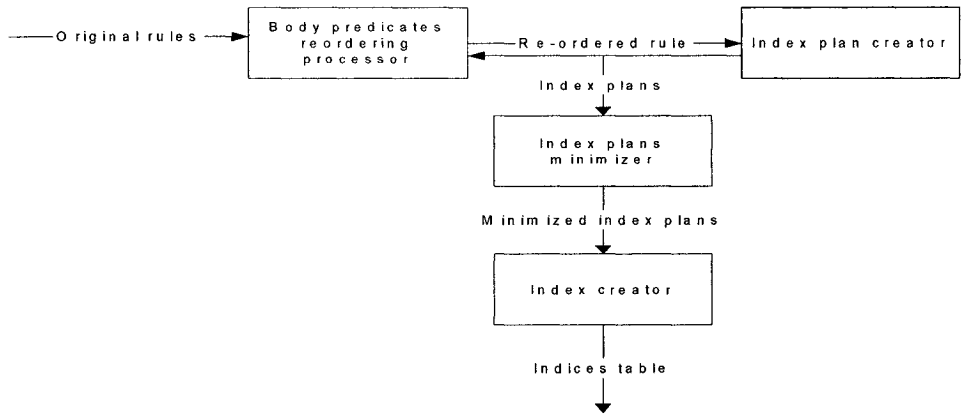


Figure 4.3: Index creation procedure

4.3 Query Optimizer Component

After the transformation done by DT, rules are further transformed by the Query Optimizer (QO) using rule rewriting, predicates reordering, depending on what kind of optimization technique is to be applied.

The query optimization applied in our system is actually a "combined" query technique, in the sense that it considers static and dynamic optimization methods together. During the compilation period, QO applies static optimization methods such as predicate reordering and rule rewriting. During the evaluation, it further considers and incorporates run-time query optimization method, e.g., the Semi-Naive with Partition (SNP) method. The QO dynamically switches the evaluation method between SNP and SN, based on the dynamic cost evaluation (see section 5.3.3).

Notice that even though a p-program is declarative rather than procedural, some optimization techniques may modify it into a procedural optimized program. For instance, stratification technique partitions a p-program into several strata and evaluates them stratum-by-stratum in sequence of lower-stratum-first. On the other hand, SN and

SNP strategies require some rules in the program not to be evaluated at some iterations. All such requirements make the evaluation procedural. To determine how the rules should be evaluated and in what order, the QO will attach some annotations or directives to each rule. The Query Processor will interpret these annotations at run-time.

For the bottom-up Naive evaluation, there is no optimization to be done by the query optimizer. The rules of an input program are simply passed to the evaluation processor as the output of the query optimizer.

4.4 Query Processor Component

Query Processor (QP) is the core of our system. It takes as input the annotated optimized program, generated by QO, and the EDB facts. The annotations in the optimized program provide the execution hints and directives to the QO to execute the program. The processor computes the selected rules by fetching the matched facts and do so iteration by iteration until no more "improved" fact-certainty pair is generated. When the fixpoint is reached, the processing terminates and returns the fact-certainty pairs that satisfy the binding patterns in the query. Figure 4.4 shows the interaction among subcomponents of QP.

Considering the fact that the number of body predicates is unknown in advance, we employ a recursive technique for QP.

Meanwhile, our program evaluator has well-defined "get-next-rule" and "get-next-fact" interfaces with DM to access the rule table, EDB table, and the IDB table. These interfaces are independent of how those tables are defined, which allows a user to choose

different optimization techniques to evaluate the input programs. This is useful as it might need to change the structures of data representation in the future.

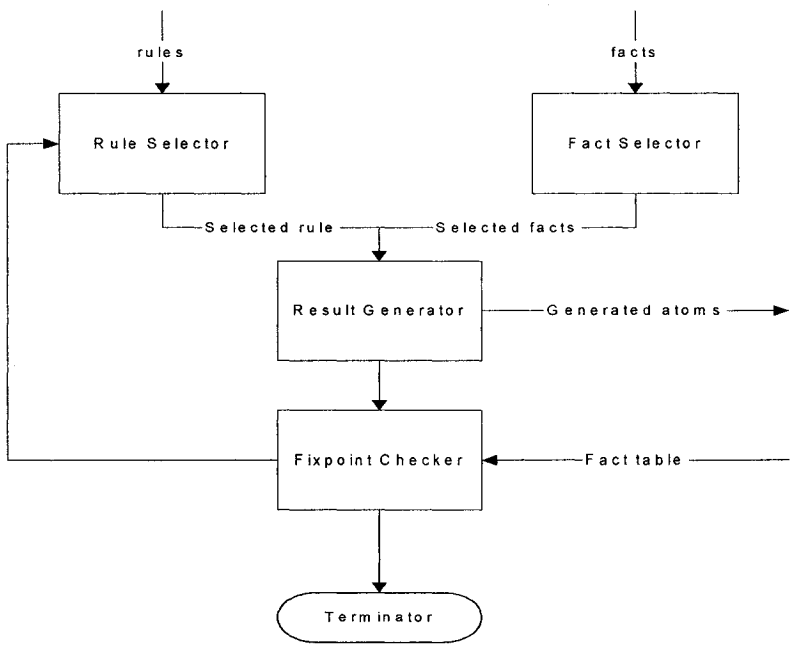


Figure 4.4: Program evaluation procedure

Unlike the fixpoint checker in classical deductive database system, such as CORAL [27], our fixpoint checker not only checks whether some new facts are generated at a particular iteration, but also checks whether the certainty associated with each fact is improved. That is, the notion of "new" is extended in our context, compared to the standard case. Therefore, the evaluation will continue if some certainties are improved while no new fact is generated. Because of the continuity property of the fixpoint operator T_P of the parametric framework, some facts may improve continuously, causing the evaluation to terminate only at ω [19]. To terminate the evaluation properly and reasonably, we introduced a limit, or a precision, to certainties. In this situation, we defined that an evaluation is considered reaching the fixpoint once the improvement of all certainties are smaller than the precision.

4.5 Library Manager Component

The Library Manager (LM) is a relatively independent component in our system in the sense that it is not concerned much with program's evaluation. It is an interface introduced to support user's interaction with the system to add new desired combination functions. There are three categories of certainty combination functions in a p-program: disjunction functions, propagation functions, and conjunction functions [19]. The most popular functions are maximum (*max*), minimum (*min*), product (*) and the probability independent function (*ind*) [36, 19]. These functions are already introduced in the functions library. However, users may need to define and use new functions to do certainty computations, as long as these functions satisfy certain properties required [19]. LM is specially designed for this purpose. Users may store their function(s) into a text file and include it in the library. LM then reads this function definition and embeds it into the library source file. Finally, the library source file is recompiled and linked to our system.

Chapter 5

System Implementation

We have two implementations of our system: in Windows and in Unix, both in C/C++ language. In both, we implement a fragment of the parametric framework over the certainty domain [0, 1] and with arguments of string data type only. Extending the implementation to support other data types and other certainty domain should be easy and should not pose new technical challenges, we believe.

In our implementation, we split the system into several subsystems according to the system architecture described in chapter 4, and provide a set of interfaces to each subsystem. We will highlight several key issues in some subsystem's implementation and provide some technical details of the development of these subsystems.

5.1 Data Representation

The choice of suitable data structures for efficient data representation, search, and access is crucial. We have used C++ classes or C structures to implement our deductive databases with uncertainty. We used C++ classes for the data types that need to be initialized with the default values, and used structures in C for others. During defining data types, pointers are widely applied for two reasons: 1) dynamically allocate main memory for data and, 2) save main memory usage and hence increase the efficiency. For example, Figure 5.1 shows how a fact-certainty pair p(10,3): 0.5 is represented internally. This fact has "p" as its name, and "10" and "3" as its first and second terms. The certainty

of this atom is 0.5. The detail explanation of data representation will be presented in the following sections.

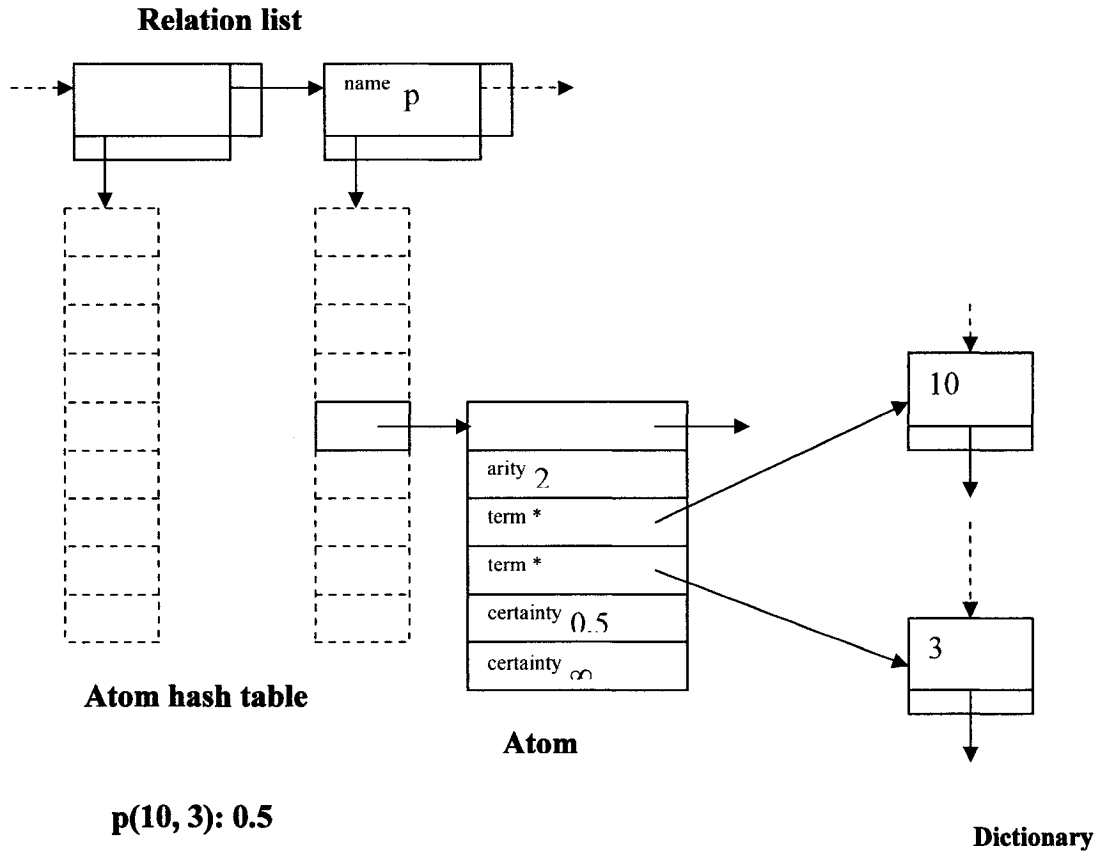


Figure 5.1: Internal representation of p(10,3): 0.5

5.1.1 Representation of Dictionary

The Dictionary in our system is a set of constants that may occur in the input program. It includes constants in the program facts and rules. The member of Dictionary is in primitive data types. In our system, they are implemented as string data type (for terms and predicate names) and real number (for certainty). The reason for our choice of these data types is for the ease of implementation and the parametric framework does not consider functional terms. Furthermore, since the certainty derivation involves

combination functions, the certainties are represented as real number in [0, 1]. In our system, the Dictionary is implemented as a linked list (see Figure 5.1).

The fixpoint evaluation looks at the input rules and the EDB facts, and generates IDB facts. However, the introduction of the Dictionary in our implementation helps to avoid copying terms and reduces memory requirement. For example, if we do not create Dictionary for an input program, the EDB facts must be stored with all its terms. Assume that there are two facts: study_in(John, Concordia_University) and study_in(Mark, Concordia_University) to be stored, the system has to store the term "Concordia_University" twice, one for each fact. If there are a thousand persons who are studying in Concordia, then the term "Concordia_University" is to be copied a thousand times. This will need much time to create and more space to store. On contrast to creating many copies of the same term, the system may create a Dictionary for the input program; only one copy of each term is stored. In case of storing a fact, no term copy is generated but a pointer to the corresponding element in Dictionary.

In addition to avoiding term reconstruction, the structure of the dictionary also provides increased efficiency when we need to compare two terms during evaluation. Moreover, in case two terms of type string are compared, it is not needed to check them character-by-character but just to check whether they refer to the same memory location. The memory address comparison is faster than string comparison, especially for long strings.

5.1.2 Representation of Atoms

In logic programming and deductive databases, an atom is also called a fact. It corresponds to "tuple" in relational databases. There are two kinds of atoms in deductive

databases: extensional data base (EDB) facts and intensional database (IDB) facts. EDB facts are those atoms that are explicitly introduced by the user in the program, whereas IDB facts are those that are derived from the input program. In our system prototype, data structures of these two kinds of atoms are identical, and treated similarly during the fixpoint evaluation.

In our implementation, all the atoms are defined as the objects of a C++ class *fact*. The atom definition in our system can be considered as a fact-certainty pair. The structure of the "fact" part is similar to the definition of fact in the standard case. A main difference with the standard case is that we also record the certainty associated with the atoms. This information is stored in "certainty" part of the pair. There are two certainty values for each atom object: the certainty assigned at the beginning of the current iteration (called supported certainty) and the certainty generated at the current iteration (which we call generated certainty). The supported certainty is used by the conjunctions function for a successful derivation, and the generated certainty records the result of applying the disjunction functions at the current evaluation iteration.

For each atom object, there is an array listing the terms that the object holds. As mentioned earlier, the term array contains no copies of the list of the terms but a list of pointers to the elements of the Dictionary. Since non-ground facts are not allowed, the terms of a fact are all constants. Therefore, every pointer will be pointing to specific element of the Dictionary. Since the arity of each predicate is fixed, it is both convenient to implement the list of terms as an array, which is efficient for searching the terms.

Note that we do not define the predicate name for each atom. This is because, as shown in Figure 5.1, all the atoms are grouped according to their predicate name. Each category of atoms forms a relation. The relation name is exactly as the predicate name.

5.1.3 Representations of Relations and Fact Table

A relation corresponds to a set of atoms with the same predicate name. The predicate name is the name of that relation. Different fixpoint evaluation techniques may require different information to be kept for each relation. However, the basic structure is the same for the relations. For instance, each relation has a name, definition (EDB or IDB) and a structure containing its tuples. There are two kinds of such structures in our system: linked list and hash table. As we were interested to measure our indexing method, we had to support "pure" Naive evaluation, which does not use any index. In this situation, linked list is the best structure suitable to implement the relations, i.e., the fact tables. On the other hand, when using indices, as in SN and SNP, we use hash tables to implement relations. Hash table is widely described as an efficient storage structure for in main-memory data [22, 30]. In the rest of this section, we will focus on the structure of hashed relations and fact tables.

To construct a hash table, the first task is to determine the hash key. In our prototype, we consider for each atom, its first argument as the default key for the corresponding relation. We do not want to take the whole set of terms as hash-key for the purpose of avoiding the overhead of combining those terms into a key. As mentioned before, the data type of all terms is string. However, the output of a hash function is usually an integer. Therefore, a conversion from string to integer must be done. Unfortunately, not every conversion function is desirable. For instance, consider a simple conversion that

returns the sum of all ASCII codes of the characters in the string. This function returns the same value for strings "abc", "cba", and "acb", hence resulting in collision. Instead, we could use a hash function that randomizes the input to reduce the collision, yet is efficient to compute. For any input string, we use the following hash function, which uses {8, 3, 2} as the set of factors:

For every character X in position n of the input string, let

$$m = \begin{cases} ASCII(X) * 8 & if\ n \bmod 3 = 0; \\ ASCII(X) * 3 & if\ n \bmod 3 = 1; \\ ASCII(X) * 2 & if\ n \bmod 3 = 2; \end{cases}$$

the hash-key is then defined as $\sum m$.

Notice that this conversion function does not guarantee a collision-free hashing operation, but improves the situation. For example, strings "abc", "cba" and "acb" are converted to 1268, 1280 and 1269, respectively.

Selecting a hash function and the hash size are issues dealt with before locating data into hash table. We simply chose a hash function: $h(key) = key \bmod hash\_size$.

For each entry of the hash table, there exists a sub-linked-list such that all atoms hashed into this cell are linked into this sub-linked-list. The size of each sub-linked-list is determined by the distribution of atoms and the hash-size. Since searching the terms in each sub-linked-list is sequential, a short list and therefore a large hash-size are preferred. However, too large hash-size might cause a waste of space in the hash table and a high maintenance cost. So, depending on the input program, the hash-size is determined dynamically. To simplify our implementation, we would like to let users to choose specific hash-size. Our default value for the hash-size is 131.

For Naive and Semi-Naive techniques, relations are kept as integrated entities. However, for Semi-Naive with Partition, IDB relations are partitioned into two parts: "improved" part and "non-improved" part. Under this scheme, it is not necessary to maintain two portioned relations for each IDB relation. We only need to create indices for each partition while keeping one copy of the whole relation. A benefit of this organization is that we can avoid duplicated creation and maintenance of IDB relations. The creation of partition indices pays off by applying these indices for efficient join of relations.

To have a uniform access, all relations are clustered together. The integration of all relations is called a fact table. It contains all the atoms (EDB and IDB) of a given program. In our implementation, the fact table is constructed as a list of relations, except for the stratification technique, which is a list of stratum and each stratum contains a sub-list of relations that are in the same stratum. Figure 5.1 shows the structure of fact table and illustrates how a relation links to another one. To search for a relation, we use sequential search. Since the number of relations is often much less than the number of tuples in the relation, it is not necessary to build an index for the relations list. As an efficient insertion for linked list, all new relations are inserted simply at the head of the relation list (or suitable stratum sub-list).

Even for input programs with a large number of relations, in most cases, we do not need to worry about the inefficiency due to sequential search (without index). This is because many input programs are stratifiable and their evaluation is done stratum by stratum from lowest level to the highest level. For the evaluation of each stratum, the size of the related relation list of a stratum is still small.

5.1.4 Representation of Rules and Rule Table

A rule in our DDB+Uncertainty is implemented as a list of predicates. It contains three parts: metadata, rule head, and rule body. The metadata contains particular information about a rule. This includes rule ID, rule certainty, and the combination functions associated with the rule. Rule head indicates the relations to which the derived atoms belong and the source of each term. Rule body says how relations join to generate an atom for the head. Notice that, basically, the result of joins of relations is independent from the sequence of joins, but our system evaluates the joins from left to right. This sequence, on the other hand, is important for creation of indices. A particular order of body relations should be chosen to reduce the number of indices. This issue is studied in section 5.2.

Note that, each EDB fact is considered as a special rule with an empty body, interpreted as being true, always, and the propagation and conjunction functions being *min* or *. In our implementation, EDB facts are excluded from considering them as rules when applying Semi-Naive methods. This is because they need not to be reevaluated once they are inserted into the facts table at compile time; their associated certainty will never change during the fixpoint evaluation. Hence, the rule table contains only rules that have some EDB or IDB predicate(s) in the body.

Another important point to mention here is that different evaluation techniques require different extra information to be included in the rule metadata to facilitate their evaluation. In Figure 5.2, we show the internal representation of the rules in the following input program, $P_{5.1}$:

$$p(X,Y) \xleftarrow{0.8} e(X,Y);\langle min,*,max\rangle.$$

$$p(X,Y) \xleftarrow{0.5} e(X,Z),p(Z,Y);\langle min,*,max\rangle.$$

In the rest of this section, we will discuss some features of each part of a rule representation for different techniques.

