Query Processing and Optimization in Deductive Databases with Certainty Constraints

Jinzan Lai

A thesis

in

The Department

of

Computer Science & Software Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of Master of Computer Science at

Concordia University

Montreal, Quebec, Canada

December 2008

© Jinzan Lai, 2009

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-63321-2
Our file Notre référence
ISBN: 978-0-494-63321-2

NOTICE:

The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

ABSTRACT

Query Processing and Optimization in Deductive Databases with Certainty Constraints

Jinzan Lai

Uncertainty reasoning has been identified as an important and challenging issue in the database research presented in the Lowell report [ea05]. Many logic frameworks have been proposed to represent and reason about uncertainty in deductive databases. Based on the way in which uncertainties are associated with the facts and rules in programs, these frameworks have been classified into:"annotation based" (AB) and "implication based" (IB). [Shi05] has investigated the relative expressive powers of AB and IB frameworks and has introduced the notion of certainty constraints, which makes them equivalent in terms of expressive power. Due to this equivalence, we developed transformation algorithms operating between AB and IB frameworks.

With presence of certainty constraints in rule bodies in logic programs, query processing and optimizations become more complicated. The bottom-up query evaluation algorithms Naive, Semi-Naive, and Semi-Naive with Partition in parametric framework [SZ04, SZ08] do not consider certainty constraints. We extend these algorithms by incorporating a new checker module and develop extended evaluation algorithms which deal with certainty constraints. We have developed the proposed techniques and conducted many experiments to measure efficiency. Our results and benchmarks indicate that the proposed techniques and strategies yield a useful and efficient evaluation engine for deductive databases with certainty constraints.

Acknowledgments

I express my deep gratitude and respect to my supervisor Prof. Nematollaah Shiri, whose wisdom, knowledge and experiences shape numerous insightful conversations, without which many ideas in this research would have not been well developed. His support, guidance and patience helped me to make this thesis possible.

My father Chuanzu Lai, my mother Meixue Wang and my sister Niwang Lai deserve my special thanks and affections. Their boundless love and dedication have always been the inspiration throughout my life. To them I dedicate this thesis.

I am grateful to Qiong Huang for his many discussions and suggestions in this thesis. I would also like to thank my friends Jiewen Wu, Heng Keng and Shu Zhang for their helpful comments in this thesis.

I would like to express my gratitude to the faculty members and staff in our department for their teaching, research seminars and services. I also wish to express my sincere thanks to my colleagues Ahmed Alasoud, Jocelyne Faddoul, Nasim Farsinia for providing a stimulating and fun working environment.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Uncertainty pervades life and can arise from many sources. It is present in most tasks that require intelligent behavior, such as planning, reasoning, problem solving, decision making, classification and many others dealing with real world entities and data. The following are some typical examples of uncertainty applications in real life:

- **Bank risk analysis:** based on a client's age, financial conditions, and credit history, banks estimate the risk of loan to the client.
- **Medical diagnosis:** estimation of multiple parameters of different nature varying from linguistic information to images and from sociological knowledge to signal data.
- **Weather forecasting:** uses particular weather information including temperature, humidity, wind speed, and cloud ceiling and distribution to predict short and long term weather.
- **HIV vaccine modeling:** Microsoft researchers use machine learning techniques to model uncertain information for HIV vaccine.

Consequently, management of uncertain data is central to the development of computer based systems that can successfully execute these tasks. In order to deal with uncertainty, we need to be able to represent it and reason about it efficiently. Uncertainty reasoning has been identified as an important database research direction in Lowell report [ea05], which is an assessment of the state of the database research as well as a prediction concerning what problems and problem areas deserve additional focus.

Ideas from AI and databases merged to give birth to deductive databases technology, a confluence of virtues of AI and DB technologies which was intended [GKT91] in the area of uncertainty reasoning. Deductive databases are developed by combining logic programming with relational databases to manage and handle large amount of data efficiently.

1.1 Motivation

During the last two decades, there has been numerous research and development on uncertainty reasoning, which resulted in a number of frameworks being proposed. Classical logic database programming, with its advantages of modularity and its powerful top-down and bottom-up query processing techniques, has been a primary choice for incorporating uncertainty. These frameworks vary in different ways including the way in which uncertainty values are associated with the facts and rules in the programs. Based on this, [LS96] classified the approaches of these frameworks into "annotation based" (AB) and "implication based" (IB).

AB and IB frameworks seem somewhat orthogonal and unrelated. They may have different mathematical foundation of uncertainty and use different combination functions. There are many research conducted on them independently, but the relation between AB and IB frameworks has not been completely studied. In [Shi05], the

author investigated the relation between these two frameworks and introduced the notion of *certainty constraints*. It builds a bridge between these two frameworks. The purpose of this notion has two fold: first, it supports the operations of selection and join by certainty, which are often useful in query formulation and processing in the context of uncertainty; second, it "relates" the expressive power of the two approaches and provides a basis to establish their equivalence.

When AB and IB frameworks are allowed to extend certainty constraints in the rule body, they become equally expressive. This is an important result "connecting" IB and AB frameworks. We should point out that AB and IB frameworks refer to two families of frameworks, rather than two specific languages. The parametric framework [LS96] is a generic IB framework that unifies and generalizes all the IB frameworks. It is thus a good choice as the representative of IB frameworks. Motivated by the equivalence built by certainty constraints, the objective of our work was to provide a uniform environment to evaluate and experiment with logic programs in AB and IB frameworks at the same time. This uniform evaluation scheme is also useful towards developing tools for uncertainty reasoning.

There have been some implementations of logic frameworks with uncertainty. In [LL96], Leach and Lu implemented a top-down query processing system containing ca-resolution for annotated logic programming which uses set as semantics structure. For set-based IB frameworks, a top-down implementation was introduced in [LS96] to evaluate programs in parametric framework on top of the XSB system [SSW94]. CORAL [RSSS94] has some capability to support uncertainty reasoning, but is limited to fuzzy logic. In [SZ04], a fragment of parametric framework was developed, which supports the multiset-based reasoning system to evaluate parametric programs efficiently in a bottom-up fashion. In this research, we extend logic programs with certainty constraints. However, query processing is more complicated in our context

due to the presence of certainty constraints in the rule bodies and no existing query processing can handle certainty constraints. We proposed a new one by reusing the existing query processing techniques from the context of parametric framework [LS96] and incorporating them with certainty constraints checker.

1.2 Contributions of The Thesis

The main contributions of this research are summarized as follows:

1. We develop a unified query processing scheme which can evaluate any AB and IB frameworks with certainty constraints at the same time. In addition, we develop the transformation between AB and IB frameworks (Section 3.4)

2. We extend query processing to handle deductive databases with certainty constraints in Chapter 4. In addition, we develop a stratification technique which will further improve the efficiency for some programs (Section 6.3.2).

3. We develop a prototype system (Chapter 5), called AILOG, and study its performance. For this, we created a number of test cases, conducted numerous experiments. The experiments and results are reported in Chapter 7.

1.3 Thesis Outline

The rest of this thesis is organized as follows. In Chapter 2, we provide background knowledge about logic frameworks with uncertainty. This includes a classification of logic frameworks with uncertainty and a review of parametric frameworks. Some existing implementations of AB and IB frameworks are also discussed in this chapter. Stratified evaluation of programs datalog with negation is presented in section 2.4.

In Chapter 3, we review the concepts of certainty constraints, and study extended generic IB framework (EGIB) and extended generic AB framework (EGAB). We also present the transformation from the EGAB program to the EGIB program, and vice versa.

In Chapter 4, we study performance and efficiency query processing for programs with certainty constraints. We propose an algorithm to deal with the Certainty Constraints and incorporate it different evaluation methods, e.g., Naive, Semi-Naive and Semi-Naive with Partition algorithms.

Our system prototype AILOG is introduced in Chapters 5 and 6. Chapter 5 gives the system architecture and describes the interconnection among its different modules: Chapter 6 discusses the implementation details. We specially discuss the issues: (1) data representation,(2) relation representation, and (3) evaluation techniques.

In Chapter 7, we report details of our experiments to study the efficiency of different query processing techniques. We report the experimental results together with analysis which provide insight to the proposed query processing scheme to handle certainty constraints. Chapter 8 provides concluding remarks and discusses possible future research directions.

Chapter 2

Background and Related Work

In this chapter, we review basics of logic programming and deductive databases with uncertainty, and introduce the concepts and notations we use in our work. We begin with a review of the *parametric framework* [LS96], which is a generic IB framework that unifies and/or generalizes the IB frameworks. We also review an implementation of the parametric framework [SZ04] and its query optimization techniques. We assume the reader is familiar with the foundations of logic programming [Llo87] and deductive databases [CGT89], such as rules, facts, EDB and IDB predicates, model and fixpoint theories.

Logic database programming, with its declarative and modular advantages, and with its powerful top-down and bottom-up query processing techniques, has been a primary choice for modeling uncertainty. Numerous frameworks have been proposed by extending the standard logic programming and deductive databases with uncertainty. As in the standard case, these frameworks offer declarative semantics of programs, and are supported by a sound and complete (or sometimes weakly complete) proof theory and corresponding fixpoint semantics.

These frameworks differ in several ways. In terms of mathematical bases, these frameworks combine deduction with different forms of uncertainty, including fuzzy,

probabilities, possibilities, hybrid of numeric and symbolic formalism. These frameworks may also differ in the way in which uncertainties are associated with the facts and rules in a program and the way in which they manipulate uncertainties. On the basis of reporting uncertainty in the programs, these frameworks are classified into *annotation based* (AB, for short) and *implication based* (IB, for short) [LS96]. Examples of the AB frameworks include annotated logic programming of Subrahmanian [Sub87], Kifer and Li [KL88], probabilistic logic programming of Ng and Subrahmanian [NS92, NS93], and the generalized theory of annotated logic programming (GAP) proposed by Kifer and Subrahmanian [KS92], Examples of IB frameworks include van Emden [vE86a], Fitting [Fit88, Fit91], Dubois et al. [DLP91], Lakshmanan and Sadri [LS94b, LS94a], Lakshmanan and Shiri [LS96]. For a comprehensive comparison of these approaches, please refer to [LS96, LS01a]. Next we review the basis of IB and AB frameworks from [LS01a].

2.1 Implication Based Approach

In the IB approach, a rule is an expression of the form:

$$r: \quad A \xleftarrow{\alpha} B_1, \ldots, B_n$$

where A and B_i's are atomic formulas of the form $p(X_1, \ldots, X_n)$, each X_i is either a variable or a constant symbol, and α is a value in the range [0,1] indicating the certainty of the rule. As in the standard case, when n=0, we refer to r as a *fact*. Intuitively, this rule asserts that "the certainty that the conjunction in the rule body implies the head is α." In a sense, α controls the "propagation" of truth from the rule body to the head. Van Emden [vE86b] was the first to propose a framework, based on fuzzy logic, in which deduction is combined with certainty. Rule evaluation

in Van Emden's framework is as follows. For this, we use the notion of valuation v which is basically a function that associates with each ground atom a certainty value. To evaluate a rule, we first determine the certainty of the rule body as a whole obtained by using the minimum of the certainties of B_i's which is given by a valuation v, which maps a ground atom to a certainty value in $[0, 1]$. The certainty of the body so obtained is then multiplied ($\times$) by the rule certainty α, which generates a certainty for the atom A derived by the rule. Alternative derivations of the same ground atom A are combined into a single certainty. This is the basic rule evaluation in any IB framework, but the certainty domain and/or certainty functions could be different in different frameworks. There is a triplet of certainty functions $\langle f_d, f_p, f_c \rangle$ involved in an evaluation of logic programs with uncertainty, in which f_d is the disjunction function, f_p is the propagation function, and f_c is the conjunction function. The order of functions in this triplet also indicates, from right to left, the order in which these functions are applied when evaluating a rule, as described above. The collection of these functions in a program is referred to as *"certainty functions."* As in the Van Emden's language, if all the rules in a program have the same triplet of disjunction, propagation, and conjunction functions, we do not need to explicitly show combination functions with every rule for ease of presentation. In addition to these three types of combination functions, the uncertainty domain is the fourth parameter describing or distinguishing among different frameworks.

2.2 Annotation Based Framework

In the AB approach, a rule is an expression of the form:

$$r: \quad A : f(\beta_1, \ldots, \beta_n) \leftarrow B_1 : \beta_1, \ldots, B_n : \beta_n.$$

In this approach, the implication in a rule is like the standard case, but each atom in the rule is associated with a certainty value or certainty variable. Symbol f in the rule head is a computable n-ary function, which combines the role of conjunction and propagation functions in IB frameworks, and β_is are annotation constants or variables ranging over an appropriate certainty domain. This rule asserts that "the certainty of A is at least(or is in) $f(\beta_1, \ldots, \beta_n)$, where the certainty of B_i is at least (or is in) β_i, for $1 \leq i \leq n$." Alternative derivations of the same atom from a program are "combined" using a disjunction function associated with the head predicate.

It has been discussed that the AB approach is more expressive in general than the IB approach to some extent, because a limited relationship between two annotations can be expressed in the AB approach. For instance, a rule in an AB framework could use annotation constants or use repeated annotation variables. Through variables, the AB approach can simulate the IB approach.

Even though there have been numerous frameworks proposed for AB frameworks, there has been little progress in their effective and efficient implementation. In [LL96], Leach and Lu discuss technical and implementation issues in the context of an AB framework with set-based semantics. A top-down query processing, containing elements of constraint solving called ca-resolution, was developed for annotated logic programming (ALPs). A computer implementation of ca-resolution for ALPs was built, and many query optimization techniques were developed and/or used to improve its efficiency.

2.3 Parametric Framework

In this section, we review the *parametric framework* [LS96], an IB framework, which unifies and/or generalizes all the IB frameworks to uncertainty. By "tuning" the parameters of the *parametric framework* appropriately, it can simulate the computation

of any known IB framework.

Definition 2.3.1. *A parametric program (p-program) P is a 5-tuple $(\mathcal{T}, \mathcal{R}, \mathcal{D}, \mathcal{P}, \mathcal{C})$, whose components are defined as follows:*

- *$(\mathcal{T}, \preceq, \otimes, \oplus)$ is assumed to be a complete lattice, where $\mathcal{T}$ is a set of truth 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 is denoted by $\top$.*

- *$\mathcal{R}$ is a finite set of parametric rules (p-rules), each of which is a statement of the form:*

$$r: \quad A \xleftarrow{\alpha} B_1, \ldots, B_n; \langle f_d, f_p, f_c \rangle$$

  where A, $B_1, \ldots, B_n$ are atoms, and $\alpha \in T - \{\perp\}$ is the certainty of r.

- *$\mathcal{D}$ is a mapping from the predicate symbol s in the program P to the collection $\mathcal{F}_d$ of disjunction functions, which associates with each predicate symbol in P a disjunction function f_d.*

- *$\mathcal{P}$ is a mapping from the rules in the program P to the collection $\mathcal{F}_p$ of propagation functions, which associates with each p-rule in P a propagation function f_p.*

- *$\mathcal{C}$ is a mapping from the rules in the program P to the collection $\mathcal{F}_c$ of conjunction functions, which associates with each p-rule in P a conjunction function f_c.*

Atom A in the p-rule r above is called the rule *head*, and the conjunction $B_1, \ldots, B_n$ is called the rule *body*. A *fact* is a special case of a p-rule in which n=0s. We use $\pi(A)$ to denote the predicate symbol of atom A, and use $Disj(\pi(A))$ to denote the disjunction function associated with $\pi(A)$. We use rules and p-rules interchangeably.

Let $\mathcal{L}$ be an arbitrary first order language that contains infinitely many variable symbols, finitely many predicates and constants, and no function symbols. While $\mathcal{L}$ does not contain function symbols, it contains symbols for the families of disjunction $\mathcal{F}_d$, conjunction $\mathcal{F}_c$, and propagation $\mathcal{F}_p$ functions, to all of which we collectively refer as "combination functions" denoted by $\mathcal{F} = \mathcal{F}_d \cup \mathcal{F}_p \cup \mathcal{F}_c$. We refer to each element in $\mathcal{F}$ as a *combination function.* In order to make derivations in the parametric framework meaningful, some conditions are assumed to hold on the combination functions in $\mathcal{F}$. We will introduce or list these conditions in general and indicate for which family of functions they are assumed to hold.

1. *Monotonicity:* $f(\alpha_1, \alpha_2) \preceq f(\beta_1, \beta_2)$, whenever $\alpha_i \preceq \beta_i$, for i=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. That is, the result of f cannot be "more" than any one of its arguments.

4. $Bounded - Below : f(\alpha_1, \alpha_2) \succeq \alpha_i$, for i=1,2. That is, the result of f cannot be "less" than any one of its arguments.

5. $Commutativity : f(\alpha, \beta) = f(\beta, \alpha), \forall \alpha, \beta \in \mathcal{T}$.

6. $Associativity : f(\alpha, f(\beta, \gamma)) = f(f(\alpha, \beta), \gamma), \forall \alpha, \beta, \gamma \in \mathcal{T}$.

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

8. $f(\dot{\emptyset}) = \bot$, where $\bot$ is the least element in $\mathcal{T}$, and $\dot{\emptyset}$ denote the empty multiset.

9. $f(\dot{\emptyset}) = \top$, where $\top$ is the greatest element in $\mathcal{T}$.

10. $f(\alpha, \top) = \alpha, \forall \alpha \in \mathcal{T}$.

11. $f(\alpha, \beta) \succ \bot, \forall \alpha, \beta \succ \bot$.

Let $\mathcal{T}$ be a certainty lattice and $\mathcal{B}(\mathcal{T})$ be the set of finite multiset over $\mathcal{T}$. Then a disjunction or conjunction function is a mapping from $\mathcal{B}(\mathcal{T})$ to $\mathcal{T}$, a propagation function is a mapping from $\mathcal{T} \times \mathcal{T}$ to $\mathcal{T}$.

Postulate 1. *Every disjunction function in $\mathcal{F}_d$ should satisfy properties 1, 2, 4, 5, 6, 7, and 8.*

Postulate 2. *Every propagation function in $\mathcal{F}_p$ should satisfy properties 1, 2, 3, 10, and 11.*

Postulate 3. *Every conjunction function in $\mathcal{F}_c$ should satisfy properties 1, 2, 3, 5, 6, 7, 9, 10, and 11.*

In [LS01a], the family of disjunction functions in $\mathcal{F}_d$ are classified into three, called types 1 to 3, defined as follows.

Definition 2.3.2. (Types of Disjunction Functions) Let $f_d \in \mathcal{F}_d$ be a disjunction function in the parametric framework. Then we say:

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

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

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

Note the difference between types 2 and 3. Whenever α and β are different from $\top$ and $\bot$, the certainty value returned by a type 2 disjunction function is always better when supplied with "better" argument values, while a type 3 disjunction function may return $\top$ for some such values. For example, the probability independence function, defined on $ind(\alpha, \beta) = \alpha + \beta - \alpha\beta$ is type 2, whereas $f = min(\alpha + \beta, \top)$ is of type 3. As shown in [LS01a], The presence of type 2 or type 3 disjunction functions may cause a fixpoint evaluation of p-program not to terminate in finite time.

A restricted top-down and bottom-up implementations of the parametric framework have been introduced in [SV97]. The top-down implementation was built on top of the XSB system [SSW94]–a powerful logic programming system. The bottom up implementation mentioned uses CORAL, which is a deductive system with many useful features and directives, such as multiset and aggregation annotations, through which a user can "influence" the run-time environment and evaluation process. These two prototype systems easily support type 1 disjunction functions, however more work is required to develop systems which support manipulation of multisets in the context of the parametric framework with type 2 and type 3 disjunction functions.

Another implementation of a fragment of the parametric framework is [SZ04], which extends the standard bottom-up evaluation to allow multisets. This resulted in multiset-based Semi-Naive and Semi-Naive with Partition over the certainty domain $[0, 1]$.

2.4 Stratified Evaluation

In this section, we discuss the stratification technique which was originally proposed to evaluate standard Datalog programs with negation (denoted by *Datalog*$^\neg$). It computes the *perfect model* of such programs.

To increase the expressive power of *pure Datalog*, several extensions of *pure Datalog* have been proposed in the literature, including incorporation of negation in the so called *Datalog*$^\neg$. Such programs may have more than one minimal Herbrand model, which explains the source of problems for the semantics of *Datalog*$^\neg$ programs: this raises the question which minimal Herbrand model should be chosen as the *perfect model* of the program. One way to compute this perfect model is stratification when possible, which approximates the Close World Assumption (CWA). Stratified

evaluation is based on predicate dependency graph induced by a program. The program evaluation then proceeds stratum-by-stratum in the order of lower stratum first [CGL86]. Figure 2.1 shows a *Datalog*$^{\neg}$ program and Figure 2.2 shows its predicate dependency graph.

$$r_1 : P \leftarrow Q, \neg R.$$
$$r_2 : R \leftarrow S, A.$$
$$r_3 : S \leftarrow R, B.$$
$$r_4 : C \leftarrow A, B.$$
$$r_5 : Q \leftarrow C.$$

Figure 2.1: Program $P_{2.1}$

Definition 2.4.1. Let P be a *Datalog*$^{\neg}$ program. A predicate dependency graph obtained from P is a directed graph $pdg(P)$, in which vertices are predicates in P and edges are defined as follows:

- The pdg(P) has a positive edges (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
- The pdg(P) has 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

Not all *Datalog*$^{\neg}$ programs are stratified. A *Datalog*$^{\neg}$ program P is stratified *iff* its predicate dependency graph does not contain a cycle involving a negative edge. If P is stratified, then there is a procedure construct a stratification of P. Any stratified program P can be partitioned into disjoint sets of clauses $P = P^1 \cup \ldots \cup P^i \cup \ldots \cup P^n$, where P^i is called a stratum. Each IDB predicate of P has its defining rules within one stratum. Each stratum of P^i contains only clauses whose negative literals correspond

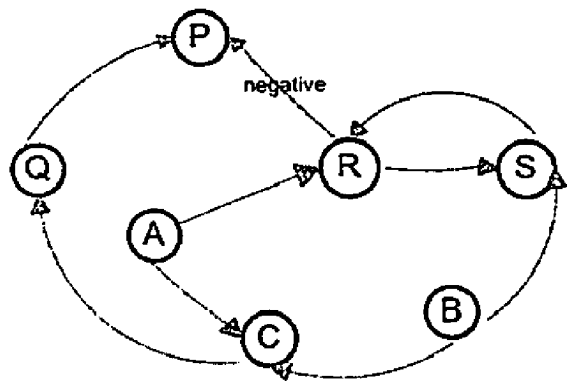

Figure 2.2: The dependency graph of program $P_{2.1}$

to predicates defined in a lower stratum. The partition of P into $P^1, \ldots, P^n$ is called a stratification of P.

Definition 2.4.2. 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, g) is in pdg(P), then $m(r_p) \geq m(r_q)$

2. if a negative edge (p, g) is in pdg(P), then $m(r_p) > m(r_q)$

Where r_p is a rule in P whose head is predicate P.

A valid stratified evaluation always evaluates a negative subgoal in a rule defining P before defining P. Note that a stratified program may have several different stratifications. For example, $P_{2.1}$ has the following stratification: $P_1 = \{r2, r3\}$, $P_2 = \{r1, r4, r5\}$, where $P_{2.1} = P_1 \cup P_2$. Another alternative stratification of $P_{2.1}$ is $P_1 = \{r4\}$, $P_2 = \{r2, r3, r5\}$, $P_3 = \{r1\}$, where $P_{2.1} = P_1 \cup P_2 \cup P_3$. All these stratifications are equivalent [ABW88], i.e, the result of evaluation of a stratified *Datalog*$^\neg$ is independent of the stratification used. Many algorithms have been proposed to find a desired stratification of a given program in *Datalog*$^\neg$ [CGT90, Ull89].

