EFFICIENT QUERY PROCESSING IN DEDUCTIVE DATABASES: THE *LogicBase* APPROACH

by

Ling Liu

B.Sc., Tsinghua University, China, 1986

M.Sc., Tsinghua University, China, 1988

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
in the School
of
Computing Science

SIMON FRASER UNIVERSITY
March 1995

APPROVAL

Name: Ling Liu

Degree: Doctor of Philosophy

Title of thesis: EFFICIENT QUERY PROCESSING IN DEDUCTIVE DATABASES: THE *LogicBase* APPROACH

Examining Committee: Dr. Ze-Nian Li
Chair

Dr. Jiawei Han, Senior Supervisor

Dr. Woshun Luk, Supervisor

Dr. William Havens, Supervisor

Dr. Laks V.S. Lakshmanan, External Examiner

Dr. Tiko Kameda, S.F.U. Examiner

Date Approved: February 3, 1995

SIMON FRASER UNIVERSITY

PARTIAL COPYRIGHT LICENSE

Title of Thesis/Project/Extended Essay

Efficient Query Processing in Deductive Databases: The Logic Base Approach.

Author: _____

(signature)

Ling Liu

(name)

March 30, 1995

(date)

To Connie

Abstract

The thesis investigates the design and implementation of a deductive database system prototype, *LogicBase*, and several query processing and optimization techniques in deductive database systems.

LogicBase adopts the compilation-based query processing method, where logic programs are compiled into highly regular forms. A query is evaluated on the compiled form by performing iterative relational operations. *LogicBase* facilitates a detailed query analysis to select an appropriate query evaluation strategy and to generate an efficient query processing plan, thus to achieve declarativeness and efficiency.

An important feature of *LogicBase* is its ability to handle constraints, which are manipulated to determine the safety of an evaluation plan and to reduce search space in some expensive query processing. Constraints and monotonicity are investigated to benefit constraint pushing and derivation. Moreover, interaction of constraints with different programs is studied.

A set of query processing strategies are proposed for efficient evaluation of programs with multiple linear recursions, which extends the method of handling single linear recursions by accessing the union of separate relations. Furthermore, an extension of counting method to deal with cyclic data path is devised, which transforms the counting method into the propagations of relative distances over cyclic paths in a directed acyclic graph constructed from strongly connected components of the original data graph.

Acknowledgements

I am very grateful to my senior supervisor, Dr. Jiawei Han, who lent me a great deal of support, encouragement and pleasant cooperation. I would like to thank Dr. Tiko Kameda and Dr. Laks V.S. Lakshmanan who spent extra effort to help me generate a better thesis. I would also like to express my appreciation for Dr. Woshun Luk and Dr. William Havens, who made valuable suggestions and comments.

Last but not the least, I would like to express my gratefulness to my wife and parents, who have provided all their love and support through the years of my graduate study.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

A deductive (or logic) database system combines merits of relational database systems and logic programming by taking logic as its data model and retaining query processing efficiency in relational databases. It extends the expressive power of a declarative query language from SQL to general logics.

Significant research efforts have been spent on query evaluation and optimization in deductive databases. Many methods and strategies have been proposed and implemented. Several deductive database systems have been developed. Our research at Simon Fraser University has been on the deductive database query evaluation and optimization, emphasizing the declarativeness and efficiency of query processing. A deductive database system prototype, *LogicBase*, has been designed and implemented. The initial implementation and experimentation have shown that the approach taken in *LogicBase* has many advantages over and offers an alternative to the other currently available deductive database query evaluation techniques.

Many issues concerning query processing in deductive databases, especially those in *LogicBase*, are investigated in the thesis.

In this chapter, the background and some practice for query processing in deductive databases are reviewed.

1.1 Deductive database: a first look

Similar to relational databases, data in deductive databases are stored in relations. However, not all relations need to be stored physically. Some relations are defined by logic rules such that data can be derived from other physically stored data. The primitive concept for deductive databases can be first introduced by a simple example in Figure 1.1.

$$
\begin{aligned}
&parent(aaron, brian).\\
&parent(brian, fred).\\
&parent(coleen, eve).\\
&parent(brian, greg).\\
&ancestor(X,Y) \;:-\; parent(X,Y). &(1.1)\\
&ancestor(X,Y) \;:-\; parent(X,Z), ancestor(Z,Y). &(1.2)
\end{aligned}
$$

Figure 1.1: The ancestor example.

In Figure 1.1, predicate $parent(aaron, brian)$ represents a fact that "brian" is a parent of "aaron". All of the *parent* predicates can be considered as tuples in a relation called *parent* and are physically stored in a database. However, tuples in relation *ancestor* are derived from two logic rules. Rule (1.1) states that one's parent is his or her ancestor, while Rule (1.2) states that one's parent's ancestor is also his or her ancestor. Relation *ancestor* can be derived from relation *parent*, which consists of following tuples: $ancestor(aaron, brian)$, $ancestor(brian, fred)$, $ancestor(coleen, eve)$, $ancestor(brian, greg)$, $ancestor(aaron, fred)$, and $ancestor($ *arron*, *greg*$)$. An inference engine is responsible to deduce facts implied by logic rules and to answer queries imposed on a derived relation.

In a deductive database, a relation stored physically is called an *extensional database* (EDB) relation or a *base relation.* A relation defined through a set of logic rules is called an *intensional database* (IDB) relation or a *derived relation.*

1.2 Extension to the relational approach and Prolog

A deductive database system extends a relational database system in expressiveness. A derived relation in deductive databases can be defined recursively, such as the *ancestor* relation in Figure 1.1. Many problems can be expressed and solved using recursively defined logic rules, which are not expressible in a relational database system.

Research in logic programming has contributed to the initial studies of deductive databases. A version of Prolog called *Datalog*, which uses function-free Horn clauses, is taken in deductive databases as a data model to define rules and queries. Although the Datalog syntax is similar to that of Prolog, the operational semantics and the evaluation strategies of Datalog are different from those of Prolog. Unlike Prolog, Datalog does not allow functions, and it follows the model-theoretic semantics (see Section 1.4) rather than the computational semantics in Prolog. Deductive databases inherit and extend data management facilities from relation databases. To efficiently cope with a large amount of data in query processing, set-oriented data accessing methods are used in deductive databases rather than tuple-at-a-time data accessing in Prolog. Deductive databases aim to be more declarative than Prolog in that the order among rules and the order among predicates can be independent from the evaluation strategy in deductive databases.

1.3 Basic concepts of deductive databases

The building element in a deductive database, called *atomic formula*, is a predicate of the form $p(a_1, a_2, \ldots, a_n)$, where p is the predicate name, a_i's are arguments and n is the number of arguments called *arity* of the predicate. Each argument can be either a constant or a variable. As a notational convention, a string starting with an upper-case character is a variable, otherwise it is a constant. A *literal* is either an

atomic formula (positive literal) or an atomic formula preceded by a negation sign *not*, which is a negative literal.

The rules in deductive databases are in the form of Horn clauses, as shown in 1.3. The left hand side of symbol ": −" is called *head*, and the right hand side is *body*. A rule has at most one predicate in the head, and usually one or more predicates in the body. The relationship among predicates in the body is logical AND. When multiple rules are used to define the same predicate, the relationship among these rules is logical OR. The meaning of logic rules is natural and easy to understand. It states if all predicates in a rule body are true, then the head predicate is true.

$$q :- p_1, p_2, \ldots, p_n. \tag{1.3}$$

A predicate is *ground* if all of its arguments are constant. A rule is *ground* if all of its predicates are ground. A fact is a ground rule with an empty rule body, such as *parent* in Figure 1.1. When the head of a rule is empty, the body predicates form an *integrity constraint*, which has to be satisfied in a consistent database.

A deductive database consists of a finite set of EDB predicates, IDB rules and integrity constraints. A set of rules in a deductive database is also referred to as a *program*.