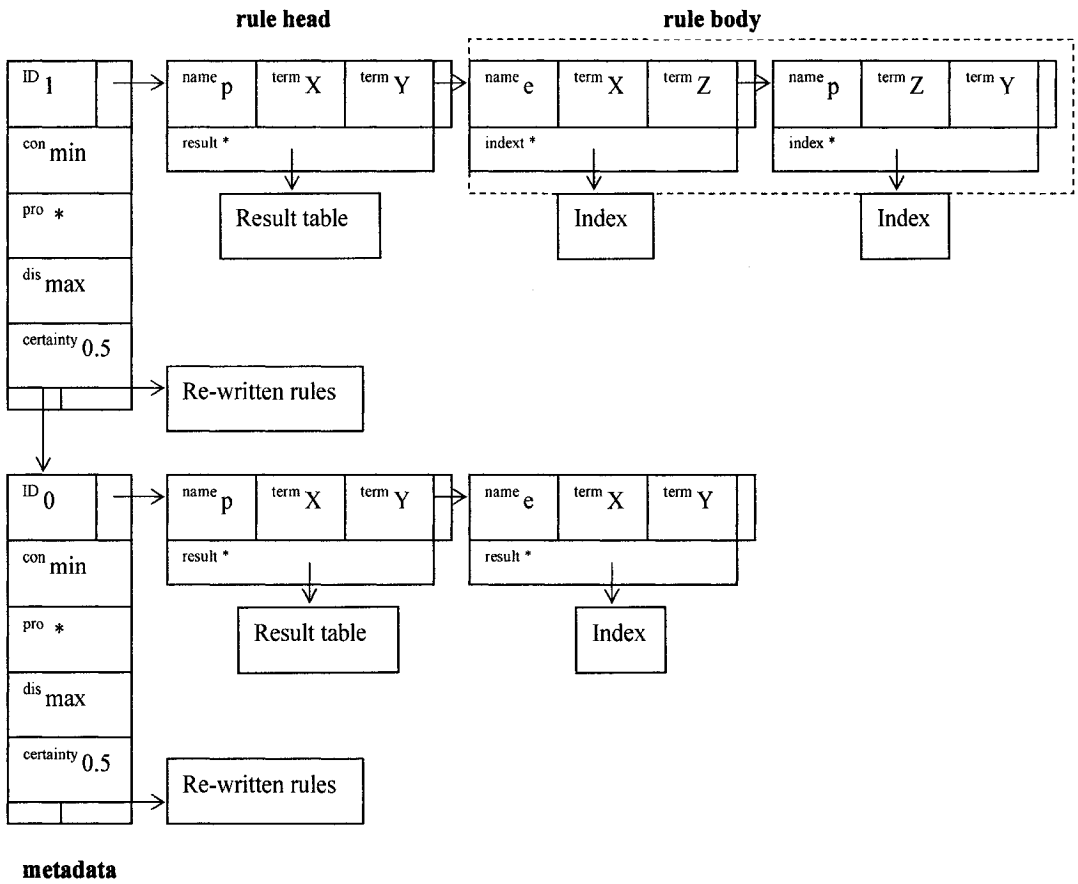


Figure 5.2: Internal representation of program $P_{5.1}$

The basic components of metadata are rule ID, rule certainty, and combination functions. These are the information we need for the Naive evaluation method. In our implementation, the data structure of the combination functions is an array with three

elements, through which, we can access any one of the associated combination functions quickly.

Besides rule ID and combination functions, SN and SNP need other information about each rule. As mentioned in chapter 3, for SN fixpoint evaluation, only those rules having something "new" in their bodies will be evaluated at a particular iteration. A Boolean expression is used for each rule to indicate whether it is to be fired or not. On the other hand, for SNP evaluation, a rule should be rewritten into a set of new rules in which the IDB relations are partitioned into two parts: the "improved" part and the "non-improved" part. Details of this rule rewriting will be discussed later. Consequently, the data structure of rule metadata should indicate which set of rules is the rewriting result of which rule. In our system, this is indicated through a pointer to the set of re-written rules.

In our implementation, the rule head links to the rule metadata. The basic components in a rule head include predicate name and terms (variables and constants). Similar to the data structure of terms in atoms, an array is used to store information about the head predicate. Since different predicates may have different arities, we use dynamic arrays as the data structure of the head predicate for the purpose of convenience and efficiency.

The main difference of the structure of the rule head in Naive and SN evaluations is that the latter requires a particular result table for each rule, whereas this is not the case in the former. For the Naive evaluation, all derived atoms are stored in the fact table. There is no need to keep a record to indicate which rule generates which atoms. On the other hand, the SN method requires bookkeeping for each rule. For instance, in case some rules need not to be fired because they will not generate "new" thing at a particular evaluation

iteration, we take the bookkeeping as the result of these rules. In our system, all the tuples generated by a particular rule are stored in a hash table.

Rule body is the last part in the rule list. It contains a list of subgoals in a particular order. The length of the rule body for each rule depends on the number of its subgoals. Even though the rule body can be created as a dynamic array, we implemented it as a linked list since the main operation on rule body is scanning rather than extraction. Furthermore, the input program is declarative, i.e., the user needs not worry about the order of rules in a program or the order of subgoals in a rule body. On the other hand, the evaluation of programs is procedural, and hence it is the system's responsibility to re-order the input program for efficiency reason. Reordering of subgoals in the body of a rule is done through operations of deletion and insertion. These operations are expensive, in general, on arrays, as they require shifting the entries in the array. Unlike array, linked list enjoys a very efficient implementation for both insertion and deletion in our context. The complexity of these operation is just O(1).

The data structure of each subgoal is the same as that of a rule head except that body predicate maintains a pointer to applicable index while the rule head keeps a pointer to its result table.

The data structure of the rule table is implemented simply as a linked list to facilitate rule re-ordering. Currently, all the rules are designed to be evaluated one-by-one from head to tail. If a stratification strategy is not applied, the specific order of rules is not required. We may simply consider the original rule order for the fixpoint evaluation. However, with the stratification strategy, the input rules must be reordered according to the order determined. The order is immaterial for the rules in the same stratum. There are two

design options to support the stratification. One is reordering the existing rule table and the other is to keep the rule table as is but creating a stratum table with pointers to the rules in rule table. We do not support stratification in our current implementation and hence the order of rules is kept as the original one.

5.1.5 Representation of Indices

Since all relations in our system are stored in Main-Memory, we use hash-based indices on a subset of the arguments of a relation. From Figure 4.3 shown in chapter 4, we can see that there is a supplementary internal data, *the index plan*, to help creation of indices. Totally, we have defined and used four main data representation: index plan, plan table, index, and index table.

Before creating an index, a corresponding index plan is created first. The index preparation contains the following pieces of information: (1) constrains of the arguments, (2) the subset of arguments for the index, and (3) a set of subgoals that the index is going to be applied to. These three kinds of information are stored in three linked lists respectively. Notice that the sequence of arguments on which we create an index is important because of the key conversion function, while the sequence of the other two lists is immaterial. The selected arguments in the sequence are converted to integers and summed together to form the argument of the index hash function. The procedure is the same as discussed in section 5.1.3, except that we allow multi-arguments here.

Figure 5.3 shows an example of index plan. The index plan is for p(X,Y,1,X) and this plan is applicable for the third body predicate in rule 2, and the fifth body predicate in rule 8. Furthermore, the list of constraints in this index plan indicates that the first and

fourth terms are identical and the third term is a constant with value "1." The index plan also shows that two arguments, the first and second terms, in sequence, are combined to form the key of the index. This information will help the reordering procedure for the body predicates to reorder the other rules so that as many body predicates as possible can share the same index structure.

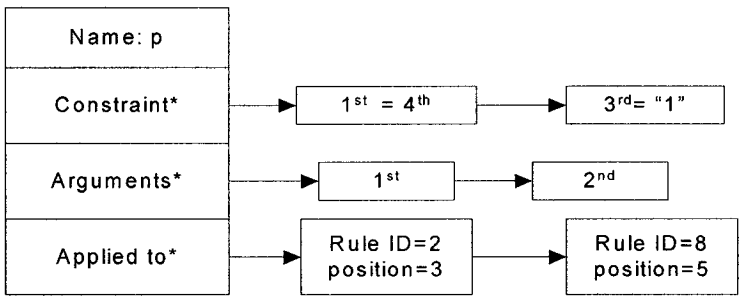


Figure 5.3: An example of Index plan

The index plan table is a partially ordered linked list. All index plans sharing the same predicate name are clustered together, while the order inside each cluster is immaterial. Clustering the index plans helps to facilitate the plan search. For example, while the system tries to create an index plan for relation "p", it first looks at the plan table to check whether the same plan has been already created. Without plan clustering, the system needs to scan the whole index plan table. With clustered plans, however, the system needs only to scan the index plans of relation "p."

The representation of indices and index table are similar to that of index plan and plan table, respectively. The difference is that the data structure for indices maintains an extra pointer to a hash table, which stores the corresponding index. In our current system, all the index hash tables have the same hash-size, which simplified our implementation of the indices, while it may waste some space and increase the cost of indexing.

5.2 Index Creation

Indexing is a basic optimization technique in databases and choosing a "right" index is a challenging issue. There is a big difference for choosing an index between relational databases and deductive databases. For relational databases, choosing extra indices on a relation are guided by the statistics on frequencies of different queries; indexing is based on those most frequent queries. It is usually the user's responsibility to tell the DBMS explicitly which index should be created. Since queries to relational database instances are often submitted after indexing is done, some new queries may not benefit from existing indices. Unlike in relational databases, the set of rules in a deductive database instance can be considered as a set of queries. These rules are recognized before indices are created and usually need to be evaluated many times. Therefore, it provides an opportunity to DDB to create a set of indices that exactly satisfy the requirements of a deductive database instance. To create a set of desired indices, our system creates a set of index plans first, then reduces duplicated plans, and finally creates indices according to the index plans.

5.2.1 Index Plan Creation

To create an index plan, the first problem to be solved is to determine on which arguments the indices should be built. Obviously, indices should be built on those common arguments (or attributes) that share the same variable(s) so that they help joining relations. We call such kind of indexing as *common attribute indexing*. As an example ($P_{5.2}$), let us consider a rule:

$$\mathrm{sgc(X,Y)} \xleftarrow{1} \mathrm{parent(X,X1)}, \mathrm{sgc(X1,Y1)}, \mathrm{parent(Y,Y1)}; \langle \mathrm{min}, \mathrm{min}, \mathrm{ind} \rangle.$$

where *X1* and *Y1* are shared variables. The second argument of "parent" and the first argument of "sgc" are common argument *X1*. Similarly, the first argument of "sgc" and the second argument of "parent" are common argument too, *Y1*. So, three indices should be created for this rule. The second argument of "parent" and each argument of "sgc" will be taken as keys for the indices, respectively. During the evaluation (from left to right), the index on the second argument of "parent" and the one on the first argument of "sgc" are used for index join. The result of the join of relations "parent" and "sgc" is then joined with the second occurrence of relation "parent", using the indices on the second argument of "parent."

Creating indices on common attribute(s) are at the relation level. This means that every fact in a relation must be assigned an index. However, in some cases, not every fact needs to be assigned an index. Let us consider the following example ($P_{5.3}$):

$$p(X,Y) \xleftarrow{0.5} q(X,1,Y), s(Y,Z,Z); \langle \min, *, \max \rangle.$$

$$q(2,1,3) \xleftarrow{0.5}; \langle \_, \min, \max \rangle.$$

$$q(2,2,3) \xleftarrow{0.5}; \langle \_, \min, \max \rangle.$$

$$s(3,5,5) \xleftarrow{0.5}; \langle \_, \min, \max \rangle.$$

$$s(3,5,6) \xleftarrow{0.5}; \langle \_, \min, \max \rangle.$$

For common attribute indexing, only the third argument *Y* of "q" and the first argument of "s" are determined as the keys of the corresponding indices. However, it is easy to see that the fact s(3,5,6) :0.5 never participates in a successful join since its second and third arguments are different. Hence, this kind of facts should be excluded when using indices. Based on this observation, a so-called *equality constraint* needs to be considered to avoid

such kind of unnecessary indexing. In this case, only a sub-set of relation "s", in which equality constraint holds on the second and third arguments, is considered when we construct an index on its first argument.

There is another constraint which we refer to as *constant constraint*, which needs to be considered during indexing. As shown in example $P_{5.3}$, not all the "q" facts will participate in a successful join; only those "q" facts that have value "1" at the second argument are considered as candidates for the join. In this case, the constant constraint for "q" is that the value of its second argument must be 1. Index should be created on the third argument of tuples t in "q" such that the value of the second argument of t is "1."

If we look at the mechanism of a join deeply, we will find that, the arguments on which an index is created are actually those with bound values. When performing a join, if a tuple from the relation on the left is taken, then the common attribute(s) would be bound by some value. Hence a matched tuple in the relation on the right hand side must hold the bound value(s) in its common attribute(s). To complete a join of two relations, all tuples in the left-hand-side relation must be scanned one-by-one [22]. Since all tuples on left hand side relation need to be scanned, it is unnecessary to create an index for that relation unless there is some constraint on it. Furthermore, we can recognize a variable in a relation would be bound if this variable does appear in the relations locating on the left of this relation in a rule (based on the particular evaluation order from left to right, in our case). Otherwise, the variable not appearing in the relations on the left is free, i.e., unbound. Based on these observations, it is not hard to convince ourselves that, *given a rule in deductive databases, the maximum number of indices needed to be created is the number of body predicates containing some common attribute(s) minus one if the first*

predicate has no constraints. When the first predicate has some constraints, we need exactly as many indices as the number of body predicates containing some common attribute(s).

5.2.2 Reordering Body Predicates

While we benefit from indexing, the cost of index creation and maintenance should not be ignored. For a more efficient evaluation, we should try to reduce the number of indices. One straightforward method to do this is to move a non-constraint predicate as the left most atom in the rule body. Since the first predicate does not contain any constraint, we do not need an index for it. This reduces the number of indices by one.

Note that, except the first predicate, for every other predicate containing some common attributes, we need exactly one index to facilitate the joins. To further reduce the number of indices on a rule, some indices must be applied more than once. This means that body predicates should be reordered according to some particular sequence so that some predicates could share the same index. Let us consider example $P_{5.2}$ again. Keeping the original order, we need to create two indices: one on the first argument of "sgc" and another on the second argument of "parent"(the second occurrence). If the body predicates are re-ordered, we get the following rule:

$$\mathrm{sgc(X,Y)} \xleftarrow{0.5} \mathrm{sgc(X1,Y1), parent(X,X1), parent(Y,Y1); \langle min, *, max \rangle}.$$

we only need one index on this rule: on the second argument of "parent." It shows that a suitable order of body predicates may reduce the number of necessary indices. We call this technique as the *body predicate reordering*. The basic idea here is to share indices by predicate reordering.

It is of cause possible to extend this technique to programs with multiple rules, as long as they have some predicates that appear more than once as body predicates. These ideas are formally expressed as an algorithm for body predicate reordering, shown in Figure 5.4. The input of this algorithm is a set R of rules. The output includes (1) a set R' of reordered rules, obtained from R, and (2) a plan, defined as a set of index plans. An index plan has the form (p, v, c), where p is a predicate name, v is a subset of its arguments, and c is a set of constraints. If $p(X_1,\ldots,X_n)$ is an atom in a rule, then we use $p(\mu)$ to denote the predicate of p, where $\mu = \{ X_1,\ldots,X_n \}$.

procedure: predicate_re-ordering (R; R', Plan)

forall r: $h(v) \xleftarrow{\alpha} b_1(v_1),\ldots b_n(v_n); <f_d, f_p, f_c> \in R$

1 Wait := $\{b(v) | b \in \{ b_1,\ldots b_n\} \wedge \exists(b, v', c) \in Plan \}$;
2 New := $\{b(v) | b \in \{ b_1,\ldots b_n\} \wedge \forall(b, v', c) \notin Plan\}$;
3 Bound := $\varnothing$;
4 $r_{new} := h(v) \xleftarrow{\alpha} ; <f_d, f_p, f_c>$;
loop
forall $b(v) \in$ Wait
if $(\exists(b, v', c) \in Plan \wedge v' = v \cap Bound) \wedge c$ = constraints in $b(v)$
5 locate_predicate(r_{new}, $b(v)$, New, Wait, Bound);
end if
end forall
if New $\neq \varnothing$
pick any $p(\mu) \in$New;
6 Plan := Plan $\cup$ $(p, \mu \cap Bound, c)$; where c is constraints of $p(\mu)$
7 locate_predicate(r_{new}, $p(\mu)$, New, Wait, Bound);
end if
else
pick any $b(\mu) \in$Wait;
8 Plan := Plan $\cup$ $(b, \mu \cap Bound, c)$; where c is constraints of $b(\mu)$
9 locate_predicate(r_{new}, $b(\mu)$, New, Wait, Bound);
end else
until Wait = $\varnothing \wedge$ New= $\varnothing$
10 append_rule(R', r_{new});
end forall
end procedure

Figure 5.4: An algorithm for body predicate re-ordering

Our re-ordering technique partitions the body predicates of a rule into two disjoint categories: Wait predicates and New predicates (lines 1, 2). The Wait predicates are those that have some index plans but none is applicable in the current situation, while New predicates are those that are not indexed yet. Obviously, only Wait predicates have the opportunity to share some existing indices through reordering. All New predicates need new index plans. Once an index plan for a New predicate is determined, the status of some New predicates may change. Some New predicates may move from New predicates to Wait predicates if they have an index plan. An update is required for both New and Wait predicates (see procedure locate_predicate in Figure 5.5). Another significance of creating an index plan is that it changes the status of some free variables to bound. Therefore, some Wait predicates may get a chance to share existing indices and thus added to the tail of the current reordered rule r_{new} (line 5). In case no Wait predicate can share the existing indices, a New predicate is picked up and appended to r_{new} (lines 6, 7). If no New predicate remains, unfortunately, one of Wait predicates would be chosen to change the situation of the index plan and the status of the remaining unbound variables (lines 8, 9). Hopefully, the remaining Wait predicates may benefit from these changes. Reordering of the predicates in a rule body is completed when all Wait and New predicates are considered. This reordering of predicates is done for all rules in the given program, one by one.

Notice that our re-ordering technique does not guarantee that the reordered program generates the minimum number of indices. This is because our technique ignores the order of rules and the selection order of New predicate when a new index plan is to be

created. However, considering these two issues in our algorithm would perhaps make it very expensive and hence not be pursued.

```
procedure: locate_predicate(r_new, p(μ), New, Wait, Bound)
    if p(μ)∈Wait
        Wait := Wait – { p(μ) };
    end if
    else
        Wait := Wait ∪ {b(ν)| b = p ∧ b(ν) ∈ New};
        New := New- Wait;
    end else
    append_body(r_new, p(μ));
    Bound := Bound ∪ {var | var∈μ};
end procedure
```

Figure 5.5: The procedure of locate_predicate

5.2.3 Index Containment and Index Creation

When creating index plans, a category of index plans attracted our attention. Some index plans are on the same predicate with identical set of arguments as index keys but different constraints. In this situation, it's possible that one uniform set of constraints may help to integrate these index plans into a single index plan such that all the facts that satisfy any of these index plans also satisfy this single index plan. We called *this single index plan contains other index plans*. To be precise, *index plan A contains index plan B if and only if 1) A and B are on the same predicate and same arguments, and 2) the facts satisfy the constraints in B also satisfy the constraints in A.* In our system, if index plan *A* contains index plan *B*, index plan *B* will be replaced by *A* and plan *B* can be ignored. The creation of index *B* is saved.