Chapter 3

Deductive Databases with Certainty Constraints

Numerous logic frameworks have been proposed for management of uncertainty in logic programming and deductive databases. As previously mentioned, these frameworks vary in various ways including the way uncertainties are associated with the facts and the rules in programs. On the basis of this, these frameworks are classified into *annotation based* (AB) and *implication based* (IB) frameworks, with relative pros and cons [LS01a]. Another way to compare these two approaches is their relative expressive power. This was studied in [Shi05], in which it was shown that certainty constraints in a key notion that connects and relates the two approaches which were otherwise considered as orthogonal and unrelated. This chapter provides details and reviews the results on expressive power.

3.1 Certainty Constraints

A useful operation in deductive databases and logic programming with uncertainty is to select from a relation, every tuple whose associated certainty value is not "less"

than some specified threshold, e.g., as in the query "find all red objects whose degree of redness is at least 0.75." This operation is called *select by certainty.* Another useful operation is *join by certainty*, which amounts to prescribing that a pair of tuples from two relations can be joined provided their associated certainties stand in a certain relationship. These *select/join by certainty* operations can be viewed as a filtering mechanism, using which we can determine a (possibly empty) subset of tuples which may contribute to the final answer, when processing a query.

A certainty constraint in an IB framework is a built-in predicate of the following forms:

- $\omega t(A)\ \theta\ \sigma$
- $\omega t(A)\ \theta\ \omega t(B)$

where A and B are atoms, θ is a comparison operator in $\{\prec, \preceq, =, \neq, \succ, \succeq\}$, $\omega t(A)$ is a special predicate *weight* indicating certainty of A, and σ is value in the certainty domain.

A certainty constraint is a conjunction of one or more such expressions above. We may also consider $\omega t(A)$ as a function call, returning the current certainty associated with A during the query processing. We assume throughout this thesis that σ always appears on the right side of θ.

In order for the comparison of the expressive power of IB frameworks and AB frameworks to be fair and justified, we also allow certainty constraints in AB frameworks. A certainty constraint in an AB framework is of the following forms:

- $V_i\ \theta\ \sigma$
- $V_i\ \theta\ V_j$

where V_i is a *certainty constant* or *certainty variable*, θ is a comparison operator as before, and σ is a certainty value in $\mathcal{T}$.

A permissible program P must satisfy the following *safety conditions*:

- Each fact is ground.
- Each variable that occurs in a rule head must also occur in the rule body.
- A predicate that occurs in a certainty constraint in a rule must also occur as a subgoal in the body of the same rule.

We will use *CC* as an abbreviation for Certainty Constraints.

3.2 Extended Parametric Framework (EGIB)

As mentioned earlier, when we compare the expressive power of IB and AB frameworks, we they refer to two families of frameworks, rather than two specific frameworks. Since the parametric framework [LS96] unifies and/or generalizes the IB frameworks, it was taken as the representative of all IB frameworks, the generic IB framework (GIB), and extended with certainty constraints [Shi05]. We refer to the extended generic IB framework as EGIB.

An EGIB rule (i–rule, for short) in the EGIB framework is an expression of the form:

$$r : p(\overline{X}) \stackrel{a}{\leftarrow} q_1(\overline{Y}_1), \ldots, q_n(\overline{Y}_n), C_r; \langle f_d, f_p, f_c \rangle.$$

where C_r is a (possible empty) conjunction of IB Certainty Constraints. The definitions of rule head, rule body and combination function is the same as in the parametric framework. For consistency reason, we require that all the i-rules in an EGIB program with the same head predicate to be associated with the same disjunction function.

As in [LS01a] a declarative semantic of EGIB programs is defined on the notions of satisfaction of EGIB programs by valuations.

Definition 3.2.1. (Valuation) A valuation ν of an EGIB program is a mapping from the Herbrand base B_p of P to the certainty domain $\mathcal{T}$, which assigns each ground atom A in B_p a certainty value in $\mathcal{T}$, that is, $\nu(A) \in \mathcal{T}$.

Definition 3.2.2. (CC-satisfaction) Let $C_r \equiv C_1, \ldots, C_k$ be the conjunction of CC specified in the body of a rule $r \in P$, where $C_l, 1 \leq l \leq k$, is either a certainty constraint of the form (1) $wt(q_i(\overline{Y_i}))\ \theta\ \sigma$, or (2) $wt(q_i(\overline{Y_i}))\ \theta\ wt(q_j(\overline{Y_j}))$, where θ is a comparison operator, and $\sigma \in \mathcal{T}$. Then we say:

(a) ν satisfies C_l, denoted $\models_\nu C_l$, iff $C_l \equiv \nu(B_i)\ \theta\ \sigma$ is true in case (1), and $C_l \equiv \nu(B_i)\ \theta\ \nu(B_j)$ is true in case (2), where B_i and B_j are ground instances of subgoals $q_i(\overline{Y_i})$ and $q_i(\overline{Y_i})$, respectively.

(b) $\models_\nu C_r$ iff $\models_\nu C_l$, for all $l, 1 \leq l \leq k$. That is, ν satisfies C_r iff ν satisfies every certainty constraint C_l in C_r.

We use the definitions of CC-satisfaction and satisfaction of p-program to formalize programs satisfaction in EGIB framework.

Definition 3.2.3. (Satisfaction) Let P be any EGIB program,
$r \equiv (p(\overline{X}) \stackrel{\alpha}{\leftarrow} q_1(\overline{Y}_1), \ldots, q_n(\overline{Y}_n), C_r; \langle f_d, f_p, f_c, \rangle)$
be any $i-$rule in P, and ν be any valuation of P. We use P^* to denote the ground instantiation of P. Let $\rho \equiv A \stackrel{\alpha}{\leftarrow} B_1, \ldots, B_n, C_r; \langle f_d, f_p, f_c \rangle \in P^*$
be any ground instance of r. Let $C_r \equiv C_1, \ldots, C_k$ be the conjunction of CC specified in the body of $r \in P$. Then,

(a) ν satisfies ρ, denoted $\models_\nu \rho$, iff $\nu(A) \succeq f_p(\alpha_r, f_c(\{|\nu(B_1), \ldots, \nu(B_n)|\}))$ and $\models_\nu C_r$.

(b) $\models_\nu r$ iff ν satisfies every ground instance of r.

(c) ν satisfies P, denoted $\models_\nu P$, iff (1) $\forall r \in P : \models_\nu r$, and (2) $\forall$ A$\in B_p : \nu(A) \succeq f_d(X)$, where X=$\{|f_p(\alpha, f_c(\{|\nu(B_1), \ldots, \nu(B_n)|\}))|(A \xleftarrow{\alpha} B_1, \ldots, B_n, C_r;$ $(f_d, f_p, f_c,)) \in P^*$ and $\models_\nu C_r|\}$.

We point out that unlike in standard logic programming and deductive database, a valuation ν which satisfies every $i-$rule r in P is not guaranteed to satisfy P itself [KL88, LS01a]. In order for ν to also satisfy P, condition c(2) above must also be satisfied, i.e, ν satisfies P if for every each atom $A \in B_p$, the certainty assigned to A by ν is not less than $f_d(X)$, where X is the multiset of certainties associated with A derived from all ground instances of ρ in P^* such that$\models_\nu \rho$ and whose head is A.

Example 3.2.1. Consider a medical application where uncertain knowledge about particular diseases and symptoms are represented as the following EGIB rules, in which we assume the triplet of certainty functions associated with each rule is "$\langle max, pro, min \rangle$".

r_1: *disease(X,D)* $\xleftarrow{0.8}$ *has(X,S), symptom(D,S), wt(has(X,S))*$\succeq$ *0.8,*
wt(symptom(D,S))$\succeq$ *0.9;*$\langle max, pro, min \rangle$
r_2: *disease(X,D)* $\xleftarrow{0.9}$ *family_history(X,D), hereditary(D),*
wt(family_history(X,D))$\succeq$ *0.8,*
wt(hereditary(D))$\succeq$ *0.7;*$\langle max, pro, min \rangle$.

The use of certainty constraints can be viewed as a filtering mechanism. The certainties that do not satisfy the constraint will be automatically filtered. For instance, r_1 determines the likelihood that a person X has disease D. It asserts that if X has a symptom S with a certainty of at least 0.8, then S is correlated with D with a certainty of at least 0.9. Note that this also results in filtering out conclusions with "insignificant" certainties.

3.3 Extended Generic AB Framework (EGAB)

To compare AB and IB approaches, we need a suitable, comparable representative of AB frameworks. However there is no such framework. For this, we consider the proposed generic AB framework (GAB) in [Shi05], which does not correspond exactly to any existing AB framework, but includes essential features of the AB approach. In order for the comparison of GAB and GIB to make sense, we extend the GAB framework to use multiset as its underlying semantic structure, as the GIB framework. Moreover, the combination functions allowed in the GAB framework are required to have properties as in the GIB framework described in Section 2, such as monotonicity, continuity, associativity, an commutativity, etc. The annotation functions allowed in the GAB framework satisfy the corresponding postulates on the combination functions in the GIB parametric framework.

Next we extend the GAB framework further to allow CCs in the rule bodies. Let us call the result as EGAB framework. An $a-$rule r in the EGAB framework is an expression of the form:

$$r : p(\overline{Y}) : f(V_1, \ldots, V_n) \leftarrow q_1(\overline{Y_1}) : V_1, \ldots, q_n(\overline{Y_n}) : V_n, C_r$$

where annotation V_i is either *certainty constant* or a *certainty variable*, and for $1 \leq i \leq n$, $q_i(\overline{Y_i})$ is an atom. All variables (object or annotation) appearing in an $a-$rule are implicitly universally quantified. The constraints C_r is a conjunction of boolean expressions of the form $V_i\ \theta\ V_j$ or $V_i\ \theta\ \sigma$, where θ is a comparison operator in $\{\prec, \preceq, =, \neq, \succ, \succeq\}$, and σ is a certainty value in $\mathcal{T}$. Note that the annotation function f used in the rule head of an AB framework (including GAB) plays the roles of both conjunction and propagation functions. Therefore, we replace the annotation function $f(V_1, \ldots, V_n)$ in the rule head with $f_p(\alpha_r, f_c(V_1, \ldots, V_n))$.

In an EGAB program [Shi05], disjunction functions are not explicitly indicated. In order to be compatible with the EGIB framework, we explicitly indicate the disjunction functions in the EGAB program. Therefore an EGAB rule ($a-$rule, for short) after this modification is an expression of the form:

$$r : p(\overline{Y}) : f_p(\alpha_r, f_c(V_1, \ldots, V_n)), f_d \leftarrow q_1(\overline{Y_1}) : V_1, \ldots, q_n(\overline{Y_n}) : V_n, C_r$$

Note that the EGAB framework so defined is strictly more expressive than the GAB framework, and hence more expressive than any existing AB framework. This is because in EGAB, we can express any comparison relation between two annotations, while in existing AB frameworks, relationship between annotations can only be expressed by using annotation constants or by repeated annotation variables, which introduces equality "=" on certainty variables. To see why this is more expressive, suppose we want to perform a join operation on relation q and r for tuples $t_q \in q$ and $t_r \in r$, such that the certainty of t_q is not "less" (w.r.t. the ordering $\preceq$ on $\mathcal{T}$) than the certainty of t_r. While this cannot be expressed in any AB framework, including the basic GAB, it can be expressed as $a-$rule, as follows:

$$p(\overline{X}) : f_p(\alpha_r, f_c(V_1, V_2)), f_d \leftarrow q(\overline{X}) : V_1, r(\overline{X}) : V_2, V_1 \geq V_2.$$

3.4 Equivalence of EGIB and EGAB Frameworks

In this section, we review the results from [Shi05] which mainly establish the equivalence of the EGAB and EGIB frameworks. We use D to denote the extensional database, which is a collection of input atom-certainty pairs. Given an (EGIB or EGAB) program P and a collection D of facts, we use $P(D)$ to denote the set of

atom-certainty pairs derived by applying P to D. Using the bottom-up Naive evaluation method, we have the following results [Shi05].

Propositon 3.4.1. Given any program P_A in the EGAB framework, there exists a program P_I in EGIB such that P_A and P_I produce the same atom–certainty pairs on every input database D. That is, $P_A(D)$=$P_I(D)$, for every database instance D.

This is established by showing how an $a-$rule in EGAB is transformed into an $i-$rule in EGIB, and vice versa. We first describe the process to transform an $a-$rule r in the EGAB framework into a $i-$rule r' in the EGIB framework. Any $a-$rule r in P_A is of the following form:

$$r : p(\overline{Y}) : f_p(\alpha_r, f_c(V_1, \ldots, V_n)) \leftarrow q_1(\overline{Y_1}) : V_1, \ldots, q_n(\overline{Y_n}) : V_n, C_r$$

As a normalization step, we replace every annotation constant V_j in r with a certainty variable V_j', and add the certainty constraint $V_j' = V_j$ to C_r. For the constraints C_r, every annotation variable V_i in C_r is replaced with $wt(q_i(\overline{Y_i}))$, for $1 \leq i \leq n$. This yields C_r', to be used in r'. The combination functions of r' and the rule certainty for r' are extracted from the annotation associated with the head of r. From left to right, f_p is the propagation function in r', α_r is the rule certainty α_r in r', and f_c is the conjunction function in r'. Using the above method, the transformed $i-$rule r' is as follows:

$$r' : \quad p(\overline{X}) \xleftarrow{\alpha_r} q_1(\overline{Y}_1), \ldots, q_n(\overline{Y}_n), C_r'; \langle f_d, f_p, f_c \rangle.$$

Next, we describe the transformation from an $i-$rule s in the EGIB framework to an $a-$rule s' in the EGAB framework. Consider the following input $i-$rule s:

$$s : \quad p(\overline{X}) \xleftarrow{\alpha_r} q_1(\overline{Y}_1), \ldots, q_n(\overline{Y}_n), C_s; \langle f_d, f_p, f_c \rangle.$$

Corresponding to this $i-$rule, we have the following $a-$rule s' in the EGAB framework:

$$s': \quad p((\overline{Y})) : f_p(\alpha_s, f_c(V_1, \ldots, V_n)) \leftarrow q_1(\overline{Y_1}) : V_1, \ldots, q_n(\overline{Y_n}) : V_n, C'_s$$

where C'_s are the certainty constraints C_s in which the weight term $wt(q_i(\overline{Y_i}))$ is replaced by annotation variable V_i, for $1 \leq i \leq n$.

The resulting rule s' obtained through the above transformation procedure is equivalent to the given rule s. That is, s and s' derive the same atom-certainty pairs on any database instance. This equivalence at the rule level implies equivalence at the program level, since both rules define the same atom certainty pairs and use the same disjunction function. The following example illustrates the transformation method.

Following the proposed transformation method, P_A is transformed into the EGIB program P_I, assuming that in P_A, max is the disjunction function associated with the predicate *disease*. Note that we can also view this as the transformation from P_I to P_A.

Example 3.4.1. Consider the followingEGAB program P_A and its transformed EGIB program P_I.

r_1: *disease(X,D): pro(0.8, min(V_1, V_2))* $\stackrel{a}{\leftarrow}$ *has(X, S):*V_1, *symptom(D, S):*$V_2, V_1 \geq 0.8, V_2 \geq 0.9$.

r_2: *disease(X,D): pro(0.9, min(V_1, V_2))* $\stackrel{a}{\leftarrow}$ *family_history(X,D):* V_1, *hereditary(D):*$V_2, V_1 \geq 0.8, V_2 \geq 0.7$.

r_1': $disease(X,D) \overset{0.8}{\leftarrow} has(X,S),\ symptom(D,S),\ wt(has(X,S)) \geq 0.8,$
$wt(symptom(D,S)) \geq 0.9; \langle max, pro, min \rangle$
r_2': $disease(X,D) \overset{0.9}{\leftarrow} family\_history(X,D),\ hereditary(D),$
$wt(family\_history(X,D)) \geq 0.8,$
$wt(hereditary(D)) \geq 0.7; \langle max, pro, min \rangle.$

3.5 Advantages and Disadvantages: A Discussion

In this section, we will discuss the advantages and disadvantages when incorporating certainty constraints within logic programs: first, it breaks the continuity of the fixpoint operator T_p of the EGIB framework, while increasing its expressive power; second, it reduces the cost of the program evaluation; third, it changes the termination behavior of the program evaluation.

3.5.1 Continuity and Expressive Tradeoff

Adding certainty constraints strictly increases the expressive power of GIB and GAB frameworks. This addition also provides a common ground for comparing these two frameworks. When certainty constraints are added, it has been shown that EGAB and EGIB have the same expressive power [Shi05]. However, the increased expressive power of the GIB framework obtained by adding certainty constraints comes at a price: the continuity of the fixpoint operator T_p of the EGIB framework is lost. This is consistent with the fact that T_p is not continuous in general for the AB frameworks. The following example illustrates this point.

Example 3.5.1. Consider the following EGIB program P, in which the underlying

certainty lattice is $\mathcal{T} = [0, 1]$.

$$r_1 : p(X, Y) \xleftarrow{1} e(X, Y); \langle ind, pro, min \rangle.$$

$$r_2 : p(X, Y) \xleftarrow{1} e(X, Z), p(Z, Y); \langle ind, pro, min \rangle.$$

$$r_3 : r(X, Y) \xleftarrow{1} p(X, Y), q(X, Y), wt(p(X, Y)) \geq 1; \langle ind, pro, min \rangle.$$

where $ind(\alpha, \beta) = \alpha + \beta - \alpha\beta$ is the disjunction function associated with predicate p and r. Suppose the EDB is:

$$D = \{e(1, 1) : 0.5, e(1, 2) : 0.5, q(1, 2) : 1\}$$

Using a fixpoint evaluation technique, the certainty associated with r(1,2) is 0, obtained at the limit ω. That is, $T_p^{\omega}((1, 2)) = 0$. However, we can easily see that $T_p^{\omega+1}((1, 2)) = 1$, indicating that the fixpoint operator T_p is not continuous – a desired property that is lost.

3.5.2 Reducing the cost of evaluation

If a certainty constraint enforces restrictions on rule deductions, the compilation of certainty constraints, together with the corresponding rules, reduces the cost of query evaluation. This can be easily shown by noting the effectiveness of the compilation of certainty constraints with its corresponding deduction rule(s) if the rule set is nonrecursive or is a bounded recursion. We examine the case of linear recursions. Consider such a rule below:

$$r(X) \xleftarrow{\alpha_r} p(Y), r(Z); \langle f_d, f_p, f_c \rangle.$$

where X, Y, and Z are arguments, and p is an EDB predicate. Suppose we add the certainty constraints $C \equiv C_1, \ldots, C_k$ to the rule body, which yields:

$$r(X) \stackrel{\alpha_r}{\leftarrow} p(Y), r(Z), C_1, \ldots, C_k; \langle f_d, f_p, f_c \rangle.$$

This addition of constraints C form a conjunction with the rule body. Since C restricts derivation of the rule, if they are evaluated together with the normal subgoal, stronger certainty constraints are enforced on the rule, which reduces the number of tuples we may set for r in every iteration, and, thus reducing the cost of query evaluation.

The case of a non-linear recursive rule can be deduced similarly.

3.5.3 Termination

Another impact of adding constraints to rule bodies is on termination and complexity properties. Evaluation terminates in some programs in the EGIB framework, while the corresponding programs without certainty constraints may not terminate. This is illustrated in the following example.

Consider an EGIB program that includes rules r_1 and r_2 defined in Example 3.5.1. The fixpoint evaluation of this program does not terminate on the input set D, defined in Example 3.5.1. Now consider the EGIB program Q in which rule r_2 is replaced with r_2' below:

$$r_2' : p(X, Y) \stackrel{1}{\leftarrow} e(X, Z), p(Z, Y), wt(e(X, Z)) \geq 1, wt(p(Z, Y)) \geq 1; \langle ind, pro, min \rangle.$$

That is, Q=$\{r_1, r_2'\}$. The fixpoint evaluation of Q on D terminates in two iterations. This is because the first certainty constraint in r_2' ensures that this rule never fires, since $wt(e(X, Z)) \geq 1$ will only be satisfied at step ω, when the fixpoint is reached.

Chapter 4

Evaluation of Deductive Databases with Certainty Constraints

Numerous query processing and optimization techniques have been proposed in standard logic programming and deductive databases, and implemented in many existing systems. Please refer to [CGT89] for a detailed description of these techniques. These techniques are often set-based and, hence, duplicate derivations of the same atom do not affect the query results in the standard case. However, one of the major sources of inefficiency in the bottom-up Naive evaluation method is duplicate derivations of atoms after their first derivation. The conventional bottom-up Semi-Naive technique was designed for reducing such redundancy. Several versions of this method can be found in the literature [Ban86, CGL86].

However, when extending deductive databases to incorporate uncertainty, it may be crucial for correctness to collect derivations as multisets, since otherwise it might result in an incorrect final model. In [LS96], the authors developed a Naive evaluation with multiset as the semantic structure in the context of the parametric framework, which collects "all" derivations as a multiset, to which a user defined disjunction function is then applied at the end of each iteration. Furthermore, considering the

source of inefficiency in Naive evaluation, they also proposed a multiset-based Semi-Naive (SN) evaluation technique for parametric programs, which was implemented in [SZ04]. They also proposed a refinement of the SN evaluation, called Semi-Naive with Partition (SNP), which further improves the performance.

When the parametric framework is further extended with certainty constraints, the aforementioned Naive, Semi-Naive, and Semi-Naive with Partition methods should also be extended to handle certainty constraints. A desired evaluation scheme should be applicable in both cases when certainty constraints are present or absent.

In this chapter, we review basis of multiset-based evaluation technique N, SN and SNP in the parametric framework in section 4.1. We then discuss some details of implementation of the Certainty Constraint Checker (CC-Checker, for short) in section 4.2. In section 4.3, we discuss two strategies of embedding CC-Checker into multiset-based N, SN and SNP evaluation techniques.

4.1 Query Processing in the Parametric Framework

When uncertainty is present in deductive databases, existing inference systems, such as CORAL and XSB, are limited to non-recursive programs, or recursive programs with type 1 disjunction functions associated with recursive predicates. Unlike type 1 disjunction functions, type 2 disjunction functions are sensitive to duplicates. When atom–certainty pairs are collected as a set S, multiple occurrences of the same atom–certainty pair $(A : \alpha)$ will only be counted once, and, hence, the combined certainty of A, after applying the corresponding disjunction function will be α. For example, if the disjunction function is *ind*, then $ind\ (\alpha, \alpha) = 2\alpha - \alpha^2 \neq \alpha$. Thus, the derived atom-certainty pairs need to be collected as a multiset rather than a set.

Certainty constraints are treated as built-in predicates, so evaluation of the EGIB rule consists of two parts. The first part focuses on regular subgoals evaluation, which is done in the same as evaluation of p-programs. The second part focuses on evaluating certainty constraints. We use the evaluation method proposed for p-program to evaluate regular predicates and implement a module to check the certainty constraints. In the following section, we briefly discuss the bottom-up evaluation methods in the parametric framework, and then extend them for our EGIB framework.

4.1.1 A Multiset–based Naive Algorithm

The basic bottom-up fixpoint evaluation in the standard datalog is called Naive evaluation. By considering the presence of certainties, the multiset-based Naive algorithm [LS96] was obtained by a "straightforward" extension of the standard case. The steps are shown in Algorithm 1.