Definition 1.1 *A predicate s is said to imply a predicate r ($s \Longrightarrow r$) if there is a Horn clause in IDB with predicate r as the head and predicate s in the body, or there is a predicate t where $s \Longrightarrow t$ and $t \Longrightarrow r$ (transitivity). A predicate r is recursive if $r \Longrightarrow r$. If $r \Longrightarrow s$ and $s \Longrightarrow r$, r and s are mutually recursive and are at the same* deduction level. *Otherwise, if $r \Longrightarrow s$ but not $s \Longrightarrow r$, r is at a lower deduction level than s.*

Definition 1.2 *A rule is linearly recursive if its body contains exactly one recursive predicate, and that predicate is defined at the same deduction level as that of the head predicate. A rule is nested linearly recursive if its body contains more than one recursive predicate but there is only one defined at the same deduction level as that of*

the head predicate. A rule is nonlinearly recursive if its body contains more than one recursive predicate defined at the same deduction level as that of the head predicate.

Definition 1.3 *A recursion is (single) linear if all of its recursive predicates are at the same deduction level and every recursive predicate is defined by one linearly recursive rule and at least one nonrecursive (exit) rule. A recursion is multiple linear if all of its recursive predicates are at the same deduction level and every recursive predicate is defined by one or more linearly recursive rules (but at least one is defined by multiple linearly recursive rules) and at least one nonrecursive rule. A recursion is nested linear if every recursive predicate in the recursion is defined by one linearly or nested linearly recursive rule (but at least one is defined by a nested linearly recursive rule) and at least one nonrecursive rule. A recursion is nonlinear if it contains some nonlinearly recursive rule(s).*

Linear recursions are particularly important, because most of the "real life" recursions are linear, and there are efficient methods to process queries on linear recursions [10].

A query in deductive database is a literal of the form "$? - q(a_1, a_2, \ldots, a_n)$", where predicate q can be defined either by an EDB or an IDB predicate and arguments a_i's can be either constants or variables. A constant in query represents *query instantiation* and a variable designates an argument being inquired. The answer to the query is the set of all the instances of predicate q according to the program and the EDB relations.

Query evaluation in deductive databases is realized by mapping a deduction rule to a corresponding relational expression and propagating query instantiation into the relational expression. To aid analysis of query evaluation, a predicate is associated with a notation *binding*, which is a string of characters 'b' or 'f'. If the i-th argument in a predicate p is instantiated, then the i-th character in p's binding is b (bound); if p's i-th argument is a variable, then the i-th character is f (free). If a predicate p has a binding string of x, the predicate is represented in the *adorned form* as p^x. For example, the adorned form for query "$? - ancestor(john, Y)$" in Figure 1.1 is

$ancestor^{bf}$. For a deduction rule, propagation of instantiation in its corresponding relational expression can be illustrated by binding propagation in the deduction rule. Analysis of binding propagation reveals the way a deductive query should be evaluated. For example, given a query of $ancestor^{bf}$ in Figure 1.1, the binding is propagated in Rule (1.2) as following:

$$ancestor^{bf}(X, Y) :- parent^{bf}(X, Z), ancestor^{bf}(Z, Y).$$

Here, the instantiation on X in the rule head enables the evaluation of *parent* (by relational selection), which instantiates Z, and produces adorned predicate $ancestor^{bf}$ in its body again. Such binding analysis reveals the query $ancestor^{bf}$ can be evaluated by performing a number of relational selection operations on *parent* relation.

1.4 Semantics of logic rules

Intuitively, the meaning of a logic rule is that the head predicate is true if all of the body predicates are true. In other words, if all of the variables in a rule are substituted by constants and the substitution makes the right hand side of the rule true, then the left hand side is also true.

Formally, there are three ways to define meanings of logic rules: proof-theoretic interpretation, model-theoretic interpretation and computational definition. They are briefly introduced here following the concepts in [132].

Proof-theoretic interpretation establishes the meaning of logic rules by treating rules as axioms, and applying them on the facts in the database to prove other facts. For example, the meaning of the *ancestor* rules in Figure 1.1 is obtained by first applying Rule (1.1) on base relation *parent*, then by repeatedly applying Rule (1.2) on *parent* and derived relation *ancestor*. Each time, a logic rule is applied in the "forward" direction, with its body predicates as the conditions and the head predicate as the conclusion.

In model-theoretic interpretation of logic rules, an interpretation of a set of predicates assigns truth or falsehood to every possible instance of those predicates, where the arguments of those predicates are chosen from some infinite domain of constants. An interpretation is represented by the true instances of the predicates. An interpretation is a *model* if the assignment of the interpretation makes the rules true, which means the rules are satisfied under all of the instances from the interpretation.

The number of models for a given set of logic rules may be infinite. A model M for a given set of logic rules is a *minimum model* if no other model for the set of logic rules is a subset of M (if only truth assignments are accounted). If there is only one minimum model for a given set of logic rules, such model is the *least model.* A Datalog program has a nice property that it has the least model, and the interpretation under its least model coincides with its proof-theoretic interpretation. However, when negation is introduced into logic rules, the uniqueness of minimum model is not guaranteed.

Computational definition to define meanings of logic rules is to provide an algorithm to determine whether a fact is true or false by executing them using the algorithm. Prolog is such an example. The set of facts found under such an algorithm is not necessarily the set of all the facts for which a proof exists. Computational interpretation enables efficient computation of logic rules.

To ensure efficient implementation, the approach adopted in deductive databases is to translate logic rules into a sequence of relational operations. It can be shown that for Datalog without negation, such interpretation produces the least model under model-theoretic interpretation.

1.5 Query processing in deductive database

Inclusion of logic rules in deductive database poses challenge to query processing. In this section, some of the representative processing strategies are briefly introduced.

For non-recursive queries, since the relational algebra is similar to the logic in Datalog, a non-recursive logic rule can be mapped into a relational expression, such that the derived relation corresponding to the predicate in the rule head can be calculated by performing a sequence of relational operations on the relations corresponding to those predicates in the rule body. For example, for the following rule:

$$uncle(X, Y) :- parent(X, Z), brother(Z, Y).$$

the derived relation *uncle* can be computed as following:

$$uncle(X, Y) = \Pi_{X,Y}(parent(X, Z) \bowtie brother(Z, Y))$$

Query processing and optimization techniques well established in the relational database [29, 38, 63, 78, 76, 119, 127, 130, 132, 146] can be applied to process non-recursive queries in deductive databases. Processing of non-recursive queries is similar to the processing of queries posed on views in relational databases.

For queries on recursive rules, such a mapping has to consider recursive predicates, which are not directly computable based on the mapping to relational expressions. Recursive query evaluation methods such as the naive and semi-naive evaluation [8, 7], the Henschen-Naqvi technique [59], the Query/Subquery approach [141] and the chain-based evaluation method [55] are developed to compute the answers for queries according to the semantics of recursive programs. Query optimization methods such as the magic sets method [9, 103] and the counting method [9, 112] are used to rewrite a recursive program into an equivalent but more efficient one to evaluate.

Query evaluation approaches can be classified as *bottom-up* or *top-down* methods. A bottom-up method starts from the exit rule and applies recursive rules to EDB and IDB relations to produce derived literals until the derived relation is generated and the query is answered. The top-down method starts from a query and propagates instantiations from the query into the recursive rules. The query is either processed by the exit rule or new subqueries are generated according to the recursive rules in order to process the original query. New queries are processed in the same top-down fashion. In general, bottom-up methods are simple, top-down methods are more efficient by computing less fruitless intermediate literals.

1.5.1 Naive and semi-naive evaluation

Naive/semi-naive evaluation [5, 24, 8, 34, 116] is one of the first bottom-up evaluation methods.

Naive evaluation of a recursion works by first evaluating its non-recursive rule set, then iteratively evaluating the set of recursive rules on database relations (EDB and IDB) generated so far, until no new tuple can generated (thus *a fixed point* is reached, for a Datalog program without negation, the fixed point is the least model for the recursion and is called its *least fixed point*). Following is the algorithm for the naive evaluation from [133].

Algorithm 1.1 *Naive Evaluation.*

Input: A collection of safe rules and relations $R_1, \ldots, R_k$, for EDB predicates mentioned in the bodies of these rules.

Output: If it is finite, the least fixed point for the rules, with respect to the given EDB relations. If the fixed point is infinite, an infinite sequence of approximations that approaches the least fixed point as a limit is produced.

Method: 1. Relation P_i for each IDB predicate p_i is assumed empty in the beginning.