Let us conduct partially ordered sets, called *index constraints*, as follows: $< Plan(p,\upsilon), \preceq >$, where $Plan(p,\upsilon)$ is a set of index plans created for the same subset

of arguments υ of the same relation p. For $A, B \in Plan(p,\upsilon)$, we say $B \preceq A$ iff plan A contains plan B. Obviously, the upper bound of a partially ordered set contains all the index plans in this set. A desired index plan for a partially ordered set of index plans is its least upper bound (*lub*). So, *looking for a uniform set of constraints to create a single index plan is actually looking for the least upper bound of constraints.*

There are two cases for finding the $lub(Plan(p,\upsilon))$ of an index plan set $Plan(p,\upsilon)$. One is that $lub(Plan(p,\upsilon))$ is one of the existing plans. In this case, we simply take this index plan as our desired index plan for $Plan(p,\upsilon)$. For example, consider $Plan(p,\upsilon)$ contains three index plans:

A: (p, <1st, 2nd >), 4th = 5th , 3rd ='1'
B: (p, <1st, 2nd >), 4th = 5th
C: (p, <1st, 2nd >), 4th = 5th , 3rd =5th

All these index plans are about predicate p and on the 1st and 2nd arguments. All plans require the 4th argument to be equal to the 5th. Plan A further requires that the 3rd argument has value 1 and plan C requires the 3rd argument to be identical to the 5th. It is not hard to see that every fact that satisfies A and C also satisfies B. The constraints of plan B is the $lub(Plan(p,\upsilon))$ in this case. Hence plan B is kept. Plans A and C are ignored in our system.

Another case of finding the least upper bound is a bit complicated. There is no index plan in the partially ordered set $Plan(p,\upsilon)$ that can be considered as the least upper bound. In this situation, the least upper bound of $Plan(p,\upsilon)$ is constructed and added to $Plan(p,\upsilon)$. For example, consider the following three index plans:

E: (p, <1st, 2nd >), 4th = 5th, 3rd ='1'

F: (p, <1st, 2nd >), $4^{th} = 5^{th}$, 3^{rd} ='2'

G: (p, <1st, 2nd >), $4^{th} = 5^{th}$, $3^{rd} = 5^{th}$

Plans E and G are the same as A and C in the previous example. Compared to B, plan F has one extra constraint: the 3rd argument should have value 2. Now, unlike B, plan F is not the least upper bound because $E \not\preceq F$ and $G \not\preceq F$. Actually, none of these plan contains others. Therefore, A new index plan, *New plan*, should be built as their least upper bound. That is:

New plan: (p, <1st, 2nd >), $4^{th} = 5^{th}$, 3^{rd} ='1' $\vee$ 3^{rd} = '2' $\vee$ $3^{rd} = 5^{th}$

This new plan that contains E, F, and G is then inserted into the plan table and plans E, F and G are ignored. Notice that, even though the new plan may have a large size (of the number of the satisfied facts) in a relation, compared to plans E, F, and G, its size is not larger than the sum of these three plans. Therefore, in this case, selecting the least upper bound still pays off.

Using the notion of index containment, our system can further reduce the number of indices, which in turn reduces the cost of indexing. Once the final index plans are determined, the index creation becomes simple. All the facts in the fact table are scanned and their addresses are stored in suitable positions in the hashed index table, for each applicable index.

5.3 Query Optimization

During the last two decades, a number of query optimization strategies have been introduced and implemented developed for the standard deductive databases [10, 5, 13, 26, 12, 15, 9, 28]. These strategies can be classified into top-down and bottom-up

optimization. However, neither of these techniques is directly applicable to deductive databases with uncertainty. The main reason is that they are set-based, while we need multiset-based techniques, in general, for DDB+Uncertainty. In our system implementation, we consider various techniques, including multiset-based Semi-Naive and Semi-Naive with Partition methods to increase the efficiency of programs' evaluation. The stratification method is almost done; we need to complete the interface between query processing and optimization modules, which we are currently working on. In chapter 3, we described the algorithms for Semi-Naive (SN), Semi-Naive with Partition (SNP) and the desired stratification. We will discuss implementation details of these techniques in following sections.

5.3.1 Rule Rewriting Technique

As discussed in chapter 3, the SNP method tries to avoid repeated derivation by focusing on inferring improved fact-certainty pairs. By improved, we mean a new fact or old fact with better certainty. We point out that derivations of improved fact-certainty pairs can originate from the joins of atoms in which at least one atom is improved at the last iteration. Based on this point, we use similar ideas of rule rewriting introduced for Datalog [3, 2, 8] in our context. Let us consider the following p-program:

$$\text{r1}: \text{p(X, Y)} \xleftarrow{\alpha} \text{e(X, Y)}; < f_{\text{d}}, f_p, f_c > .$$

$$\text{r2}: \text{q(X, Y)} \xleftarrow{\alpha} \text{e(X, Y)}; < f_{\text{d}}, f_p, f_c > .$$

$$\text{r3}: \text{p(X, Y)} \xleftarrow{\alpha} \text{p(X, Y), q(Z, Y)}; < f_{\text{d}}, f_p, f_c > .$$

where "e" is EDB predicate and "p" and "q" are IDB predicates.

Clearly, the definition of "e" does not change during the evaluation, while the definitions of "p" and "q" may change. So rules r1 and r2 yield new fact-certainty pairs for "p" and "q" only at iteration 1. For other iterations, no new fact-certainty pairs will be inferred by these two rules since there are no new fact-certainty pairs for "e." For r3, only the joins of improved fact-certainty pairs of "p" or "q" may yield new fact-certainty pairs. If we partition "p" and "q" into two parts: Δ and Λ, where Δ contains all the improved fact-certainty pairs in a relation and Λ holds the rest of the fact-certainty pairs in that relation, r3 can be rewritten to as follows:

$$\text{r3\_1}: \text{p(X, Y)} \xleftarrow{\alpha} \Delta\text{p(X, Y), q(Z, Y)}; < f_d, f_p, f_c > .$$

$$\text{r3\_2}: \text{p(X, Y)} \xleftarrow{\alpha} \Lambda\text{p(X, Y)}, \Delta\text{q(Z, Y)}; < f_d, f_p, f_c > .$$

$$\text{r3\_3}: \text{p(X, Y)} \xleftarrow{\alpha} \Lambda\text{p(X, Y)}, \Lambda\text{q(Z, Y)}; < f_d, f_p, f_c > .$$

at every iteration *i*, the result of r3 is equivalent to that of rewritten rules. If we look at the re-written rules carefully, we find that the evaluation of rule r3_3 is redundant since it involves joins of non-improved fact-certainty pairs in "p" and "q" and is done at some previous iterations. So we can bookkeep the result of r3_3 and avoid its evaluation.

We now present our rewriting technique based on the aforementioned idea, for the general case. Let ***P*** be a p-program over EDB predicates B's and IDB predicates T's. Consider the following rule in ***P***.

$$\text{S}(\mu) \xleftarrow{\alpha} \text{B}_1(\upsilon_1), ..., \text{B}_n(\upsilon_n), \text{T}_1(\tau_1), ..., \text{T}_m(\tau_m); < f_d, f_p, f_c > .$$

This rule can be rewritten as the following set of rules:

$$\text{r}_1: \text{S}(\mu) \xleftarrow{\alpha} \text{B}_1(\upsilon_1), ..., \text{B}_n(\upsilon_n), \Delta\text{T}_1(\tau_1), \text{T}_2(\tau_2), ..., \text{T}_m(\tau_m); < f_d, f_p, f_c > .$$

$$\vdots$$

$$r_j: \; S(\mu) \xleftarrow{\alpha} B_1(\upsilon_1),...,B_n(\upsilon_n), \Delta T_1(\tau_1), ..., \Delta T_{j-1}(\tau_{j-1}),$$
$$\Delta T_j(\tau_j), T_{j+1}(\tau_{j+1}),..., T_m(\tau_m); < f_d, f_p, f_c >.$$

$$\vdots$$

$$r_m: \; S(\mu) \xleftarrow{\alpha} B_1(\upsilon_1),...,B_n(\upsilon_n), \Delta T_1(\tau_1), ..., \Delta T_{m-1}(\tau_{m-1}), T_m(\tau_m); < f_d, f_p, f_c >.$$

$$r_{m+1}: \; S(\mu) \xleftarrow{\alpha} B_1(\upsilon_1),...,B_n(\upsilon_n), \Delta T_1(\tau_1), ..., \Delta T_m(\tau_m); < f_d, f_p, f_c >.$$

This kind of rule rewriting has been shown to be equivalent to the original rule for classical Datalog rules. However, recall that the evaluation of classical deductive database programs is set-based rather than multi-set-based. We next prove the correctness of this rewriting in our context before applying this technique.

Theorem 6 *The rewritten rules generated by the rewriting technique above produce the same multiset-based result as the original rules.*

Proof:

First of all, note that the rewriting technique is not concerned with combination functions in the rule. These functions are used when evaluating the rules. Therefore, if all the inferences of the original rule are kept by corresponding rewritten rules, the theorem is proved.

The result of the evaluation of a rule can be considered as the multiset-based extension of the head predicate. This multiset-contains all the result of joins of body predicates. For a general rule above, the body can be considered as a sequence of joins (for convenience, we ignore the argument list of the predicates):

$$B_1 \bowtie B_2 \bowtie ... \bowtie B_n \bowtie T_1 \bowtie T_2 \bowtie ... \bowtie T_m \qquad (1)$$

If the IDB predicate T_1 is partitioned into Δ and Λ, as shown earlier, expression (1) can be re-expressed as:

$$B_1 \bowtie \ldots \bowtie B_n \bowtie (\Delta T_1 \cup \Lambda T_1) \bowtie T_2 \ldots \bowtie T_m \qquad (2)$$

This expression can be further transformed into:

$$B_1 \bowtie \ldots \bowtie B_n \bowtie \Delta T_1 \bowtie \ldots \bowtie T_m \qquad (3\_1)$$

$$\cup\, B_1 \bowtie \ldots \bowtie B_n \bowtie \Lambda T_1 \bowtie \ldots \bowtie T_m \qquad (3\ 2)$$

This expression shows that the original rule is equivalent to the union of two rules (3_1 and 3_2). Notice that this union is the multiset union, which retains duplicates. Now, consider sub-expression in formula 3_2. If we further partition T_2 into the "proved" and "non-improved" parts, this sub-expression can be further transformed into:

$$B_1 \bowtie \ldots \bowtie B_n \bowtie \Lambda T_1 \bowtie \Delta T_2 \ldots \bowtie T_m \qquad (3\_2\_1)$$

$$\cup\ B_1 \bowtie \ldots \bowtie B_n \bowtie \Lambda T_1 \bowtie \Lambda T_2 \ldots \bowtie T_m \qquad (3\_2\_2)$$

If we continue this transformation until all the IDB predicates are partitioned, expression (1) would be rewritten as the following multiset union of sub expressions:

$$B_1 \bowtie \ldots \bowtie B_n \bowtie \Delta T_1 \bowtie T_2 \ldots \bowtie T_m$$

$$\cup\ B_1 \bowtie \ldots \bowtie B_n \bowtie \Lambda T_1 \bowtie \Delta T_2 \bowtie T_3 \ldots \bowtie T_m$$

$$\cup\ B_1 \bowtie \ldots \bowtie B_n \bowtie \Lambda T_1 \bowtie \Lambda T_2 \bowtie \Delta T_3 \ldots \bowtie T_m$$

$$\vdots$$

$$\cup\ B_1 \bowtie \ldots \bowtie B_n \bowtie \Lambda T_1 \bowtie \Lambda T_2 \bowtie \ldots \bowtie \Delta T_m$$

This set of multiset unions contains *m+1 sub-expressions*, which can be considered as a set of rules with the same head. Each sub-expression forms a body of a rewritten rule. The rewritten rules are the same as described above, and hence not reproduced here.

The correctness of our transformation above based on multiset concept follows, upon noting that the transformation respects the multiset-based operation of union and join. □

If we look at the rewritten rule r_{m+1}, we may note that no joins in this rule involve any new facts. That is, the evaluation of this re-written rule is completely redundant. If we have a proper bookkeeping of the result of this rewritten rule, its evaluation can be ignored and the redundancy can be avoided. This is actually the essence of our rule rewriting.

5.3.2 Information Backtracking

Instead of evaluating an original rule, we evaluate all but the last rewritten rules. At the completion of each iteration, and before starting a new iteration, the evaluation result is recorded. This recording allows skipping evaluation of the last rewritten rule. The IDB predicates are repartitioned at the end of the current iteration to prepare for the next.

The reason for repartitioning is because of the non-monotonicity of non-improved partitions. Some facts in the non-improved partition might be improved and hence removed from non-improved partition and added to the improved partition. Consequently, the related results in the bookkeeping need to be moved out too. This brings out the problem of bookkeeping maintenance.

The solution for maintaining the bookkeeping strongly depends on the information associated with the results. It is very easy to update the bookkeeping if they include the

information of how they were derived, which means the system not only records the result (fact-certainty pairs) but also the instance of the tuples involved in the joins, i.e., the ground body. In this case, we simply look up the associated ground body of each record in the bookkeeping and check whether they involve some improved facts. If yes, this result is removed from the bookkeeping. The following example ($P_{5.4}$) shows how the bookkeeping is updated when "enough" information is recorded.

$$\text{r1}: \text{p(X, Y)} \xleftarrow{1} e(\text{X, Y}); < \text{ind}, \text{min}, \text{min} > .$$

$$\text{r2}: \text{p(X, Y)} \xleftarrow{1} p(X, Z), e(\text{Z, Y}); < \text{ind}, \text{min}, \text{min} > .$$

Suppose there are four EDB fact-certainty pairs: (e(1,2): 0.5), (e(2,3): 0.5), (e(1,3): 0.5) and (e(3,4): 0.5). Using our rewriting technique, r2 can be rewritten as:

$$\text{r2\_1}: \text{p(X, Y)} \xleftarrow{1} \Delta p(X, Z), e(\text{Z, Y}); < \text{ind}, \text{min}, \text{min} > .$$

$$\text{r2\_2}: \text{p(X, Y)} \xleftarrow{1} \Lambda p(X, Z), e(\text{Z, Y}); < \text{ind}, \text{min}, \text{min} > .$$

As mentioned before, r2-2 can be ignored and we only evaluate r2-1. At iteration 1, r1 and r2-1 are evaluated as usual. Since Δp is empty, only r1 generates new facts, shown below, which are book kept for the next iteration.

Bookkeeping = { (p(1,2): 0.5) ← (e(1,2): 0.5);
(p(2,3): 0.5) ← (e(2,3): 0.5);
(p(1,3): 0.5) ← (e(1,3): 0.5);
(p(3,4): 0.5) ← (e(3,4): 0.5) }

We remark that actually, we record only ground rules in this case. Applying the associated disjunction function, the result is converted from a multi-set to a set, which is the extension of "p" at this iteration. All the facts in "p" are new and hence p = Δp.

At iteration 2, r1 is no longer applied; and we only apply r2-1 at subsequent iterations. This generates some new facts, shown below:

{ (p(1,3): 0.5) ← (p(1,2): 0.5),(e(2,3): 0.5);
(p(2,4): 0.5) ← (p(2,3): 0.5):(e(3,4): 0.5);
(p(1,4): 0.5) ← (p(1,3): 0.5),(e(3,4): 0.5) }

With multiset-based union operation, this result is integrated with the bookkeeping, generated at the last iteration and forms the new bookkeeping.

Bookkeeping = {(p(1,2): 0.5) ← (e(1,2): 0.5);
(p(2,3): 0.5) ← (e(2,3): 0.5);
(p(1,3): 0.5) ← (e(1,3): 0.5);
(p(3,4): 0.5) ← (e(3,4): 0.5);
(p(1,3): 0.5) ← (p(1,2): 0.5),(e(2,3): 0.5);
(p(2,4): 0.5) ← (p(2,3): 0.5),(e(3,4): 0.5);
(p(1,4): 0.5) ← (p(1,3): 0.5),(e(3,4): 0.5) }

Before repartitioning, relation "p" is redefined by combining the results in the bookkeeping into a set of fact-certainty pairs through the associated disjunction function. During this redefinition of "p", fact p(1,3) is identified as being improved, from certainty 0.5 to 0.75. Furthermore, two new facts, p(2,4) and p(1,4), are generated. These three facts are then added into the improved partition (Δp):

Δp = {(p(1,3): 0.75), (p(2,4): 0.5), (p(1,4): 0.5)}

Since the certainty of p(1,3) is improved, all the related derivations in the bookkeeping are replaced with the new one. In this example, (p(1,4): 0.5) is inferred using p(1,3) and hence, we remove the instance (p(1,4): 0.5) ← (p(1,3): 0.5), (e(3,4): 0.5) from the bookkeeping.

At iteration 3, the only derivation we obtain is:

(p(1,4): 0.5) ← (p(1,3): 0.75),(e(3,4): 0.5).

This is added into the bookkeeping and "p" is redefined again. While re-partitioning "p", we find that no fact is included in Δp. In this case, no new fact-certainty pair will be derived in subsequent iterations, and hence the evaluation terminates.

According to this procedure, it is very easy to find and remove the derivations from the bookkeeping. We simply look at the body of a ground rule to check whether there is any fact whose certainty is improved. If there is at least one of such facts, remove the record from bookkeeping while repartitioning.

However, there is a disadvantage here. Supporting such bookkeeping needs a very large space. All the ground rules (at least the ground IDB instances) need to be recorded. Suppose a fact-certainty pair needs *n* bytes of memory to store. Also suppose there are m IDB body predicates in a rule on average. To bookkeep a derivation, n*(m+1) bytes are needed, in which *n* bytes for a result and *n*m* bytes for ground body, which is large when *m* is large. Furthermore, large space usually means expensive search. To avoid this, in our implementation, we avoid instantiating all rules and bookkeeping them. Only the results (the instances of rule head) are recorded.

A problem we may face in such an implementation is that, when we need to remove some records from the bookkeeping, because of some improved facts, we do not know which fact-certainty pairs are derived from those facts. To determine which pairs are to be removed, we introduce a *backtracking* method to update the bookkeeping during the evaluation. Let us have a look at the previous example $P_{5.4}$ again.