Algorithm 1 Multiset-based Naive Algorithm [LS96]

Input: P, D
// P is a parametric-program, a set of p-rules
// D is the input instance database of atom-certainty pairs as the EDB.
Output: $lfp(T_{P\cup D})$

1: **for all** $A \in B_P$ **do**
2: $\quad \nu_0 := \bot$
3: $\quad M_1 := \{|\alpha|(A:\alpha) \in D|\}$
4: $\quad \nu_1(A) := f_d(M_1(A))$, where $f_d := Disj(\pi(A))$
5: **end for**
6: $newset_1 := \{A|(A:\alpha) \in D\}; i := 1;$
7: **while** $(newset_i \neq \varnothing)$ **do**
8: $\quad i := i + 1$
9: $\quad$ **for all** $(r : A \xleftarrow{\alpha_r} B_1, \ldots, B_n, C_r; \langle f_d, f_p, f_c\rangle) \in P^*$; **do**
10: $\quad\quad M_i(A) := \{|f_p(\alpha_r, f_c(\{|\nu_{i-1}(B_1), \ldots, \nu_{i-1}(B_n)|\}))|\};$
11: $\quad\quad \nu_i(A) := f_d(M_i(A))$,where $f_d := Disj(\pi(A))$
12: $\quad$ **end for**
13: $\quad newset_i := \{A|A \in B_p, \nu_i(A) \succ \nu_{i-1}(A)\};$
14: **end while**
15: $lfp(T_{P\cup D}) := \nu_i$

Initially, every atom is assigned the least certainty value, $\perp$, i.e, every atom is initially assumed to be false. In each following iteration, all facts and rules are applied, and the derived atom-certainty pairs are collected as a multiset. Note that when firing a rule, the best certainty found in the previous iteration of each subgoal in the rule body is used to compute the certainty of the rule head. The multiset of certainties of each atom A, shown as $M_i(A)$, derived at iteration i are combined into a single certainty for A. More precisely, $T_p(\nu)(A) = f_d(M_i(A))$, where f_d is the disjunction function associated with the predicate A, and $M_i(A)$ is a multiset of certainties such that, $M_i(A) = \{|f_p(\alpha_r, f_c(\{|\nu(B_1), \ldots, \nu(B_n)|\}))|(A \xleftarrow{\alpha_r} B_1, \ldots, B_n; \langle f_d, f_p, f_c\rangle) \in P^*|\}$. The bottom-up evaluation of T_p is then defined as in the standard case. It was also shown that T_p is monotone and continuous for any p-program P, and that the least fixpoint of T_p, denoted $lfp(T_p)$, is equivalent to the declarative semantics of P. This evaluation process continues until some iterations in which no atom is derived with a "better" certainty [LS01a].

4.1.2 A Multiset–based Semi–Naive Algorithm

As shown in [LS96], since some disjunction functions are sensitive to duplicates, a "straightforward" extension of the Semi–Naive evaluation from the standard deductive database will not be suitable for evaluating p–programs. The following example illustrates what may go wrong.

Steps 10 and 11 in the Naive evaluation above is a source of the inefficiency of the evaluation, similar to its counterpart in standard deductive databases; an atom derived at iteration i will continue to be derived in every subsequent iteration. This redundant computation also includes atoms whose certainties did not change in the previous iteration. To improve this situation, we need to identify and fire only the rules which have "new" subgoals in the body, i.e., the subgoal is "new derivations"

either a new fact or a fact which was derived before but its certainty was improved in the previous iteration. Firing rules will then be restricted to only those which have something new in the rule body. This helps minimize useless computation, and hence leads to increased efficiency of evaluation [SZ04]. This multiset-based, Semi-Naive evaluation is presented in Algorithm 2, to which we refer as SN. In this algorithm,

Algorithm 2 A Multiset-based Semi-Naive Evaluation [SZ04]

Input: P,D
// P is a parametric-program
// D is the set of atom-certainty pairs.
Output: $lfp(T_{P\cup D})$

1: **for all** $A \in B_P$ **do**
2: $\nu_0 := \bot$
3: $M_1 := \{|\alpha|(A:\alpha) \in D|\}$
4: $\nu_1(A) := f_d(M_1(A))$,where $f_d := Disj(\pi(A))$
5: **end for**
6: $newset_1 := \{A|(A:\alpha) \in D\}; i := 1;$
7: **while** $(newset_i \neq \varnothing)$ **do**
8: $i := i + 1$
9: **for all** $A \in B_P$: **do**
10: **if** $\exists(r : A \xleftarrow{\alpha_r} B_1,\ldots,B_n;(f_d,f_p,f_c)) \in P^*$
such that $\exists B_j \in newset_i$, for some $j \in \{1,\ldots,n\}$:
then
11: $M_i(A) := M_{i-1}(A)$
12: **for all** $(r : A \xleftarrow{\alpha_r} B_1,\ldots,B_n;(f_d,f_p,f_c)) \in P^*$
such that $\exists B_j \in New_i, forsomej \in \{1,\ldots,n\}$: **do**
13: $M_i(A) := M_i(A) - \{|\sigma^r_{i-1}(A)|\} \bigcup \{|\sigma^r_i(A)|\}$,where
14: $\sigma^r_i(A) := f_p(\alpha_r, f_c(\{|\nu_{i-1}(B_1),\ldots,\nu_{i-1}(B_n)|\}))$
15: **end for**
16: $\nu_i(A) := f_d(M_i(A)), where f_d := Disj(\pi(A))$
17: **else**
18: $\nu_i(A) := \nu_{i-1}(A);$
19: **end if**
20: **end for**
21: $newset_i := \{A|A \in B_p, \nu_i(A) \succ \nu_{i-1}(A)\}$
22: **end while**
23: $lfp(\Gamma_{P\bigcup D}) := \nu_i$

every ground atom A is associated with a pair (M_i, σ_i), where M_i is a multiset which

includes all certainties of atom A derived so far from different rules. Each element in $M_i(A)$ is of the form $(r : \alpha)$, indicating a derivation of atom A with certainty α from rule r. It also includes different derivations of the same atom through the same rule and within the same iteration. If the SN algorithm does not take into account these multiple derivations, and collects them as a set, it will lead to wrong results. The value σ_i is the certainty of A obtained by aggregating the certainties in M_i. If in the next iteration, we apply a rule r which uses A as a subgoal in the body, σ_i is used as the certainty of A in deriving the certainty of the rule head.

At the beginning, each ground atom is associated with a pair $\langle\dot{\varnothing},\perp\rangle$. Then, the certainties of some atoms will be improved by the EDB facts. This is done in lines 1 to 4. The "new" atom–certainty pair (a new atom or an old atom with a "better" certainty compared to the last iteration) will be recorded in set $newset_i$. This is initially done in line 6, and updated in subsequent iterations in line 21. Every rule which has at least one improved subgoal (included in $newset_i$) will be selected and fired. This is done in line 10. At step $i+1$, we remove from M_{i+1} all atom–certainty pairs $(A : \alpha)$ derived by r, and replace them by new atom–certainty pair $(A : \beta)$ derived by r. In line 16, we apply the disjunction function $Disj(\pi(A))$ associated with the predicate symbol of A to obtain a single certainty for A. Rules that do not have anything new as a subgoal in the body will not be applied new derivations, and, hence their old derivations are preserved (line 18). After applicable rules are fired at iteration i, the atoms with improved certainties will be identified and added to set $newset_i$ (line 12). If this set is empty, then the evaluation terminates and the result ν_i will be returned, as shown in line 23. Otherwise, the execution proceeds to the next iteration.

There is a *correctness requirement* as follows. *If disjunction functions of type 2 and/or type 3 are presented in a p–program, derivations by the same rule should not be*

combined across different iterations. This is also required in our context of certainty constraints. Note that combining derivations across iterations is an optimization that, for example, the Coral system takes advantage of, which also explains the need for new evaluation systems in our context.

4.1.3 A Multiset-based SN with Partition Algorithm

As discussed in the previous section, SN evaluation is more efficient compared to Naive method since it avoids the computation of some derivations that do not yield "better" certainties. More accurately, the algorithm only fires those rules which have at least one "new" subgoal obtained in the last iteration. However, not all redundant computations can be avoided by the SN evaluation. For example, if there are several derivations in a rule r, it is possible that only some of these derivations by r which have improved subgoal(s). But for correctness, the proposed SN algorithm removes "all" the derivations by r whose uncertainties have not improved. Therefore, some unnecessary computations are repeated. The following example explains this point. Let r be a rule in a p-program P:

$$r : p(X,Y) \stackrel{\alpha}{\leftarrow} q(X,Z), t(Z,Y); \langle f_d, f_p, f_c \rangle.$$

where p and q are IDB predicates, and t is an EDB predicate. Suppose there are three instances of r which generates three derivations d_1, d_2, and d_3 of atom p(1,2) at iteration i, as shown below, with certainty values β_1, β_2, and β_3, respectively.

$$d_1 : p(1,2) \stackrel{1}{\leftarrow} q(1,3), t(3,2); \langle f_d, f_p f_c \rangle.$$
$$d_2 : p(1,2) \stackrel{1}{\leftarrow} q(1,4), t(4,2); \langle f_d, f_p, f_c \rangle.$$
$$d_3 : p(1,2) \stackrel{1}{\leftarrow} q(1,5), t(5,2); \langle f_d, f_p, f_c \rangle.$$

In the SN algorithm described above, the multiset associated with p(1,2) at iteration i would be $M_i(p(1,2)) = \{|r : \beta_1, r : \beta_2, r : \beta_3|\}$. Suppose that at the end of iteration i, only the certainty of q(1,5) is improved, say from δ to γ. When the evaluation proceeds to the next iteration, all these three derivations by r in the SN algorithm are removed from $M_i(p(1,2))$, since q is used in r. Clearly evaluation d_3 again may yield something new, but not d_1 and d_2. Based on this observation, the Semi-Naive with Partition algorithm (SNP, for short) was proposed to eliminate this redundant re-evaluation of d_1 and d_2 in SN. It partitions every IDB relation, e.g. p and q, into two parts: the "improved" part and the "non-improved" part. The improved part contains all atom–certainty pairs that are generated or improved in the last iteration, while the non-improved part contains the rest of atom-certainty pairs. In the SNP algorithm, every ground fact A is associated with a pair $\langle C_i, \sigma_i \rangle$, 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, $\sigma_i = f_d(C_i(A))$. The elements in $C_i(A)$ are 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, which somehow records the information about the "source" of a derived certainty. In other words, it helps track the subgoal contributed to the derivations. In case a fact B is improved at iteration i, we check $C_i(A)$ to see whether some element of $C_i(A)$ indicating $B \in S_B$. If so, it implies that the certainty of this element depends on the certainty of B. We then remove this element from $C_i(A)$. The SNP algorithm only focuses on derivations in which at least one of their subgoals is improved in the last iteration. The SNP evaluation is shown in Algorithm 3. An extention of this partitioning technique will be introduced in section 6.3.1 to handle certainty constraints.

Algorithm 3 Multiset-based Semi-Naive with Partition Algorithm [SZ08]

Input: P, D
// P is a parametric-program
// D is the set of atom-certainty pairs
Output: $lfp(T_{P \cup D})$

1: **for all** $A \in B_P$ **do**
2: $C_1(A) := \{|(\alpha : \phi)|(A : \alpha) \in D|\}$
3: $\nu_1(A) := f_d(C_1(A)(\alpha))$,where $f_d := Disj(\pi(A))$
4: **end for**
5: $newset_1 := \{A|(A : \alpha) \in D\}; i := 1;$
6: **while** $(newset_i \neq \varnothing)$ **do**
7: $i := i + 1$
8: **for all** $A \in B_P$: **do**
9: $C_i(A) := C_{i-1}(A)$
10: **for all** $B \in newset_1 \wedge (\alpha, S_B) \in C_{i-1}(A) \wedge B \in S_B$: **do**
11: $C_i(A) := C_i(A) - \{|(\alpha, S_B)|\}$
12: **end for**
13: **for all** $(r : A \xleftarrow{\alpha_r} B_1, \ldots, B_n; \langle f_d, f_p, f_c \rangle) \in P^* \; \exists B_j \in New_i,$ where $j \in \{1, \ldots, n\}$: **do**
14: $C_i(A) := C_i(A) \cup \{|(\sigma_i^r(A), S_B)|\},$
15: where $\sigma_i^r(A) := f_p(\alpha_r, f_c(\{|\nu_{i-1}(B_1), \ldots, \nu_{i-1}(B_n)|\}))$, and
16: $S_B := \{B_j | B_j \in IDB \wedge j \in \{1, \ldots, n\}\}$
17: **end for**
18: $\nu_i(A) := f_d(C_i(A)(\alpha)), where \;\; f_d := Disj(\pi(A))$
19: **end for**
20: $newset_i := \{A | A \in B_p, \nu_i(A) \succ \nu_{i-1}(A)\}$
21: **end while**
22: $lfp(T_{P \cup D}) := \nu_i$

4.2 Handling Certainty Constraints

We develop a module, called the Certainty Constraints Checker (or CC-Checker, for short) to test the satisfiability of certainty constraints in the EGIB programs. Recall that C_r is a conjunction of certainty constraints in the form of "$wt(A)\ \theta\ wt(B)$" or "$wt(A)\ \theta\ \sigma$". In order to test the satisfiability of C_r, each certainty constraint in C_r has to be tested. Unsatisfiability of any certainty constraint in C_r implies unsatisfiability of C_r.

There is only one C_r associated with each EGIB–rule, but each rule may have many

ground instances. If C_r is a ground instance of a certainty constraint in such a ground rule, the satisfiability test of other ground rules should not reuse the grounded C_r. We need a mechanism to avoid this reuse problem. For this, we first ground C_r according to the ground rule. After the satisfiability test, we reset C_r to the original "empty" state. To be compatible with evaluation of the parametric program, the proposed CC-Checker will always return true when the certainty constraint C_r associated with the EGIB rule is empty.

Algorithm 4 provides details of the CC-Checker. It returns the value true when the conjunction of certainty constraints in the corresponding ground rule is satisfied, and returns false otherwise.

Algorithm 4 Certainty Constraints_Checker

Input: A conjunction of certainty constraint C_r, a ground instance of rule r
// $C_r \equiv C_{r1} \wedge C_{r2}$, $C_{r1} \equiv C_1 \wedge, \ldots, \wedge C_k$,$C_{r2} \equiv C_0' \wedge, \ldots, \wedge C_j'$
// C_{r1} is a conjunction of certainty constraints $wt(A)\theta\sigma$,
// C_{r2} is a conjunction of certainty constraints $wt(A)\theta wt(B)$.
Output: Boolean

```
1: if C_r is empty then
2:     return true
3: end if
4: Initialize C_r using corresponding ground atoms in instantiated rule r.
   // Check the satisfiability of the first type certainty constraints.
5: for i = 0 to k do
6:     if (!Satisfiable1(C_i)) then
7:         Reset C_r to its original form in r
8:         return false
9:     end if
10: end for
   // Check the satisfiability of the second type certainty constraints.
11: for i = 0 to j do
12:     if (!Satisfiable2(C_i')) then
13:         Reset C_r to its original form in r
14:         return false
15:     end if
16: end for
17: Reset C_r to its original form in r
18: return true
```

Recall that the certainty constraint of the form $wt(\mathrm{p}(\overline{X}))\ \theta\ \sigma$ is actually a *selection by certainty* operation, which selects tuples from a relation, whose associated certainty value holds true with respect to the operator θ, and the specified threshold σ. A certainty constraint of type $wt(p(\overline{X}))\ \theta\ wt(q(\overline{Y}))$ is a *join by certainty* operation, which prescribes that a pair of tuples from two relations can be joined, provided their associated certainties stand in the relationship defined by the operator θ. We can view the operations of selection and join by certainty as a filtering mechanism, which can determine a (possibly empty) intermediate relation during the evaluation process. We can also consider $wt(p(\overline{X}))$ as a function call, returning the "current" certainty associated with the ground instance of $p(\overline{X})$ during rule evaluation.

For certainty constraint $wt(\mathrm{p}(\overline{X}))\ \theta\ \sigma$, we develop the algorithm "*Satisfiable_1*" to test its satisfiability. If the certainty of a ground atom satisfies the relationship defined by the operator with a specified threshold σ, the algorithm returns true; otherwise, it returns false. The description of this algorithm is shown in Algorithm 5.

For certainty constraint of type $wt(p(\overline{X}))\ \theta\ wt(q(\overline{Y}))$, the algorithm "*Satisfiable_2*" is used to test its satisfiability. If not all arguments of both predicates in this constraint are grounded, we return "true" as a strategy to postpone its constraint evaluation until all arguments are instantiated. When both of arguments are grounded, we test the certainties of these two predicates to see whether they stand in the relationship defined by the operator θ. The test result returned is true if the condition holds; otherwise, it returns false. The details of this algorithm is shown in Algorithm 6.

Algorithm 5 Satisfiable_1

Input: Certainty constraint 1 $C_i \equiv wt(A)\theta\sigma$
// C_i is an expression of the form $wt(A)\theta\sigma$
Output: Boolean
1: Let operator=map(θ)
// map θ to a corresponding integer
// θ is a member of $\{\prec, \preceq, =, \neq, \succ, \succeq\}$ with values 0, 1, 2,..., 5, respectively
2: **if** A is not a ground atom **then**
3: **return true**
4: **end if**
5: **if** operator=0 **then**
6: **return** $wt(A) \prec \sigma$
7: **end if**
8: **if** operator=1 **then**
9: **return** $wt(A) \preceq \sigma$
10: **end if**
11: **if** operator=2 **then**
12: **return** $wt(A) = \sigma$
13: **end if**
14: **if** operator=3 **then**
15: **return** $wt(A) \neq \sigma$
16: **end if**
17: **if** operator=4 **then**
18: **return** $wt(A) \succ \sigma$
19: **end if**
20: **if** operator=5 **then**
21: **return** $wt(A) \succeq \sigma$
22: **end if**

4.3 Incorporating CC-Checker within Evaluation Process

There are two ways to invoke the module CC-Checker. First, the evaluation algorithm calls this module after regular subgoals in the rule body are evaluated. If it returns true, then it generates the atom-certainty pair for the head predicate. Otherwise, it moves on to the next derivation. As the second way, the evaluation algorithm invokes the CC-Checker on the fly while it is evaluating regular subgoals of the rule body,

Algorithm 6 Satisfiable_2

Input: Certainty constraint 2: C_i' // $wt(A)\theta wt(B)$
Output: Boolean
// map θ to a corresponding integer
// θ is a member of $\{\prec, \preceq, =, \neq, \succ, \succeq\}$ with values 0, 1, 2,..., 5, respectively

```
1: if A and B are not ground atoms then
2:     return true
3: end if
4: if operator=0 then
5:     return wt(A) ≺ wt(B)
6: end if
7: if operator=1 then
8:     return wt(A) ⪯ wt(B)
9: end if
10: if operator=2 then
11:     return wt(A) = wt(B)
12: end if
13: if operator=3 then
14:     return wt(A) ≠ wt(B)
15: end if
16: if operator=4 then
17:     return wt(A) ≻ wt(B)
18: end if
19: if operator=5 then
20:     return wt(A) ⪰ wt(B)
21: end if
```

rather than postponing checking constraints.

The advantage of the first way is its simple implementation by directly adding the CC-Checker to the evaluation algorithm. Its disadvantage is that we can only find out that the certainty constraints are not satisfiable when evaluation of the body is done, i.e., perhaps after some useless computations. The second way is more efficient, as it terminates derivation immediately when the CC-Checker fails during the rule evaluation.

We use the following rule r_1 to illustrate the advantage and disadvantage of these

two ways.

$$r_1 : t(X,Y) \xleftarrow{1} p(X,X), q(X,Y), wt(p(X,X)) > 0.5; (f_d, f_p, f_c).$$

Suppose EDB=$\{p(1,1) : 0.5,\ p(2,2) : 0.5,\ p(3,3) : 0.6,\ q(1,5) : 0.6,\ q(2,7) : 0.6,\ q(2,8) : 0.6,\ q(3,9) : 0.6\}$. Adopting the first way, we construct the following fragment of the rule r_1 during the join process:

$$p(1,1),\ q(1,5),\ wt(p(1,1)) > 0.5$$

However, this fails when we perform the CC-Checker, due to the unsatisfiability of certainty constraint $wt(p(1,1)) > 0.5$. Similarly, the following partial evaluations will be built during the evaluation process, but none of them will contribute to the

$$\{p(1,1), q(1,6), wt(p(1,1) > 0.5\}$$
$$\{p(2,2), q(2,7), wt(p(2,2)) > 0.5\}$$
$$\{p(2,2), p(2,8), wt(p(2,2)) > 0.5\}$$

head predicate because of the unsatisfiability of the associated certainty constraint. Only derivation $\{p(3,3), q(3,9), wt(p(3,3)) > 0.5\}$ will produce atom-certainty pair $\{t(3,9) : 0.36\}$ for the head predicate. Therefore, if we can prune in advance, the tuples in relation P which do not satisfy certainty constraints, the unproductive work above could be avoided.

The second way overcomes the unproductive work above. There are two types of certainty constraints in an EGIB rule, which are checked separately. We apply the "select-before-join" principle. For every predicate, we perform a "select" preprocess, which will prune the unnecessary tuples from a relation according to the predicate arguments, before the relation participates in the join process. For instance, for the

predicate $p(X,X)$ in the rule r_1, the tuples $p(1,2):0,5$, $p(2,3):0.5$ will be pruned from the select process, since these tuples do not satisfy the constraint that the first argument should equal the second argument. The "select" module was developed to perform this task. Now we also need to consider the first type of certainty constraint checking in the "select" module. We develop a new module, "extend_select", using which not only can check constraints based on predicate arguments, but also can chck the first type of certainty constraints.

Consider again the above predicate $p(X,X)$ as an example. After "select" module, tuples $p(1,1):0.5$, $p(2,2):0.5$, $p(3,3):0.6$ will be kept. However, after "extend_select" module, only $\{p(3,3):0.6\}$ will be kept, because the other two do not satisfy $wt(p(X,X)) > 0.5$.

The second type of certainty constraints are checked during the join process, rather than postponing them to the end. Let us use rule r_2 below to illustrate this case.

$$r: \quad p(X,Z) \xleftarrow{1} a(X,Y), q(Y,Z), wt(a(X,Y)) > 0.6,$$
$$wt(a(X,Y)) > wt(q(Y,Z)); \langle ind, pro, pro \rangle.$$

Suppose EDB=$\{a(1,2):0.5,\ a(2,3):0.6,\ a(3,4):0.7,\ q(4,6):0.6,\ q(5,6):0.8,\ q(4,7):0.8\}$. We assume the order of occurrence of subgoals in the rule body is the same as the order of the evaluation order of subgoals. When performing the join of the subgoals , we first select predicate a, and then apply the "extend_select" module to its corresponding relation a. This yields tuple $a(3,4):0.7$. Then we select predicate p, and invoke the "extend_select" module to the corresponding relation q, which yields the tuples: $q(4,6):0.6$, $q(5,6):0.8$, $q(4,7):0.8$. In the join process, the tuples in the left relation are chosen one by one to join with suitable tuples in the right relation. Now, we select a(3,4):0.7, which matches the tuples in relation q as it satisfies the "join constraint" in q, which requires that the first argument of relation q is the same

as the second argument of relation a. This means tuples q(4,6): 0.6,q(4,7): 0.8 are suitable to join with a(3,4). Actually, q(4, 6): 0.6 will not contribute since condition $wt(a(X,Y)) > wt(q(Y,Z))$ does not hold. Thus we incorporate this type of certainty constraint checking in the join process.