2. Suppose during evaluation, we have approximations $P_1, \ldots, P_m$, for the IDB predicates, $p_1, \ldots, p_m$. The next approximation for p_i is obtained by

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

as follows.

(a) for each of n rules defining p_i, construct a relation for the head from that of the body as follows:

i. For each non-built-in predicate $q_i(t_1, \ldots, t_k)$ in the rule body, obtain the corresponding relation Q_i for q_i as follows. Let Q_i be

empty initially. Let Q'_i be the relation for predicate q_i (Q'_i is one of the P relations if q_i is an IDB predicate or one of the R relations if q_i is an EDB predicate). For each tuple $q'_i(s_1, \ldots, s_k)$ in Q'_i, if there is a term matching γ for $q_i(t_1, \ldots, t_k)$ and tuple $q'_i(s_1, \ldots, s_k)$, add tuple $(s_1, \ldots, s_k)$ into Q_i.

ii. compute the join $Q = Q_1 \bowtie \ldots \bowtie Q_n$ (omit Q_i if q_i is a built-in predicate).

iii. apply to Q a selection for each of the built-in predicate if any.

iv. perform corresponding selection and projection on Q to obtain relation for the rule head.

(b) compute union on all relations from each rule of p_i; the result is P'_i.

(c) compare each P_i with P'_i; if $P'_i = P_i$ for all i, then the least fixed point is encountered. Else for each P'_i which is a proper superset of P_i, replace P_i with P'_i. Repeat step (2). □

Improvement can be made on the naive evaluation. In every iteration of applying a recursive rule on P, a portion of P is reevaluated since it also participated in iterations before. Semi-naive evaluation [8, 7, 116] removes such redundancies by performing a relational difference operation at every iteration, so that only the newly generated tuples of P participate in the next iteration of evaluation.

Semi-naive evaluation method is shown as Algorithm 1.2.

Algorithm 1.2 *Semi-naive evaluation.*

Input: same as in Algorithm 1.1.

Output: same as in Algorithm 1.1.

Method: 1. For each recursive IDB predicate p, construct the corresponding differential IDB predicate Δp as follows. For each recursive rule defining p as follows:

$$p :- g_1, \ldots, g_n.$$

replace each recursive predicate p in the rule body with Δp and produce a rule for Δp as given below:

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

2. Initialize the relation P for each IDB predicate p to be empty, and initialize the relation ΔP for the differential predicate Δp by applying the *EVAL* procedure in Algorithm 1.1, but only to those rules with no IDB predicate.
3. If all of ΔP's are empty, the least fixed point is encountered, each IDB predicate p has a relation P.
4. Otherwise, replace each IDB relation P by $P \cup \Delta P$.
5. For each IDB predicate p, compute a new differential relation $\Delta P'$ for each of the differential rules for p by applying Algorithm 1.1, using EDB relations $R_1, \ldots, R_k$, the current IDB relations $P_1, \ldots, P_m$, and the differential relations $\Delta P_1, \ldots, \Delta P_m$ as needed, and compute union over all of the differentiated rules for p.
6. For each IDB predicate, compute $\Delta P = \Delta P' - P$, and go to step 3. □

The naive/semi-naive evaluation methods can be applied to a wide range of recursive programs, both linear and non-linear. However, these methods generate the whole derived relation regardless what is being inquired. The magic sets method is proposed to remedy such a problem.

1.5.2 Magic sets method

The magic sets method [9, 13, 109, 103, 114] is a query optimization method which tailors the database according to the query instantiation so that only the "useful" portion of a database is used in the evaluation, thus substantially reducing the evaluation cost. The method can be understood best through an example. Consider the ancestor example in Figure 1.1 with a query of "? – $ancestor(peter, Y)$". Suppose the *parent*

relation contains a large number of (e.g., more than 100,000) tuples. The query is to find all of *peter*'s ancestors. Obviously, among all of the people recorded in the *parent* relation, only a small number of them are related to the derivation of *peter*'s ancestors, the rest has nothing to do with the query evaluation. The essential idea of the magic sets method is to extract the query-relevant portion of the *parent* relation to replace *parent* in the query processing. To obtain the query-relevant portion, the original rules are rewritten to incorporate the query instantiation into the rule body. In practice, usually a superset of the query relevant portion is obtained for ease of implementation. The following shows the rewritten program for *ancestor*:

$$
\begin{aligned}
&magic\_ancestor(peter).\\
&\quad magic\_ancestor(Y) \ :- \ magic\_ancestor(X), parent(X,Y).\\
&\quad ancestor(X,Y) \ :- \ magic\_ancestor(X), parent(X,Y).\\
&\quad ancestor(X,Y) \ :- \ magic\_ancestor(X), parent(X,Z), ancestor(Z,Y).
\end{aligned}
$$

The newly added predicate *magic_ancestor* (called *magic predicate*) represents the relevant portion of the *parent* relation to the query constant "peter". The rules for deriving the magic predicate are generated based on how the bindings are passed from the head of a recursive rule to the body. The original rules are rewritten to include the magic predicates so that only the relevant tuples in *parent* are used in the evaluation. The rewritten program is equivalent to the original one in that they produce the same answer to the given query. The magic sets method can also be applied to optimize evaluation of a non-recursive program [89].

1.5.3 Counting method

Counting method [9, 113, 115] works efficiently with the linear recursive query processing. To see how it works, let's examine the "same generation" example.

$$sg(X,X) \ :- \ person(X). \tag{1.4}$$

$$sg(X,Y) \ :- \ parent(X,Xp), \ sg(Xp,Yp), \ parent(Y,Yp). \tag{1.5}$$

Suppose the query is "$? - sg(janet, Y)$". For Y to be an answer it has to satisfy the following condition:

$$parent(janet, X_1), parent(X_1, X_2), \ldots, parent(X_{k-1}, X_k), person(X_k),$$
$$parent(Y_{k-1}, X_k), \ldots, parent(Y_1, Y_2), parent(Y, Y_1)$$

Here k starts from 1 to a level at which iteration no more intermediate answers can be generated. There are an equal number of instances of the predicate *parent* at both sides of the *person* predicate instance. The counting method introduces an explicit integer called *counting level* to denote such a number, which counts up (down) during the processing of the *parent* instances at the left (right) hand side of *person*.

The *sg* program can then be rewritten as follows to incorporate the equal number requirement:

$$\begin{aligned} u\_sg(janet, 0). & \\ u\_sg(Xp, I) &:- parent(X, Xp),\ u\_sg(X, I-1). \\ d\_sg(X, I) &:- u\_sg(X, I),\ person(X). \\ d\_sg(Y, I-1) &:- parent(Y, Yp),\ d\_sg(Yp, I). \\ sg(janet, Y) &:- d\_sg(Y, 0). \end{aligned}$$

The predicate *u_sg* represents the processing of *sg* during the counting up phase, and *d_sg* during the counting down phase.

Counting method is more efficient than the magic sets method for linear recursion, with the worst time complexity of $O(ne)$ and $O(e^2)$ respectively [84]. However, counting method requires that the base relation in the "up" portion be acyclic, otherwise the method does not terminate.

1.5.4 Logic programming approaches

Prolog is a well-known declarative programming language. There are many projects designed to apply Prolog's logic processing to databases [15, 23, 60, 67, 86, 91, 143,

144]. However, Prolog's SLD computation mechanics (depth-first search with backtracking resolution strategy) [77, 108] does not suit the database application well. Its tuple-at-a-time access is costly for the database applications. Termination of an evaluation in Prolog is not guaranteed and is dependent on the orders of predicates in the rule body and/or the orders of rules in the program. There are also redundancy in the computation of answers in Prolog.

The SLG resolution [27] in the XSB system [118] (see also 1.8.4) is an evaluation method combining the Prolog evaluation with memoing. Memoing for logic programming [33, 136, 85, 98, 145] maintains a table of goal/subgoal calls and their return values during a query evaluation. If the same call is made again, the answer to such a query is retrieved from the table rather than the query being executed again, thus removing redundant computation and providing better termination of evaluation.