The bookkeeping for iteration 1 is

Bookkeeping ={(p(1,2): 0.5), (p(2,3): 0.5), (p(1,3): 0.5), (p(3,4): 0.5)}

The definitions of "p" and "Δp" are the same, as described above. The only difference is that there is no ground body associated with each result in the bookkeeping. At the end of iteration 2, we get

Bookkeeping ={(p(1,2): 0.5),(p(2,3): 0.5), ***(p(1,3): 0.5)***,(p(3,4): 0.5),
(p(1,3): 0.5),(p(2,4): 0.5),(p(1,4): 0.5)}

After combining the same facts using their associated disjunction functions, p(1,3) is derived with an improved certainty: from 0.5 to 0.75. To remove derivations related to (p(1,3): 0.5) from the bookkeeping, we apply the fact (p(1,3): 0.5) to each rule and derive again the related facts. Through this re-derivation, we identify (p(1,4):0.5) and then remove it from the bookkeeping. This re-derivation is done at each iteration to update the bookkeeping until the fixpoint is reached. Figure 5.6 shows the procedure we developed that implements the Semi-Naive with Partition method and backtracking.

procedure : Semi-Naive_Partition_backtracking(P,D; lfp($P \cup$D))

1 set P' to be the rules in P with ***no IDB*** predicate in the body ;
$T^0 := \varnothing$, $BK^0{}_T := \varnothing$, for ***IDB*** predicate T defined in P;
$M\Delta^1{}_T :=$ P'(I)(T), for all ***IDB*** predicate T;
$BK^1{}_T := BK^0{}_T \cup M\Delta^1{}_T$;
$T^1 := f_d(BK^1{}_T)$; $\Delta^1{}_T := f_d(M\Delta^1{}_T)$, for all ***IDB*** predicate T;

2 $i := 1$;
repeat
forall *IDB* predicate T, where $T_1,\ldots,T_m$ are the ***IDB*** predicates used in the definition of T:

3 $M\Delta^{i+1}{}_T := Eval^i{}_T(P(T), T_1^{i-1},\ldots,T_m^{i-1},T_1^i,\ldots T_m^i,\Delta^i{}_{T1},\ldots,\Delta^i{}_{Tm})$;

4 $BK^{i+1}{}_T := BK^i{}_T \cup M\Delta^{i+1}{}_T$;

5 $T^{i+1} := f_d(BK^{i+1}{}_T)$;

6 $Change^i{}_T := \{(T(\mu),C_i) | (T(\mu),C_{i+1}) \in T^{i+1} \wedge (T(\mu),C_i) \in T^i \wedge C_{i+1} > C_i \}$

7 $T^{i-1} := T^i - Change^i{}_T$;

8 $\quad$ $\text{Remove}^{i+1}{}_T := \text{Eval}^i{}_T(I, T_1^{i-1}, \ldots T_n^{i-1}, T_1^i, \ldots T_n^i, D\Delta^i{}_{T1} \ldots D\Delta^i{}_{Tn})$

9 $\quad$ $BK^{i+1}{}_T := BK^{i+1}{}_T - \text{Remove}^{i+1}{}_T$;

10 $\quad$ $T^i := T^i - \text{Change}^i{}_T$;

11 $\quad$ $\Delta^{i+1}{}_T := T^{i+1} - T^i$;

end forall;

12 $\quad$ $i := i + 1$;

until $\Delta^i{}_T = \varnothing$ for each IDB predicate T.

lfp($P \cup D$) := $\{T^i | \text{ T is an IDB predicate in } P\}$

end procedure

Figure 5.6: Procedure for Semi-Naïve evaluation with partition and backtracking

This procedure classifies rules in the input p-program into two partitions: rules with no IDB predicate in the body, and rules with some IDB predicates in the body. We first evaluate the former type of rules. This requires a single iteration, and no backtracking. The result of this iteration is used to initialize the bookkeeping and partitions (lines 1-2). For every iteration onward, new results of the evaluation are added (line 4) to the bookkeeping (see procedure $Eval^i{}_T$ in Figure 5.7). If the certainty of some facts is improved, the system backtrack the results derived by these facts (line 6) and removes them from the bookkeeping (lines 7-9). These operations are done iteratively, until every partition Δ_T is empty.

It is clear that backtracking is a kind of redundancy in the evaluation processing. However, it is necessary when having no details of a derivation. This redundancy is a tradeoff for saving space.

procedure : $\text{Eval}^i{}_T(P(T), T_1^{i-1}, \ldots, T_m^{i-1}, T_1^i, \ldots\ldots T_m^i, \Delta^i{}_{T1}, \ldots, \Delta^i{}_{Tm}; T)$

Forall r: $S(\mu) \leftarrow B_1(\nu_1), \ldots, B_n(\nu_n), T_1(\tau_1), \ldots, T_m(\tau_m); (f_c, f_p, f_d)$. in P, where $S = T, T_1, \ldots, T_m$ are ***IDB*** predicates, $B_1, \ldots, B_N$ are ***EDB*** predicates.

1 rewrite r with rewriting technique.

2 evaluate all rewritten rules with the input value.

end forall;

T:= {(f,c) | (f,c)∈ S } ;

end procedure

Figure 5.7: The procedure for evaluation of rewritten rules

5.3.3 Run-Time Decisions

The essence of query optimization techniques is having an efficient evaluation of the query. Since an evaluation has to deal with any input program, a particular optimization technique may be efficient for some programs and be poor for others [28]. Focusing on a set of optimization techniques, we expect that our system can automatically (at least partially) choose the suitable ones in order to obtain better performance across a wide range of programs. Since such selection is usually decided during the evaluation of a program, we call it run-time decision.

As mentioned in section 5.3.2, to avoid some repeated computations, SNP method may introduce some overhead for re-derivations when backtracking is applied, it is then possible that for some input programs, SNP may not perform as expected. To illustrate this input, let us consider the following rule:

$$S(\mu) \xleftarrow{\alpha} B(\upsilon), P(\tau), Q(\eta); < f_d, f_p, f_c > .$$

where B is an EDB predicate, and P, Q are IDB predicates. Let P^i and Q^i be the definitions of predicates P and Q at iteration i. At iteration $i+1$, the unchanged partition of P and Q, i.e., ΛP^{i+1} and ΛQ^{i+1} are not necessarily the same as P^i and Q^i respectively. This is because the certainty of tuples in P^i and/or Q^i may have improved and hence are

moved to the corresponding Δ partitions. Let us define $\Delta P^{i+1} = P^i - \Delta' P^{i+1}$ and $\Delta Q^{i+1} = Q^i - \Delta' Q^{i+1}$, where $\Delta' P^{i+1}$ and $\Delta' Q^{i+1}$ contain facts with improved certainties. By our back tracking method, the derivations that involve a fact in $\Delta' P^{i+1}$ or $\Delta' Q^{i+1}$ need to be redone and removed from the bookkeeping. The number of possible joins in these re-derivations is

$$\text{Cost} = |B| * |\Delta' P^{i+1}| * |Q^i| + |B| * |\Delta P^{i+1}| * |\Delta' Q^{i+1}|$$

where |X| represents the number of tuples in relation X. On the other hand, the number of redundant joins that SNP tries to avoid in this case would be:

$$\text{Saving} = |B| * |\Delta P^{i+1}| * |\Delta Q^{i+1}|$$

This is the number of joins of tuples done by evaluation of the last re-written rule. It is clear that SNP is useful in case Cost is smaller than Saving. However, this may not be always the case, as shown in the following example:

Let: $|B| = 100$;

$|\Delta' P^{i+1}| = 30$; $|\Delta P^{i+1}| = 50$;

$|\Delta' Q^{i+1}| = 40$; $|\Delta Q^{i+1}| = 110$;

The number of possible re-computing joins is

$$\text{Cost} = |B| * |\Delta' P^{i+1}| * |Q^i| + |B| * |\Delta P^{i+1}| * |\Delta' Q^{i+1}|$$
$$= 100*30*(110+40) + 100*50*40$$
$$= 650000$$

The number of possible joins of tuples that can be avoided is

$$\text{Saving} = |B| * |\Delta P^{i+1}| * |\Delta Q^{i+1}|$$
$$= 100*50*110$$
$$= 550000$$

To avoid 550,000 joins, SNP needs to re-compute 650,000 joins. Obviously, SNP is not beneficial in this example.□

In general, for a rule in form:

$$S(\mu) \xleftarrow{\alpha} B_1(\upsilon_1),...,B_n(\upsilon_n), T_1(\tau_1),...,T_m(\tau_m); < f_d, f_p, f_c >.$$

where B_j is EDB predicate, for $j \in \{1,...,n\}$ and T_k is IDB predicate, for $k \in \{1,...,m\}$, the cost of SNP, at iteration i, would be

$$\begin{aligned} \text{Cost} = & |\Delta' T_1^{i+1}| * |T_2^{i}| *...* |T_n^{i}| +...+ \\ & |\Lambda T_1^{i+1}| *...* |\Delta' T_{k+1}^{i+1}| *...* |T_n^{i}| +...+ \\ & |\Lambda T_1^{i+1}| *...* |\Lambda T_{n-1}^{i+1}| * |\Delta' T_n^{i}|; \end{aligned}$$

where $\Delta' T_j^{k}$ contains all T_j facts with improved certainties at iteration k, and ΛT_j^{k} contains all T_j facts whose certainties are not improved at this iteration.

The saving by SNP would be

$$\text{Saving} = |\Lambda T_1^{i+1}| *...* |\Lambda T_n^{i+1}|;$$

For each iteration, if Cost < Saving, then SNP method is used. Otherwise, we use the SN method. Since the size of each partition of all IDB relations may change during different iterations, a desired evaluation would use the preferred method, SN or SNP, at each iteration, depending on the values of Cost and Saving defined above. Another situation to be considered is that different rules may require different techniques at a particular iteration. For the above two reasons, we have introduced this run-time decision capability and implemented it in our system.

Besides selecting a desired evaluation method dynamically, the query optimizer also decides whether a rule needs to be evaluated at a particular iteration at run-time. We can view a rule in a Datalog program to indicating a "method" to infer the tuples of the head predicate, through a series of join, selection, and projection operations. There is, however, no guarantee that a rule will always infer tuples for its head predicate. This is

the case when the rule has a predicate in the body with no tuple. In this situation, we consider not to evaluate this rule and avoid unnecessary computations. This decision at run-time is important for a query optimizer. Since a rule is normally evaluated from left to right, the two left most body predicates join first. The result then joins with the third body predicate, and so on (details of program evaluation will be discussed in section 5.4). In case there is an empty body predicate, all these joins will actually be a waste. To avoid this waste, rules having an empty body predicate should be identified and excluded from the evaluation. A simple check for identifying rules with empty predicate body could be done before starting each iteration.

Another important run-time decision is on indexing decision. When the size of a relation is small, creating and/or using an index may cause an overhead and is discouraged. For example, if an IDB predicate has 10 facts, the overhead of creating a hash table and searching it could be much more costly than forgetting all about it and using sequential search. Similarly, when indexing is needed, the size of hash tables should be determined at run-time. Currently, the consideration of the size of a relation and of a hash table is excluded from our indexing for simplifying the implementation. However, it is an important issue for our further research.

5.3.4 Rule Re-ordering and Stratification

In section 3.4, we argued that the stratified evaluation is an efficient technique for p-programs since it avoids intermediate derivations by delaying the evaluation of higher strata until the final evaluation results of those lower strata are obtained. Since we store the input rules, in our implementation, as a list and evaluate them from head to tail, rules

must be partitioned into different strata and reordered, and evaluated according to their strata.

We consider two main steps to stratify the input rules. First, we create the predicate dependency graph (PDG) of the input p-program. This PDG is then used to determine the strata as well as the rules each stratum includes. Given a p-program P, the predicate dependency graph of P, denoted as PDG(P), is a directed graph whose nodes are the predicates in P and for every rule $H \xleftarrow{\alpha} B_1,...B_n$ in P, there is an edge in PDG(P) from $\pi(B_i)$ to $\pi(H)$, for $0 < i \leq n$. Recall that $\pi(B_i)$ denotes the predicate of atom B_i. Figure 5.8 shows an example dependency graph.

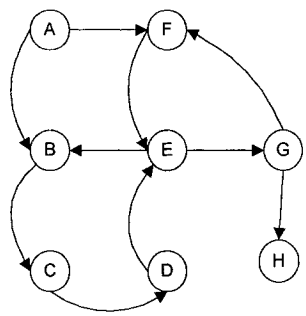


Figure 5.8: An example of predicate dependency graph

Stratification is actually based on the notion of Strongly Connected Components (SCC) construction, which identifies all cycles containing maximum number of predicates in a dependency graph. There are numbers of efficient algorithms for finding SCC in a directed graph [33, 12, 25, 6]. These algorithms basically consider two interleaved traversals of a graph. First, a depth-first search traverses all the edges and constructs a depth-first spanning forest. Second, once a so-called root of an SCC is found, all its descendants that are not elements of previously found components are marked as elements of this component. In our implementation, we use Tarjan's algorithm [33],

which we modified slightly as follows. We first find, for each IDB predicate p, the cycles in which p is involved, using depth-first search method. Two stacks are used to record the cycle path and to support backtracking. Once all cycles are identified, cycles sharing some common elements are then merged to form a bigger cycle.

For a simple illustration, consider the PDG shown in Figure 5.8. There are two cycles in the graph: C_1:{B,C,D,E} and C_2:{E,F,G}. Since C_1 and C_2 share node E, every node in C_1 has at least a path to reach the elements in C_2. For example, node B in C_1 can reach node F in C_2 through path $B \rightarrow C \rightarrow D \rightarrow E \rightarrow G \rightarrow F$ and node G in C_2 can connect node D in C_1 through path $G \rightarrow F \rightarrow E \rightarrow B \rightarrow C \rightarrow D$. Therefore, all the elements in C_1 and C_2 should belong to the same SCC. These two cycles are thus merged to form an SCC. An SCC is formed until all cycles in which any two of them share some common elements are merged. In Figure 5.8, element B, C, D, E, F and G construct an SCC.

Unlike in the conventional SCC, we also consider SCCs of a single node as well. In Figure 5.8, node A and H form two separate SCCs.

The final step of our stratification method is to stratify SCCs. The principle of stratification is to include all the dependants of a predicate p in some strata below the one which includes p. Obviously, all EDB predicates should be included in the lowest stratum. The algorithm for stratifying SCCs is shown in Figure 5.9, which inputs are a set of rules $\boldsymbol{R}$ in a p-program and a set of SCCs ($\mathbf{S}_{SCC}$) in $\boldsymbol{R}$, and its output is a set of stratified SCCs. For an SCC X and atom h, we use $h \in X$ in the algorithm to mean $\pi(h) \in X$.

Procedure: stratify_SCC($\boldsymbol{R}$, $\mathbf{S}_{SCC}$; $\mathbf{S}_{SCC}$)

$\forall_{X \in S_{SCC}} X :: tag = 1$

i:=1

loop

forall $X \in \mathbf{S}_{SCC} \wedge X :: tag = i$

let $x \in \{ y \mid y \in Y \wedge Y \in \mathbf{S}_{SCC} \wedge Y :: tag = i \}$

if $x \in \{b_i \mid r : h \leftarrow b_1, b_2, \ldots b_n \in \boldsymbol{R} \wedge h \in X$, for $i \in \{1,\ldots,n\}\}$

$X :: tag$ ++;

end forall

i++;

until $\{Y \mid Y \in \mathbf{S}_{SCC} \wedge Y :: tag = i\} = \phi$

end procedure

Figure 5.9: The stratification algorithm

The basic idea of this algorithm is that, initially, it puts all the SCCs in the lowest stratum. If an element *X* of an SCC is dependant on element *Y* of another SCC, it then moves the SCC containing *Y* to a higher stratum. This process is repeated until no SCC needs to be relocated. For the example in Figure 5.8, the SCCs {*B, C, D, E, F, G*}, {*A*}, and {*H*} are initialized as stratum 1. Since *A* depends on *F* and *B*, which are elements of {*B, C, D, E, F, G*}, {*A*} is relocated to stratum 2. Similarly, {*B, C, D, E, F, G*} is also relocated to stratum 2 because *G* depends on *H*, which is an element of another SCC. Since *H* depends on nothing, {*H*} remains in stratum 1. Furthermore, {*A*} is relocated to stratum 3 since *A* depends on *F* and *B*. Eventually, no SCC needs to be relocated and the desired stratification is done as follows:

stratum 1	{H}
stratum 2	{B, C, D, E, F, G}
stratum 3	{A}

Note that predicate stratification entails rule stratification. That is, every rule defining *H* is in stratum 1, every rule defining *A* is in stratum 3, and the rest of the rules are

collectively in the middle stratum, in this example.

5.4 Query Evaluation

In this section, we discuss query processing strategies. For this, we will discuss a number of related concepts and techniques, which include materialization, pipelining, and dynamic variable binding. A precision control mechanism is also introduced in this section. We provide a brief description of these, before we discuss them in detail.

There are two main categories of query evaluation strategies: bottom-up and top-down. In our system, we focus on the bottom-up fixpoint evaluation, which attempts to infer all possible fact-certainty pairs and returns the ones subsumed by the user query. As mentioned before, evaluation of a rule actually performing a series of joins of tuples defined by the body predicates. There are two possible ways to perform these joins: materialization and pipelining. We will discuss these two options in details in section 5.4.1. During the joins, a *variable table* is created and updated to provide a match pattern to fetch the facts. Dynamic updating of the variable table will be discussed in section 5.4.2. Recall that evaluation of some p-program may reach the fixpoint at ω, i.e., it will not stop in finite time. This allows a fixpoint evaluation to continue until a desired level of precision is obtained. A user should thus interact with the system to provide this precision. This issue is further discussed in section 5.4.3. When stratification is incorporated into our system, an input p-program may be divided into several modules. In this case, the system must coordinate the evaluations of different modules. Section 5.4.4 will provide some suggestions to achieve this.

5.4.1 Materialization and Pipelining

The materialization approach to query evaluation considers a sequence of joins of body predicates as sequential tasks. The join between two body predicates is independent of and completed before starting the next join. The result of a join is stored in memory as an intermediate relation, which is then joined with the next predicate, the result of which is another intermediate relation. This process is repeated until all the body predicates are joined. There are some concerns with materialization. First of all, the system must reserve some space for intermediate relations. In case the intermediate relations are large, the space required will be large as well. In addition to creation of these intermediate relations, we also need to maintain and search them. These clearly will introduce an overhead, which could be too expensive to bear. Another problem with materialization originates from the repetition of making match patterns. For each instance of join, the system must fetch a fact from the intermediate relation and bind values to shared variables. These bound, shared variables then form a pattern to restrict the fact fetching in subsequent relations.

Unlike materialization, which can be viewed as a "piece-meal" computation, the pipelining approach considers a sequence of joins of body predicates as an "integrated" task, so that it is not needed to construct intermediate relations. Once a tuple is obtained from a join operation, it is "piped" to the next join operation, and so on, until all instantiated body predicates are evaluated. Only the final tuples of head predicate are materialized into memory. No other instantiated rules are evaluated until the current instantiated rule is completed. Pipeline approach solves the problems of materialization explained above. There is no space requirement for intermediate relations and hence no

overhead of intermediate relation creation and maintenance. After an instance of join is done, only the variables bound by the last fetched fact are freed for the next join, and hence the system does not need to re-initialize shared variables.

Clearly, both materialization and pipelining have advantages and disadvantages. However, the advantages of materialization are mainly due to the disk I/O cost. If all the data for an evaluation are stored on disks, materialization is often preferred over pipelining for its less number of disk I/O, depending on the main memory available. However, since our system is an in main-memory system, we are not concerned much with disk and disk I/O. Furthermore, there is no conventional aggregation during the evaluation of a rule body in our context—less reason to adopt the materialization method. On the basis of the above discussion and observation, we have implemented pipelining in the evaluation approach in our system.

An advantage of materialization is reusability. In case the result of a join is going to be reused, materialization can be used to avoid re-computing the same joins. However, it seems less likely that such opportunities exist for p-programs to include repeated joins. We ignore this advantage in our consideration for the evaluation approach.