In our system prototype, we implemented the second way instead of the first one, because it can save time and space by avoiding some unproductive computations of intermediate relations. We integrate the above procedure within a Multiset-based Naive algorithm for p-programs, and develop the Extended Multiset-based Naive algorithm for EGIB programs. Algorithm 7 provides details of this algorithm. For this, we add line 10 of Algorithm 7 between lines 12 and 13 in Algorithm 2. We also add the line 10 of Algorithm 7 between lines 13 and 14 in Algorithm 3, and develop an Extended Multiset-based Semi-Naive with Partition algorithm for evaluating EGIB programs.

Algorithm 7 Extended Multiset-based Naive Algorithm with Certainty Constraints

Input: P,D

// P is the set of p-rules with Certainty constraints.

// D is the set of atom-certainty pairs.

Output: $lfp(P(D))$

// P(D) denote the set of atom-certainty pairs derived by applying P to D.

1: **for all** $A \in B_P$ **do**
2: $\quad \nu_0 := \perp$
3: $\quad M_1 := \{|\alpha|(A:\alpha) \in D|\}$
4: $\quad \nu_1(A) := f_d(M_1(A))$, where $f_d := Disj(\pi(A))$
5: **end for**
6: $newset_1 := \{\pi(A)|(A:\alpha) \in D\}; i := 1;$
7: **while** $(newset_i \neq \varnothing)$ **do**
8: $\quad i := i + 1$
9: $\quad$ **for all** $(r : A \xleftarrow{\alpha_r} B_1, \ldots, B_n, C_r; \langle f_d, f_p, f_c \rangle) \in P^*$; **do**
10: $\quad\quad$ **if** $(CertaintyConstraint\_Checker(C_r))$ **then**
11: $\quad\quad\quad M_i(A) := \{|f_p(\alpha_r, f_c(\{|\nu_{i-1}(B_1), \ldots, \nu_{i-1}(B_n)|\}))|\};$
12: $\quad\quad$ **end if**
13: $\quad$ **end for**
14: $\quad newset_i := \{A|A \in B_p, \nu_i(A) \succ \nu_{i-1}(A)\};$
15: $\quad \nu_i(A) := f_d(M_i(A))$, where $f_d := Disj(\pi(A))$
16: **end while**
17: $lfp(P(D)) := \nu_i$

Chapter 5

System Architecture

This chapter presents a system prototype of deductive databases with uncertainty, called AILOG, which implements the certainty constraints checking and the optimization techniques proposed. AILOG is a single user, in-memory deductive database system. It also serves as a test bed for evaluating the optimization techniques proposed in chapter 4. To ensure portability, this system is implemented in Java. Figure 5.1 gives an overview of the system architecture.

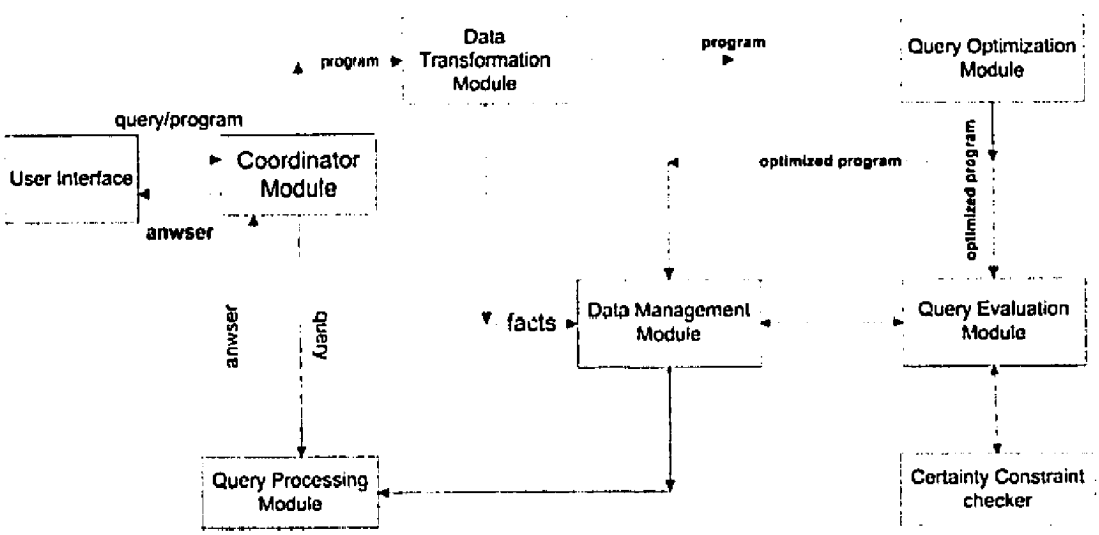

Figure 5.1: System architecture

As shown in the figure, AILOG consists of six main modules:

1. **Coordinator Module** (CM) which is the central part of the system and is responsible for the overall program evaluation flow. It takes input from the User Interface and creates subtasks for the Data Transformation Module, Query Optimization Module, Query Processing Module ,and Data Management Module to process user queries. This module also returns the result to the user through the interface.

2. **Data Transformation Module** (DTM) is actually a "parser". As the input, it takes a logic program with uncertainty in either the EGIB or the EGAB language, already stored on disk as a text file. It checks the input for correct syntax and transform it into an internal representation. Furthermore, the DTM can be used to transforms an EGIB program to an equivalent EGAB program, and vice versa.

3. **Data Management Module** (DMM) is responsible for maintaining and manipulating the in-memory data in relations and storing the "optimized" programs.

4. **Query Evaluation Module** (QEM) takes as an input the internal representation of the program produced by DTM. It also takes database relations. The annotations in the program provide execution hints and directives. The QEM *interprets* the internal representation of the optimized program.

5. **Query Processing Module** (QPM) accepts the query, parses its syntax, and then executes the query represented in some internal form. The user program interacts with QPM, which in turn, interacts with the DMM. The QPM isolates the user from execution details. The user specifies a query, and the QPM determines how to compute the answers.

In the following sections, we will explain these modules and their interactions in more details.

5.1 Coordinator Module

The Coordinator Module, which acts as the heart of AILOG, is responsible for the overall evaluation process.

Figure 5.2 shows an sequence diagram indicating how the CM interacts with the user and the other five modules in a time sequence diagram. As shown in the figure, the CM first gets program and configuration parameters from the user. It then stores the configuration parameters in a *"Config"* table, a data structure which saves the user preference for later use, and invokes the Data Transformation Module to parse the input program and transform it into an internal representation for further processing. The CM module then calls QOM to optimize the program based on the optimization parameter specified in the *"Config"* table. After the optimization, the CM will choose the corresponding strategy to evaluate the "optimized" program by sending the optimized program to QEM.

Queries typed by the user in the user interface do not require rewriting. For such queries, CM calls QPM and returns the result to the user. Complex queries are typically defined and stored in advance as some "program modules" that export predicates (view) with associated "query forms". For such queries, QOM will optimize the program module and a query form, and generate a rewritten program. QEM evaluates the optimized program and returns the result back to the user.

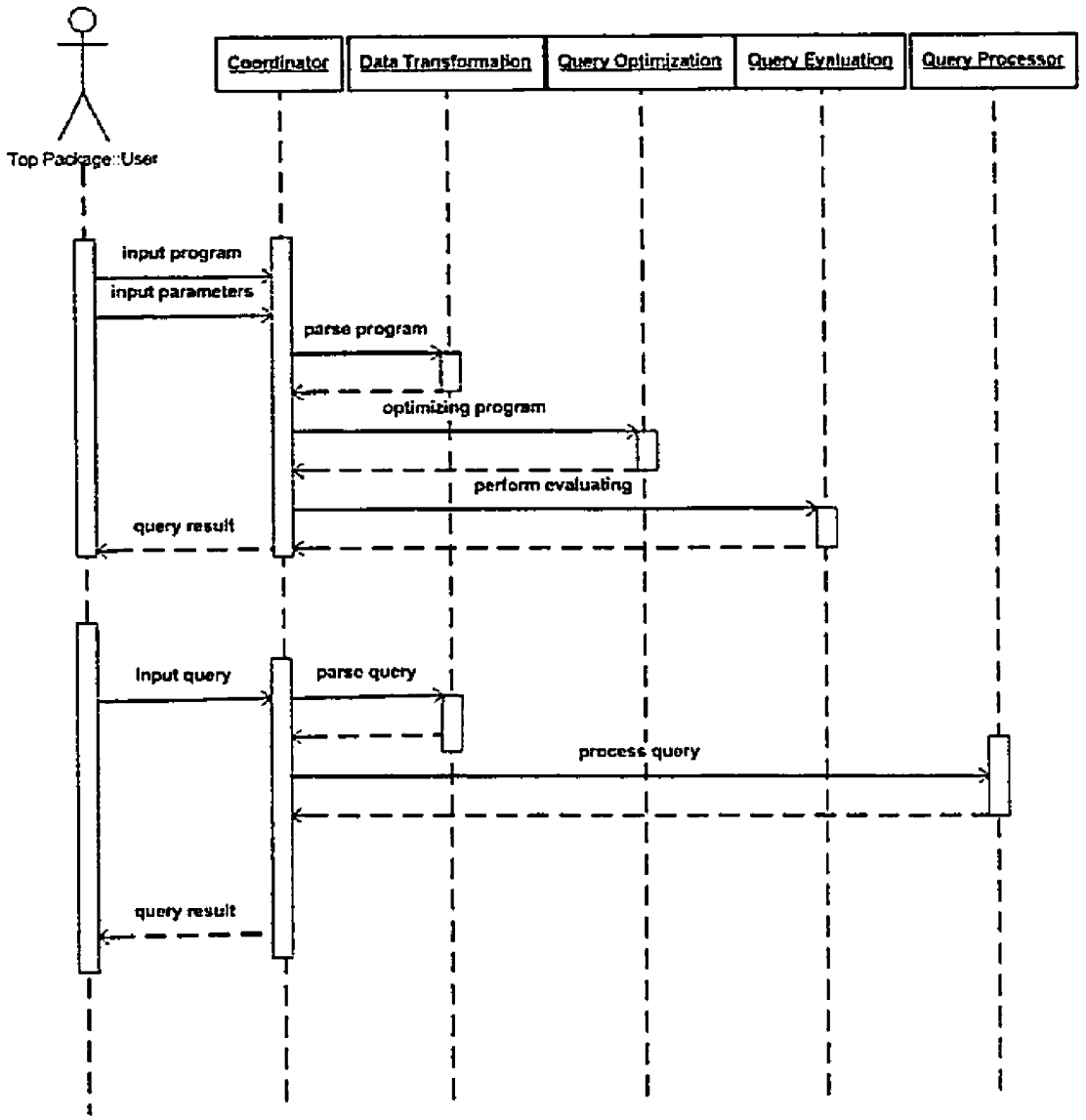

Figure 5.2: General sequence diagram of the system prototype

5.2 Data Transformation Module

Figure 5.3 shows the model diagram of DTM. The input (EGIB or EGAB) program is usually stored in the secondary memory as a text file, before it is submitted to the system for evaluation. Once submitted, the DTM will call the corresponding (EGAB or EGAB) parser in DTM. Since there are two types of input programs in our system,

i.e, EGIB and EGAB programs, we develop a parser for each kind of syntax. They are called Parser_IB and Parser_AB. The appropriate parser is chosen and the program is then checked for its syntax. If there is no syntactic error in the input program, it is separated into three parts: facts and rules, certainty constraints, and the query. Every part is passed to corresponding transformation procedures, which transforms the input into its internal representation. In this transformation procedure, we take advantage of the object-oriented data model, such as *inheritance*, for presentation and manipulation ease. The output of DTM, which contains transformed facts and rules, certainty constraints, and the query, is then passed to modules DMM, QOM, and QPM, respectively. In addition, based on the user configuration parameters specified, the input logic program could be transformed into its desired EGIB or EGAB program and stored in the secondary memory as a text file.

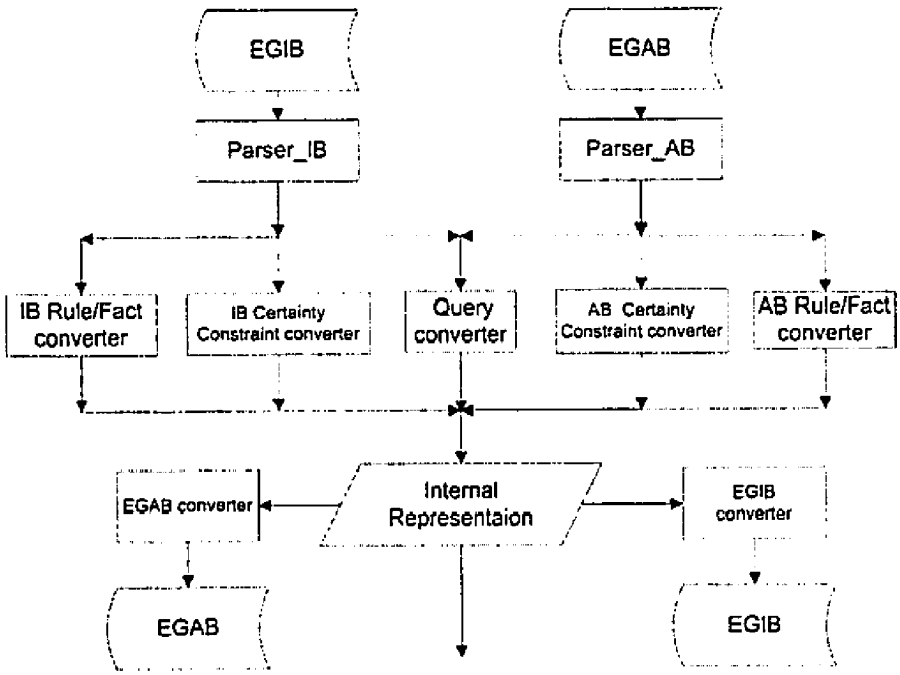

Figure 5.3: Data Transformation procedure

In the parsing process, regardless of which of Parser_IB or Parser_AB is used, we perform program safety checks. The program safety checking consists of rule safety checking, fact safety checking, and certainty constraints safety checking. If any one of these three safety checks fails, the program safety checks fails. This terminates the parsing process and a suitable error message is displayed through the User Interface.

5.3 Data Management Module

AILOG currently supports only in-memory data. The DMM is responsible for maintaining and manipulating relations and programs that are in-memory. The facts, rules, and query are represented as Java classes. In order to have efficient search and access, we integrate these classes and provide uniform access for other modules. The DM creates EDB relations for the transformed facts from the DT facts converter. It also stores and manages IDB relations defined during the evaluation process dynamically. We store and manage IDB and EDB relations uniformly in our system. We use a data structure called *fact_table* to "link together" the relations. Furthermore, the "optimized" program generated by the QOM is stored in the data structure *rule_table*.

We create and use a workspace called *SymbolTable*, also called a workspace, to manage database relations and program. A user can have several named workspaces, copy relations from one workspace to another (or simply make a relation in one workspace visible from another without copying), update relations in a workspace, or run queries against a workspace. It is also possible to save a workspace as a text file; persistency is left as a possible future work.

We use hash-based indices for in-memory relations, in order to speed up query evaluation. These indices include two levels. At the relation level, we use hash tables to store the key and the reference to a relation. At the tuple level, we use hash tables to store the key and the reference to a tuple. We create and use indices to access the

relation and tuples during the query evaluation.

The DMM offers a well defined "get-next-tuple" interface to other modules, especially to QEM, which accesses relations frequently. This interface is independent of how the relations are defined, so it is quite flexible. Such a high-level interface is quite useful, since it allows different modules to be evaluated using different strategies.

5.4 Query Evaluation Module

This module is the central component for the efficiency of AILOG. It takes the "optimized" program from the QOM and the EDB facts as input. Based on the user parameters and execution hints in the "optimized" program, QEM will apply the appropriate strategies to execute the program. The QEM derives the selected rules by populating the matched facts iteration by iteration until no "new" fact-certainty pair is produced. Recall that definition of "new" is extended from the standard case to our context of uncertainties to include not only the new facts, but also an old fact with a "better" certainty. When the least fixpoint is reached, the evaluation terminates and returns the least model, which consists of atom-certainty pairs.

Our Extended Multiset-based Naive (EN) algorithm, Extended Multiset-based Semi-Naive (ESN) algorithm, and Extend Multiset-based Semi-Naive with Partition (ESNP) algorithm, described in Chapter 4, are implemented in the QEM, which calls the CC-checker. The stratification method is also implemented here, but cannot be applied in isolation. This method should be used in conjunction with one of the three evaluation algorithms for improved performance.

Since the notion of "new" is extended in our context of uncertainty, the termination behavior of the evaluation changes as well, that is, the evaluation continues if the certainty of any fact is improved. Because of the continuity property of the fixpoint operator T_p, in some cases, the evaluation will take ω steps to reach the fixpoint. For

practical reason, we take advantage of this property to allow an evaluation to proceed until a desired precision is achieved. When the amount of improvement of certainty is less than a user specified error, the evaluation process will be terminated.

The CC-Checker is a sub-module of QEM. When a rule of an "optimized" program has certainty constraints in the rule body, the CC-checker is invoked during query processing to evaluate the certainty constraint parts.

5.5 Query Processing Module

Our QPM accepts a query, chooses a good execution plan, and executes. Simple queries (selecting facts from a single relation or multiple joined relations, for instance), which can be typed in at the user interface, is dealt with by the QPM in our system. In this case, QPM only needs to verify the query syntax, and execute it against the database, since it it already derived all atom-certainty pairs from the program. We use "?" to indicate a query, e.g., "$?p(a, Y), wt(p(a, Y)) \geq 0.5$", which asks for a variable Y such that p(a,Y) is true with a certainty of at least 0.5. The database (which is the least fixpoint model) is represented by a set of relations linked together. The QPM needs to find the corresponding relation through the DMM for the query, and creates a view of the corresponding relation according to the query and its bindings expressed in the query predicate. The query result is then returned to the user.

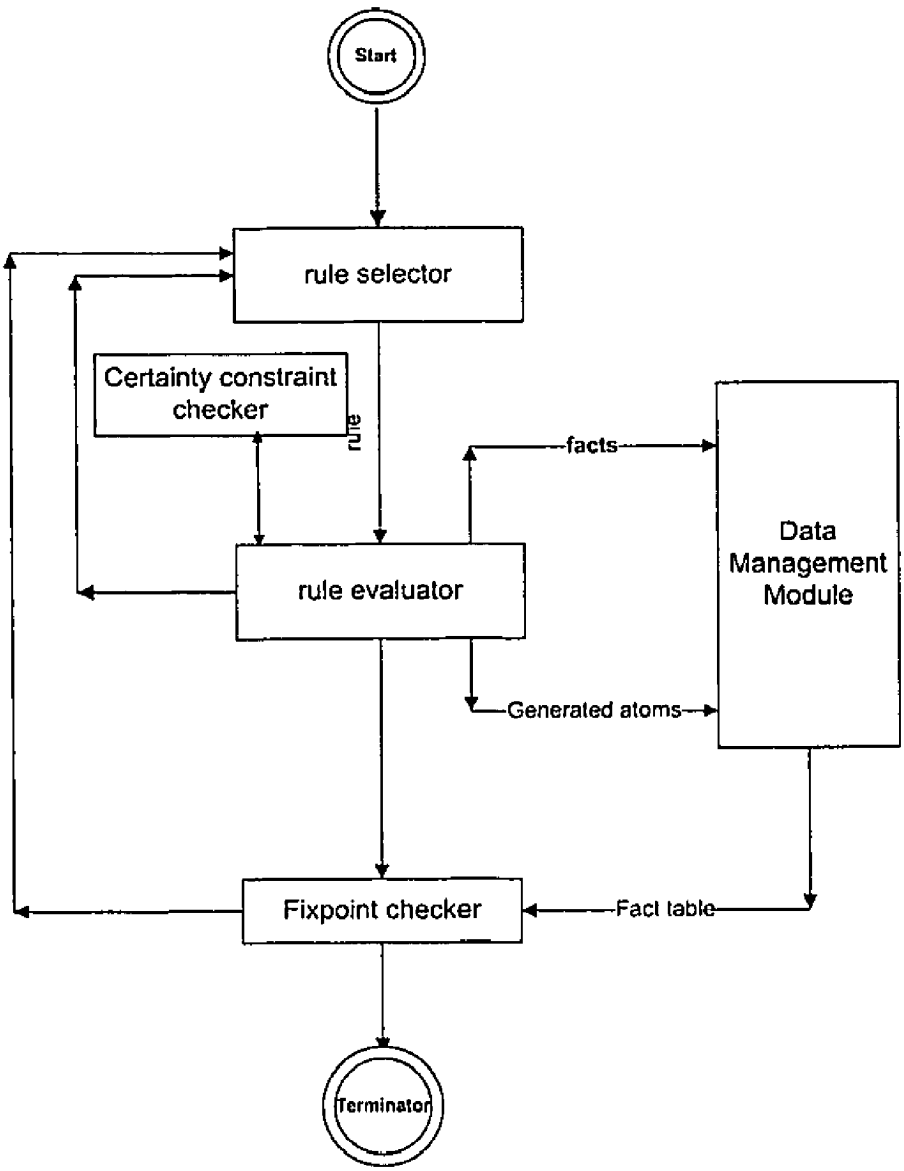

Figure 5.4: Program evaluation procedure

Chapter 6

System Implementation

Our system prototype AILOG was implemented in Java. Following the system architecture described in chapter 5, we split the system implementation into subsystems, and develop proper interfaces for these subsystems. In this chapter, we explain some technical details of our implementation.

6.1 Data Structure

We use Java which is an object-oriented programming language, in order to take advantage of object oriented data model in representing data structures in our implementation. The efficiency with which such data can be processed depends largely on our representation. The relations are represented as follows:

1. Each workspace is a collection of relations
2. Each relation is a collection of tuples
3. Each tuple is a list of arguments
4. Each argument has a type, such as Integer (int), String, etc.

5. Rule table is a list of rules

6. Each rule is implemented as a list of predicates, certainty constraints, and combination functions.

6.1.1 Structure of Tuple

The notion of fact in logic programming and deductive databases corresponds to a tuple in relational databases. There are two types of facts in logic programming and deductive databases: an Extensional Database (EDB) fact, which is explicitly mentioned in the program, and an Intensional Database (IDB) fact, which is implicitly mentioned in the program and is derived from the rules and facts in the program. In our implementation, we treat them the same and use class Tuple to represent both types of facts.

With the presence of uncertainty in deductive databases, each fact is then associated with a certainty, represented as an fact–certainty pair. The major difference from the standard case is that we need to record the certainty of each fact. To develop the corresponding structure for fact-certainty pairs, we add the "certainty" part to the Tuple class.

For each object of the Tuple class, we associate a list of all terms that the object holds. Since we have a program safety checker in the parsing stage, we do not allow non-ground facts in our work, and, hence, the terms in the list associated with an object are all constants. It is convenient to use list to implement the list of terms, which is also fast in search. The Tuple class defines a number of methods to access and manipulate tuple objects. These methods include *addColumn(String s)*, *setColumn(int index, String s)*, *comparesTo(Tuple t)* and *factSafeCheck(String name)*.

Note that each object of the Tuple class need not explicitly have a relation name, because all the tuples will be clustered according to their predicate name to form a

relation, and the relation name is exactly the same as the predicate name.