1.5.5 Compilation and chain-based evaluation

The evaluation and optimization methods introduced so far treat recursive programs as a general form and use the same strategy to process all of them. Chain-based method [55, 46, 45] compiles a recursive program to extract and make use of information about how the recursive program behaves. Recursive programs are compiled into a regular form called *chain* by expanding the recursive rules. All linear recursive programs and most of the non-linear programs with a natural interpretation can be compiled into chains [46, 58]. Since a compiled program is highly regular, the behavior of a recursive program can be analyzed and a set of evaluation strategies (within the scope of the chain-based evaluation method) can be applied on the compiled form to generate answers to the query efficiently, according to the binding passing patterns in the program and the efficiency criteria. Therefore, evaluation may be either top-down, bottom-up, or hybrid to suit different programs. Various chain-based evaluation strategies will be presented in Chapter 3.

For the ancestor example shown in Figure 1.1, the derived relation for the predicate

ancestor can be represented as follows by expanding the recursive rule:

$$\begin{aligned} ancestor(X,Y) = parent(X,Y) \cup \\ parent(X,X_1) \bowtie parent(X_1,Y) \cup \\ parent(X,X_1) \bowtie parent(X_1,X_2) \bowtie parent(X_2,Y) \cup \\ \cdots\cdots \end{aligned} \tag{1.6}$$

which can be represented roughly as $parent \bowtie parent^*$ ($parent^*$ designates a sequence of joins on $k \geq 0$ *parent* relations). Such a compiled form reveals that the *ancestor* query can be processed as a transitive closure.

1.6 More about compilation of logic programs

The basic concepts and principles for compiling logic programs into chain forms are introduced in this section.

A function-free linear recursion can be compiled into a highly regular chain form or a bounded form [58], which has a relational expression similar to that of *ancestor* in formula (1.6). A recursion with function symbols can be transformed into its function-free counterpart by a *function-predicate transformation* which maps a function together with its variables to a predicate that carries the result of the function with an extra (functional) variable [47]. For example, function "+" in a predicate $p(X+Y,Z)$ can be transformed into a functional predicate $plus(X,Y,Sum)$ where Sum carries the result of the function evaluated, and the predicate p becomes $p(Sum,Z)$. A program is *rectified* if all of its function symbols are transformed and the head predicates of a set of logic rules defining the same predicate have identical variables. For example, the *ordered* recursion shown in Rules (1.7) and (1.8) is rectified as shown in Rules (1.9) and (1.10). Notice that $[Y|Ys]$ denotes a list construction function which results in a list with Y as the head and Ys as the rest of the list.

$$ordered([X]). \tag{1.7}$$

$$ordered([X,Y|Ys]) \ :- \ X \leq Y, ordered([Y|Ys]). \tag{1.8}$$

$$ordered(XYYs) \ :- \ cons(X, [], XYYs). \tag{1.9}$$

$$ordered(XYYs) \ :- \ X \leq Y, cons(X, YYs, XYYs), cons(Y, Ys, YYs), ordered(YYs). \tag{1.10}$$

In *LogicBase*, the compilation of a linear recursive program into chains is performed by expanding the recursive rule until regularity can be found. The regularity of a compiled recursion is that every argument in the head predicate is connected to the corresponding argument position in the recursive predicate in the body of the expanded rule via a set of chain predicates in the expansions of the recursive rule.

In the compilation of a linear recursion, the first expansion refers to the (transformed) recursive rule itself. The *i-th expansion* of a recursive rule is the unification of the recursive predicate in the body of the $(i-1)$-st expansion with the head of the (transformed) recursive rule.

For example, for the recursive program *mod* shown in Rules (1.11) and (1.12), the recursive rule in (1.11) is transformed into Rule (1.13) by the function-predicate transformation, whose second expansion becomes (1.14).

$$mod(X, Y, Z) \ :- \ X < Y, X = Z. \tag{1.11}$$

$$mod(X, Y, Z) \ :- \ X \geq Y, X_1 = X - Y, mod(X_1, Y, Z). \tag{1.12}$$

$$mod(X, Y, Z) \ :- \ X \geq Y, minus(X, Y, X_1), mod(X_1, Y, Z). \tag{1.13}$$

$$mod(X, Y, Z) \ :- \ X \geq Y, minus(X, Y, X_1), X_1 \geq Y, minus(X_1, Y, X_2), mod(X_2, Y, Z). \tag{1.14}$$

$$mod(X_0, Y, Z) \ :- \ X_0 \geq Y, minus(X_0, Y, X_1), \ldots, X_{k-1} \geq Y, minus(X_{k-1}, Y, X_k), X_k = Z, X_k < Y. \tag{1.15}$$

When the k-th expansion of *mod* unifies with the exit rule, it becomes (1.15) which consists of a total of k pairs of "$X_{j-1} \geq Y, minus(X_{j-1}, Y, X_j)$" (for $j = 1, \ldots, k$) between the head predicate and the predicates of the exit rule body. Two predicates in an k-th expansion form are *connected* if they share a common variable. A group of such connected predicates is called a *chain element* (or *chain predicate* if

the element consists of only one predicate). The regularity of a recursive program can be characterized by its chain element, because in each expansion of a recursive rule, such chain element is added into the expanded rule. The appearance of the chain element is called a *chain iteration* (or simply *iteration*) in the expanded rule, since it corresponds to the unit to be evaluated in each iteration of chain-based evaluation. More specifically, a chain iteration added to the expanded rule in the k-th expansion is called the k-th chain iteration. The predicates in successive chain iterations have different argument values. They are propagated from one chain iteration to the next via share variables during a query evaluation.

As a notational convention, a predicate p in the i-th chain iteration is denoted as $p_{(i)}$. The j-th argument of a predicate p is represented as $p : j$, and the value of the argument $p : j$ in the i-th iteration is represented as $p_{(i)} : j$.

The recursion *mod* is compiled into a *single chain*, where the connection is through one chain element. Some programs can be compiled into *multiple chains* (predicates in the recursive rule are partitioned into multiple groups according to their connections). For example, the same generation recursion *sg* shown in Rules (1.4) and (1.5) can be compiled into two chains: $parent(X, X_P)$ and $parent(Y, Y_P)$, respectively.

In general, a linear recursion can be compiled to an n-chain recursion ($n = 1$ for single chain recursion, $n > 1$ for multiple-chain recursion) or a bounded recursion which is equivalent to a set of non-recursive rules. The compilation is performed automatically [58], the methods and algorithms for the compilation are introduced in Chapter 3.

For a compiled n-chain recursion, the end of a compiled chain which connects to the query is called *query end*; whereas the other end (connected to the exit rule) is *exit end*. The evaluation of a single chain recursion is essentially a traversal along the chain, either from the query end to the exit end (called *chain-exit evaluation*), where bindings are passed from the query to the first chain iteration, from the i-th chain iteration to the $(i+1)$-st ($i > 0$) iteration, and finally to the exit rule; or in the reverse direction from the exit end to the query end (called *exit-chain evaluation*).

For a multiple-chain recursion, the query and the exit rule are connected via several chains which are to be synchronized during evaluation. Evaluation is performed by either (1) starting with some chains by the chain-exit evaluation, then evaluating the exit rule, and finally evaluating the remaining chains by the exit-chain evaluation, or (2) evaluating all the chains by the chain-exit evaluation, or (3) starting with the exit rule and by performing the exit-chain evaluation on each chain. Notice that all of the participating chains should be synchronized in evaluation in the spirit of the counting method.

The regularity of compiled chains greatly benefits constraint analysis and the generation of efficient query evaluation plans. Dedicated (and often simpler) algorithms can be applied to each category of recursions: for bounded recursions, nonrecursive query processing algorithms are adequate; for single-chain recursions, transitive closure algorithms and chain-based evaluation algorithms are applicable; for multiple-chain recursions, counting, magic sets, and the chain-based evaluation methods are applicable. A suboptimal evaluation plan can be selected from among several candidates based on binding passing, termination judgement, constraint pushing, and evaluation efficiency.