5.4.2 Dynamic Variables Binding

Before performing the joins of body predicates, a variable table is created, for each rule *r*, recording the status of each variable occurring in the body predicates of *r*. Initially, we associate an un-restricted match pattern ∅ with *r*. When a fact of the current body predicate is obtained, the variables occurring in this predicate become bound. This in turn makes the common variables, which form a match pattern, direct the selection of facts from the rest body predicates in *r*, to the right. During a sequence of joins of the body

predicates, more and more variables become bound, causing the variable table and match pattern to be updated accordingly. In the pipelining approach, all the variables become bound when an instance of a sequence of joins is completed. These bound variables actually form an anonymous ground tuple containing all the bound variables as its attributes. A projection is then applied on this anonymous ground tuple to produce a fact for the head predicate when all terms of the head predicate are variables. When a fact is generated, it is not necessary to re-initialize all variables in the variable table. There may be other facts that belong to the predicates to the right, which satisfy the current match pattern. In this case, all the variables bound by the fact from the right most body predicate must change from bound to free. These free variables become bound later when another matching tuple is found. In some situation, we not only need to unbind variables, but also need to undo match pattern. Undoing match pattern is caused by the "freeing" of shared variables. In case there is no more suitable fact for the current match pattern, another fact of the left-hand-side predicate is fetched. Therefore, the variables, including common variables, bound by the previous fact from the body predicate on the left, become unbound. Accordingly, we undo the match pattern. After that, variables are re-bound and match pattern is re-built according to the newly fetched fact. This new match pattern is then applied to find matched facts in the next predicate, to the right.

Let us consider example $P_{5.2}$ from in section 5.2.1 and use it to demonstrate how variables are dynamically bound and unbound:

$$r: \text{sgc}(X, Y) \xleftarrow{1} \text{parent}(X, Z), \text{sgc}(Z, W), \text{parent}(Y, W); \langle \text{min}, \text{min}, \text{ind} \rangle.$$

initially, the following variable table is constructed for rule *r*; in which all variables are marked as unbound:

X	Z	W	Y
unbound	unbound	unbound	unbound

Suppose we have the following fact-certainty pairs in relations "parent" and "sgc":

parent	sgc
(John, Mike): 0.5	(Mike, Marry): 0.5
(Eric, Peter): 0.5	(Mike, Paulo): 0.5
(Jim, Paulo): 0.5	(Rudy, Nichol): 0.5

When evaluating *r*, we first fetch (John, Mike): 0.5 from "parent." This causes variables X and *Z* in "parent" to bind to "John" and "Mike", respectively. The variable table is then updated accordingly. Since *Z* is a common variable of "sgc" and the first occurrence of "parent", the current match pattern is {Z = "Mike"}. Using to this match pattern, the system fetches from predicate "sgc", the first fact matched, i.e., (Mike, Marry): 0.5. Once a matched fact is fetched, the variable table is updated again. Now, *W* is bound by "Marry." The match pattern is in turn updated to {Z = "Mike", W="Marry"}, since variable *W* is a shared variable with the second "parent" predicate. However, there is no fact in "parent" that matches the current match pattern. An "undo" operation is thus performed on variable table and the match pattern. All variables bound by tuple (Mike, Marry): 0.5, *W*, in this case, in relation "sgc" are changed to free. Then the next suitable fact of "sgc" is fetched, (Mike, Paulo):0.5, in this case. Variable *W* is then bound by "Paulo" and a new match pattern {Z = "Mike", W="Paulo"} is constructed. According to this pattern, tuple (Jim, Paulo): 0.5 in "parent" is fetched and variable *Y* is bound by "Jim." Now, all variables in the variable table are bound, resulting a successful derivation of an atom, sgc(John, Jim):0.5, by projecting on variables *X*, *Y* in the head.

The variable table now looks as follows:

X	Z	W	Y
John	Mike	Paulo	Jim

To find the next possible fact in "parent", variable *Y* is changed to free. However, scanning the whole relation "parent" reveals that no more fact exists that matches the pattern {Z = "Mike", W="Paulo"}. Therefore, *W* becomes free and the match pattern is undone to {Z="Mike"}. The current variable table is as follows:

X	Z	W	Y
John	Mike	unbound	unbound

Again, no more fact in "sgc" matches the pattern, and hence, variables *X* and *Z* become free and the match pattern is undone to its initial state, ∅. The second tuple (Eric, Peter):0.5 in "parent" is fetched and variables *X* and *Z* are bound again. This process continues until all facts corresponding to the first occurrence of predicate "parent" are considered.

5.4.3 Precision Controlling

As introduced in chapter 2, evaluations of some p-programs may terminate only at the limit, i.e., iteration step ω. This may occur when (1) the input program is recursive, and (2) the data is cyclic, and (3) the disjunction function associated with recursive pedicate is type 2 or 3. Recall that the result of such a disjunction function is often "better" than its input arguments [19]. Since the disjunction function produces "better" certainty for a fact, this fact can then contribute to a new derivation of itself, causing its certainty to improve even further. Since the underlying fixpoint operator is monotone and continuous [19], such recursive derivation goes on and terminates only at ω. Even though this is not a

desired situation in practice, it indicates an opportunity for approximation to the program model, should take advantage of this, be desired. To avoid an infinite evaluation, while approximating to the least model, we introduce a "precision check" technique. This technique essentially takes advantage of the continuity property of the fixpoint operator to "approach" the fixpoint. A certainty precision, denoted as Δ to our system, is provided by the user. For instance, $\Delta=10^{-5}$. At the end of each iteration, the system checks whether the certainty of each fact has grown more than Δ, compared to the previous iteration. If for any fact, the answer is negative, then the evaluation continues to the next iteration. Otherwise, it terminates, since it has "approached" the fixpoint, even though it is not there yet.

5.4.4 Stratified Evaluation

As in Datalog, a p-program contains two kinds of predicates: EDB predicates and IDB predicates. Different kinds of predicates have different operations when they are indexed (see section 5.2) and partitioned (see section 5.3). Especially when SNP is applied, only IDB predicates are partitioned. Since there is a cost for partitioning and maintenance of the partitions, we should try to reduce this cost as much as possible. Stratified evaluation may help to reduce this cost.

Stratification divides an input p-program into several strata and the system evaluates it stratum by stratum in the order, starting with the lowest stratum. During the evaluation, each stratum can be considered as an independent sub-program. Therefore, the definition of an IDB predicate in a lower stratum can be considered as an EDB predicate for some higher strata since the definitions of such IDB predicates will not changed in the rest of the evaluation.

In conclusion, the idea of stratified evaluation introduced helps to reduce the overhead of evaluation as much as possible by considering all the IDB predicates in lower-level strata as EDB predicates in the higher-level strata.

We have not completed the stratified evaluation in our current implementation due to time constraint, and it is left as a future work. We hope the reader is convinced, by the arguments provided, that stratified evaluation could be an important evaluation scheme, in conjunction with a wealth of techniques and tricks introduced in this research and by others.

Chapter 6

Performance Analysis and Evaluation

To evaluate the performance of query optimization techniques implemented in our system, we conducted a number of experiments under different p-programs. The evaluation time of experiments has been considered as the main parameter of performance. In our experiments, the following techniques (or combined techniques) are evaluated:

- Naive: the basic fixpoint evaluation technique (discussed in chapter 2), through which bottom-up fixpoint evaluation is conceptually defined.
- Naive with Indexing: a set of indices is generated, by analyzing the input program, for shared attributes of each body predicate (discussed in section 5.2). To keep the number of indices "reasonable", we applied subgoal reordering strategy on each rule.
- Semi-Naive: a basic *multiset-based* Semi-Naive evaluation technique proposed in [32], which extends the standard Semi-Naive technique for uncertainty. Same as in the Naive technique, we use indices in conjunction with this evaluation method.
- Semi-Naive with Partition: the basic *multiset based* Semi-Naive technique is further improved by partitioning IDB predicates into "improved" and "non-improved" parts to avoid repeated computation (discussed in section 3.3 and 5.3). Indices are also used with this method.

This chapter is divided into 5 sections. Section 6.1 will describe the experimental environment. In section 6.2, we will introduce the kinds of p-program we used in our performance evaluation. The selection and generation of test data for each set of experiments will be discussed in section 6.3. Our experimental results and analysis are provided in section 6.4 and 6.5. More specifically, Section 6.4 will focus on the indexing and section 6.5 will focus on Semi-Naive (SN) and Semi-Naive with Partition (SNP) evaluations. In these experiments, we record the evaluation time and use it to compare the various evaluation schemes. We remark that there were no benchmark program and test data to be used in our context. However, the test data and programs we created are representative of a wide range of size and complexity.

6.1 Experiment Environment

Our experiments were carried out on an IBM desktop computer with a Pentium 4 CPU of 2.4GHz, 512MB main memory, 80GB hard disk, and run under Windows 2000 professional operating system. The block size of main memory remained default.

6.2 Test Programs Selection

Recursion is an attractive feature of deductive databases, standard or otherwise [35, 9, 28, 7]. There are two kinds of recursive programs: linear and non-linear programs. For linear program, only one mutually recursive predicate exists in the body of a recursive rule. In case the body of a recursive rule contains more than one mutually recursive predicate, the program is called a non-linear program. In our experiments, we considered four test programs, *p1* to *p4*. These test programs are defined in Figure 6.1. Program *p1* and *p2* compute the transitive closure of a binary predicate, edge, where *p1* is linear and *p2* is

non-linear. Program *p3* and *p4* compute the so-called same-generation relationship, where *p3* is the regular one, and *p4* is the same program extended with supplement magic predicates and rules. All these four programs use 1 as certainty of rules with some certainty combination functions, as defined in [19]. In the rest of this section, each test program will be described briefly.

$r_1 : p(X, Y) \xleftarrow{1} e(X, Y) ; \langle f_c, f_p, f_d \rangle.$

$r_2 : p(X, Y) \xleftarrow{1} e(X, Z), p(Z, Y); \langle f_c, f_p, f_d \rangle.$

(a) Test program *p1*

$r_1 : p(X, Y) \xleftarrow{1} e(X, Y) ; \langle f_c, f_p, f_d \rangle.$

$r_2 : p(X, Y) \xleftarrow{1} p(X, Z), p(Z, Y); \langle f_c, f_p, f_d \rangle.$

(b) Test program *p2*

$r_1 : sg(X, Y) \xleftarrow{1} flat(X, Y) ; \langle f_c, f_p, f_d \rangle.$

$r_2 : sg(X, Y) \xleftarrow{1} up(X, Z1), sg(Z1, Z2), flat(Z2, Z3), sg(Z3, Z4), down(Z4, Y); \langle f_c, f_p, f_d$

(c) Test program *p3*

$r_1 : msg(1) \xleftarrow{1} ; \langle f_c, f_p, f_d \rangle.$

$r_2 : supm2(X, Y) \xleftarrow{1} msg(X), up(X, Y); \langle f_c, f_p, f_d \rangle.$

$r_3 : supm3(X, Y) \xleftarrow{1} supm2(X, Z), sg(Z, Y); \langle f_c, f_p, f_d \rangle.$

$r_4 : supm4(X, Y) \xleftarrow{1} supm3(X, Z), flat(Z, Y); \langle f_c, f_p, f_d \rangle.$

$r_5 : sg(X, Y) \xleftarrow{1} msg(X), flat(X, Y); \langle f_c, f_p, f_d \rangle.$

$r_6 : sg(X, Y) \xleftarrow{1} supm4(X, Z), sg(Z, W), down(W, Y); \langle f_c, f_p, f_d \rangle.$

$r_7 : msg(X) \xleftarrow{1} supm2(Z, X); \langle f_c, f_p, f_d \rangle.$

$r_8 : msg(X) \xleftarrow{1} supm4(Z, X); \langle f_c, f_p, f_d \rangle.$

(d) Test program *p4*

Figure 6.1: Test programs in performance experiment

Figure 6.1 (a) shows the details of program *p1*. *p1* finds pairs of nodes (X,Y) in the input directed graph such that node "Y" is reachable from node "X." The certainty associated with each pair in the result represents the probability of this connection. In this linear transitive closure program, the size of one joined relation increases until the fixpoint of the evaluation, while the size of another joined relation remains unchanged during the whole evaluation.

Program *p2* is presented in Figure 6.1 (b). It has the same function as program *p1* except that it uses different method to find the reachable nodes. It is a non-linear transitive closure program. As can be seen, there are two occurrences of the recursive predicate "p" in the body of rule r_2. Here, unlike *p1*, the size of more than one body predicate increases during the evaluation.

Program *p3* defines pairs of individuals that are at the same level in a family tree (see Figure 6.1 (c)). This is slightly different version of the so-called same-generation program. Notice that program *p3* cannot find out all same generation pairs unless a specific path, indicated by the second rule r_2, is satisfied. This program was chosen in our performance evaluation since r_2 has a long chain of EDB and IDB predicates in the body. In *p3*, "up", "flat" and "down" are EDB predicates and the rest are IDB predicates.

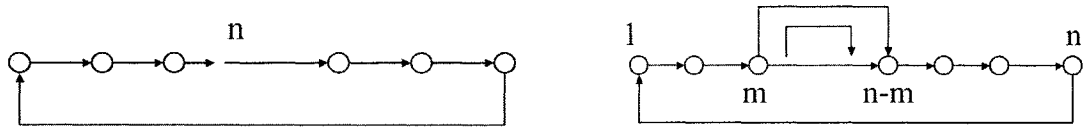
Program *p4* is a rewritten version of same-generation program *p3*, using the supplementary magic set technique [5]. It is exhibited in Figure 6.1 (d). Program *p4* contains a number of rules and many its predicates are IDB predicates. This program was chosen in our experiment since the original same-generation program has been widely used for evaluating optimization techniques in classical deductive databases [26, 10, 4, 15]. We need to mention that the modified program is not equivalent to the original

program with uncertainty even though it does not affect our choice as a test program. An interesting feature of this program in our context of performance evaluation is that it includes many simple rules and most of which define small relations.

For all test programs, we have used *ind* as the disjunction function associated with all rules. This function can be viewed as a representative of disjunction type 2 and 3 (described in chapter 2), which are very important to our experiments. The conjunction and propagation functions are either *min* or *product*. In some cases, *product* is undesired because it often generates a smaller value than its arguments. This causes problem for the precision of the uncertainty values computed for derived facts. Too small certainty value may make the derived fact have least certainty and therefore terminate the further derivation quickly.

6.3 Test Data Selection and Generation

In the context of standard Datalog programs, there have been a number of data sets widely used to measure the efficiency of query processing and optimization techniques [26, 10, 4, 15]. We adopt these data sets in our context and developed program moduler that generates suitable large data sets with uncertainty. In total, 9 categories of input data have been chosen in our performance experiment. Figure 6.2 and 6.3 exhibits these data graphically. For programs *p1* and *p2*, we consider two different collections of EDB facts with different numbers of tuples. The collection CT_n of data set includes cyclic data of the form: e(0,1), e(1,2),..., e(n,0), for n = 10K, where $1 \leq K \leq 15$. Figure 6.2 (a) shows an example of CT_n.



(a) CT_n Data (b) M_n Data

Figure 6.2: Data set for program *p1* and *p2*

Incorporating with type 2 or type 3 disjunction functions, such as *ind*, both *p1* and *p2* improve the certainty associated with each IDB facts at every subsequent iteration. For *p1*, there is unique path between every pair of nodes connected. For example, a connection from node *i* to node *j* is only derived by e(i,i+1) joins p(i+1, j). There is no other join deriving this connection. Unlike *p1*, there are many paths in *p2* defining the pairs of connected nodes. To be more precise, *p2* has *n+1* different joins to define each such pairs, namely, the joins of p(i, k) joins p(k, j), where $k \in \{0,\ldots,n\}$. Compared to *p1*, there are more derivations at each iteration of evaluating *p2*. Under the same data set, this causes the evaluation of *p2* to terminate in less number of iterations than *p1* 's evaluation. However, the amount of computations of evaluating *p2* at each iteration is more.

The second data set we used for evaluating *p1* and *p2* is $M_{n,m}$, shown in Figure 6.2 (b). It includes *n* nodes and circle connections for each interval of *m* nodes. This data set constructs multi-circles instead of the single circle in CT_n. The number of paths between two nodes is relatively large and the workload at every evaluation step is also more.

Besides CT_n and $M_{n,m}$, we constructed several other data sets for *p3* and *p4*. These data sets are described as follows:

A_n: Figure 6.3 (a) shows the structure of A_n. It looks like a triangle shape made by layers of nodes. As the number of layers increases, the size (in terms of the number of nodes) of the bottom layer increases. There is at most one matched path from a node to its same

generation node. It means that if there is an inference for an answer, this inference must be unique. To find a same generation node in level *i*, the derivation must go to the top-level layer first and then down to the layer *i*. The more layers there are, the more workload is involved in the derivations. This data set is used to measure the speed of various evaluation methods under the environment that least derivation is achieved.

B_n: The data set B_n contains *n* layers of nodes. Each layer has 8 nodes, which form a double linked list. There are four arcs connecting a lower layer and its immediate higher layer. Two are upward and the other two are downward. Figure 6.3 (b) shows the structure of B_n. When we look at this data set carefully, we note that there are exactly 3 answers to query ?sg(1,X) for *p4* whenever the number of layers *n* is larger than 2. When *n* increases, there is no more answer generated from *p4* but the certainties of these answers may be improved since more paths contribute to the answers. Unlike *p4*, program *p3* has more answers when the number of layers increases.

C_n: The data set C_n is very similar to B_n. Each layer in C_n includes a single linked list of 8 nodes. Each node has an arc connecting to the corresponding node in the higher layer. All arcs connecting a higher layer to its immediate lower layer are bi-directional. Figure 6.3 (c) shows the structure of C_n. There are exactly 4 answers for query ?sg(1,X) from *p4* whenever the number of layers *n* is larger than 2. As in B_n, when *n* increases, there is no more answer generated from *p4* but the certainties of these answers may be improved since more paths contribute to the answers. Even though all nodes have an arc connecting the corresponding node in higher layers, the number of derivations of *p3* under C_n may not be more, compared to using B_n. This is because the flat arcs are changed to single direction and hence no derivation exists from a right node to the left.