6.1.2 Structure of Relations and Fact table

A relation is a collection of tuples which have the same predicate name, and the name of the relation is the same as that of the predicate. Although different fixpoint evaluation techniques may require different information to be kept for each relation, the basic structure is more or less the same for all. For example, each relation has a name, a structure containing its tuples and member functions. There are two kinds of structures in our system: list and hash table. For the tuples stored in the list, we do not need any index. Each element of a list stores a reference to the tuple object. The disadvantage of this structure is that, if we request a specific tuple from a relation, it needs to go through each element in the list, which could be lengthy. Hash table is widely used as an efficient storage structure for in-memory data[HGMW02, Sha00].

Actually, in our system prototype, we provide a mechanism to take advantage of both list and hash table. First, we add the tuples to the list based on their occurrence, since we use a nested loop join mechanism for performing join of the body predicates. It is thus convenient to go through the list from top to bottom. Second, we use a key-generator to generate a key for every tuple in the list, and to map the key values to the corresponding references of the tuples using a hash table. The hash table structure supports an efficient look-up operation: given a key (generated from the tuple), it finds the corresponding value (reference of the tuple) efficiently.

To construct a hash table, the first task is to determine the hash key. In our system prototype, we combine the relation name to which the tuple belongs with the values of arguments. The result of this combination is a string, which is converted to an integer. We do this by using the default hashcode() method of the Java String

class. That is, the hash code for a String object is computed as

$$s[0] * 31^{(n-1)} + s[1] * 31^{(n-2)} + ... + s[n-1]$$

where $s[i]$ is the *ith* character of the string, n is the length of the string (the hash value of the empty string is zero). For example, the string "abc", "cba", "acb" are converted to integers 96354, 98274, 96384, respectively.

After converting the key from string to integer, a hash function is applied to get the index corresponding to the key in the hash table, as follows.

$$h(key) = (key \ \& \ \ 0x7FFFFFFF) \ \ \% \ \ hash\_size$$

Note that no hash function is collision-free. At the same time, the hash table is open, i.e., in case of a collision, a single bucket stores multiple hashed values, which must be searched sequentially. The size of multiple entries in a single bucket is determined by the distribution of tuples and the hash-size, since searching in a single bucket is sequential. Therefore, a smaller number of entries in a bucket and a larger hash-size are preferred. However, too large a hash-size may result in a waste of space and a high maintenance cost. Depending on the input program, an EDB relation has a fixed size, which can be obtained in advance. When we create a hash table for an EDB relation, we can set the initial size. The size of an IDB relation, on the other hand, increases during an evaluation, for which we create a hash table using a default initial size. When a higher hash table capacity is needed, its size will automatically increase. To achieve this, we can control the *initial capacity* and *load factor* when we initialize the hash table.

For the Naive and Semi-Naive algorithms, relations are integrated, i.e., there are no partitions. However, in the Semi-Naive algorithm with Partition, an IDB relation

is partitioned into two parts: the "improved" part and the "non-improved" part. We use two lists to: one to store the "improved" part and the other one to store the "non-improved" part. Actually, we divide the *references* of tuples into two parts. The tuples move from the nonimproved to the improved part. We only need to move their references accordingly , rather than them, which results in great time saving.

We integrate all the relations as a collection, to which we refer as *fact_table*. It includes the EDB and IDB relations of the user program. For efficiency, a new relation is simply inserted at the end of the collection. The *fact_table* provides a unified access to the relations. For the collection structure, it offers sequential search to a relation. If an input program has a large number of relations, such a sequential search (without index) would result in low efficiency. Therefore, we build a hash table for the relations, using relation names as keys and the references to the relations as values. The building hash table overhead for *fact_table* pays off well when the number of input relations is large. Figure 6.1 shows relations and how they are linked to each other.

6.1.3 Structure of Rules and Rule table

Each rule in our system prototype is implemented as a list of predicates. It contains three parts: rule head, rule body and parameters. The rule head contains a predicate, which indicates the relation that the derived facts belong to. The rule body consists of three parts: a list of normal predicates, a list of first type certainty constraints and a list of second type certainty constraints. The first part indicates how the relations join to generate facts for the head relation. The second and the third parts indicate the restriction applied during the process of joining relations. Note that, the result of the join of relations is independent from the sequence of joins. Our system prototype evaluates joins from left to right as the default order. The parameters include the

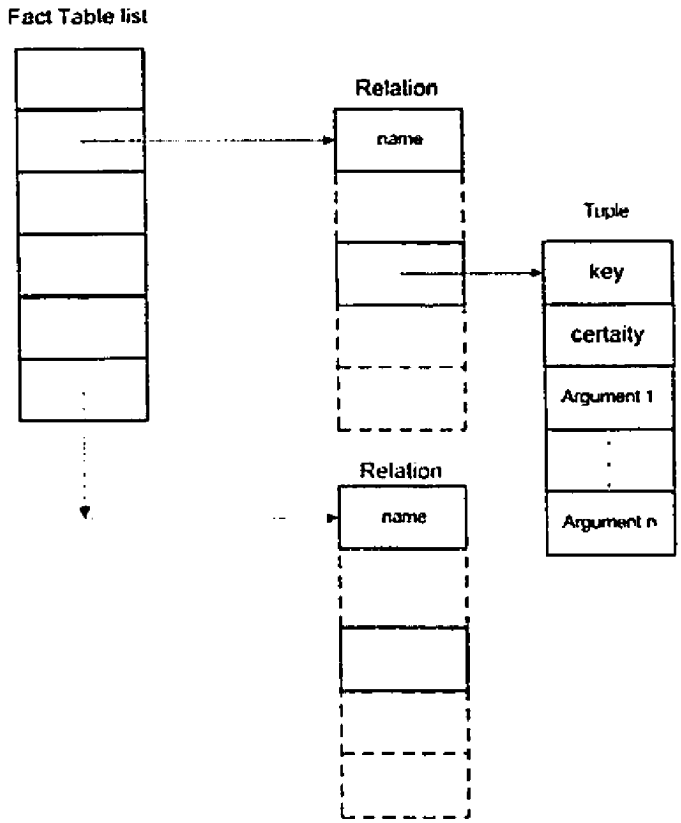

Figure 6.1: Internal representation of fact tables

rule certainty and the combination functions associated with the rule.

Note that each fact is considered as a special rule with an empty body, interpreted as being true, and hence there is no need for disjunction, propagation and conjunction functions. In our implementation, we do not treat facts differently from the rules. We parse and store them directly into relations when parsing the program. This is because we do not need to re-evaluate these facts once they are stored into relations at compile time, and their associated certainties will never change during fixpoint evaluation. Therefore, our rule table only stores rules that have some EDB or IDB predicate(s) in the body. Figure 6.2 shows the internal representation of the rules in

for following program $P_{6.1}$

$$p(X,Y) \overset{0.9}{\leftarrow} e(X,Y), wt(e(X,Y)) > 0; \langle ind, pro, pro \rangle.$$

$$p(X,Y) \overset{0.9}{\leftarrow} e(X,Y), p(Y,Z), wt(e(X,Y)) > 0, wt(p(Y,Z)) > 0,$$
$$wt(e(X,Y)) > wt(p(Y,Z)); \langle ind, pro, pro \rangle.$$

In our implementation, the predicate is the basic component in the rule head and body. The predicate structure has a predicate name and arguments (variables and constants). Since different predicates may have different arities, we use a dynamic array as the underlying container for arguments, for convenience.

As shown in Figure 6.2, the rule head contains a head predicate, and its body is made up of three collections. The first collection contains a list of predicates in a particular order. Since different rules may have different numbers of subgoals, a dynamic array is suitable to represent it. The second and third collections contain first type certainty constraints and second type certainty constraints, respectively. They could also be null if the rule does not contain the corresponding constraint. All the predicates appearing in the CCs must appear in the first collection, otherwise it violates the program safety condition. This situation will be detected at compile time and reported as an error. In the CCs structure, to save space, we do not store the predicate, but rather store on the reference of a predicate, which points to the predicate in the rule body. The last part of a rule stores parameters, which includes rule certainty, and its associated disjunction, propagation, and conjunction functions.

All these aforementioned structures are wrapped as members of the Rule class. We have also defined some other useful operations in the Rule class, including member functions. The rule class is the internal representation of both the EGIB and EGAB programs, so transformation between an EGIB rule and an EGAB rule is

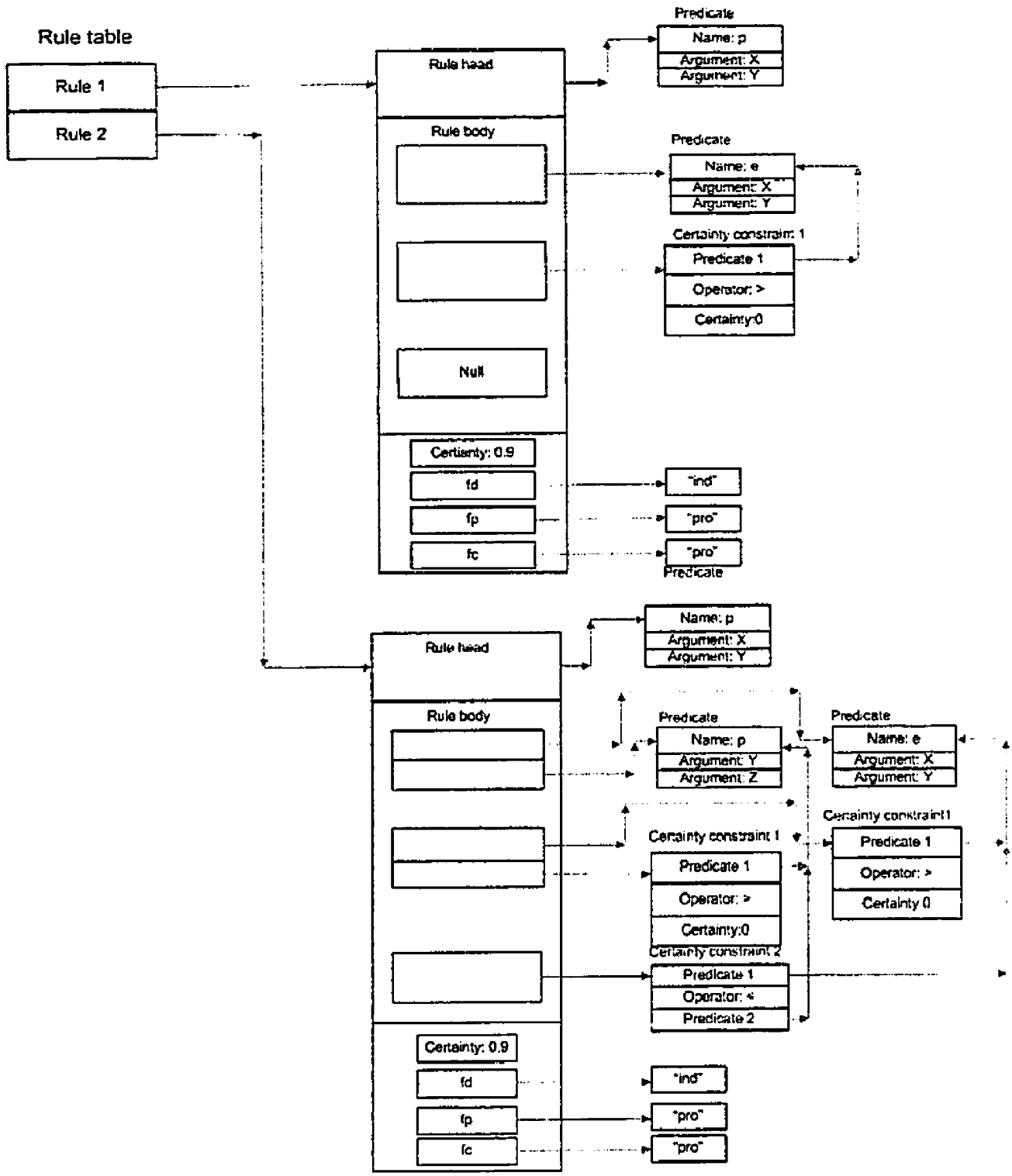

Figure 6.2: Internal representation of rule table

actually implemented here. We define an *to_IB* function to transfer the internal representation into an EGIB rule, and define an *to_AB* function to transfer the internal representation into an EGAB rule.

The program safety check contains three parts: the fact safety checking, rule safety checking, and certainty constraints safety checking. In the Rule class, we define a *rule_safety_check* function to check rule safety and a *certainty_constraint_check* function to check certainty constraint safety.

The data structure *rule_table* was implemented as a collection. When the stratification technique does not apply, all the rules are stored in one collection, as a specific order of rules is not required. The evaluation technique gets rules from this collection one by one. If stratification applies, the input rules are reordered according to the determined order, and the order of applying rules in the same stratum is immaterial. We create several stratum tables corresponding to the strata of rules, and each stratum table stores the references of the rules in the *rule_table* . The evaluation technique proceeds by evaluating the rules stratum by stratum.

6.1.4 Storage Structure for Multisets

In deductive databases with certainty constraints, we are interested in multisets over B=$B_P \times \mathcal{T}$, where B_P is the Herbrand base of the given logic program P, and $\mathcal{T}$ is the set of certainty values used. If X is a multiset over B, then every element in X is of the form $(A : \alpha) : m$, where $A \in B_P$ is a ground atom, $\alpha \in \mathcal{T}$ is a certainty associated with A, and $m \geq 0$ is the multiplicity of the basic part (A, α) in X.

When uncertainty is present, the fixpoint evaluation is sensitive to duplicates. Therefore, we need to use a multiset to store derived facts and, after every iteration, we need to combine duplicate derivations of an atom into a certainty using the disjunction function associated with the predicate of the atom. This procedure is called

normalization [SV97], which intuitively combine the certainty of each atom A into a certainty value, for all atoms $A \in B_P$. Formally, given a multiset X over $B_P \times \mathcal{T}$, the normalization of X is defined a set X' whose content is "equivalent" to the multiset X as follows:

$$X' = \{|(A : \beta)|(A : \alpha_j) : m_j \in X, 1 < j < l_A, \pi(A) = p_i, \beta = f_i(Y)|\}$$

where $p_1, \ldots, p_k$ are all the predicate symbols in the program, f_i is the disjunction function associated with p_i, $1 \leq i \leq k$, and Y is a multiset that includes m_j copies of a certainty value α_j, for all derivations of atom A. Recall that for an atom A, $\pi(A)$ denotes the predicate symbol of A.

We need a data structure that can serve as a container of the multiset, and can perform *normalization* on the multiset. This motivates our design of multiset storage. Since the Semi-Naive with Partition algorithm has a different need for a multiset from the Naive and Semi-Naive algorithms, we create Multiset1 structure for the former and Multiset2 structure for the latter two.

For the data structure Multiset2, we use a dynamic array as a container for the input program, in which each rule has a corresponding element in the dynamic array. Each element of the array stores a data structure, which is similar to the *relation* structure we defined in section 6.1.2, but allows duplicate tuples for multiple derivations of different fact–certainty pairs obtained by rule r at some iterations. If none of the subgoals of a rule has something "new", we keep the fact–certainty pairs unchanged in the relation structure for next iteration. If there is some subgoal in the rule whose certainty is improved, the fact–certainty pairs obtained in the last iteration will be removed and replaced by the new fact-certainty pairs obtained by re-evaluation of the rule.

We define a member function in the Multiset2 to perform disjunction function

on the multiset structure. The fact–certainty pairs, after applying disjunction, are stored in a hash table, which is actually a set whose content is "equivalent" to the original multiset.

For Multiset1, we use a dynamic array to store a list of hash tables, one for each rule. Each entry of the hash table is a pair $(S_B, tuple)$, where S_B is a set of all the IDB subgoals in one derivation, indicating data contributed to a derived certainty. This bookkeeping helps track the derivation sources and is crucial for efficient SNP evaluation by limiting re-evaluation to derivations with improved subgoals. Consider a hash table corresponding to a rule r. For a ground instance of r, if there is no new subgoal, we do not re-evaluate r, and keep the fact–certainty pair unchanged. If there is a new subgoal in a ground instance of the rule, we generate S_B for any such this derivation and use it as an index to get the corresponding old fact–certainty pair derived by r, remove it, and replace it with the new fact–certainty pair.

We also define a function in the Multiset1 structure to perform disjunction functions. The description of this function is the same as the one described for Multiset2.

6.2 Query Compilation

The Query compilation module is responsible for parsing the user input, checking if it is well-formed, and, at the same time, transforms them into the internal representation defined in section 6.1. There are two parsers in our system, which are transparent to the user. One parser is responsible for the EGIB programs, and the other one is responsible for the EGAB programs. Figure 6.3 shows the parsing process, which consists of two phases. The first phase is scanning or token generation, by which the input program viewed as a character stream is divided into meaningful symbols defined by the grammar. The second phase is parsing or syntactic analysis, which checks whether the tokens form an allowable expression. This is usually done with

reference to a context-free grammar, which recursively defines components that can make up an expression and the order in which they must appear. The second phrase outputs an abstract parse tree.

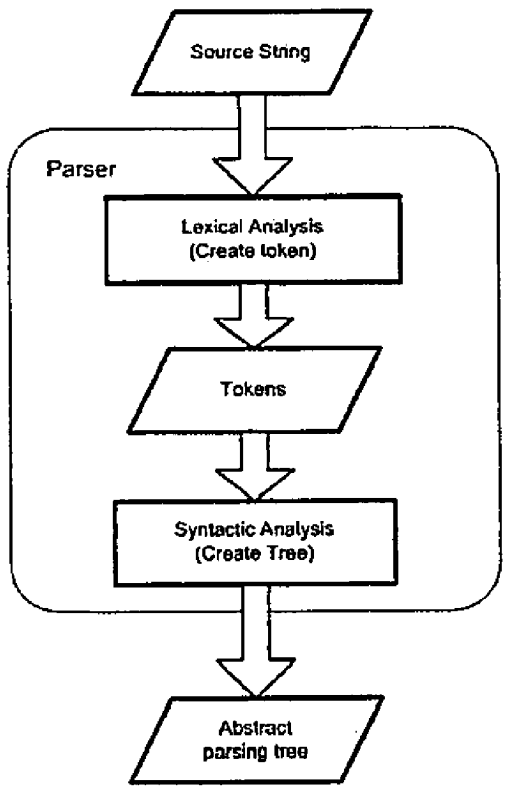

Figure 6.3: The parsing process

6.2.1 Scanner

As in conventional parsers, the scanner is a Lexical Analyzer which reads the characters of the source language, treat as a stream of characters. Moreover, it collects consecutive characters into basic logical units of the source language. A logical unit, which is meaningful in the source language, is called a token. In our system prototype, we represent a token by the Token class, which has a field to specify the type of token. A simple token, such as " > ", may be represented by a token whose type field is set

to an integer value, e.g, 18 in our implementation. We associate each possible type appearing in our program with a corresponding Integer (int) in the Ipub interface, which allows us to use tokens in Java switch statements [Dep].

The language we deal with contains names of predicates and arguments. Names are also called *identifiers*. A name in this sense is typically a nonempty sequence of letters and digits, beginning with a letter. For clarity and ease of implementation, our logic programming language has so-called *keywords* or *reserved names*, which are sequences of letters that should not be used as names of arbitrary objects. For instance, we have the keywords "begin", "end", "max", "min", etc., which cannot be used as predicate names or arguments. Each keyword should be represented by a distinct token. For these keywords, the corresponding tokens may be BEGIN, END, MAX, MIN, defined as suitable integer constants, such as:

final static int BEGIN = 1 ;

final static int END = 2;

final static int MAX = 51;

final static int MIN= 52 ;

...

A token may distinguish names from keywords as follows: whenever a token that looks like a name is found, its string value is compared to the keywords in a list. The token is classified as a keyword if it is in the list; otherwise it is classified as a name. If the set of keywords becomes larger, we should use more efficient means, such as a hash table, to find the token corresponding to a given string. For instance, by using a hash table.

There are several member functions defined in the scanner to perform the key functions, as follows.

- char Nextchar(): it gets the next character of the input stream of the source

program, and also keeps track of the character position.

- int CheckCharType(char c): it is a character classification method. If the given character is a letter, it returns 1; if the given character is a digit, it returns 2; if the given character is the horizontal tab, the new line feed, or carriage return, it returns 3; if the given character is null, it returns 4; for any other given characters, it returns 5.

- Token GetNextToken(): it constructs the next token from the input character stream and returns it as the next token.

6.2.2 Parser

In our implementation, the scanner we built is shared by the IBParser and the AB-Parser, which are LL(1) parsers for EGIB and EGAB programs, respectively. An LL(1) parser is a top-down parser for a subset of context-free grammars. It parses the input string from left to right, and produces leftmost derivation of the input string using one token of look-ahead . In addition to parsing the input token stream, our parser also transforms the input into an abstract syntax tree for increased flexibility, as described in section 6.1. A desired parser class in our context should have the following features:

- Provide an interface to other classes involved

- Parse through all tokens in the token stream and properly report errors

- Parse all rules, facts and query into an abstract syntax tree, which is implemented as java classes and subclasses

To achieve the above mentioned performance, each parser consists of a set of mutually recursive parsing methods for nonterminals in the language. Each parsing

method is a member function of the Parser class.

6.3 Query Optimization

A number of top-down and bottom-up query optimization strategies have been introduced for standard deductive databases. However, when uncertainty is present, those techniques are not applicable directly mainly because their underlying semantics are set based. Therefore, in [SZ04], which implements a fragment of the parametric framework, a Multiset-based Semi-Naive algorithm and Multiset-based Semi-Naive algorithm with Partition were developed and shown to be effective. These proposed evaluation methods cannot be directly applied in our context with certainty constraints. In our work, we consider various techniques introduced in [SZ04] and extend them to handle CCs. In chapter 4, we introduced the Extended Multiset-based Semi-Naive and Extended Multiset-based Semi-Naive with Partition algorithms. We will describe important implementation details in this section. We will also describe details of our implementation of the stratification evaluation technique.

6.3.1 Relation Partitioning

The Semi-Naive algorithm with Partition focuses on derivations that may generate improved fact-certainty pairs. Derivations of new fact-certainty pairs may result from joining relations which have, in the last iteration, at least a new fact or a fact with improved certainty. Evaluation of program will be restricted to those derivations in which new facts may contribute. This is done through partitioning every IDB relation R into two parts in our implementation: Δ and $\wedge$, where Δ is the improved part of R, and $\wedge$ is the non-improved part of R which includes the rest of fact-certainty pairs. For a relation R, we use R[0] to refer to Δ and use R[1] to refer to $\wedge$. Let P be

an EGIB program using EDB predicates $B_i's$ and IDB predicates $T_j's$. Consider the following generic rule r in the input program P:

$$r : S(\mu) \overset{\alpha_r}{\leftarrow} B_1(\nu_i), \ldots, B_n(\nu_n), T_1(\tau_1), T_2(\tau_2), \ldots, T_m(\tau_m), C_r; \langle f_d, f_p, f_c \rangle.$$

We associate a counter with IDB predicates $T_1(\tau_1), T_2(\tau_2), \ldots, T_m(\tau_m)$, which ranges from 0 to $2^m - 1$, incremented each time. It has corresponding states from $\underbrace{00\ldots00}_{m}$, $\underbrace{00\ldots01}_{m}$, ..., $\underbrace{11\ldots11}_{m}$. Using a data structure to store the state, each element in the data structure corresponds to an IDB predicate, in the order in which the predicate appears in the rule body. If the value of the element is 0, the improved part Δ of the relation corresponding to an IDB predicate should be used in the join process. If this value is 1, it means that the unimproved part of the relation should be used in the join process.