1.7 Extension of Horn Clause Programs

1.7.1 Negation

Datalog programs without negation have a unique least model, furthermore, the fixed point obtained from the bottom-up evaluation coincides with the least model. Therefore the bottom-up evaluation on negation-free programs is guaranteed to generate correct and complete answer. However, when negation is introduced into a recursive program, its least model may not exist anymore. There could be several minimum models, each can be an interpretation of the program. For example, the following

program has two minimum models: {p(a),q(b)} and {p(b), q(a)}.

$$
\begin{aligned}
& r(a). \\
& r(b). \\
p(X) \ & :- \ not\ q(X), r(X). \\
q(X) \ & :- \ not\ p(X), r(X).
\end{aligned}
$$

Much research has been done on the semantics of negated programs [4, 25, 37, 139, 102, 101, 110]. The solution to the non-uniqueness of minimum models is to give an intended model as the interpretation [107].

Stratified negation [4, 25, 137, 92] is an important class of programs where a recursion is not defined through a negation. Stratified programs have intuitive semantics and efficient evaluation methods [12, 6, 72]. Each derived predicate in a stratified negated program can be given a stratum level so that the program can be evaluated from the predicate with the lowest stratum to the highest. Upon evaluation of a predicate with a stratum, all negated IDB predicates in its rule body should be available, because they have lower strata and should have been evaluated earlier.

Extensions to the stratified negation include locally stratified negation [102], modularly stratified negation [68, 110, 104]. A program is *locally stratified* if a recursion is defined through a negation, but if all the variables in each rule are substituted by constants then the resulting instantiated rules do not have a recursion defined through negation. The *modular stratification* concept extends the connections among data in the local stratification to connections among strongly connected components.

Other models such as the *well-founded model* [139] are proposed to deal with general query evaluation involving negation.

1.7.2 Aggregation

Aggregation or set-grouping is an important feature in relational databases. Much research has been done to study aggregation in deductive databases [12, 14, 35, 36, 138,

69, 80, 90, 110, 111, 117, 149]. The handling of aggregation in deductive databases resembles that of negation. If an aggregation is defined through a recursion in a program, it is necessary to find an appropriate semantic model to determine the meaning of the program. The following bill-of-material example illustrates such a case, where the cost of a part is recursively defined to be the sum of costs of its subparts.

$$
\begin{aligned}
bom(Part, sum(< Subcost >)) &:- subpart\_cost(Part, Subpart, Subcost).\\
subpart\_cost(Part, Part, Cost) &:- basic\_part(Part, Cost).\\
subpart\_cost(Part, Subpart, Cost) &:- assembly(Part, Subpart, Quantity),\\
&\quad bom(Subpart, Subcost),\\
&\quad Cost = Subcost * Quantity.
\end{aligned}
$$

Before the aggregation function *sum* is performed on *Subcost* attribute, all tuples in *subpart_cost* have to be available and grouped according to the *Part* attribute. However, the predicate *subpart_cost* is defined through the *bom* predicate, and thus the aggregation and the recursion are dependent on each other.

Similar approaches to the query processing in negated programs are adopted to deal with query processing involving aggregation. The *stratified aggregation* [12] is a class of programs where no recursion is defined through an aggregation, which has intuitive semantics and efficient evaluation methods. Weaker forms of stratification such as *group stratification* and *modular stratification* [90, 110] are defined in a similar way to the local stratification and the modular stratification in negated programs. For the bill-of-material example, since *assembly* contains no cycle, no cost of a part is defined through an aggregation on itself. Thus, it is a group-stratified program and can be evaluated according to the hierarchical order in the *assembly* relation. Well-founded and the stable models are also proposed to deal with more general aggregation [69] in cases that stratification cannot be found. Monotonic program is discussed in [31, 32, 90, 111, 138] where the aggregated value on a partially derived data relation has monotonicity as the data relation grows.

1.7.3 Deductive and object-oriented databases

Deductive databases assume logic as both the specification language and the computational formalism, but it only supports flat data structures. On the other hand, objected-oriented database supports complex data types and concepts of object and data abstraction, but lacks the declarativeness and a logic semantics. Deductive and object-oriented database (DOOD) stems from the merging of these two separate approaches to yield benefits in each approach.

F-logic [73, 74] is proposed to represent and to reason features in object-oriented database by logic. The significance of F-logic is that it provides a logic foundation for object-oriented databases, thus enabling the integration of deductive and object-oriented paradigms. F-logic has a higher-order syntax to deal with inheritance, methods and schema of objects, but a natural first-order semantics to support efficient query evaluation.

Some deductive database systems such as LDL++ [149], CORAL++ [125] and LOGRES [21] support integration of deductive and object-oriented databases by incorporating object-oriented features into deductive databases.

Another issue concerning DOOD, schema integration and evolution, is discussed in [79].

1.8 Deductive Database Systems and Prototypes

Many deductive database systems and prototypes have been developed and reported in recent years, such as ADITI [135], COL [1], ConceptBase [64], Coral/CORAL++ [105, 125], EKS-V1 [142], Glue-NAIL! [87], Hy+ [30], LDL/LDL++ [28], LogicBase [55], LOGRES [21], LOLA [16], XSB [118]. It is widely recognized that the system implementation is a vital part in database research. We briefly overview and compare some representative deductive database systems in this section.

1.8.1 LDL

Developed at MCC, LDL (Logic Data Language) [28, 93] is one of the first functional deductive database systems available to the database researchers and developers.

The design philosophy of LDL is to extend the relational data model to logic data model and to support database management system features. The first system built at MCC tried to couple Prolog with a relational database system. Valuable lessons were learned that Prolog was not suitable to database application because its dependency on orders among rules and predicates. Thus MCC started to develop a general-purpose declarative logic language supporting full database features.

The highlights of LDL include a declarative data model combining a relational language and the expressive power of Prolog. LDL supports traditional relational DBMS features such as crash recovery and transaction management. Logic rules can be defined in LDL recursively including linear and non-linear recursive rules. The query optimizer in LDL employs the magic sets and the counting transformations for linear rules and special transformation for right- or left-linear rules. LDL extends Horn Clause programs by supporting stratified negation and stratified set grouping and aggregation. LDL++ is a direct successor to LDL which incorporates object-oriented features such as object identity and inheritance, while still retaining its relational database value concept [120], interoperability with other programming languages and DBMS.

1.8.2 Glue-Nail

NAIL! (Not Another Implementation of Logic!) was developed at Stanford University [88, 131] to study query optimization method in deductive databases. NAIL! supports general recursions and stratified negations. Before a query is evaluated, NAIL! analyzes the binding passing using rule/goal graph to select an appropriate evaluation order. Then according to the type of programs to be evaluated, proper query evaluation methods such as magic sets, counting, left- or right-linear evaluation are

applied.

It was later found that a declarative system alone cannot meet all the application demands, therefore, a procedure language called Glue was developed to augment NAIL! with procedural control, I/O operation and update features [87, 99], which becomes Glue-Nail. A predicate can be an EDB relation, a temporary local relation, a NAIL! rule or a Glue procedure. A Glue-Nail program consists of one or more modules of Glue procedures and NAIL! rule sets, each module can be compiled separately into a target language. Declarative and query-oriented program is expected to be written in NAIL! rules, while Glue is expected to take care of the interface and EDB update functions. Query evaluation strategies of NAIL! are incorporated into the target language during compilation. The target language for Glue and NAIL! structures are gathered into a single file for execution of the query evaluation.

Impedance mismatch between declarative NAIL! and procedural Glue is minimized by providing same or close data types and objects, and "all solution" computation.

1.8.3 Coral

The Coral project [105, 106] gained experience from the LDL system . A generalized magic sets method, *magic templates* [103], provides the foundation for query processing in Coral, where general recursive programs are supported. Coral also supports modularly stratified negation and modularly stratified aggregation and set-grouping. Coral employs a number of evaluation strategies, which can be applicable to different programs.

Coral provides *module* mechanism for organizing programs. Each module exports a derived predicate, which can be considered to be the definition for that predicate. With the module structure, a number of different optimization strategies can be integrated, and user can influence the evaluation strategies. It is up to the user to determine the basic evaluation approach for each module. A distinct feature of Coral is its

support for non-ground tuples. Storage manager of an extensible database called EXODUS [22] provides disk-resident data management, transaction and crash-recovery for Coral.

Coral++ [125] is a recent extension to Coral to incorporate object-orientation features.