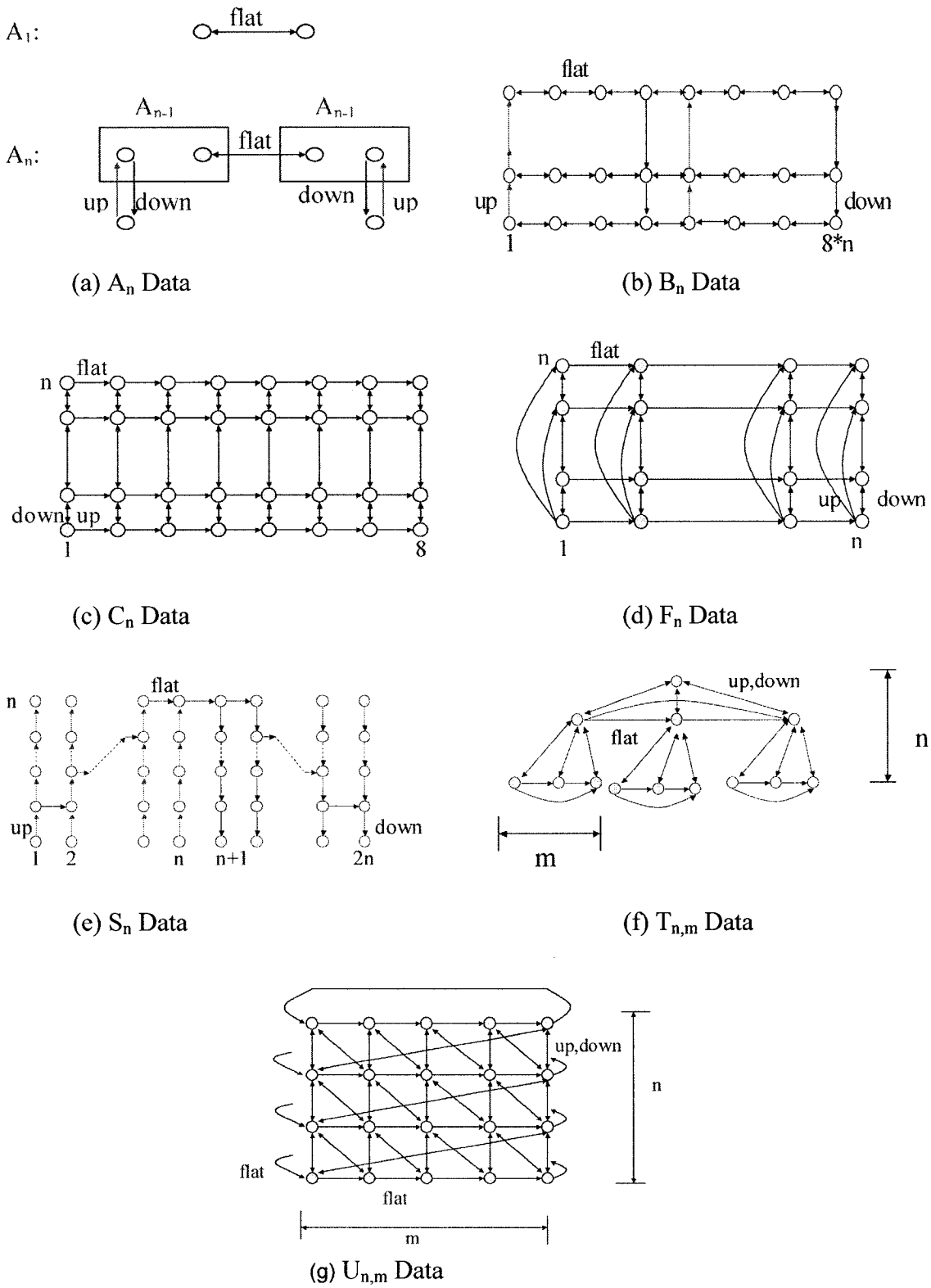


Figure 6.3: Data set for program *p3* and *p4*

F_n: F_n is a variant of C_n. Unlike in C_n, the length of each layer and the number of layers in F_n are flexible. This structure makes C_n graph to a square shape (see Figure 6.3 (d)). The number of layers is same as the length of each layer. Each node in the lowest layer has an extra arc to the corresponding node of each higher layer. All these extra arcs are upward (named "up"). Since the length of each layer is flexible, the number of query answers (both *p3* and *p4*) increases when *n* increases. However, the increase for *p4* is faster than that for *p3*.

S_n: S_n includes $2n$ linked lists, each having *n* nodes. The first *n* lists have upward links, while the second *n* lists have inverted links. There is only one link between two nearest linked lists. The graphical description of S_n is shown in Figure 6.3 (e). It is easy to see that there is only one path for node 1 to its satisfied same generation node 2n and the derivation should be done recursively from top to bottom. Compared to the size of data set, the number of derivations is small for S_n.

$T_{n,m}$: $T_{n,m}$ data set represents an *m*-ary tree, where n indicates the height of the tree and m indicates the number of children a node has. Figure 6.3 (f) shows $T_{2,3}$, as an example of this data structure. $T_{n,m}$ provides many paths between any two same generation nodes. When the height *n* increases by 1, the total number of arcs in the tree increases by $3m^{n-1}$. That is, the increase of the size of data set is exponential.

$U_{n,m}$: This data set is another variant of C_n. It makes the number of layers *n* and the length of each layer *m* flexible. Each layer is renovated to be a cycle-linked list. Moreover, the connections between two nearest layers become more frequent. From Figure 6.3 (g), we can see that each node has two bi-directed arcs to nodes at its immediate upper layer. These arcs cause $U_{n,m}$ to provide a large number of paths between

two same generation nodes. Hence, the number of derivations in a given program becomes large.

All above different types of data sets are used in our experiments, as they represent different types of derivations in terms of the number of derived facts as well as the structure of the facts. In our experiments, A_n and S_n, used for *p3* and *p4*, result in fewer derivations. There is only one output for A_n and S_n as the input of *p3*. Compared to A_n, the number of iteration to reach fixpoint is less when using S_n. On the other hand, B_n and C_n take moderate number of different paths to obtain the output. Both B_n and C_n take about the same number of iterations to reach the fixpoint, while C_n has more failed joins than B_n. Data sets F_n, $T_{n,m}$, and $U_{n,m}$ take large number of different paths to obtain the output but with fewer number of iterations. We expect that these data sets to be rich enough representations to support a fair and through evaluation of the proposed execution scheme.

The data sets data sets A_n and B_n are adopted from [15], C_n, F_n and S_n from [26], and $T_{n,m}$ and $U_{n,m}$ are adopted from [4]. Unless specified otherwise, the certainty associated with every fact in these data sets is 0.5. We developed in C++ the program module for these data generators, which take as input parameters the number of layers and the length of each layer, and produce the desired test data.

6.4 Index Performance Evaluation

For the obvious reason, we used indexing for efficient query processing in our context. For this, we used basic heuristic to determine a reasonable number of indices. To measure the gain of indices considered, we run *p1*, *p2*, *p3* and *p4* on all data sets introduced in

section 6.3. We used the elapsed time as the performance parameter. The elapsed time is defined as the period from submitting the program to listing the last answer to query. It includes compilation time (such as data transformation, indexing and so on), and execution time. We will provide some of these evaluation results in this section. Appendix A presents all theses results. Column "Data Set Class" in each result tables indicates the kind of data set used. Column "Output Tuples" shows the number of answers to the query. Column "Number of Iterations" gives the number of iterations required to reach the fixpoint. Columns "Indexed" and "Non-Indexed" correspond to the evaluation elapsed time (in seconds) when using or not using indices, respectively. The ratios non-indexed/indexed of elapsed time indicate the speedup, shown as the last columns in each result table.

Data Set Class	Output Tuples	Number of Iterations	Non-Indexed	Indexed	Speed-Up (Non_Indexed/Indexed)
CT_{10}	100	71	0.19	0.07	2.71
CT_{30}	900	122	7.882	1.462	5.39
CT_{50}	2500	151	53.397	6.499	8.22
CT_{70}	4900	167	260.355	16.584	15.70
CT_{90}	8100	181	797.817	32.897	24.25
CT_{110}	12100	188	1756.59	52.705	33.33
CT_{130}	16900	187	3033.74	72.104	42.07
CT_{150}	22500	188	4733.2	97.701	48.45

Table 6.1: Running *p1* on CT_n with/without indexing

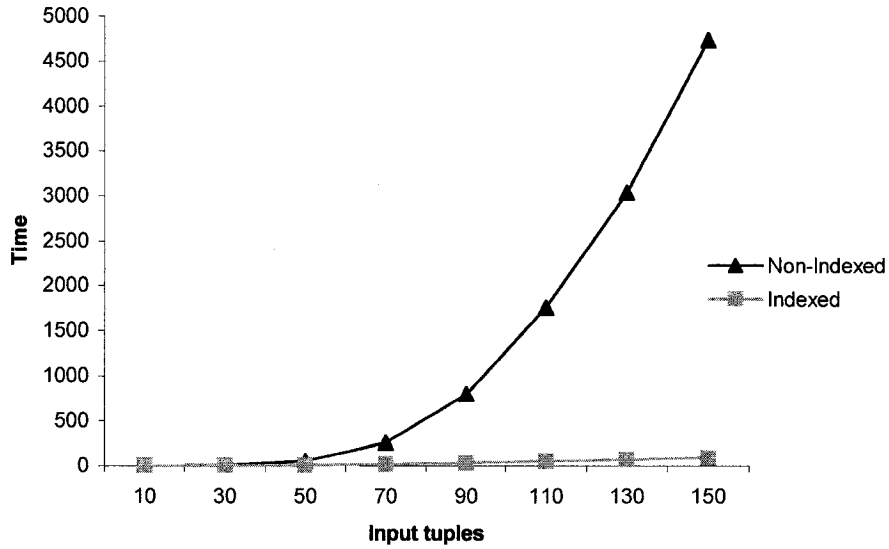


Figure 6.4: Running p1 on CT with/without indexing

Table 6.1 and Figure 6.4 show the experiment result of running *p1* on different size of data set CT_n. The certainty assigned to each EDB facts is 0.9. Disjunction function is *ind*, and both propagation and conjunction functions are * (the product). As expected, indexing improves the performance significantly. The results indicate a wide range of speed up from 2.71 (CT_{10}) to 48.45 (CT_{150}). Table 6.1 also shows that as the size of the input data set CT_n increases, the number of derived facts as well as the number of evaluation iterations increase.

Table 6.2 and Figure 6.5 show the measurement of evaluating *p2* on $M_{n,m}$. The certainty of each EDB fact is 0.5 and the combination functions are same as in *p1*. The speed up obtained as can be seen from the table ranges from 5.15 for $M_{20,4}$ to 28.17 for $M_{100,4}$. By running *p2* on the data set $M_{n,m}$, a smaller number of input facts (compared with output) may derive a large number of IDB facts within a small number of iterations.

Data Set Class	Output Tuples	Number of Iterations	Non-Indexed	Indexed	Speed-Up (Non_Indexed/Indexed)
$M_{20,4}$	400	8	1.292	0.251	5.15
$M_{35,4}$	1225	8	9.554	1.432	6.67
$M_{50,4}$	2500	8	36.793	4.146	8.87
$M_{65,4}$	4225	9	214.238	15.232	14.06
$M_{80,4}$	6400	9	588.026	30.134	19.51
$M_{100,4}$	10000	9	1538.08	54.608	28.17

Table 6.2: Running *p2* on $M_{n,m}$ with/without indexing

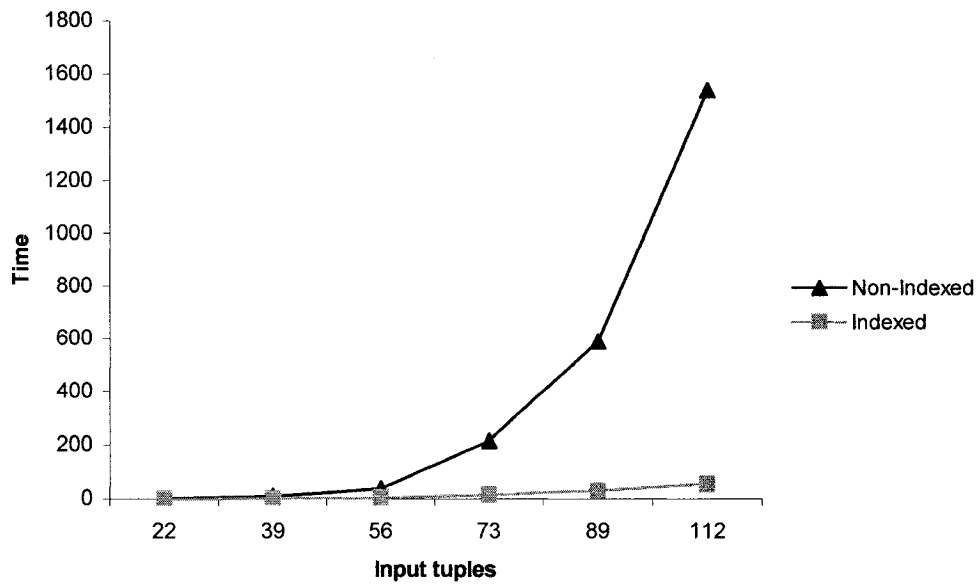


Figure 6.5: Running p2 on M with/without indexing

The indexing scheme developed improves the speed up when running *p3* on data set $T_{n,m}$, as shown by table 6.3 and Figure 6.6. The speed-up gained ranges from 1 for $T_{2,4}$ to 11.47 for $T_{6,4}$. Results also indicate that indexing may be more beneficial when the number of derivations is large. In other words, when the number of derived facts is not large, the cost of creating indices may not pay off.

Data Set Class	Output Tuples	Number of Iterations	Non-Indexed	Indexed	Speed-Up (Non_Indexed/Indexed)
$T_{2,4}$	6	10	0.01	0.01	1.00
$T_{3,4}$	22	19	0.1	0.1	1.00
$T_{4,4}$	86	25	1.683	0.661	2.55
$T_{5,4}$	342	30	53.667	8.332	6.44
$T_{6,4}$	1366	34	3114.6	271.48	11.47

Table 6.3: Running *p3* on $T_{n,m}$ with/without indexing

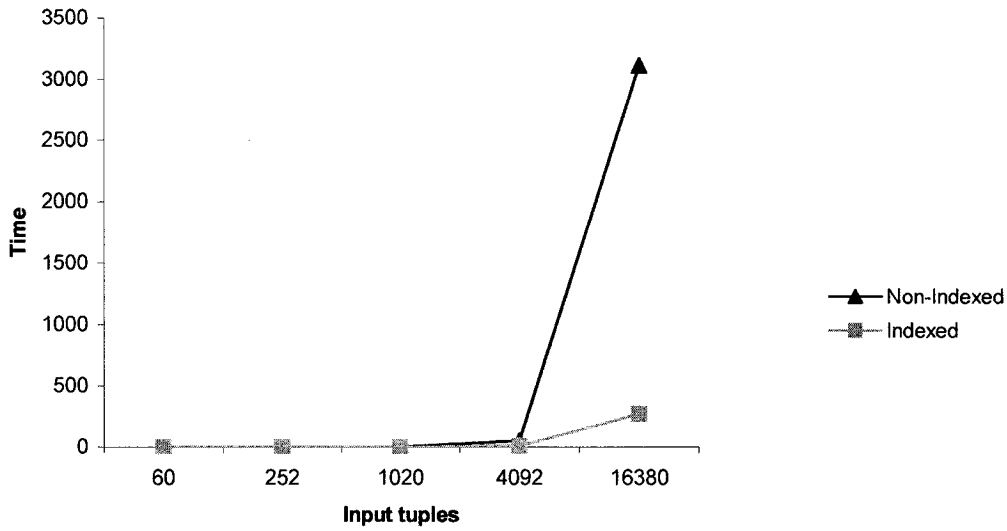


Figure 6.6: Running p3 on T with/without indexing

Looking at the tables in Appendix A, we may find that all the test cases of *p1, p2* and ***p4*** provide a consistent result for indexing technique. These results show that indexing outperforms non-indexing very much. The conducted speed-up increases when the input size increases. The property of scalability is held by our indexing technique. Furthermore, we observe that the larger number of iterations the evaluation takes, the more time it saves. The large number of iteration means the created indices are to be used for a large number of times. The high-frequent indices application brings out the large benefits of indexing. Even though the evaluation takes less iteration, the total number of derivations

remains large if a large number of IDB facts (comparing with EDB facts) are derived at each iteration. Since fact derivation may take the advantage of indexing search of facts, a large number of derivations accumulate more benefits of indexing and make the cost of indexing pay-off. In this situation, indexing may still improve the performance over non-indexing (see Table 6.2).

However, our indexing technique being simple does not always bring benefits to the evaluation. Indexing may perform worse in case the cost of its creation and maintenance is larger than what it saves (in terms of search time). The experiment result of *p3* running on some data sets (A_n, B_n, C_n and S_n, for example) demonstrated this kind of situation. As an example, Table 6.4 and Figure 6.7 show the result of *p3* running on C_n. We can see that, compared with non-indexing, evaluation with indices took more time. For evaluation of C_{128}, indexing took 223.942 seconds to complete, while non-indexing took 83.971 seconds only, about one third.

Data Set Class	Input Tuples	Output Tuples	Number of Iterations	Non-indexed		Indexed	
				Number of Joins	Time	Number of Joins	Time
C_4	76	51	5	434	0.08	8,314	0.23
C_{16}	352	268	5	2,606	1.031	69,072	1.533
C_{32}	720	499	5	5,502	5.428	281,712	7.831
C_{64}	1456	1011	5	11,294	21.17	1,137,072	42.741
C_{128}	2928	2035	5	22,878	83.971	4,568,112	223.942

Table 6.4: Running *p3* on C_n with/without indexing

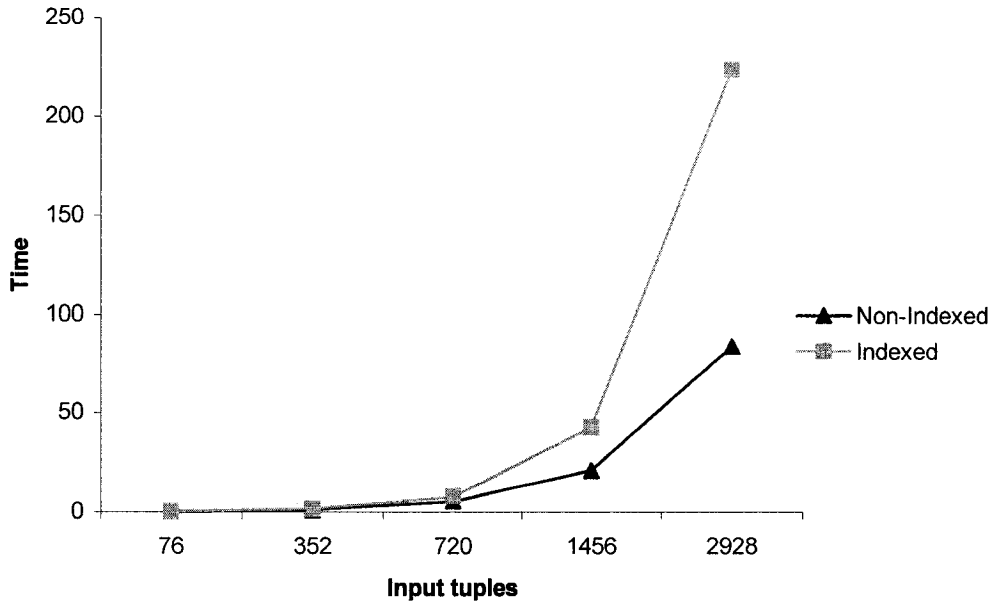


Figure 6.7: Running p3 on C with/without Indexing

The reason why our indexing technique did not result in better performance is due to our subgoal reordering scheme being primitive. Let us look at *p3* again. According to our re-ordering technique, the second rule r_2 in *p3* is reordered as follow:

$$sg(X, Y) \xleftarrow{1} flat(Z2, Z3), up(X, Z1), sg(Z1, Z2), sg(Z3, Z4), down(Z4, Y); \langle f_c, f_p, f_d \rangle.$$

Predicate "flat" was moved as the left most sub-goal in the body because of its size being the largest. Notice that the second predicate "up" has no common attributes with the first predicate. This results in performing Cartesian product at the run time, hence performing useless large joins. Since the original rule maintains some attribute(s) of each predicate bound, the number of joins is less than that of re-ordered rules and therefore evaluation without indexing would be more efficient. The experiment result in Table 6.4 confirms this analysis. If we compare the results of columns "Number of Joins" under "Indexed" and "Non-indexed", we find that the number of joins using indices is always more than

that without index. For the data set C_{128}, the total number of joins for indexing is 4,568,112, while it is only 22,878 for non-indexing.

To avoid this undesired situation, our reordering technique could be improved by guaranteeing that all body predicates in a reordered rule must have at least one bound attribute, when evaluating the subgoals from left-to-right for side-way information passing.