The evaluation of the rule r is divided into 2^m steps. Each state has a corresponding step, used to guide the evaluation at each step, determining which part of the IDB relation should participate in the join process. Before we perform the join, we check the corresponding part of the IDB relation denoted by the state element. If any one of them is empty, the evaluation of this step terminates, and we continue to check the next state, because an empty relation does not contribute to any successful derivation. Furthermore, the last state $\underbrace{11\ldots11}_{m}$ means every IDB predicate should use the part $\wedge$ in its corresponding relation to participate in the join process. But this join does not have any new fact-certainty pairs involved and hence it does not generate any new fact-certainty pairs either. Thus when the counter reaches $2^m - 1$, we do not evaluate the corresponding step.

For rule $r' : p(X, Y) \overset{\alpha}{\leftarrow} p(X, Y), q(Z, Y); \langle f_d, f_p, f_c \rangle$, only the joins of improved fact-certainty pairs in "p" or "q" may yield new fact-certainty pairs. Using rewriting

technique [SZ04], this rule can be rewritten as follows:

$$r'_1 : p(X,Y) \xleftarrow{a} \Delta p(X,Y), q(Z,Y); \langle f_d, f_p, f_c, \rangle;$$
$$r'_2 : p(X,Y) \xleftarrow{a} \wedge p(X,Y), q(Z,Y); \langle f_d, f_p, f_c, \rangle;$$
$$r'_3 : p(X,Y) \xleftarrow{a} \wedge p(x,Y), q(Z,Y); \langle f_d, f_p, f_c, \rangle;$$

Using Relation Partitioning (RP, for short), every rule has a series of corresponding states pattern, and each state has a corresponding rewritten rule. The rule r' has states pattern 00, 01, 10, and 11. These states have corresponding rules, using state pattern as suffix, r'_{00}, r'_{01}, r'_{10}, and r'_{11}. In our RP method, 0 stands for Δ and 1 stands for $\wedge$, so these rewritten rules have the following concrete forms:

$$r'_{00} : p(X,Y) \xleftarrow{a} \Delta p(X,Y), \Delta q(Z,Y); \langle f_d, f_p, f_c \rangle;$$
$$r'_{01} : p(X,Y) \xleftarrow{a} \Delta p(X,Y), \wedge q(Z,Y); \langle f_d, f_p, f_c, \rangle;$$
$$r'_{10} : p(X,Y) \xleftarrow{a} \wedge p(X,Y), \Delta q(Z,Y); \langle f_d, f_p, f_c, \rangle;$$
$$r'_{11} : p(X,Y) \xleftarrow{a} \wedge p(X,Y), \wedge q(Z,Y); \langle f_d, f_p, f_c, \rangle;$$

Between the above two groups of rules, we can see that $r'_2 = r'_{10}$, $r'_3 = r'_{11}$, $r'_1 = r'_{00} \cup r'_{01}$. The rule level equivalence means that these two groups of rules produce the same results on the same input set of fact-certainty pairs, we thus have the following results, which extends the one in [SZ08].

Theorem 6.3.1. *The RP method proposed above produces the same result as the rewritten rule generated from the rule rewriting technique in [SZ08].*

Proof 1. 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 results of the joins of body predicates. For the general rule r above, the rule body can be considered as a sequence of joins (for convenience, we ignore EDB predicates and the argument list of the predicates):

$$T_1 \bowtie T_2 \bowtie T_3 \ldots T_{m-2} \bowtie T_{m-1} \bowtie T_m \tag{1}$$

Using the rewriting technique [SZ04], expression 1 can be further transformed into:

$$r_1 : \Delta T_1 \bowtie T_2 \bowtie T_3 \ldots T_{m-2} \bowtie T_{m-1} \bowtie T_m$$

$$r_2 : \wedge T_1 \bowtie \Delta T_2 \bowtie T_3 \ldots T_{m-2} \bowtie T_{m-1} \bowtie T_m$$

$$r_3 : \wedge T_1 \bowtie \wedge T_2 \bowtie \Delta T_3 \ldots T_{m-2} \bowtie T_{m-1} \bowtie T_m$$

$$r_{m-2} : \wedge T_1 \bowtie \wedge T_2 \bowtie \wedge T_3 \ldots \Delta T_{m-2} \bowtie T_{m-1} \bowtie T_m$$

$$r_{m-1} : \wedge T_1 \bowtie \wedge T_2 \bowtie \wedge T_3 \cdots \wedge T_{m-2} \bowtie \Delta T_{m-1} \bowtie T_m$$

$$r_m : \wedge T_1 \bowtie \wedge T_2 \bowtie \wedge T_3 \cdots \wedge T_{m-2} \bowtie \wedge T_{m-1} \bowtie \Delta T_m$$

$$r_{m+1} : \wedge T_1 \bowtie \wedge T_2 \bowtie \wedge T_3 \cdots \wedge T_{m-2} \bowtie \wedge T_{m-1} \bowtie \wedge T_m$$

In RP method, the generic rule R has a series of state patterns, and each state has a corresponding rewritten rule. The state patterns of R are $\underbrace{00\ldots00}_{m}$, $\underbrace{00\ldots01}_{m}$, ..., $\underbrace{11\ldots11}_{m}$, so the corresponding rewritten rules are $r_{\underbrace{00\ldots00}_{m}}$, $r_{\underbrace{00\ldots01}_{m}}$, ..., $r_{\underbrace{11\ldots11}_{m}}$. For each rule r_i in the rule group r_1, ..., r_m, we can find an equivalent subset of rules

in rule group $r_{\underbrace{00\ldots00}_{m}}, r_{\underbrace{00\ldots01}_{m}}, \cdots, r_{\underbrace{11\ldots11}_{m}}$.

$$r_{m+1} = r_{\underbrace{1\ldots1}_{m}}$$

$$r_m = r_{\underbrace{1\ldots1}_{m-1}0}$$

$$r_{m-1} = r_{\underbrace{1\ldots1}_{m-2}0X} = r_{\underbrace{1\ldots1}_{m-2}00} \cup r_{\underbrace{1\ldots1}_{m-2}01}$$

$$r_{m-2} = r_{\underbrace{1\ldots1}_{m-3}0XX} = r_{\underbrace{1\ldots1}_{m-3}000} \cup r_{\underbrace{1\ldots1}_{m-3}001} \cup r_{\underbrace{1\ldots1}_{m-3}010} \cup r_{\underbrace{1\ldots1}_{m-3}011}$$

$$r_i = r_{\underbrace{1\ldots1}_{i-1}0\underbrace{X\ldots X}_{m-i}}$$

$$r_1 = r_{\underbrace{1\ldots1}_{0}0\underbrace{X\ldots X}_{m-1}}$$

where $\underbrace{1\ldots1}_{i-1}0\underbrace{X\ldots X}_{m-i}$ indicates the state pattern which has $i-1$ 1 as the prefix, 0 in the ith position. X means this position could be 1 or 0. $r_{\underbrace{1\ldots1}_{i-1}0\underbrace{X\ldots X}_{m-i}}$ is a multiset union of all the rules which have $\underbrace{1\ldots1}_{i-1}0$ as prefix.

The correctness of the theorem follows noting that rule level equivalence implies the equivalence of two groups of rules. □

If we look at the rewritten rule r_{m+1}, we can see that no joins in this rule involve any new facts. That is, evaluation of this rule is redundant for not contributing to anything new. If we can identify such rules, redundant computation can be avoided. That is actually the essence of our rule rewriting technique.

6.3.2 Stratification

Stratified evaluation in deductive databases with negation helps select a distinguished minimal model of Datalog$^\neg$, in a natural and intuitive way. In that context, we have

concerned with correctness and not efficiency. In our work, we observe that a "desired" stratification could result in the increased efficiency in evaluation of programs with certainty constraints. The following example illustrates this point.

r_1: p(X,Y)$\stackrel{1}{\leftarrow}$ e(X,Y),wt(e(X,Y))>0;$\langle ind, pro, pro\rangle$.
r_2: p(X,Y)$\stackrel{1}{\leftarrow}$ e(X,Y),p(X,Y),wt(e(X,Y))>0,wt(p(X,Y))>0;$\langle ind, pro, pro\rangle$.
r_3: p(X,Y)$\stackrel{1}{\leftarrow}$ p(X,Y),wt(p(X,Y))>0;$\langle , ind, pro, pro\rangle$.

Figure 6.4: Stratified program $P2$

Let $\{e(1,2) : 0.8\}$ be the EDB. We note that $P2$ has a stratification with strata: $P^1 = \{r_1, r_2\}$, and $P^2 = \{r_3\}$, where $P2 = P^1 \cup P^2$. Let us first consider evaluation of $P@$ in the usual way using the Naive method. The evaluation results at each iteration are as follows.

$$I_1 = \{p(1,2) : 0.8, e(1,2) : 0.8\}$$

$$I_2 = \{p(1,2) : 0.928, e(1,2) : 0.8, q(1,2) : 0.8\}$$

$$I_3 = \{p(1,2) : 0.94848, e(1,2) : 0.8, q(1,2) : 0.928\}$$

$$I_4 = \{p(1,2) : 0.9517568, e(1,2) : 0.8, q(1,2) : 0.94848\}$$

$$I_5 = \{p(1,2) : 0.9517568, e(1,2) : 0.8, q(1,2) : 0.9517568\}$$

$$lfp(P_2) = \{p(1,2) : 0.9517568, e(1,2) : 0.8, q(1,2) : 0.9517568\}$$

We can easily see that the certainty of q(1,2) depends on the certainty of p(1,2). When the certainty of p(1,2) also improves, the certainty of q(1,2) improves accordingly. This process repeats until p(1,2) reaches its "best" possible certainty. At iterations 2, 3, and 4, rule r_3 is evaluated using the certainties of p(1,2) in I_1, I_2, and I_3, respectively, which is less than the certainty of p(1,2) in I_4. Clearly, the evaluation of

rule r_3 using these intermediate values of p(1,2) is redundant. If we postpone these evaluations of r_3 until p(1,2) reaches its "best" possible value, we can save time by avoiding the computation of rule r_3 in iterations 2, 3, and 4. This efficiency could be achieved by evaluating program $P2$ stratum by stratum. The result of stratified evaluation is shown as follows. We use I^{P^i} to denote the result of evaluating P by restricting to rules in stratum P^i.

$$I_1^{P^1} = \{p(1,2) : 0.8, e(1,2) : 0.8\}$$
$$I_2^{P^1} = \{p(1,2) : 0.928, e(1,2) : 0.8\}$$
$$I_3^{P^1} = \{p(1,2) : 0.94848, e(1,2) : 0.8\}$$
$$I_4^{P^1} = \{p(1,2) : 0.9517568, e(1,2) : 0.8\}$$
$$I_1^{P^2} = \{q(1,2) : 0.9517568\}$$
$$lfp(P_2) = I_4^{P^1} \cup I_1^{P^2} = \{p(1,2) : 0.9517568, e(1,2) : 0.8, q(1,2) : 0.9517568\}$$

We first evaluate stratum P^1 iteration by iteration until we reach the fixpoint of P^1 under certain precision control, explained in section 6.4.4. We then record the result and apply it as the input for evaluation of the next stratum P^2. The least fixpoint of $P2$ is the union of the interpretations of each stratum. Through this example, we can see that stratified evaluation could improve efficiency by avoiding the computation using an intermediate result. Note that while more iterations may be needed by a higher strata to compute the final result, more computations using intermediate result can be avoided by stratified evaluation.

Definition 6.3.1. Let $\mathcal{P}$ be an EGIB program. The predicate dependency graph of $\mathcal{P}$, denoted by pdg($\mathcal{P}$), is a graph whose vertices are the IDB predicates in $\mathcal{P}$ and for a rule $H \overset{\alpha_r}{\leftarrow} B_1, \ldots, B_n, T_1, \ldots, T_m, C_r; (f_d, f_p, f_c)$, where B_is are EDB predicates

and T_is are IDB predicates, and there is an edge in pdg($\mathcal{P}$) from $\pi(T_i)$ to $\pi(H)$, for $0 < i \leq m$.

$$H \overset{\alpha_r}{\leftarrow} L; \langle f_d, f_p, f_c \rangle.$$
$$G \overset{\alpha_r}{\leftarrow} H, F, C; \langle f_d, f_p, f_c \rangle.$$
$$L \overset{\alpha_r}{\leftarrow} G, E; \langle f_d, f_p, f_c \rangle.$$
$$E \overset{\alpha_r}{\leftarrow} F; \langle f_d, f_p, f_c \rangle.$$
$$D \overset{\alpha_r}{\leftarrow} E, B; \langle f_d, f_p, f_c \rangle.$$
$$F \overset{\alpha_r}{\leftarrow} D, C; \langle f_d, f_p, f_c \rangle.$$
$$B \overset{\alpha_r}{\leftarrow} A; \langle f_d, f_p, f_c \rangle.$$
$$A \overset{\alpha_r}{\leftarrow} C; \langle f_d, f_p, f_c \rangle.$$
$$C \overset{\alpha_r}{\leftarrow} B; \langle f_d, f_p, f_c \rangle.$$

Figure 6.5: Program $P3$

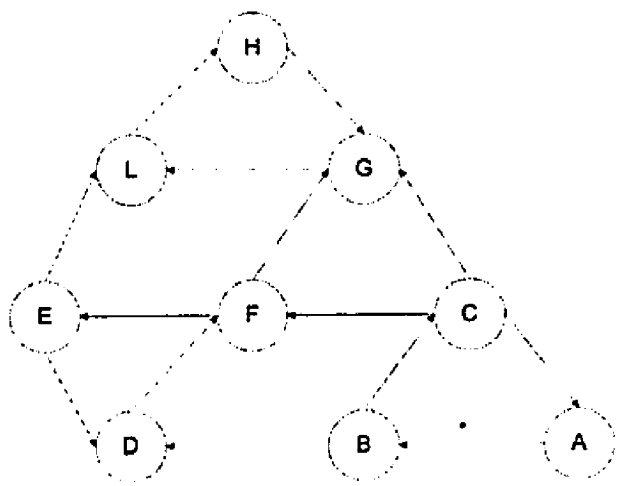

Figure 6.6: Predicate dependency graph of program $P3$

Definition 6.3.2. A Strongly Connected Component (SCC) of a directed graph G=(V, E) is a maximal set of vertices $C \subseteq V$, such that for every pair of vertices u and v in C, there is a path from u to v, and vice versa.

Stratification of a program contains two steps: first, it identifies SCCs in the predicate dependency graph of the program; second, it identifies the levels between every two SCCs, and then puts them into different levels accordingly.

We propose the following algorithm for step 1, which is slightly modified from Tarjan's algorithm [Tar72].

Algorithm 8 Strongly_Connected_Components

Input: $\mathcal{P}$

// $\mathcal{P}$ is the abstract representation of input program with certainty constraints.

Output: SCCs

// SCCs is a vector which contains SCC in the generating order.

1: construct predicate dependency graph G=pdg($\mathcal{P}$) from $\mathcal{P}$

2: perform a depth-first search to compute ending times e[u] for each node.

3: compute the transpose G^T of G.

4: call $DFS\_SCC(G^T)$, but in the main loop of DFS_SCC, consider the nodes in the order of decreasing e[u](as computed in line 2), generate strongly connected component, and then store them in vector SCCs.

Algorithm 8 basically includes two graph traversals: first, using a depth-first search method, it traverses all the edges and creates a depth-first spanning forest; second, it performs a depth-first search again and traverses the transpose[1] of the original graph based on the descending ending-time marked by the first DFS traversal. Once a so-called root of a SCC is found, its descendants which are not members of previously found components are recorded as a member of this component.

After running Algorithm 8 on the graph of Figure 6.6, it outputs a stratification consisting of three SCCs; SCC_1 contains vertices $\{H, L, G\}$, SCC_2 contains vertices $\{E, F, D\}$, and SCC_3 contains vertices $\{C, B, A\}$.

The final step of our stratification method is to stratify SCCs. The principle of stratification is to include all dependants of a predicate P in some stratum below the one which contains P.

[1]The transpose of a directed graph G=(V, E) is the graph G^T=(V, E^T), where E^T=$\{(v, u) \in V \times V : (u, v) \in E\}$. That is, G^T is G with all its edges reversed.

Algorithm 9 Stratify_SCC

Input: SCCs

// SCCs is a collection of Strongly Connected Components in the input program partitioned into strata.

Output: SCCs

// SCCs is a vector which contains SCC in a stratification order

1: $\forall X \in SCCs$ set X.level=0
2: **repeat**
3: **for all** $X \in SCC$ **do**
4: X.check=true
5: let $x \in \{y | y \in Y \wedge Y \in SCC\}$
6: **if** $x \in \{b_i | r : h \leftarrow b_1, b_2, \ldots, b_n \in R \wedge h \in X$,for i $\in \{1, \ldots, n\}\}$ **then**
7: Y.level=X.level+1
8: Y.check=false
9: **end if**
10: **end for**
11: **until** X.check=true, $\forall X \in SCC$.

In this algorithm, we initially set the level property of all strata to zero. If a member X of a SCC depends on a member Y of another SCC, the algorithm then moves the latter to a higher level. The process repeats until no SCC needs to be relocated. For instance, in Figure 6.6, the level property of SCCs $\{H, L, G\}$, $\{E, F, D\}$, $\{C, B, A\}$ are set to zero initially. Since F depends on C, which is a member of $\{C, B, A\}$, the level of $\{C, B, A\}$ is changed to 1. Similarly, the level of $\{E, F, D\}$ is changed to 1 because L depends on E. The SCC $\{E, F, D\}$ needs to be checked again because of the change in its level. Since F depends on C, which is a member of $\{C, B, A\}$, the level of $\{C, B, A\}$ is reset to 2, which is obtained from level of SCC $\{E, F, D\}$ plus one. Finally, no SCC needs to be relocated and the stratification is obtained as follows:

$$\textit{Stratum } 2: \quad \{C, B, A\}$$

$$\textit{Stratum } 1: \quad \{E, F, D\}$$

$$\textit{Stratum } 0: \quad \{H, L, G\}$$

Based on predicate stratification, we develop a mapping from predicate to rule stratification. Therefore, every rule defining predicates C, B or A is mapped to stratum 2; every rule defining E, F or D is mapped to stratum 1; and every rule defining H, L, and G is mapped to stratum 0, the lowest level.

6.4 Query Evaluation

There are numerous query processing techniques proposed in standard deductive databases. The two main approaches of such techniques are: top–down and bottom–up. In our work, we considered the bottom up, fixpoint evaluation, which derives "all" possible fact–certainty pairs from the program. This is done through a series of rule evaluations, each of which involves a series of join operations. We apply a materialization mechanism to perform these joins. We will discuss this in detail in section 6.4.1. The basic join mechanism used in our current implementation is nested-loops. During the joins, Argument Constraints are created to provide a matching pattern to filter out useless facts. Argument Constraints will be discussed in section 6.4.2. When stratification is used in our work, an input program may be divided into several strata, and, therefore, we should develop a corresponding stratified evaluation. We will discuss stratified evaluation in section 6.4.3. The fixpoint evaluation of programs with uncertainty may not terminate in finite time in general. To compute a desired model, the user can provide an error threshold for practical reasons. We will discuss this control mechanism in section 6.4.4.

6.4.1 Materialized Evaluation

The Materialized evaluation executes joins in a rule body as a sequential process. The join process is divided into several joins, each between two consecutive predicates, and

each is independent and completed before starting the next. A temporary relation is created to store the result of a join, which is then joined with the next predicate. The result is stored as another temporary relation. This process is repeated until the last predicate in the body is joined.

Borrowing the heuristic "push selection in join" used in performing join operation in relational database, we apply a select operation to the relation corresponding to the selected predicate. The selected tuples in the relation must not only follow the predicate Argument Constraint, but also Certainty Constraints of first type related to the predicate. The selected tuples are a subset of the original relation, called view. This select operation prunes unsuccessful joins in advance, and, hence, results in increased efficiency. This operation can be incorporated within materialized evaluation seamlessly.

6.4.2 Argument Constraint

There are two groups of Argument Constraints in our implementation. The first group is created from an individual predicate, which is used to obtain a subset of relations corresponding to the predicate, by filtering out the unsatisfiable tuples. The second group is created between two predicates, which is used to facilitate the join process. Consider the following rule:

$$p(X,Y) \overset{0.5}{\leftarrow} q(X,1,Y), s(Y,Z,Z), C_r; \langle max, pro, min \rangle.$$

Suppose EDB=$\{p(2,1,3) : 0.5, p(2,2,3) : 0.5, s(3,5,5) : 0.5, s(4,5,5) : 0.5, s(3,5,6) : 0.5\}$. Executing the join manually, we note that q(2,2,3): 0.5 will not participate in any successful join; only those tuples in relation q that have value "1" as the second argument are considered as candidates for the join. The so-called *constant constraint*

for q is that the value of its second argument must be 1.

In addition to *constant constraint*, there is another Argument Constraint in the first group, which is referred to as *equality constraint*. It is also clear that the fact s(3,5,6):0.5 never participates in a successful join, since its second and third arguments are different. This kind of fact also needs to be excluded from the join process. The *equality constraint* that requires the second and the third argument to be equal can filter out useless tuples. Like *constant constraint*, only a subset of relation "s" in which the tuples satisfy the equality constraint may be fruitful to keep.

The second group of Argument Constraint, to which we refer as *common argument constraint*, needs to be considered during the join process. As shown in the above example, the third argument Y of predicate q and the first argument Y of predicate s are the same. We thus create a *common argument constraint* for predicate q and s, based on the common argument Y. It means that the value of argument Y in predicates q and s need to be equal in order to have a successful join. Let us consider a mechanism to perform a join. To complete a join of two relations, all tuples on the left-hand-side need to be scanned one by one [HGMW02]. If a tuple t from the relation on the left is selected, then the common argument(s) would be bound by some value. Using this, all the tuples in the other subgoal are examined. Only the tuples that have the same bound arguments will be joined with tuple t.

6.4.3 Stratified Evaluation

Stratification partitions an input program into several strata, which should be evaluated stratum-by-stratum, from the highest stratum to the lowest. During the evaluation, each stratum is considered as an independent sub-program. Since the IDB predicates in the higher strata will not change when evaluating lower strata, such IDB predicates can be considered as EDB for some lower strata.

This stratified evaluation improves efficiency by avoiding computations of using intermediate certainty and by considering all the IDB predicates in higher strata as EDB for the lower strata. However, there are some restrictions on when stratification can be applied. When all the following three conditions hold, the stratification technique of an EGIB program may go wrong.

1. Data contains cycle.

2. Recursive predicates (direct recursive or indirect recursive)

3. Type 2 or 3 disjunctive function associated with IDB predicates.

Program $P_{6.2}$ satisfies the second and third conditions. When there is a cycle in the EDB, the stratified evaluation may go wrong. Program $P_{6.3}$ is a stratification of $P_{6.2}$. Both programs have the following EDB: $\{a(1,2) : 0.5, a(2,1) : 0.5, a(1,1) : 0.5, q(1) : 1.0\}$. Under precision control set to 10^{-3}, if we use the Naive method to evaluate these two programs, we get two different results:

$$lfp(P_{6.2}) = \{p(1,2) : 0.314746, p(1,1) : 0.3413093, p(2,2) : 0.03888607,$$
$$p(2,1) : 0.1623307, q(1) : 1.0, q(2) : 0.5\}$$
$$lfp(P_{6.3}) = \{p(1,2) : 0.314746, p(1,1) : 0.34158635, p(2,2) : 0.03888607,$$
$$p(2,1) : 0.16175783, q(1) : 1.0, q(2) : 0.5\}$$

p(X,Y)$\overset{0.5}{\leftarrow}$q(X), a(X,Y), wt(q(X))>0, wt(a(X,Y))>0;$\langle ind, pro, pro\rangle$.
q(Z)$\overset{1.0}{\leftarrow}$q(X), a(X,Z), wt(q(X))>0, wt(a(X,Z))>0; $\langle ind, pro, pro\rangle$.
p(X,Y)$\overset{0.5}{\leftarrow}$q(X), a(X,Z), p(Z,Y), wt(q(X))>0, wt(a(X,Z))>0, wt(p(Z,Y))>0;$\langle ind, pro, pro\rangle$.