1.8.4 XSB

XSB system developed at SUNY Stony Brook [118] is a Prolog-based logic programming system, which employs SLG resolution (memoing or tabling). SLG extends Prolog's top-down tuple-at-a-time evaluation by adding tabling to make evaluation finite and non-redundant on Datalog, and by adding scheduling strategy and delaying mechanics to support well-founded negation. Another important feature is its support for the more expressive HiLog data model. HiLog [26] is a higher order logic with a first order semantics, which can be evaluated efficiently.

XSB's query engine is implemented at the emulator level to make use of the efficiency of WAM (Warren Abstract Machine). The HiLog syntax predicates are compiled into SLG-WAM instructions to execute. It is reported such implementation is efficient [118]. Being an extension to Prolog, XSB is a memory resident system. Indexing and hashing are extended to suit database applications. The interface with disk-resident data is provided by ASCII files.

1.8.5 LogicBase

LogicBase is being developed at Simon Fraser University [55]. The design goal is to implement the chain-based query evaluation method with an emphasis on efficient compilation and query evaluation of *application-oriented recursions* in deductive databases. LogicBase identifies different classes of recursions and compiles recursions into chain or chain-like forms when appropriate. Queries posed to the compiled recursions are

analyzed systematically with efficient evaluation plans generated and executed, mainly based on a chain-based query evaluation method. Stratified negation is supported in LogicBase. Stratified aggregation will be supported in the future. Its most important feature is the pure declarativeness achieved through query-independent compilation and chain-based evaluation. Although other deductive database systems are declarative, they still depend on the orders of rules and predicates in a program to a certain extent. *LogicBase* incorporates a number of evaluation strategies into chain-based evaluation, including the bottom-up, top-down and the counting methods.

The compilation approach in *LogicBase* enables a detailed analysis of a recursive program, which facilitates the handling of functions and constraints in deductive databases. Thus *LogicBase* system can safely evaluate many queries on programs involving functions that cannot be handled by other approaches.

1.8.6 Overview of the Thesis

After presenting the principles concerning deductive databases, the rest of the thesis focuses on the problem of efficient query processing and presents author's contribution to its solution.

Constraints and monotonicity are discussed in Chapter 2. When functions are introduced into recursive rules, safety (whether the evaluation will terminate) becomes a vital issue. It is shown that constraints and monotonicity among arguments in the recursive rules can be employed to guarantee the termination of an evaluation for many programs. For those programs with multiple level recursive rules, the interaction among constraints and monotonicity is studied. Appropriate methods are proposed to propagate constraints in an efficient way to help terminate an evaluation. Furthermore, a novel technique is proposed to prune the large search space of some problems by constraint propagation and enforcement.

Chapter 3 presents the design and implementation of *LogicBase*. It gives an overview of the compilation and query processing methods in *LogicBase* as well.

Chapter 4 investigates query processing in multiple linear recursion. Programs contains multiple linear recursive rules are first classified into different categories. A set of query processing techniques centering around side-relation unioned processing are proposed to efficiently process queries. Side-relation unioned processing is to replace multiple EDB relations in different rules by their union, such that the original multiple recursive rules are replaced by a single recursive rule with an unioned EDB relation.

In Chapter 5, compressed counting is proposed to answer a query using counting method in EDB relations which contain cycles. The counting method is more efficient than the magic sets method, but it suffers inability to terminate when there is cycle in an EDB relation. The compressed counting precompiles an EDB relation and generalizes it into a direct acyclic graph (compressed graph) whose notes represent the strongly connected components in the original relation. Information about how data are connected cyclically in the EDB relation with respect to those strongly connected components is derived. Query processing is realized by propagation of the information over the compressed graph, which is much smaller than the data graph corresponding to the EDB relation and can be done efficiently.

The last chapter discusses the advantages and restrictions on the query processing methods in *LogicBase*, and presents performance evaluation with respect to the top-down evaluation and bottom-up evaluation with the magic sets optimization. The chain-based query processing approach is a promising new direction toward a declarative language for database query processing.

Chapter 2

Monotonicity and Constraint Pushing

The study of monotonicity and constraint pushing is motivated by the design and implementation of the *LogicBase* deductive database system. Problems like safety (termination) of the evaluation of functional programs, complexity of search space inherent in query processing in deductive databases led to the issues of derivation and push of constraint by monotonicity.

One of the most important features of a deductive database is the declarative semantics, i.e., which is independent of the modes of queries and the ordering of rules and predicates in the program. Declarativeness in deductive databases and logic programming relieves users of the worry about how to solve the problem. To make sure that a declarative program is safe and efficient, constraints and monotonicity are needed to accertain termination of query evaluation and to reduce the search space of problem solving.

In this chapter, methods are explored for discovery of monotonicity constraints in deductive databases and declarative logic programs and for push of constraints in the evaluation of multiple level (nested) linear recursive programs with function symbols. The study shows that monotonicity detection and constraint pushing play

an important role in program termination and efficient query evaluation.

This chapter is organized as follows. Section 2.1 introduces constraint pushing by a few examples. Constraints are categorized in section 2.2. Principles for monotonicity and constraint pushing in recursive programs are presented in section 2.3. Methods for constraint propagation and termination judgement are proposed in section 2.4. Reduction of search space by pushing monotonic list constraints is investigated in section 2.5.

2.1 Introduction

A deductive database program is considered to be a logic program with declarative semantics. The following is the classical eight-queens [126] example used in many AI studies.

Example 2.1 A queen in the chess game can attack in all directions (up, down, and diagonal). The problem is to place 8 queens on an 8×8 chess board so that none of the queens attacks each other. If the eight-queens problem is generalized to n queens on an $n \times n$ chess board, it is called the n-queens problem. Treated as a purely declarative logic program, the n-queens program is presented in Figure 2.1.

For the declarative program *nqueens*, queries with different modes, such as "? – $nqueens(5, Qs)$" (to find all chess placements for 5 queens), or "? – $nqueens(N$, [3, 5, 2, 4, 1])" (given a chess board, verify whether it is a valid n-queens placement and if it is, return the number of queens), should be evaluated efficiently and completely (finding all the answers) and terminate properly, independent of the ordering of rules and predicates in the program.

The chain-based evaluation method executes the program in three steps: (1) compile the program into a set of normalized recursions; (2) perform query binding and constraint analysis to determine whether query evaluation may terminate and, if termination can be guaranteed, select an appropriate evaluation strategy and generate

$$nqueens(N, Qs) :-$$
$$\quad range(1, N, Ns),\ queens(Ns, [], Qs). \tag{2.1}$$
$$range(M, N, [M|Ns]) :-$$
$$\quad M < N,\ M_1\ is\ M + 1,\ range(M_1, N, Ns). \tag{2.2}$$
$$range(N, N, [N]). \tag{2.3}$$
$$queens(Unplaced, Safe, Qs) :-$$
$$\quad select(Q, Unplaced, Unplaced_1),\ not\ attack(Q, Safe),$$
$$\quad queens(Unplaced_1, [Q|Safe], Qs). \tag{2.4}$$
$$queens([], Qs, Qs). \tag{2.5}$$
$$attack(X, Xs) :- attack(X, 1, Xs). \tag{2.6}$$
$$attack(X, N, [Y|Ys]) :-$$
$$\quad X\ is\ Y + N;\ X\ is\ Y - N. \tag{2.7}$$
$$attack(X, N, [Y|Ys]) :-$$
$$\quad N_1\ is\ N + 1,\ attack(X, N_1, Ys). \tag{2.8}$$
$$select(X, [X|Xs], Xs). \tag{2.9}$$
$$select(X, [Y|Ys], [Y|Zs]) :- select(X, Ys, Zs). \tag{2.10}$$

Figure 2.1: The declarative n-queens recursion defined in the Prolog syntax.

an efficient evaluation plan; and (3) carry out the query evaluation according to the query evaluation plan.

As a result, the method generates efficient query evaluation plans for reasonable query bindings, such as "$? - nqueens(4, Qs)$", "$? - nqueens(N, [2, 4, 1, 3])$", "$? - nqueens(N, [3, X, Y, 2])$", but returns a warning without evaluation for unsafe queries, such as "$? - nqueens(N, [2|L])$". □

The *LogicBase* deductive database system prototype [54, 55] evaluates queries declaratively on a subset of logic programs: linear and nested linear recursions. One strength of this implementation is the discovery of monotonicity behavior of a program and utilization of different kinds of constraints in the evaluation, which will be

analyzed in detail in this chapter.