6.5 Semi-Naive Technique : Performance Evaluation

A main objective of our work in this research is the performance evaluation of our implementation of SN [32] and SNP methods for DDB+uncertainty. We designed and implemented the basic SN evaluation and SNP evaluation in our system and conducted extensive experiments of running *p1* to *p4* on the various data sets introduced earlier and report the results as follows. As a retrospect, Naive technique was also evaluated by the same test cases. Same as indexing evaluation, we measure the elapsed time as the performance parameter. Some of these results are presented here. The complete set of tables can be found in Appendix B. The tables are of the same format as discussed in section 6.4.

Table 6.5 and Figure 6.8 show the experimented result of running *p1* on different size of data set $M_{n,m}$. The certainty assigned to each EDB facts is 0.5. The disjunction function is *ind*, and both propagation and conjunction functions are *min*. The result shows that SNP performs better than SN, which in turn performs better than Naive method. By running *p1* on $M_{n,m}$, SN performs about 8% better than Naive because SN does not save very much of the repeated computation. If we compare 5^{th} and 6^{th} columns in Table 6.5, we can see that SNP outperforms SN by about 1.87 times for $M_{35,4}$ to 5.41 times for $M_{80,4}$.

Data Set Class	Output Tuples	Number of Iterations	Naive	Semi-Naive	Semi-Naive with partition	Speed-Up (Naive/SN)	Speed-Up (SN/SNP)
$M_{20,4}$	400	22	0.12	0.12	0.04	1.00	3.00
$M_{35,4}$	1225	37	0.791	0.711	0.381	1.11	1.87
$M_{50,4}$	2500	52	2.364	2.153	0.831	1.10	2.59
$M_{65,4}$	4225	67	5.858	5.358	1.592	1.09	3.37
$M_{80,4}$	6400	82	11.307	10.565	2.544	1.07	4.15
$M_{100,4}$	1000	102	24.235	23.043	4.256	1.05	5.41

Table 6.5: SN and SNP performance: running *p1* on $M_{n,m}$

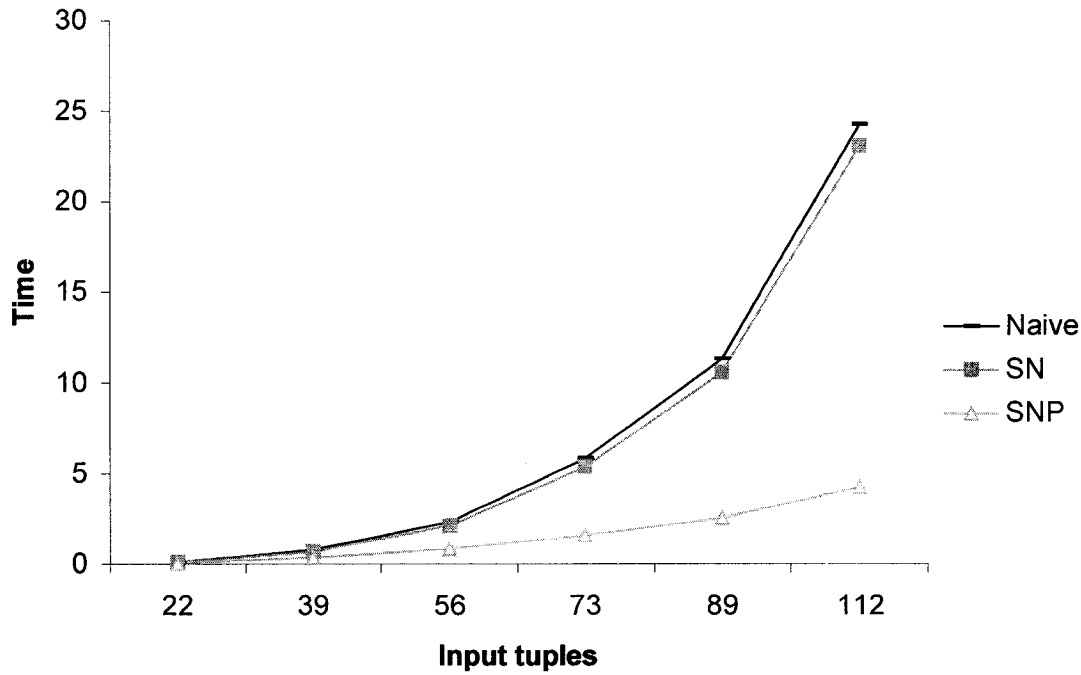


Figure 6.8: SN and SNP performance: running p1 on M

Running *p2* on CT_n gave a similar result as running *p1* on $M_{n,m}$. This result is shown in Table 6.6 and Figure 6.9. The certainty of each EDB fact is 0.5 and the certainty combination functions of conjunction, propagation, and disjunction are *, *, and *max* respectively. The experimental result indicated that SNP is about 1.5 times faster than SN, which in turn is 5% better than Naive. Table 6.6 also shows that the size of output, in terms of number of facts, is always square of the input size. If we compare the number of

iterations listed in Table 6.1, we find that under the same data set, *p1* requires more iterations to reach the fixpoint than *p2*, while the output size are same. This means that the workload for evaluating *p2*, in terms of number of joins, at each iteration is more than that of *p1*.

Data Set Class	Output Tuples	Number of Iterations	Naive	Semi-Naive	Semi-Naive with partition	Speed-Up (Naive/SN)	Speed-Up (SN/SNP)
CT_{10}	100	6	0.03	0.02	0.02	1.50	1.00
CT_{50}	2500	8	4.236	4.056	2.744	1.04	1.48
CT_{100}	10000	9	42.641	40.238	25.466	1.06	1.58
CT_{150}	22500	10	207.488	201.519	96.428	1.03	2.09

Table 6.6: SN and SNP performance: running *p2* on CT_n

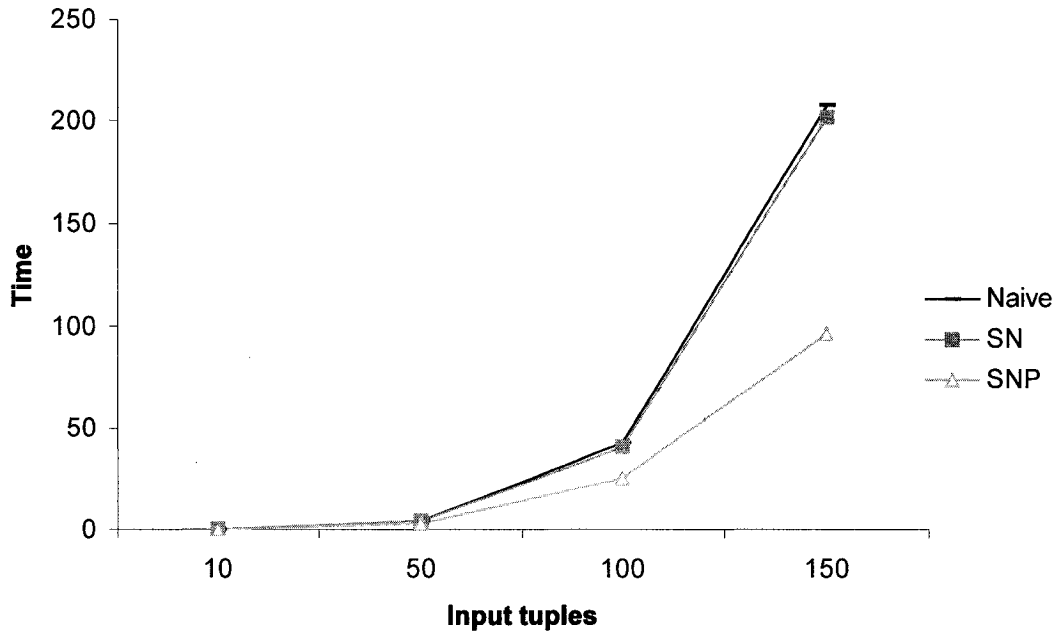


Figure 6.9: SN and SNP performance: running p2 on CT

The experimental result of running *p3* on data set A_n is shown in table 6.7 and Figure 6.10. The elapsed time of SN is about half time by Naive method. For A_9, the evaluation time used by Naive method is 560.716 seconds, while it is only 260.57 seconds for SN

method. As discussed in chapter 3, in some cases, SN cannot avoid some repeated derivations. SNP attempts to improve the situation and succeeds significantly. The speeded-up achieved by SNP over SN ranges from1.25 for A4 to 203 for A_9. The results in Table 6.7 indicate that running *p3* on A_n, there are fewer facts generated, on average, by the recursive rule at each iteration and the size of the recursive predicate is relatively large. In this situation, the repeated derivations caused by non-recursive rule are avoided by SN and hence SN has greater speeded-up over the Naive method. However, not all repeated derivations could be avoided. These repeated derivations are then defected and removed by our SNP method. Since the derivations by the recursive rule involves in many joins and hence more workload per iteration step, the speed-up of SNP over SN is higher than that of SN over Naive.

Data Set Class	Output Tuples	Number of Iterations	Naive	Semi-Naive	Semi-Naive with partition	Speed-Up (Naive/SN)	Speed-Up (SN/SNP)
A_4	44	5	0.06	0.05	0.04	1.20	1.25
A_6	188	7	2.594	1.112	0.05	2.33	22.24
A_7	380	8	15.563	6.529	0.18	2.38	36.27
A_8	764	9	96.619	41.82	0.52	2.31	80.42
A_9	1532	10	560.716	260.575	1.282	2.15	203.26

Table 6.7: SN and SNP performance: running *p3* on A_n

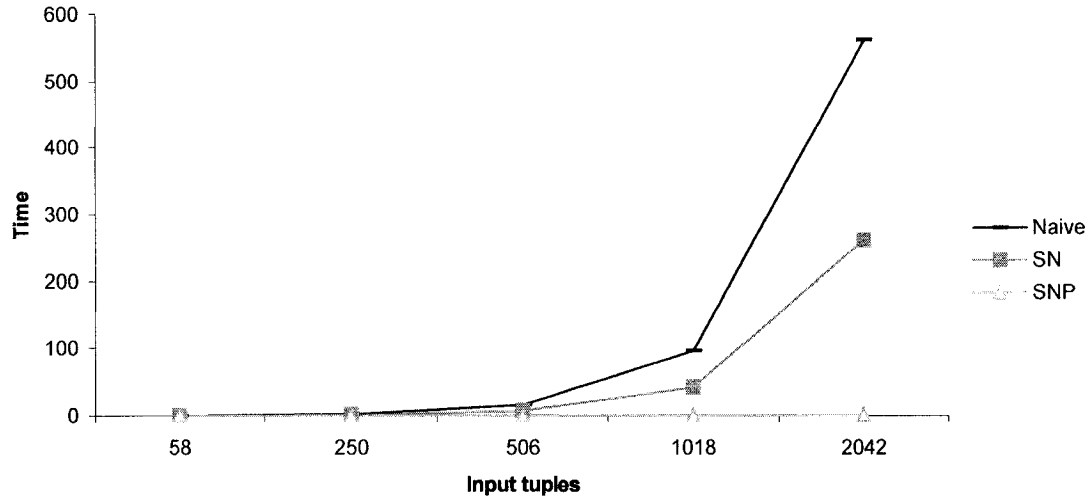


Figure 6.10: SN and SNP performance: running p3 on A

Running *p4* on S_n reveals another speed-up category. Unlike the case of running *p3* on A_n, the speedup of SN over Naive outperforms that of SNP over SN (see Table 6.8 and Figure 6.11). The higher speedup of SN over Naive is due to the feature of the input program. If we look at *p4* carefully, we see that there are always some rules in *p4* not to be evaluated at a particular iteration in SN method. This is because not every rule has a new fact for its body at every iteration. The saving of not applying such rules decreased the evaluation time. From Table 6.8, we can see that the speed-up of SN over Naive is about 29 times on data S_{96}. On the other hand, sometimes the speed-up of SNP over SN is lower than that of SN over Naive, for instance on data set S_n. Since less number of related facts could be derived during the evaluation, the amount of repeated derivations SNP could avoid is small and hence the time saved by SNP is little. However, even in such situation, SNP is at least two times faster than SN on data S_{96}.

Data Set Class	Output Tuples	Number of Iterations	Naive	Semi-Naive	Semi-Naive with partition	Speed-Up (Naive/SN)	Speed-Up (SN/SNP)
S_4	1	6	0	0	0.01	N/A	0.00
S_{16}	1	30	0.21	0.06	0.13	3.50	0.46
S_{32}	1	62	2.874	0.952	1.112	3.02	0.86
S_{64}	1	126	114.545	13.199	9.083	8.68	1.45
S_{96}	1	190	1873.464	64.092	30.484	29.23	2.10

Table 6.8: SN and SNP performance: running *p4* on Sn

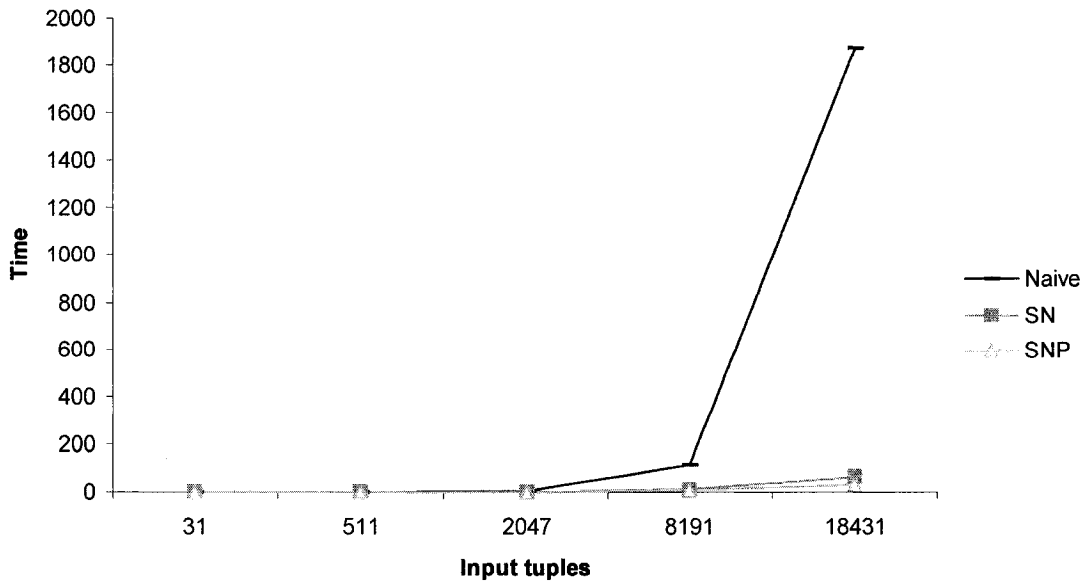


Figure 6.11: SN and SNP performance: running p4 on S

Our experiment results show that most of the SN yields faster evaluation than Naive. At least, SN in no case performed worse than Naive. Furthermore, SNP resulted in greater efficiency compared to both Naive and SN for most of test cases. Again, SNP did not perform worse than SN in any case. On the basis of our rather through experiments, we may conclude that, SNP outperforms both SN Naive methods. This conclusion also confirms to our analysis described in chapter 3.

From these experiment results, we find that the speed-ups of SN over Naive and SNP over SN may vary under different evaluation test cases. SN provided outstanding performance by running program *p3*, while SNP provided greater efficiency over SN by running program *p1*, *p2* and *p4*.

If we look at table 6.7 again, we may find that the speed-ups of SN over Naive and SNP over SN are 1.2 and 1.25 respectively, when the input data has only 4 layers. However, these speed-ups achieved range from 2.15 to 203.26, respectively, when the number of layers increases to 9. This is true for almost all test cases. As the input size of a test case increases, the speedup (both SN over Naive and SNP over SN) achieved also increases. This indicates that the larger the input data set, the higher efficiency SN and SNP provide. In other words, the proposed SN and SNP evaluation schemes are scalable.

A natural question which may arise is: "how the speedup achieved by the various evaluation schemes we proposed in our research compares with the efficiency of existing engines that could support fixpoint computation with uncertainty?" Even though our research was motivated by the fact that the Semi-Naive evaluation does not have a counterpart when uncertainty is presented, the comparison makes sense if we consider p-programs in which the certainty associated with each rule/fact is 1. This is because in this case the certainty associated with every derived atom will be 1. To answer the above question, we considered such p-programs and evaluated them in CORAL, using two features it supports, namely multisets and *aggregate-per-iteration* annotations. The latter instructs the run-time environment to apply, at every iteration, a desired aggregation function, the disjunction function in our case, to each group of identical atoms. In some cases, it took too much time that we had to terminate the execution! For *p1*, we evaluated

it on the data set CT of different sizes. The result indicated that our SNP system outperformed CORAL by a factor of 16 to 24, in all sizes.

Recall that we could not use CORAL to evaluate p-programs with certainties different than 1, since the result produced would be different to our SN (or SNP) method, a CORAL performs some short-cuts and low-level optimization that make quite sense in the standard case. In fact, this was a reason in the first place explaining why we need to build a new evaluation engine for deductive databases with uncertainty, resulting in the development and our prototype in this research. Another way we tried to answer the question was to consider evaluating, in CORAL, standard Datalog programs (without uncertainty values and combination functions) and compare the time it takes with our system, which was used to evaluate the same program but with certainties 1 and any "suitable" aggregation functions added. This resulted in CORAL to outperform our system by a factor of 1.5 to 6. The superiority of CORAL in this case was mainly due to the facts that (1) unlike our system, CORAL does not have to deal with certainty values/functions, and (2) CORAL ignores all duplicate derivations, whereas ours has to respect them in general and dealt with them in the fixpoint evaluation.

Our observation is less than conclusive, but these experiments convinced that our implementation, which does not really use sophisticated indexing and data structures is reasonably efficient, which could be further improved by adding more useful bells and whistles.

Chapter 7

Conclusion and Future Research

Our goal in this thesis was to primarily develop an efficient environment for declarative manipulation of uncertain knowledge. In this thesis, we studied evaluations of logic programs and deductive databases in the context of the parametric framework over the complete lattice [0, 1]. We assumed that arithmetic computations over real number could be carried out with arbitrary precision.

To reach our goal, we first study the applicability of classical Semi-Naive evaluation over the uncertainty computation and illustrated that the classical Semi-Naive evaluation method does not have a counterpart in logic programs with uncertainty. Our motivation in this work was then to build a bridge on this gap and proposed a desired evaluation method, which is equivalent to the Naive method. This method is called multiset-based Semi-Naive method.

For the purpose of further enhancing the evaluation efficiency, we refined multiset-based SN method by applying derivation source tracking to avoid the repeated computation during the evaluation. This SN refinement, called Semi-Naive with Partition technique actually partitions an IDB predicate into two disjoint parts: "improved" partition and "non-improved" partition. Only derivations where at least one fact-certainty pair of improved partition participates may generate "something new" and therefore should be

considered for the further evaluation. Moreover, the equivalence between SNP method and Naive method is also established.

Our efficient evaluation study is not restricted on bottom-up approach. We developed an efficient evaluation technique, called stratified evaluation, through recognizing that the intermediate certainties of non-mutually recursive subgoals do not contribute to form the final certainty of the rule head. A desired stratification is defined for the stratified evaluation. The evaluation over which may generate the maximal efficiency, in terms of avoiding intermediate certainty computations, compared to the other stratifications.

To explore the practicability of proposed query optimization techniques, a deductive database with uncertainty system has been designed and implemented. A hash-based index structure has been adopted as the base of the constructions of those query optimization techniques. We introduced a primitive indexing plan technique to reduce the number of necessary indices and hence to degrade the cost of indexing.