Figure 6.7: Program $P_{6.2}$

Level 0
p(X,Y)$\overset{0.5}{\leftarrow}$q(X), a(X,Y), wt(q(X))>0, wt(a(X,Y))>0;$\langle ind, pro, pro\rangle$.
p(X,Y)$\overset{0.5}{\leftarrow}$q(X), a(X,Z), p(Z,Y), wt(q(X))>0, wt(a(X,Z))>0, wt(p(Z,Y))>0;$\langle ind, pro, pro\rangle$.
Level 1
q(Z)$\overset{1.0}{\leftarrow}$q(X), a(X,Z), wt(q(X))>0, wt(a(X,Z))>0; $\langle ind, pro, pro\rangle$.

Figure 6.8: Program $P_{6.3}$

Practically, if any of these above conditions is violated, then the stratification technique can be applied. For condition 2 or 3, deciding whether there is a violation is easy, but for condition 1, it may be expensive to determine if the data is cyclic, especially when the EDB is large. Furthermore, when the arity of a predicate is larger than 2, the definition of cycle in the data may become unclear, which makes deciding whether the data is cyclic even more difficult.

6.4.4 Precision Control

Evaluation of some EGIB programs may terminate only at ω. This may happen when the disjunction function associated with a recursive predicate is type 2 or type 3. The result returned by such disjunction functions is often "better" than its input arguments [LS01a], and a "better" certainty for a fact contributes to a derivation with better certainty. The underlying fixpoint operator is also monotone and continuous [LS01a]. This makes the least fixpoint of the program achieved most at ω. Evaluations of programs often terminate due to the finite precision of computer memory. We need alternative fixpoint evaluations to obtain an approximation of least fixpoint model. Therefore, we introduce the notion "precision control". This control essentially takes advantage of the continuity property of a fixpoint operator to reach the fixpoint. A certainty precision, represented as *precision*, is specified by the user. In other words, a certainty precision is also the maximum error range that can be tolerated.

For example, *precision*=10^{-3}. At each iteration, Our system checks whether the amount of certainty improved is greater than the user-specified precision. If so, it is considered as a new fact, and the evaluation continues to the next iteration; otherwise, the evaluation terminates, since we have already computed an approximate model.

Chapter 7

Experiments and Results

In order to measure the performance of the query processing techniques for dealing with constraints developed in our work, a number of experiments have been conducted under different EGIB programs. The evaluation time has been measured as the main parameter of performance. In our experiments, the following techniques for certainty constraints are implemented and evaluated:

- Extended Naive (EN): an extended *multiset-based* Naive evaluation technique, which extends *multiset-based* Naive evaluation proposed in [SZ04] to handle certainty constraints.

- Extended Semi-Naive (ESN): an extended *multiset-based* Semi-Naive evaluation technique by avoiding evaluation of rules that do not include subgoals with improved certainty.

- Extended Semi-Naive with Partition (ESNP): the Extended *multiset-based* Semi-Naive evaluation technique is further improved by partitioning the IDB subgoal into "improved" and "non-improved" parts to avoid redundant computation.

- Stratification: This technique partitions the input program into strata, and

evaluates them stratum by stratum. This technique is applied together with ESN or ESNP, thus called as ESN+S or ESNP+S.

7.1 Experiment Environment

For our experiments, we used Java running time environment (JRE) 1.5.0 installed on a regular Dell desktop computer with Pentium 4 CPU of 2.4GHz, 3.25G RAM, 189G hard disk, and runs under Windows XP professional 2003.

7.2 Test Programs and Benchmarks

Recursion is an attractive feature and power of standard logic program and deductive databases, and our system prototype inherits this feature. There are two kinds of recursions: linear and non-linear. In a linear recursion, the head predicate appears at most once in the body. In case there are more than one such subgoals, it is then called non-linear recursion.

Figure 7.1 is an example of a linear recursive program used in our experiments program. Program $P1$ compute the same-generation cousin (SGC) with uncertainty, which has been widely used for evaluating query optimization techniques in standard deductive databases [BR86, DMP93, RSS92, RSS90]. It defines the pairs of individuals which are in the same level of a family tree if their parents are in the same level. In $P1$, "person" and "par" are EDB predicates, and "sgc" is an IDB predicate.

r_1: $sgc(X,X) \xleftarrow{a} person(X)$, $wt(person(X)) > 0$,
$wt(person(X)) \geq wt(person(X)); \langle f_d, f_p, f_c \rangle$.
r_2: $sgc(X,Y) \xleftarrow{a} par(Z,X)$, $sgc(Z,W)$, $par(W,Y)$, $wt(par(Z,X)) > 0$,
$wt(sgc(Z,W)) \geq wt(sgc(Z,W))$; $\langle f_d, f_p, f_c \rangle$
. . {Relevant facts }

Figure 7.1: Linear SGC program $P1$ with uncertainty

Figure 7.2 shows an example of a nonlinear recursive program used in our experiments. This program $P2$ is slightly different from the SGC $P1$. Notice that program $P2$ cannot find all same generation pairs unless a specific path is satisfied in the second rule. This program was chosen in our performance evaluation because the second rule includes a long chain of EDB and IDB predicates in the body. In $P2$, "up", "flat", and "down" are EDB predicates, and "sgc" is an IDB predicate.

r_1: $sgc(X,Y) \xleftarrow{a} flat(X,Y),\ wt(flat(X,Y)) > 0; \langle f_d, f_p, f_c \rangle$.
r_2: $sgc(X,Y) \xleftarrow{a} up(X,Z1),\ sgc(Z1,Z2),\ flat(Z2,Z3),\ sgc(Z3,Z4),\ down(Z4,Y),$
$wt(up(X,\ Z1)) > 0,\ wt(flat(Z2,Z3)) > 0,\ wt(down(Z4,Y)) > 0,$
$wt(sgc(Z1,Z2)) \geq wt(sgc(Z1,Z2))\ ; \langle f_d, f_p, f_c \rangle$.
...{Relevant facts }

Figure 7.2: A Non-linear example of the SGC program $P2$

Each of the above two programs $P1$ and $P2$ has two rules. To evaluate the performance of the stratification technique, we define a N×1 structure which represents a SGC program holding N sets of rules. A N×1 SGC program has N sets of rules. Each set R_{i1}, where $1 \leq i \leq N$, contains two rules. If i = 1, R_{11} includes rules r_1^{11} and r_2^{11} shown in Figure 7.1 and Figure 7.2. If $1 < i \leq N$, R_{i1} also includes two rules: r_2^{i1} and a new rule below, denoted as r_3^{i1}:

$$r_3^{i1} : sgc_{i1}(X,Y) \xleftarrow{a} sgc_{(i-1)1}(X,Y); \langle max, min, min \rangle.$$

For example, a 2×1 EGIB program for $P1$ is shown in Figure 7.3, denoted as "$P1_{2\times1}$". In the graph, the program includes 4 rules divided into two sets and each set contains 2 rules. In the evaluation, sgc_{21} cannot be derived until sgc_{11} is prepared. We apply the combination functions "$\langle max, min, min \rangle$" in r_3 in order to initialize the certainty of sgc_{21} to 0.5 at the start point.

R_{21} :

$sgc_{21}(X,Y) \xleftarrow{ps} sgc_{11}(X,Y), C_1, <\max, \min, \min>$

$sgc_{21}(X,Y) \xleftarrow{ps} par(X,Z), sgc_{21}(Z,W), par(Y,W), C_2, <ind, pro, pro>$

R_{11}

$sgc_{11}(X,X) \xleftarrow{ps} person(X), C_1, <ind, pro, pro>$

$sgc_{11}(X,Y) \xleftarrow{ps} par(Z,X), sgc_{11}(Z,W), par(W,Y), C_2, <ind, pro, pro>$

Figure 7.3: A 2×1 structured program P1

7.3 Test Data Selection and Generation

A number of EDB data sets have been widely used to measure the efficiency of query processing and optimization techniques for standard deductive database [RSS94, DMP93, BR86, KNSSS90]. We adopt these data sets in our context of deductive databases with certainty constraints and develop program modules to generate suitable large test data sets with uncertainty. In total, we use 7 data sets in our experiments, described as follows.

(1) A_n : Figure 7.4 shows the structure of the data set A_n. It looks like a triangle constructed by layers of nodes. As the number of layers increases, the size of bottom layer increases. There is at most one matched path from a node to its same generation node. That is, if there is an inference for an answer Tuple, there is at most one match. To find a same generation node at level i, the derivation process goes up to the top-level layer and then down to layer i. The more layers we have, the more computation

is required to find the match.

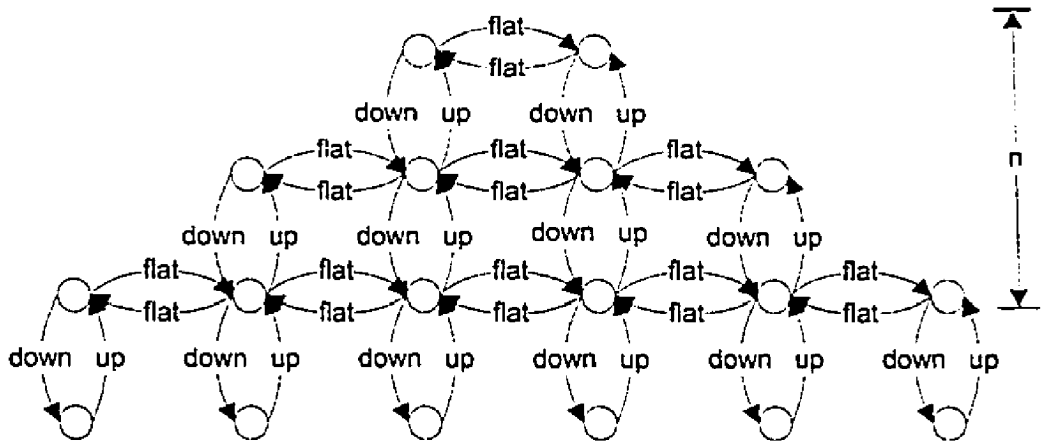

Figure 7.4: Data set A_n

(2) B_n : Figure 7.5 shows the structure of the data set B_n. It is constructed by n layers of nodes. Each layer has 8 nodes, which are connected by a double linked list. There are four edges connecting a lower layer and its immediate higher layer. Edges in column 1 and 5 are upward and those in columns 4 and 8 are downward.

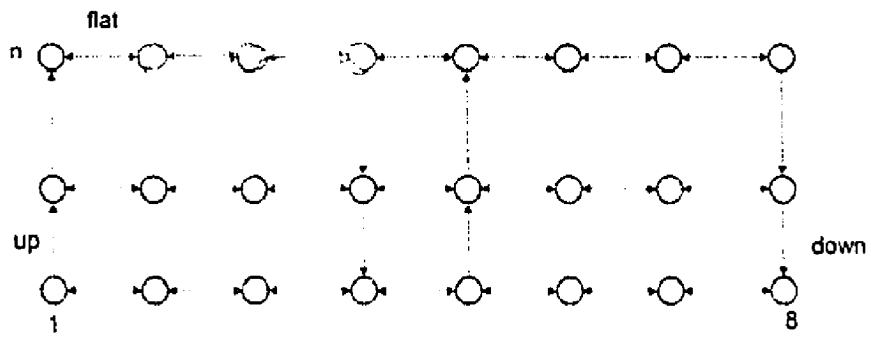

Figure 7.5: Data set B_n

(3) C_n : Figure 7.6 shows the structure of the data set C_n, which is very similar to the data set B_n. It has n layers, each of which is a single linked list of 8 nodes. Each node in C_n has a bi-directional edge connecting to the corresponding node in the immediate higher layer. Even though all nodes have an edge connecting the

corresponding node in higher layers, the number of derivations of $P2$ using the data set C_n may be less, compared to B_n. There exists no derivation from a node on the right to a node on the left since the flat edge are single directional.

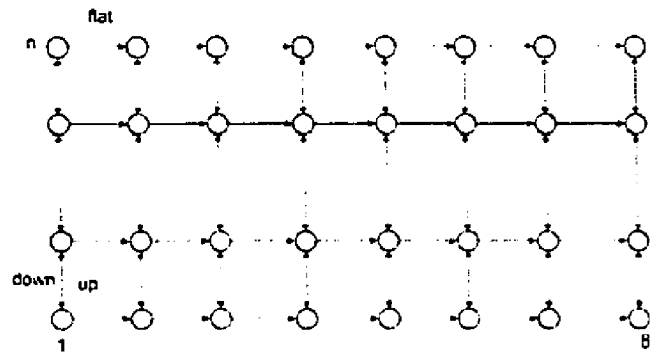

Figure 7.6: Data set C_n

(4) F_n : Figure 7.7 shows the structure of the data set F_n, which is a variant of C_n. Unlike C_n, the length of each layer and the number of layers in F_n are flexible and equal. This structure makes F_n a square shape. Each node in the lowest layer has an extra edge to the corresponding node at each higher layer. The number of query answers increases when n increases.

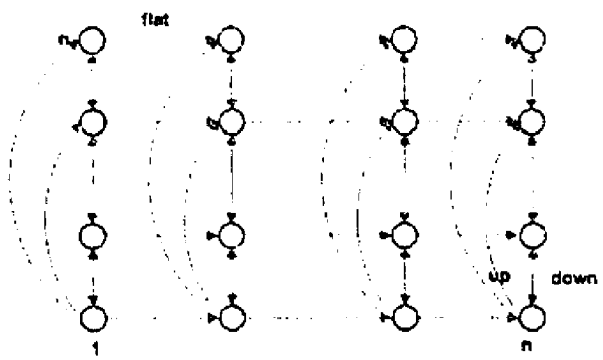

Figure 7.7: Data set F_n

(5) S_n : Figure 7.8 shows the structure of the data set S_n. It includes n upward linked lists and n downward linked lists. There is only one link between two nearest

linked lists. It is easy to see that there is only one path from node *1* to its corresponding same generation node *2n*. The derivation proceeds recursively from top to bottom.

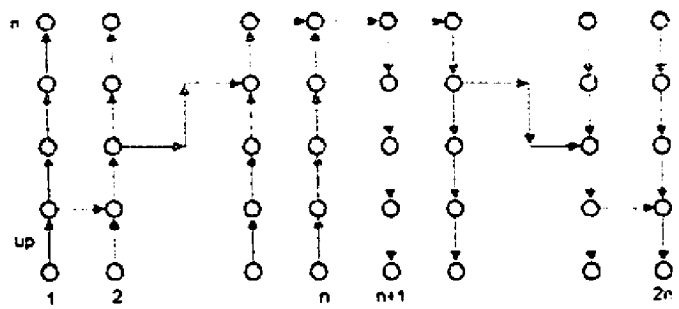

Figure 7.8: Data set S_n

(6) $T_{n,m}$: Figure 7.9 shows the structure of the data set $T_{n,m}$. It is an m-ary tree of height n with m children per node. Figure 7.9 is an example of $T_{2,3}$. When the height n increases by 1, the total number of edges in the trees increases by $3m^{n-1}$. That is, the data set grows exponentially in the height.

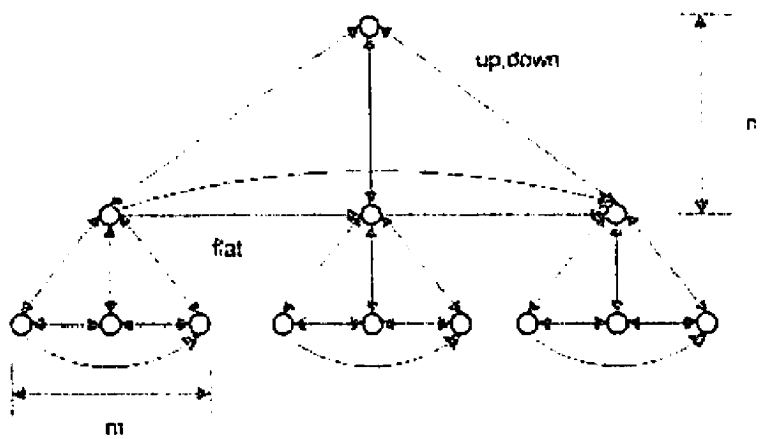

Figure 7.9: Data set $T_{n,m}$

(7) $U_{n,m}$: Figure 7.10 shows the structure of the data set $U_{n,m}$, which is a variant of C_n. It has n layers and each layer has m nodes. Each layer is linked by a cycle list.

There are more connections between two nearest layers than C_n. From Figure 7.10, we can see that each node has two bi-directional edges to nodes at its immediate higher layer. The number of derivations in a given program increases larger since the paths between two same generations nodes increase.

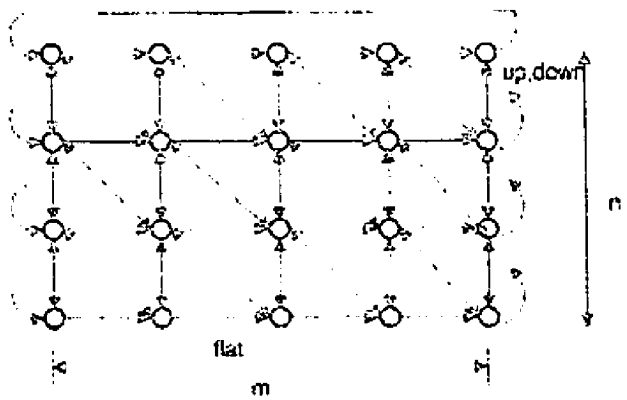

Figure 7.10: Data set $U_{n,m}$

7.4 CC-checker Performance Evaluation

To measure the overhead of adding CC-Checker to the Naive algorithm, we implemented Naive algorithm in [SZ04] and compared its performance with our Extended-Naive (EN) algorithm. If we remove the certainty constraints from $P1$ and $P2$, they become $P1'$ and $P2'$, respectively. We ran $P1$, $P2$, $P1'$, and $P2'$ on the data sets introduced in previous section 7.3. The execution time is used as the performance parameter. These evaluation results will be reported in this section. Column "Data set" in each result table denotes the type of data set used. Column "Iterations" gives the number of iterations needed to reach the fixpoint. Column "Output" shows the number of answer tuples found for the query. Columns "EN" and "N" correspond to the execution time (in seconds) when using Extend-Naive algorithm and Naive algorithm, respectively. The ratio (EN-N)/N of execution time indicates the overhead of

execution time, shown as "overhead ratio" in the last column of each result table.

Data set	Iterations	Input	Output	EN	N	overhead ratio
A_9	7	342	793	1.547	1.485	4.18%
A_{10}	7	420	1024	2.219	2.141	3.64%
A_{11}	7	506	1294	3.406	3.312	2.84%
A_{12}	7	600	1607	5.094	4.984	2.21%
A_{13}	8	702	1965	9.406	9.188	2.37%

Table 7.1: CC-Checker overhead evaluation: running $P1$ and $P1'$ on A_n

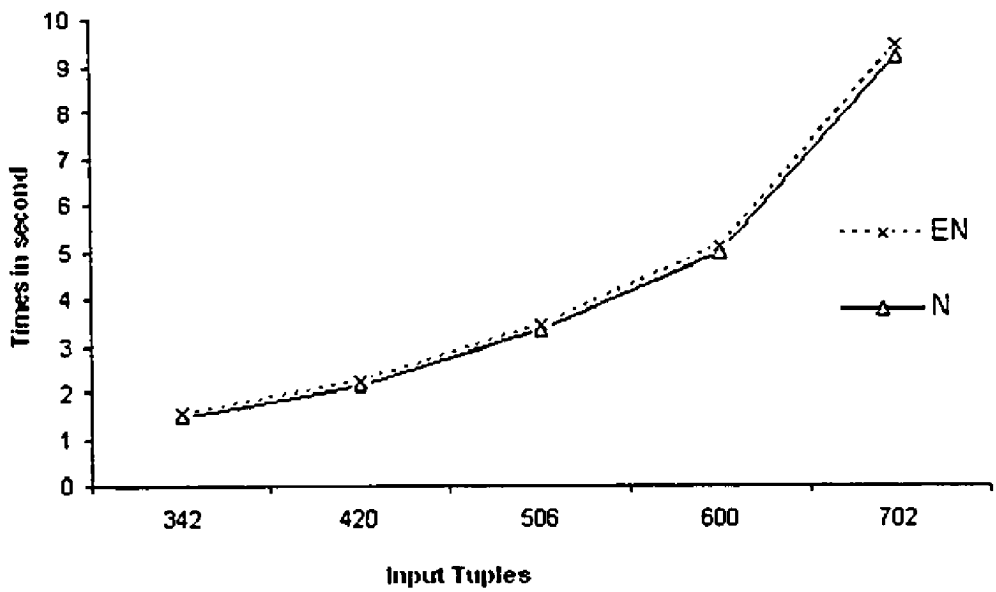

Figure 7.11: CC-Checker overhead evaluation: running $P1$ and $P1'$ on A_n

Table 7.1 and Figure 7.11 show the experiment results of running $P1$ and $P1'$ on different size of data set A_n. The certainty assigned to each EDB fact is 0.5. The disjunction function is *ind*, and propagation and conjunction functions are *pro* (the product). The results indicate a range of overhead from 2.21% to 4.18%. As expected, the overhead of adding CC-Checker to Naive algorithm is not large, and in fact, reasonable. Table 7.1 also shows that as the size of input data set A_n increases, the number of derived facts increases as well.

Data set	Iterations	Input	Output	EN	N	overhead ratio
$U_{5,10}$	4	210	560	0.719	0.703	2.28%
$U_{10,10}$	4	460	1310	2.906	2.781	4.49%
$U_{12,10}$	4	560	1610	4.0	3.828	4.49%
$U_{15,10}$	4	710	2060	5.875	5.641	4.15%
$U_{20,10}$	4	960	2810	9.671	9.375	3.16%

Table 7.2: CC-Checker overhead evaluation: running $P1$ and $P1'$ on $U_{n,m}$

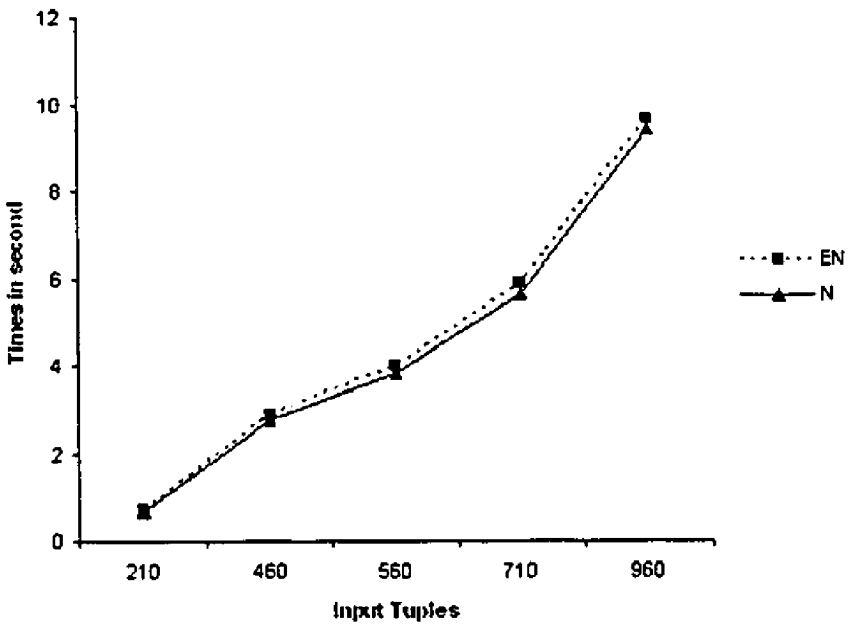

Figure 7.12: CC-Checker overhead evaluation: running $P1$ and $P1'$ on $U_{n,m}$

Table 7.2 and Figure 7.12 show the experiment result of running $P1$ and $P1'$ on data set $U_{n,m}$. The certainty assigned to each EDB fact is 0.5. The combination functions used are (ind, pro, pro). As indicated in the table, the ranges from 2.28% to 4.49%. Generally speaking, the overhead ratio decreases as the input data set increases.