A recursive program without function symbols has a finite Herbrand universe and the termination of its evaluation is guaranteed by the bottom-up evaluation. However, the Herbrand universe of a logic program with function symbols is in general infinite, and conventional bottom-up evaluation encounters the termination problem in evaluation. Constraints have been used to determine the termination or safety of query evaluation on recursive programs [2, 18, 19, 47, 20, 71, 89, 100, 123, 124, 128, 134]. Although a complete solution for termination control is undecidable [75, 121] for logic programs with function symbols, constraint enforcement has been shown to be effective as a sufficient condition for terminating the evaluation of many logic programs.

In this chapter, we explore the discovery of monotonicity behavior in declarative logic programs and the interaction between monotonicity and constraints in the evaluation of multiple level recursive programs. Constraint handling has been studied recently by different researchers [70, 124, 128]. Our techniques presented here emphasize its use in multi-level recursion where constraints in different levels interacts and depends on each other, which poses greater challenges than in a single-level program. The study is confined to linear and nested linear recursions with function symbols. In our approach, a (nested) linear recursion is first compiled into a highly regular chain form [58], which reveals the connections among arguments in logic rules of different levels and enables a detailed constraint analysis and constraint propagation in recursive programs.

The monotonic behavior of some arguments of a recursive predicate can be disclosed by the analysis of compiled recursive rules. An argument of a recursive predicate is *monotonic* in a recursive rule if there exists a strict inequality relationship under certain mapping between variables corresponding to the same argument position of a recursive predicate in both sides of the rule. A query constraint on a monotonic argument can be pushed into the rule for efficient query evaluation and termination of evaluation if it bounds the monotonic growth of argument values. Furthermore,

constraints can be inferred from functions and existing constraints and propagated to different levels of recursions, which enables evaluation of some programs that otherwise do not have a proper way to terminate.

The importance of monotonicity detection and constraint pushing in terminating recursive logic programs and reducing search space is shown in the following examples. The principles for handling constraints in these examples are discussed in later sections.

Example 2.2 Evaluation of query (2.11) on a recursion *gcd* defined in Figure 2.2 terminates based on the following constraint analysis.

$$? - gcd(6, Y, 2),\ Y < 20. \tag{2.11}$$

The constraint "$Y > Z$" derived from (2.14) and (2.15) implies that "$Y > Z$" holds in (2.13), i.e., the second argument of the recursive predicate *gcd* in (2.13) monotonically increases in bottom-up evaluation (or decreases in top-down evaluation). Thus the query constraint "$Y < 20$" can be pushed into (2.13), which, together with the constraint "$Y > 0$" in (2.13), guarantees the termination of the evaluation of query on *gcd*. Furthermore, "$Y > 0$" in (2.13) infers "$X > X_1$" in (2.15), that is, the first argument of the recursive predicate *mod* in (2.15) monotonically increases in bottom-up evaluation (or decreases in top-down evaluation). Since constraint "$X = 6$" in the query implies "$X = 6$" for the first call of $mod(X, Y, Z)$, whereas the query constraint, "$Y < 20$", infers "$X < 20$" for subsequent calls of $mod(X, Y, Z)$ based on variable connections, the evaluation of *mod* terminates as well.

Without such an inference of monotonicity and analysis of constraints on the compiled program, it is difficult to terminate the query evaluation. □

Besides termination judgement, another major benefit of constraint pushing is search space reduction by pruning futile derivatives in query evaluation. As an extension of inequality constraint, a list containing a sequence of elements with the values of the elements monotonically increasing according to certain partial order is

$$gcd(X, 0, X) \ :- \ X > 0. \tag{2.12}$$
$$gcd(X, Y, Gcd) \ :- \ Y > 0, \ mod(X, Y, Z), \ gcd(Y, Z, Gcd). \tag{2.13}$$
$$mod(X, Y, Z) \ :- \ X < Y, \ X = Z. \tag{2.14}$$
$$mod(X, Y, Z) \ :- \ X \geq Y, \ X_1 = X - Y, \ mod(X_1, Y, Z). \tag{2.15}$$

Figure 2.2: The recursion *gcd* (the greatest common divisor).

called a *monotonic list.* Such a monotonic behavior can be discovered by inference on constraints and function and pushed as a constraint into a recursion to prune a large number of (intermediate) lists which would have been generated without enforcing the constraint.

Example 2.3 Query "$? - sort([4, 2, 3, 1], X)$" on a recursion *sort* (permutation sort) defined in Figure 2.3 is to sort a list of elements by first enumerating all of the possible permutations and then selecting the ordered one. A monotonicity relationship among the elements of the list Ys in $ordered(Ys)$ can be derived based on its definition in (2.21) and (2.22). A special built-in constraint, $monolist(Ys, \leq)$, which means elements in list Ys must be in non-decreasing order, can be pushed into the body of the rule (2.17) to enable derivation of Xs with monotonic elements, which can be further pushed into the body of rule (2.20) to be applied on $[Y|Ys]$. It should be noted that the special constraint *monolist* is different from predicate *ordered.* Program *ordered* represents syntactial information to a query processor, whose semantics is unknown. Whereas *monolist* is a built-in predicate whose semantics and evaluation strategies are available to the query processor. The relevant modified rules are shown in Figure 2.4. Without enforcing this constraint, $permutation(Xs, Ys)$ generates $n!$ tuples, where n is the number of elements in the list to be sorted. After "filtering" by *monolist* constraint, only one tuple is fed into $ordered(Ys)$ in (2.16). □

$$sort(Xs, Ys) \ :- \ permutation(Ys, Xs),\ ordered(Ys). \quad (2.16)$$
$$permutation(Xs, [Z|Zs]) \ :- \ select(Z, Xs, Ys),\ permutation(Ys, Zs). \quad (2.17)$$
$$permutation([], []). \quad (2.18)$$
$$select(X, [X|Xs], Xs). \quad (2.19)$$
$$select(X, [Y|Ys], [Y|Zs]) \ :- \ select(X, Ys, Zs). \quad (2.20)$$
$$ordered([X]). \quad (2.21)$$
$$ordered([X, Y|Ys]) \ :- \ X \leq Y,\ ordered([Y|Ys]). \quad (2.22)$$

Figure 2.3: A permutation sort program.

$sort(Xs, Ys) :-$
$\quad permutation(Ys, Xs),\ monolist(Ys, \leq), ordered(Ys).$
$permutation(Xs, [Z|Zs]) :-$
$\quad select(Z, Xs, Ys),\ permutation(Ys, Zs),\ monolist(Xs, \leq).$
$permutation([], []).$
$select(X, [X|Xs], Xs).$
$select(X, [Y|Ys], [Y|Zs]) :-$
$\quad monolist([Y|Ys], \leq),\ select(X, Ys, Zs).$

Figure 2.4: Part of the permutation sort program in which *monolist* is pushed.

2.2 Categories of constraints

A constraint in a logic program represents certain relationship that the arguments in the program must satisfy. Equality or inequality constraints are two typical kinds of constraints. Constraints can be categorized based on their appearance and function in a logic program into the following:

1. **Query constraint** is a constraint associated with one or more arguments of a queried predicate, which puts restrictions in the head of a logic rule.

2. **Rule constraint** is a constraint appearing in the body of a recursive logic rule.

3. **Exit constraint** is a constraint appearing in the body of the exit rule of a recursion.

During recursive query evaluation, rule constraints can be applied to every iteration of a recursive query evaluation, which is not the case for query constraints or exit constraints. In general, pushing a query constraint into a recursive rule body does not generate an equivalent program. For example, for the query constraint "$Fare > 800$" in Figure 2.5 can not be pushed into rule body, otherwise, some legimate answer to the query will be left out. However, for a class of programs shown later, a program and query pair of $\langle P, Q \wedge C \rangle$ has a query-equivalent program P' such that $\langle P', Q \wedge C \rangle$ and $\langle P, Q \wedge C \rangle$ have the same answer for Q under all EDB's, where P is a logic program, Q is a query on P, and C is a set of constraints. P' is obtained by transforming (if necessary) some constraints in C and appending them to the rule body of P. By doing so, the query constraints are said to be *pushed* into P.