A bunch of experiments were conducted to verify the benefit of the proposed techniques. Our experimental results show that most of the SN yields faster evaluation than Naive. At least, SN in no case performed worse than Naive. Furthermore, SNP resulted in greater efficiency compared to both Naive and SN for most test cases. Again, SNP did not perform worse than SN in any case. The property of scalability of SN and SNP is also verified by our experimental results. The experimental results also show that the speed-ups of SN over Naive and SNP over SN strongly depend on the feature of the input program and the structure of the input data.

Until now, the study of efficient uncertainty evaluation focuses on avoiding redundant derivations during the evaluation. There is less attention paid to the cost of disk I/O. For

the classical deductive databases, many techniques have been proposed to try to minimize the amount of disk I/O by mixing up the computations from different iterations. Through these methods, the result of derivations at a particular iteration can be immediately applied for the further derivations at the same iteration. Therefore the evaluation becomes more set-oriented w.r.t. disk I/O. Unfortunately, for the evaluation of deductive database with uncertainty, it is strictly forbidden to mix up the computations from different iterations (see section 3.1). Hence, solving the problem "how to make the uncertainty evaluation more set-oriented w.r.t. the disk I/O" should be a research direction for the next step.

Another research direction about the query optimization originates from the non goal-oriented feature of the bottom-up approach. The bottom-up approach, in general, does the selection after all possible derivations, query related or unrelated, are done. To avoid the irrelevant derivation, a so-called magic set technique has been introduced by [5] to push the selection down. However, this technique cannot be borrowed in a "straight" way to the evaluation of deductive databases with uncertainty. Research is needed for "how to push the selection down while doing query processing."

References

[1] S. Abiteboul, R. Hull, V. Vianu. Foundations of Databases. *Addison-Wesley Publishing Company*, 1995.

[2] I. Balnbin and K. Ramamohanarao. A Generation of the Differential Approach to Recursive Query Evaluation. *Journal of Logic Programming*, 4(3), 1987.

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

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

[5] Catriel Beeri, R. Ramakrishnan. On the Power of Magic. *Journal of Logic Programming,* 10:255-299, 1991.

[6] Roderick Bloem, Harold N. Gabow, and Fabio Somenzi. An Algorithm for Strongly Connected Component Analysis in nlogn Symbolic Steps. *FMCAD*, 143-160, 2000.

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

[8] S. Ceri, G. Gottlob, L. Tanca. Logic Programming and Databases. *Springer-Verlag*, Berlin, 1990.

[9] Subrata Kumar Das. Deductive Databases and Logic Programming. *Addison-Wesley Publishing Company*, 1992.

[10] Marcia A. Derr, Shinichi Morishita, Geoffrey Phipps. Design and Implementation of the Glue-Nail Database System. *SIGMOD Conference,* 147-156, 1993.

[11] Didier Dubois, Jérôme Lang, Henri Prade. Towards Possibilistic Logic Programming. *International Conference on Logic Programming*, 581-595, 1991.

[12] P. Havlak. Nesting of Reducible and Irreducible Loops. *ACM Transactions on Database Systems,* 19(4):557-567, 1997.

[13] Ki-Hyung Hong, Yoon-Joon Lee, Kyu-Young Whang. Dynamically Ordered Semi-Naive Evaluation of Recursive Queries. *Information Sciences,* 96(3&4):237-269, 1997.

[14] Michael Kifer, V. S. Subrahmanian. Theory of Generalized Annotated Logic Programming and its Applications. *Journal of Logic Programming*, 12(3&4):335-367, 1992.

[15] J. Kuittinen, O. Nurmi, S. Sippu, and E. S. Soininen. Efficient Implementation of Loops in Bottom-up Evaluations of Logic Queries. In *Proc. International Conference on Very Large Data Bases Conference*, 372-379, 1990.

[16] Laks V. S. Lakshmanan. An Epistemic Foundation for Logic Programming with Uncertainty. *Foundations of Software Technology and Theoretical Computer Science*, 89-100, 1994.

[17] Laks V. S. Lakshmanan, Fereidoon Sadri. Probabilistic Deductive Databases. *Symposium on Logic Programming*, 254-268, 1994.

[18] Laks V. S. Lakshmanan, Fereidoon Sadri. Modeling Uncertainty in Deductive Databases. *Database and Expert Systems Applications*, 724-733, 1994.

[19] Laks V.S. Lakshmanan and Nematollaah Shiri. A Parametric Approach to Deductive Databases with Uncertainty. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 13:554-574, July-August 2001.

[20] Sonia M. Leach, James J. Lu. Query Processing in Annotated Logic Programming: Theory and Implementation. *J. Intell. Inf. Syst.* 6(1):33-58, 1996.

[21] J. W. Lloyd. Foundations of Logic Programming. Springer-Verlag, second edition, 1987.

[22] Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer Widom. Database Systems : The Complete Book. *Prentice Hall*, 2002.

[23] Raymond T. Ng, V. S. Subrahmanian. Probabilistic Logic Programming. *Information and Computation,* 101(2): 150-201, 1992.

[24] Raymond T. Ng, V. S. Subrahmanian. A Semantical Framework for Supporting Subjective and Conditional Probabilities in Deductive Databases. *J. Autom. Reasoning,* 10(2):191-235, 1993.

[25] Esko Nuutila, Eljas Soisalon-Soininen. On Finding the Strongly Connected Components in a Directed Graph. *Information Processing Letters,* 49(1):9-14, 1994.

[26] R. Ramakrishnan, Divesh Srivastava, S. Sudarshan. Rule Ordering in Bottom-Up Fixpoint Evaluation of Logic Programs. *IEEE Transactions on Knowledge and Data Engineering,* 6(4):501-517, 1994.

[27] R. Ramakrishnan, Divesh Srivastava, S. Sudarshan, Praveen Seshadri. The CORAL Deductive System. *VLDB Journal* 3(2):161-210, 1994.

[28] R. Ramakrishnan and J. D. Ullman. A Survey of Deductive Database Systems. *Journal* of *Logic Programming,* vol. 23, no. 2, 125-149, May 1995.

[29] Konstantinos F. Sagonas, Terrance Swift, David Scott Warren. XSB as an Efficient Deductive Database Engine. *SIGMOD Conference*, 442-453, 1994.

[30] Clifford A. Shaffer. A Practical Introduction to Data Structures and Algorithm Analysis, Second Edition. *Prentice Hall*, 2001.

[31] Nematollaah Shiri. Towards a Generalized Theory of Deductive Databases with Uncertainty. *PhD thesis, Department of Computer Science, Concordia University, Montreal, Canada*, August 1997.

[32] Nematollaah Shiri, Zhi Hong Zheng. Challenges in Fixpoint Computation with Multisets. *International Symposium on Foundations of Information and Knowledge Systems,* 273-290, 2004.

[33] R. E. Tarjan. Depth-first Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1:146-160, 1972.

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

[35] J. D. Ullman. Principles of Database and Knowledge-Base Systems volume II. *Computer Science Press*, 1988.

[36] M. H. Van Emden. Quantitative deduction and its fixpoint theory. *Journal of Logic Programming*, 4:37-53, 1986.

Appendix A – Results of Indexing Performance Testing

1. Indexing Performance Testing under program **p1**:

Data Set	Output	Iterations	Non-Indexing *	Indexing*
CT_{10}	100	71	0.19	0.07
CT_{20}	400	101	1.913	0.471
CT_{30}	900	122	7.882	1.462
CT_{40}	1600	146	23.384	3.535
CT_{50}	2500	151	53.397	6.499
CT_{60}	3600	167	130.127	11.487
CT_{70}	4900	167	260.355	16.584
CT_{80}	6400	165	439.702	22.392
CT_{90}	8100	181	797.817	32.897
CT_{100}	10000	188	1250.57	43.623
CT_{110}	12100	188	1756.59	52.705
CT_{120}	14400	188	2362.05	62.13
CT_{130}	16900	187	3033.74	72.104
CT_{140}	19600	188	3876.99	84.982
CT_{150}	22500	188	4733.2	97.701
$M_{20,4}$	400	22	0.381	0.12
$M_{35,4}$	1225	37	3.174	0.791
$M_{50,4}$	2500	52	14.37	2.364
$M_{65,4}$	4225	67	58.444	5.858
$M_{80,4}$	6400	82	183.293	11.307
$M_{100,4}$	1000	102	598.5	24.235

2. Indexing Performance Testing under program **p2**:

Data Set	Output	Iterations	Non-Indexing*	Indexing*
CT_{10}	100	9	0.191	0.05
CT_{20}	400	9	1.642	0.311
CT_{30}	900	10	9.143	1.342
CT_{40}	1600	10	25.016	3.305
CT_{50}	2500	10	62.87	7.19
CT_{60}	3600	10	161.091	13.73
CT_{70}	4900	10	354.139	21.661
CT_{80}	6400	10	658.837	32.417
CT_{90}	8100	10	1100.2	45.976
CT_{100}	10000	11	2525.49	90.34
CT_{110}	12100	11	3994.07	120.363
CT_{120}	14400	11	6153.18	158.598
CT_{130}	16900	11	9296.8	199.918
CT_{140}	19600	11	12364	252.748
CT_{150}	22500	11	16376.6	305.93
$M_{20,4}$	400	8	1.292	0.251
$M_{35,4}$	1225	8	9.554	1.432
$M_{50,4}$	2500	8	36.793	4.146
$M_{65,4}$	4225	9	214.238	15.232
$M_{80,4}$	6400	9	588.026	30.134
$M_{100,4}$	10000	9	1538.08	54.608

3. Indexing Performance Testing under program **p3**:

Data Set	Output	Iterations	Non-Indexing*	Indexing*
A_4	44	5	0.05	0.06
A_6	188	7	1.002	2.594
A_7	380	8	4.586	15.563
A_8	764	9	19.638	96.619
A_9	1532	10	77.001	560.716
B_4	64	5	0.06	0.04
B_{16}	268	5	1.021	1.412
B_{32}	540	5	4.216	8.191
B_{64}	1084	5	15.312	53.947
B_{128}	2172	5	60.407	286.412
C_4	51	5	0.08	0.23
C_{16}	268	5	1.031	1.533
C_{32}	499	5	5.428	7.831
C_{64}	1011	5	21.17	42.741
C_{128}	2035	5	83.971	223.942
F_4	17	3	0.01	0.01
F_8	165	5	0.681	0.39
F_{16}	1557	6	41.379	14.581
f_{24}	5637	7	600.684	223.392
F_{32}	13909	7	3284.13	1417.768
S_4	8	3	0	0.01
S_{16}	32	3	0.08	0.41
S_{32}	64	3	0.601	3.625
S_{64}	128	3	7.241	35.541
S_{96}	192	3	50.983	158.698
$T_{2,4}$	36	3	0.01	0.12
$T_{3,4}$	356	4	0.681	3.375
$T_{4,4}$	4132	5	44.094	339.228
$T_{5,4}$	53220	6	6672.71	94669.628
$U_{5,10}$	350	5	6.759	1.422
$U_{10,10}$	850	5	40.098	6.319
$U_{12,10}$	1050	5	60.257	9.193
$U_{15,10}$	1350	5	97.46	13.73
$U_{20,10}$	1850	5	185.347	26.619

4. Indexing Performance Testing under program **p4**:

Data Set	Output	Iterations	Non-Indexing*	Indexing*
A_4	1	52	0.1	0.06
A_6	1	220	5.488	1.301
A_7	1	444	43.723	6.069
A_8	1	892	350.765	32.517
A_9	1	1788	2815.7	169.093
B_4	3	31	0.09	0.07
B_{16}	3	53	2.874	0.641
B_{32}	3	85	18.877	2.423
B_{64}	3	149	137.448	11.226
B_{128}	3	277	1069.28	55.55
C_4	4	24	0.04	0.04
C_{16}	4	48	2.804	0.571
C_{32}	4	80	20.6	2.323
C_{64}	4	144	156.495	11.487
C_{128}	4	272	1241.47	59.396
F_4	4	11	0.01	0.01
F_8	20	20	0.361	0.12
F_{16}	96	38	24.214	1.823
F_{24}	232	56	358.625	14.151
F_{32}	432	74	2558.87	63.881
S_4	1	6	0	0
S_{16}	1	30	0.15	0.21
S_{32}	1	62	3.135	2.874
S_{64}	1	126	135.525	114.545
S_{96}	1	190	1917.03	1873.464
$T_{2,4}$	6	10	0.01	0.01
$T_{3,4}$	22	19	0.1	0.1
$T_{4,4}$	86	25	1.683	0.661
$T_{5,4}$	342	30	53.667	8.332
$U_{5,10}$	10	30	3.861	0.761
$U_{10,10}$	10	30	31.576	3.155
$U_{12,10}$	10	34	69.76	5.788
$U_{15,10}$	10	40	164.446	10.455
$U_{20,10}$	10	50	457.708	21.781

**Timing in seconds*

Appendix B – Results of SNs Performance Testing

1. SNs Performance Testing under program **p1**:

Data Set	Output	Iterations	Naive *	Semi-Naive*	Semi-Naive with Partition*
CT_{10}	100	71	0.07	0.01	0.03
CT_{20}	400	101	0.471	0.44	0.16
CT_{30}	900	122	1.462	1.351	0.501
CT_{40}	1600	146	3.535	3.345	0.951
CT_{50}	2500	151	6.499	6.089	1.533
CT_{60}	3600	167	11.487	10.856	2.373
CT_{70}	4900	167	16.584	16.714	3.285
CT_{80}	6400	165	22.392	21.411	4.176
CT_{90}	8100	181	32.897	31.345	5.428
CT_{100}	10000	188	43.623	41.97	6.77
CT_{110}	12100	188	52.705	50.382	8.128
CT_{120}	14400	188	62.13	59.736	9.684
CT_{130}	16900	187	72.104	68.979	11.336
CT_{140}	19600	188	84.982	82.358	13.389
CT_{150}	22500	188	97.701	94.836	14.761
$M_{20,4}$	400	22	0.12	0.12	0.04
$M_{35,4}$	1225	37	0.791	0.711	0.381
$M_{50,4}$	2500	52	2.364	2.153	0.831
$M_{65,4}$	4225	67	5.858	5.358	1.592
$M_{80,4}$	6400	82	11.307	10.565	2.544
$M_{100,4}$	1000	102	24.235	23.043	4.256

2. SNs Performance Testing under program **p2**:

Data Set	Output	Iterations	Naive *	Semi-Naive*	Semi-Naive with Partition*
CT_{10}	100	9	0.05	0.05	0.06
CT_{20}	400	9	0.311	0.301	0.3
CT_{30}	900	10	1.342	1.231	1.312
CT_{40}	1600	10	3.305	3.074	2.915
CT_{50}	2500	10	7.19	6.719	6.019
CT_{60}	3600	10	13.73	12.298	11.917
CT_{70}	4900	10	21.661	20.149	19.598
CT_{80}	6400	10	32.417	30.164	29.151
CT_{90}	8100	10	45.976	43.643	41.229
CT_{100}	10000	11	90.34	84.472	63.772
CT_{110}	12100	11	120.363	113.974	103.268
CT_{120}	14400	11	158.598	150.426	154.342
CT_{130}	16900	11	199.918	196.112	186.939
CT_{140}	19600	11	252.748	241.396	230.201
CT_{150}	22500	11	305.93	296.626	284.419
$M_{20,4}$	400	8	0.251	0.25	0.21
$M_{35,4}$	1225	8	1.432	1.322	1.272
$M_{50,4}$	2500	8	4.146	3.966	3.946
$M_{65,4}$	4225	9	15.232	13.439	9.684
$M_{80,4}$	6400	9	30.134	27.179	19.999
$M_{100,4}$	1000	9	54.608	51	44.444

3. SNs Performance Testing under program **p3**:

Data Set	Output	Iterations	Naive*	Semi-Naive*	Semi-Naive with Partition*
A_4	44	5	0.06	0.05	0.04
A_6	188	7	2.594	1.112	0.05
A_7	380	8	15.563	6.529	0.18
A_8	764	9	96.619	41.82	0.52
A_9	1532	10	560.716	260.575	1.282
B_4	64	5	0.04	0.02	0.02
B_{16}	268	5	1.412	0.10	0.1
B_{32}	540	5	8.191	0.36	0.341
B_{64}	1084	5	53.947	0.88	0.851
B_{128}	2172	5	286.412	2.02	2.042
C_4	51	5	0.271	0.05	0.12
C_{16}	268	5	4.407	0.12	0.091
C_{32}	499	5	24.645	0.36	0.331
C_{64}	1011	5	147.562	0.931	0.822
C_{128}	2035	5	850.693	2.253	1.863
F_4	17	3	0.01	0.01	0
F_8	165	5	0.39	0.08	0.06
F_{16}	1557	6	14.581	1.923	1.522
F_{24}	5637	7	223.392	15.373	11.336
F_{32}	13909	7	1417.768	51.213	43.843
S_4	8	3	0.01	0	0
S_{16}	32	3	0.461	0.01	0
S_{32}	64	3	3.916	0.05	0.02
S_{64}	128	3	37.263	0.711	0.06
S_{96}	192	3	164.046	5.488	0.18
$T_{2,4}$	36	3	0.12	0.01	0.01
$T_{3,4}$	356	4	3.375	0.12	0.05
$T_{4,4}$	4132	5	339.228	2.814	2.443
$T_{5,4}$	53220	6	94669.628	113.644	82.008
$U_{5,10}$	350	5	1.422	0.992	0.962
$U_{10,10}$	850	5	6.319	3.715	3.766
$U_{12,10}$	1050	5	9.193	4.707	4.767
$U_{15,10}$	1350	5	13.73	6.199	6.159
$U_{20,10}$	1850	5	26.619	9.043	8.573

4. SNs Performance Testing under program **p4**:

Data Set	Output	Iterations	Naive*	Semi-Naive*	Semi-Naive with Partition*
A_4	1	52	0.06	0.02	0.05
A_6	1	220	1.301	0.441	0.381
A_7	1	444	6.069	1.982	1.332
A_8	1	892	32.517	9.103	5.107
A_9	1	1788	169.093	42.371	20.609
B_4	3	31	0.07	0.06	0.03
B_{16}	3	53	0.641	0.551	0.471
B_{32}	3	85	2.423	2.083	1.803
B_{64}	3	149	11.226	9.273	7.31
B_{128}	3	277	55.55	42.191	29.934
C_4	4	24	0.04	0.081	0.03
C_{16}	4	48	0.571	0.511	0.471
C_{32}	4	80	2.323	2.113	1.892
C_{64}	4	144	11.487	9.784	7.981
C_{128}	4	272	59.396	47.068	33.388
F_4	4	11	0.01	0.03	0.01
F_8	20	20	0.12	0.12	0.08
F_{16}	96	38	1.823	1.672	0.932
F_{24}	232	56	14.151	13.169	4.747
F_{32}	432	74	63.881	57.613	13.95
S_4	1	6	0	0	0.01
S_{16}	1	30	0.21	0.06	0.13
S_{32}	1	62	2.874	0.952	1.112
S_{64}	1	126	114.545	13.199	9.083
S_{96}	1	190	1873.464	64.092	30.484
$T_{2,4}$	6	10	0.01	0.05	0.01
$T_{3,4}$	22	19	0.1	0.081	0.06
$T_{4,4}$	86	25	0.661	0.41	0.431
$T_{5,4}$	342	30	8.332	4.105	3.415
$U_{5,10}$	10	30	0.761	0.711	0.49
$U_{10,10}$	10	30	3.155	2.964	2.364
$U_{12,10}$	10	34	5.788	5.458	3.525
$U_{15,10}$	10	40	10.455	10.435	5.187
$U_{20,10}$	10	50	21.781	21.731	8.432

**Timing in seconds*