We ran $P1$ and $P1'$, $P2$ and $P2'$ on the data sets proposed in section 7.3. Each data set has 5 test cases with different sizes. The result is reported in Table 7.3, in which we only show data sets and corresponding overhead ratio observed. We

Data set	overhead ratio	Data set	overhead ratio
C_8	6.90%	T_{16}	6.82%
C_{16}	1.08%	T_{32}	4.88%
C_{32}	5.39%	T_{64}	5.92%
C_{64}	4.14%	T_{128}	5.18%
C_{128}	2.24%	T_{256}	3.95%
S_{32}	6.67%	$U_{5,10}$	2.28%
S_{64}	6.85%	$U_{10,10}$	4.49%
S_{128}	6.28%	$U_{12,10}$	4.49%
S_{256}	5.44%	$U_{15,10}$	4.15%
S_{512}	2.93%	$U_{20,10}$	3.16%
F_8	6.78%	F_{40}	4.87%
F_{16}	4.43%	F_{64}	3.68%
F_{32}	3.67%		

Table 7.3: CC-Checker overhead evaluation

note that all the test cases yield more or less similar overhead for dealing with CC-Checker in Naive algorithm, in despite of data size. The results in this table show that the overhead of adding CC-Checker is acceptable, and the Naive algorithm is still scalable with addition of CC-Checker. Furthermore, we observe that the larger the set of derived facts generated, the less overhead we have for handling certainty constraints. The evaluation of EN algorithm require more time than Naive algorithm for all types of data sets.

7.5 EN, ESN and ESNP Performance Evaluation

A main component in this research is the performance evaluation of our extended algorithms of ESN and ESNP for logic programs with certainty constraints. We designed and implemented these algorithms in our system prototype, and conducted numerous experiments by running $P1$ and $P2$ on various data sets of different size introduced earlier in this section. As a benchmark as well as basis for correctness, we also evaluated EN algorithm using the same test cases. To evaluate the CC-Checker,

the execution time is used as the performance measure again. The table structure and discussion is the same as discussed in section 7.4. In addition, we add three more columns. Column "ESNP" corresponds to the execution time when using the ESNP algorithm. The time ratios EN/ESN and EN/ESNP of execution time are indicated in Column "Speed-Up (EN/ESN)" and Column "Speed-Up (EN/ESNP)", respectively.

Data set	Iterations	Output	EN	ESN	ESNP	Speed-Up (EN/ESN)	Speed-Up (ESN/ESNP)
A_9	7	451	1.578	1.422	0.75	1.11	2.10
A_{10}	7	604	2.265	2.187	1.156	1.04	1.96
A_{11}	7	788	3.547	3.391	1.719	1.05	2.06
A_{12}	7	1007	5.272	5.125	2.547	1.03	2.03
A_{13}	8	1263	9.753	9.5	4.172	1.03	2.28

Table 7.4: ESN and ESNP performance: running P1 on A_n

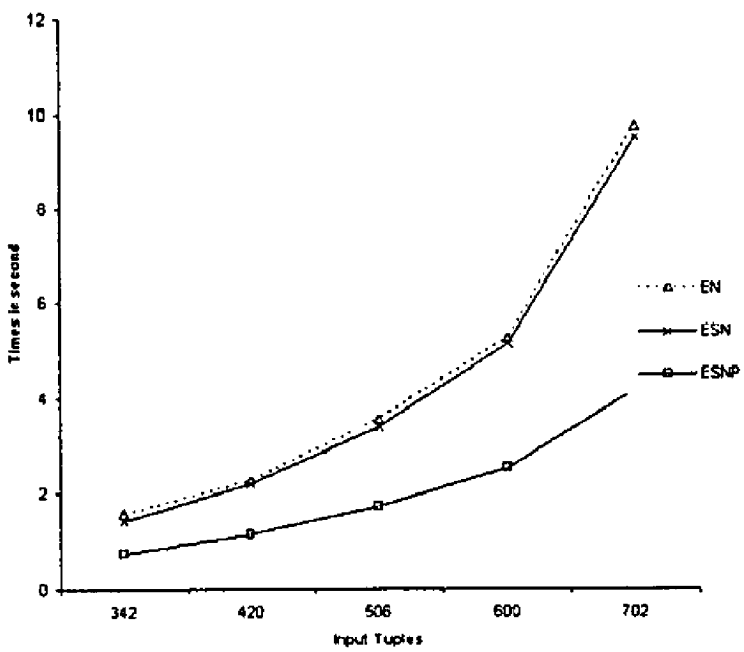

Figure 7.13: ESN and ESNP performance: running $P1$ on A_n

The experimental result of running $P1$ on data set A_n is shown in Table 7.4 and Figure 7.13. The ESN algorithm is 5% faster than EN algorithm. We ran programs $P1$ and $P2$ on the data sets described in section 7.3, and used five test cases in each data set. These results are shown on Table 7.5 in a compact form, which only shows data sets name, speed-up of ESN over EN, and speed-up of ESNP over ESN. As can be seen from this table, the speed-up achieved by ESN over EN ranges from 1.01 to 2.66, and the speed-up achieved by ESNP over ESN ranges from 1.02 to 3.13.

From these experimental results, we find that the efficiency improvement of ESN over EN is not significant, since it only avoids the computation of rule:

$$sgc(X, X) \stackrel{1.0}{\leftarrow} person(X), wt(person(X)) > 0,$$
$$wt(person(X)) \geq wt(person(X)); \langle f_d, f_p, f_c \rangle$$

but the time spent on the computation of this rule takes only a small fraction of the total computation, so the execution time of ESN and EN are very close to each other. The efficiency however, is further improved by ESNP. The speed-up achieved by ESNP over ESN is about 2.65. We also observe that the speeds-up of ESN over EN, and ESNP over ESN may be different under different test cases. ESNP results in greater efficiency compared to both EN and ESN for most test cases. We remark that the trends of ESN over EN, and ESNP over ESN are very similar to trends of SN over N and SNP over SN, as reported in [SZ04].

7.6 Stratification Performance Evaluation

Another objective of this research was to evaluate the performance of our stratification technique. For this, we use structured programs $P1_{i\times1}$ to test the various data sets introduced earlier. First, we remark that the stratification technique needs to be

Test case	Speed-Up (ESN/EN)	Speed-Up (ESNP/ESN)	Input Tuples
B_8	2.66	1.47	140
B_{16}	0.99	1.21	284
B_{32}	1.12	1.23	572
B_{64}	1.08	1.30	1148
B_{128}	1.04	1.30	2300
C_8	1.00	1.94	168
C_{16}	1.06	1.41	352
C_{32}	1.05	1.24	720
C_{64}	1.00	1.14	1456
C_{128}	1.03	1.18	2928
F_8	1.24	1.02	224
F_{16}	1.01	1.27	960
F_{32}	1.04	1.57	3968
F_{40}	1.05	2.17	6240
F_{64}	1.09	1.43	16128
S_{32}	1.52	1.00	1567
S_{64}	1.16	1.21	6207
S_{128}	1.12	1.03	24703
S_{128}	1.12	1.03	24703
S_{256}	1.13	0.98	98559
$U_{5,10}$	0.09	1.12	210
$U_{10,10}$	1.00	1.06	460
$U_{12,10}$	1.00	1.07	560
$U_{15,10}$	1.03	1.10	710
$U_{20,10}$	1.00	1.06	960
T_{16}	1.00	3.13	61
T_{32}	1.14	2.44	125
T_{64}	1.01	2.49	253
T_{128}	1.07	2.86	509
T_{256}	1.02	2.35	1021

Table 7.5: ESN and ESNP performance evaluation

used in conjunction with ESN or ESNP. We used the EN algorithm as a basis for correctness benchmark. The execution time is used as the performance parameter.

Data set	Iterations	Output	EN	ESN +S	ESNP +S	Speed-Up ($\frac{EN}{ESN+S}$)	Speed-Up ($\frac{EN}{ESNP+S}$)
A_9	7	451	1.562	1.406	0.766	1.11	2.04
A_9	8	902	3.203	2.328	1.484	1.38	2.16
A_9	9	1353	5.375	2.641	1.813	2.04	2.96
A_9	10	1804	7.735	2.922	2.172	2.65	3.56
A_9	11	2255	10.641	3.234	2.5	3.29	4.26

Table 7.6: ESN and ESNP with Partition performance: running $P1_{i\times1}$ on A_9

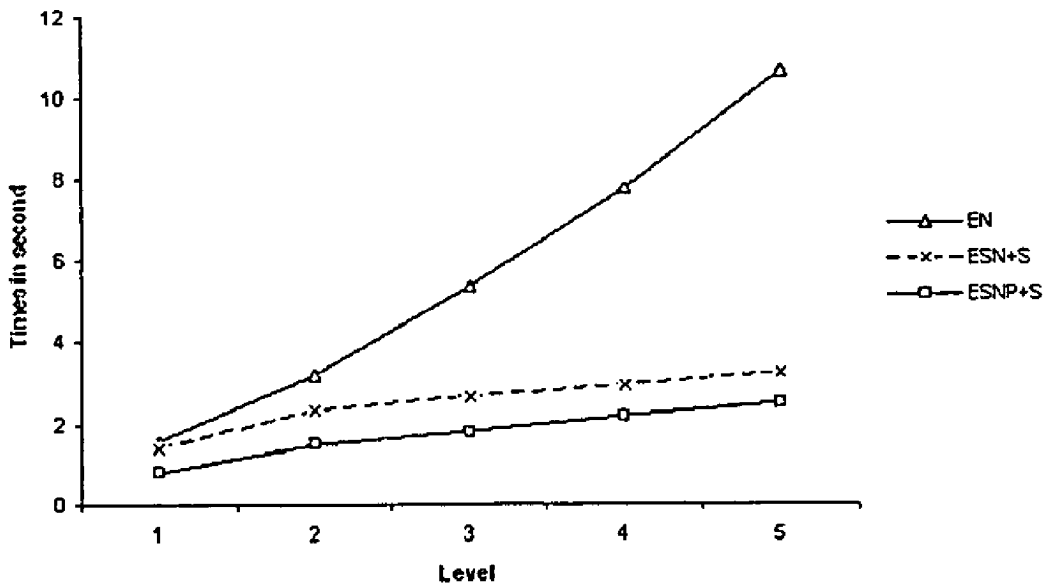

Figure 7.14: Stratification performance: running $P1_{i\times1}$ on A_9

Table 7.6 and Figure 7.14 show the experiments result of running programs $P1_{i\times1}$ on the data set A_9. The speed-up of ESN with stratification over EN ranges from 1.11 to 3.29, and the speed-up of ESNP with stratification over EN ranges from 2.04 to 4.26. We find that the speed-up increases when the number of strata of the programs increases, as expected by our analysis results.

Test case	Speed-Up (EN/ESN+S)	Speed-Up (EN/ESNP+S)	Test case	Speed-Up (EN/ESN+S)	Speed-Up (EN/ESNP+S)
B_{64}	1.03	1.30	S_{64}	1.00	1.26
B_{64}	1.41	1.60	S_{64}	1.71	1.71
B_{64}	2.02	2.26	S_{64}	2.13	2.00
B_{64}	2.59	2.82	S_{64}	2.74	2.60
B_{64}	3.13	3.34	S_{64}	3.17	3.17
C_{64}	1.01	1.18	T_{16}	1.32	2.00
C_{64}	1.72	1.91	S_{64}	1.38	2.21
C_{64}	2.34	2.51	S_{64}	1.28	2.25
C_{64}	2.92	3.09	S_{64}	1.77	2.46
C_{64}	3.47	3.63	S_{64}	2.13	3.41
F_{16}	1.01	1.31	$U_{5,10}$	0.94	1.06
F_{16}	1.32	2.06	$U_{5,10}$	1.08	1.21
F_{16}	1.90	2.54	$U_{5,10}$	1.57	1.72
F_{16}	2.48	3.25	$U_{5,10}$	2.05	2.23
F_{16}	2.96	3.67	$U_{5,10}$	2.56	2.73

Table 7.7: Stratification performance evaluation

We ran these set of programs with strata from 1 to 5 on data sets B_{64}, C_{64}, F_{16}, and $T_{4,4}$, respectively. The results are shown in Table 7.7. The speed-up of ESN with stratification over EN ranges from 1 to 3.47, and the speed-up of ESNP with stratification over EN ranges from 1.06 to 3.67. The ESNP with stratification perform better than ESN with stratification in every case. The efficiency improvement is more visible when the number of strata is large.

These experiments showed that our implementation of query processing techniques, which deal with the presence of certainty constraints in the rule body, are correct and reasonably efficient. What is also important to note is that the algorithm proposed and implemented provide a uniform environment to evaluate and experiment with logic programs and deductive databases of AB and IB approach at the same time. This uniform evaluation mechanism is useful towards developing tools for uncertainty reasoning.

Chapter 8

Conclusion and Future Research

In order to be able to handle real-life applications, it is essential not only to model and reason with uncertainty, but also do this efficiently with massive amounts of data [SSU91]. The goal of this research was to develop query processing and optimization techniques for dealing with certainty constraints. In this chapter, the work done towards this goal and the contributions of the thesis is summarized. We also provide possible directions to explore.

To reach our goal, we studied the *Certainty Constraints* as a key concept introduced in [Shi05], which relates the IB and AB approaches to uncertainty. The Parametric Framework [LS96, LS01b], which unifies and generalizes the existing IB frameworks to uncertainty in logic programming and deductive databases. With the presence of *Certainty Constraints* in the rule bodies, the Multiset-based Naive, Multiset-based Semi-Naive and Multiset-based Semi-Naive with Partition algorithms need to be extended accordingly to handle certainty constraints. Motivated by this, we developed these extended evaluation schemes with support for checking certainty constraints.

We also proposed stratified evaluation in our context, which originates from Datalog$^{\neg}$

to further improve query processing, by avoiding computation of intermediate certainties for some classes of programs. A desired stratification algorithm is defined for stratified programs which proceeds stratum-by stratum to compute the least fixpoint.

In addition to efficient evaluation of programs, we also studied and developed transformation modules between the EGIB and the EGAB frameworks, based on the study in [Shi05] which established equivalence of the two approaches in terms of expressive power when certainty constraints are allowed in both frameworks. Through this transformation, an EGAB program can be transformed into an equivalent EGIB program, and vice versa, which is then evaluated by our system efficiently.

To study the performance of our system, we designed and implemented extended fixpoint algorithms which handles certainty constraints. A number of experiments have been conducted to measure the benefits of the proposed techniques. Our experiments and results show that the overhead of adding CC-Checker to evaluation algorithms is reasonable. The extended algorithm ESNP results in greater efficiency compared to both extended Naive (EN) and semi-naive (ESN) for most test cases, and does not perform worse than ESN in any case. The scalability of ESN and ESNP to larger data sets are promising in our results. The efficiency has been further improved for some classes of programs by applying stratification technique with ESN or ESNP methods.

Our system currently supports in-memory data and runs in a stand-alone mode. We plan to extend it to support disk-resident data. Much of the design effort should focus on buffer management.

Another research direction of the query optimization is to bring in "Magic set" rewriting technique into the EGIB framework. In [Hua08], "Magic set" rewriting technique was incorporated within an implementation of the parametric framework,

which results in great efficiency improvement, obtained by extending from the standard case. To further bring "Magic set" rewriting into EGIB framework, we need to pay more attention to the certainty constraints when rewrite the rules.

Bibliography

[ABW88] K. R. Apt, H. A. Blair, and A. Walker. Towards a theory of declarative knowledge. pages 89–148, 1988.

[Ban86] Francois Bancilhon. Naive evaluation of recursively defined relations. pages 165–178, 1986.

[BR86] Francois Bancilhon and Raghu Ramakrishnan. An amateur's introduction to recursive query processing strategies. *SIGMOD Rec.*, 15(2):16–52, 1986.

[CGL86] Stefano Ceri, Georg Gottlob, and Luigi Lavazza. Translation and optimization of logic queries: The algebraic approach. In *VLDB '86: Proceedings of the 12th International Conference on Very Large Data Bases*, pages 395–402, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.

[CGT89] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989.

[CGT90] Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic programming and databases.* Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[Dep] Peter Sestoft Department. Grammars and parsing with java.

[DLP91] Didier Dubois, Jérôme Lang, and Henri Prade. Towards possibilistic logic programming. In *ICLP*, pages 581–595, 1991.

[DMP93] Marcia A. Derr, Shinichi Morishita, and Geoffrey Phipps. Design and implementation of the glue-nail database system. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 147–156, Washington, D.C., USA, 1993.

[ea05] Serge Abiteboul et al. The lowell database research self-assessment. *Commun. ACM*, 48(5):111–118, 2005.

[Fit88] Melvin Fitting. Logic programming on a topological bilattice. *Fundamental Information*, 11:209–218, 1988.

[Fit91] Melvin Fitting. Bilattices and the semantics of logic programming. *Journal of Logic Programming*, 11(1&2):91–116, 1991.

[GKT91] Ulrich Guntzer, Werner Kiesling, and Helmut Thone. New directions for uncertainty reasoning in deductive databases. In *SIGMOD Conference*, pages 178–187, 1991.

[HGMW02] Jeff Ullman Hector Garcia-Molina and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall, 2002.

[Hua08] Qiong Huang. Extending magic sets techniques to deductive databases with uncertainty. Master's thesis, Montreal, QC, Canada, 2008.

[KL88] M. Kifer and A. Li. On the semantics of rule-based expert systems with uncertainty. In *Lecture notes in computer science on ICDT '88*, pages 102–117, New York, NY, USA, 1988. Springer-Verlag New York, Inc.

[KNSSS90] Juhani Kuittinen, Otto Nurmi, Seppo Sippu, and Eljas Soisalon-Soininen. Efficient implementation of loops in bottom-up evaluation of logic queries. In *VLDB '90: Proceedings of the 16th International Conference on Very Large Data Bases*, pages 372–379, 1990.

[KS92] Michael Kifer and V. S. Subrahmanian. Theory of generalized annotated logic programming and its applications. *J. Log. Program.*, 12(4):335–367, 1992.

[LL96] Sonia M Leach and James J Lu. Query processing in annotated logic programming: Theory and implementation. *Journal of Intelligent Information Systems*, 6(1):33–58, 1996.

[Llo87] J. W. Lloyd. *Foundations of logic programming; (2nd extended ed.).* Springer-Verlag New York, Inc., New York, NY, USA, 1987.

[LS94a] Laks V. S. Lakshmanan and Fereidoon Sadri. Modeling uncertainty in deductive databases. In D. Karagiannis, editor, *Proc. Int. Conf. on Database and Expert Systems Applications, DEXA'94, Athens, Greece*, volume 856, pages 724–733, 1994.

[LS94b] Laks V. S. Lakshmanan and Fereidoon Sadri. Probabilistic deductive databases. In *Symposium on Logic Programming*, pages 254–268, 1994.

[LS96] Laks V. S. Lakshmanan and Nematollaah Shiri. A parametric approach to deductive databases with uncertainty. In *LID '96: Proceedings of the International Workshop on Logic in Databases*, pages 61–81, London, UK, 1996. Springer-Verlag.

[LS01a] Laks V. S. Lakshmanan and Nematollaah Shiri. A parametric approach to deductive databases with uncertainty. *IEEE Transactions on Knowledge and Data Engineering*, 13(4):554–570, 2001.

[LS01b] Laks V. S. Lakshmanan and Nematollaah Shiri. A parametric approach to deductive databases with uncertainty. *IEEE Transactions on Knowledge and Data Engineering*, 13(4):554–570, 2001.

[NS92] Raymond T. Ng and V. S. Subrahmanian. Probabilistic logic programming. *Information and Computation*, 101(2):150–201, 1992.

[NS93] Raymond T. Ng and V. S. Subrahmanian. A semantical framework for supporting subjective and conditional probabilities in deductive databases. *J. Autom. Reasoning*, 10(2):191–235, 1993.

[RSS90] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Rule ordering in bottom-up fixpoint evaluation of logic programs. In *Proceedings of the 16th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Brisbane*, 1990.

[RSS92] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Efficient bottom-up evaluation of logic programs, 1992.

[RSS94] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Rule ordering in bottom-up fixpoint evaluation of logic programs. *IEEE Trans. on Knowl. and Data Eng.*, 6(4):501–517, 1994.

[RSSS94] Raghu Ramakrishnan, Divesh Srivastava, S. Sudarshan, and Praveen Seshadri. The CORAL deductive system. *VLDB Journal: Very Large Data Bases*, 3(2):161–210, 1994.

[Sha00] Clifford A. Shaffer. *A Practical Introduction to Data Structures and Algorithm Analysis.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[Shi05] Nematollaah Shiri. Expressive power of logic frameworks with certainty constraints. In *18th Int'l FLAIRS Conference, Special Track on Uncertainty Reasoning*, pages 759–765, Florida, USA, 2005.

[SSU91] Avi Silberschatz, Michael Stonebraker, and Jeff Ullman. Database systems: achievements and opportunities. *Commun. ACM*, 34(10):110–120, 1991.

[SSW94] Konstantinos F. Sagonas, Terrance Swift, and David Scott Warren. Xsb as an efficient deductive database engine. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 442–453, Minneapolis, Minnesota, USA, 1994. ACM Press.

[Sub87] V. S. Subrahmanian. On the semantics of quantitative logic programs. In *SLP*, pages 173–182, 1987.

[SV97] Nematollaah Shiri-Varnaamkhaasti. *Towards a generalized theory of deductive databases with uncertainty.* PhD thesis, Montreal, P.Q., Canada, Canada, 1997.

[SZ04] Nematollaah Shiri and Zhi Hong Zheng. Challenges in fixpoint computation with multisets. In *In Proc. 3rd Int'l Symp. Foundations of Information and Knowledge Systems (FoIKS)*, pages 273–290, Vienna, Austria, 2004.

[SZ08] Nematollaah Shiri and ZhiHong Zheng. Optimizing fixpoint evaluation of logic programs with uncertainty. In *Proc. 13 CSI Int'l Comp. Conf. (CSICC)*, Kish, Iran, 2008.

[Tar72] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[Ull89] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I.* Computer Science Press, 1989.

[vE86a] Maarten H. van Emden. Quantitative deduction and its fixpoint theory. *Journal of Logic Programming*, 3(1):37–53, 1986.

[vE86b] Maarten H. van Emden. Quantitative deduction and its fixpoint theory. *J. Log. Program.*, 3(1):37–53, 1986.