In our study, the push of a set of constraints in the form of $X \prec Y$ and $X \prec c$ are studied, where $\prec$ is a partial order, c is a constant, X and Y are variables (or more precisely argument positions) in a logic program. Notice that X and Y are sometimes not directly comparable, but there may exist a mapping $\mathcal{M}$ on X and Y such that $\mathcal{M}(X) \prec \mathcal{M}(Y)$. For example, $cons(X, Y, Z)$ (a list concatenation predicate) has a constraint $length(Z) > length(Y)$, i.e., the length of the list Z is greater than that of Y. $\mathcal{M}(X) \prec \mathcal{M}(Y)$ is denoted as $X \prec_{\mathcal{M}} Y$, or simply $X \prec Y$ when the mapping can be neglicted to simplify presentation.

2.3 Monotonicity and constraint pushing

2.3.1 Monotonicity constraints and monotonic arguments

Definition 2.1 *Given a rule set r, two arguments X and Y in r and a mapping function $\mathcal{M}$, a* **monotonicity constraint** *is a relationship between X and Y if and only if $X \prec_{\mathcal{M}} Y$, according to some partial order $\prec$.*

Definition 2.2 *An argument $p : i$ in a recursive predicate p is* **monotonic** *if there exists a mapping function $\mathcal{M}$ such that in the chain containing p, $\mathcal{M}(p_{(j)} : i) \prec \mathcal{M}(p_{(j+1)} : i)$, for $j > 0$, according to partial order $\prec$. It is denoted as $\prec_{\mathcal{M}} (p : i)$.*

A monotonic argument has monotonically increasing or decreasing $\mathcal{M}$-values in the corresponding argument position in successive chain predicates.

Proposition 2.1 *Given a recursive program of the form:*

$$q(X_1, \ldots, X_n) \ :- \ p_1, \ldots, p_n, \ q(Y_1, \ldots, Y_n). \tag{2.23}$$

$$q(X_1, \ldots, X_n) \ :- \ exit. \tag{2.24}$$

argument $q : i$ of the recursive predicate is monotonic if and only if there exists a mapping function $\mathcal{M}$, such that there is a monotonicity constraint, $X_i \prec_{\mathcal{M}} Y_i$, in the recursive rule (2.23).

Lemma 2.1 *If argument $q : i$ is monotonic $\prec_{\mathcal{M}} (q : i)$ and c is a constant, then rule constraint $Y_i \prec_{\mathcal{M}} c$ or $X_i \prec_{\mathcal{M}} c$ or $c \not\prec_{\mathcal{M}} Y_i$ or $c \not\prec_{\mathcal{M}} Y_i$ in* (2.23) *terminates the chain-exit evaluation, and rule constraint $c \prec_{\mathcal{M}} X_i$ or $c \prec_{\mathcal{M}} Y_i$ or $X_i \not\prec_{\mathcal{M}} c$ or $Y_i \not\prec_{\mathcal{M}} c$ in* (2.23) *terminate the exit-chain evaluation.*

Proof. Given monotonic argument of $\prec_{\mathcal{M}} (q : i)$, the value of $q : i$ increases monotonically according to the partial order of $\prec$ under mapping $\mathcal{M}$, thus the rule constraint of $Y_i \prec \mathcal{M}c$ forms an upper bound to the value of $q : i$. Therefore the chain-exit evaluation terminates. The other cases can be similarly proved.

The constant c in the rule constraint serves as a bound on the monotonic growth of the monotonic argument $q : i$. An equality constraint of either "$X_i = c$" or "$Y_i = c$" in the rule obviously terminates both chain-exit and exit-chain evaluations. A constraint used for termination of the evaluation of a recursive query is called the *termination constraint* of the query.

Besides termination constraints, EDB predicates may also serve to terminate a monotonic argument if the variable of the monotonic argument appears in an EDB predicate. This is because EDB predicates have finite number of tuples, which limits the monotonic growth of the monotonic argument.

2.3.2 Constraint pushing via monotonic argument

According to Lemma 2.1, a rule constraint may serve directly as a termination constraint if it bounds a monotonic argument. Similarly, a query constraint or an exit constraint can be pushed into a rule from the query end or the exit end respectively via a monotonic argument based on the transitivity property of the partial order and serves as a termination constraint. Notice that for a non-recursive program, a query constraint can be pushed into the rule body directly. This is in general not so for a recursive program.

Lemma 2.2 *Given a program q defined in* (2.23) *and* (2.24)*, a monotonic argument of $\prec$ $(q:i)$, and a query "$? - q(X_1,\ldots,X_n), c \prec X_i$", where c is a constant. The program of* (2.23) *and* (2.24) *is equivalent to the following program with respect to the same query.*

$$q(X_1,\ldots,X_n) \ :- \ p_1,\ldots,p_n,\ c \prec Y_i,\ q(Y_1,\ldots,Y_n). \tag{2.25}$$

$$q(X_1,\ldots,X_n) \ :- \ exit. \tag{2.26}$$

Proof. There exists the following relationship among the values of the monotonic argument $q:i$.

$$q_{(1)}:i \prec q_{(2)}:i \prec \ldots \prec q_{(k)}:i$$

where $q_{(1)}:i$ is X_i and $q_{(2)}:i$ is Y_i in (2.23). Since $c \prec X_i$, based on the transitivity of the partial order, we have $c \prec q_{(j)}:i$ for $j=1,\ldots,k$. Since $q_{(j)}:i$ for $j=2,\ldots,k$ corresponds to the variable Y_i in the rule body in different expansions, the constraint on each $q_{(j)}:i$ is equivalent to a rule constraint of $c \prec Y_i$. Thus the original rule set

(2.23) and (2.24) with respect to the query is equivalent to the same query on (2.25) and (2.26). □

Lemma 2.2 shows that a query constraint $c \prec X_i$ can be *pushed* into a recursive rule via a monotonic argument, and becomes $c \prec Y_i$. It is noted that $c \prec X_i$ can be pushed directly into (2.23). Similarly, an exit constraint, $q : i \prec c$ in the exit rule (2.24), can be pushed into the recursive rule (2.23) based on the transitivity of the partial order, and the recursive rule (2.23) becomes (2.27).

$$q(X_1, \ldots, X_n) \ :- \ p_1, \ldots, p_n, \ X_i \prec c, \ q(Y_1, \ldots, Y_n). \tag{2.27}$$

Furthermore, for a query constraint $X_i = c$, $c \prec Y_i$ can be pushed into the recursive rule because $X_i \prec Y_i$ and $X_i = c$ infers $c \prec Y_i$. Similarly, for the exit constraint $X_i = c$ in (2.24), $X_i \prec c$ can be pushed into (2.23).

Given a monotonic argument in a recursive predicate q as $\prec (q : i)$, a constraint (either from a query or an exit rule) is *consistent* with the monotonic argument if it can be pushed into the recursive rule based on the rules for monotonicity constraint pushing.

$$\begin{aligned}
&travel(FnoList, Dep, Arr, Fare) : - \\
&\quad flight(Fno, Dep, Arr, Fare), \ cons(Fno, [], FnoList). && (2.28)\\
&travel(FnoList, Dep, Arr, Fare) : - \\
&\quad flight(Fno, Dep, Int, F_1), \ cons(Fno, L, FnoList), \\
&\quad travel(L, Int, Arr, F_2), \ Fare = F_1 + F_2. && (2.29)\\
&? - travel(FnoList, vancouver, paris, Fare), \ Fare < 1500, \\
&\quad Fare > 800. && (2.30)
\end{aligned}$$

Figure 2.5: Program *travel*.

Example 2.4 Fig. 2.5 defines a recursion, *travel* (or *connected flights*) on the EDB relation *flight*. The recursion is a single chain recursion (by compilation) with the

following chain element (with three predicates):

$$flight(Fno, Dep, Int, F_1),\ cons(Fno, L, FnoList),\ Fare = F_1 + F_2$$

The query can be evaluated by exit-chain evaluation because the exit rule is evaluated to produce $travel^{bbbb}$ in the body of recursive rule, where b in the adorned predicate $travel^{bbbb}$ indicates that the corresponding argument is *bound* [134]. A monotonicity constraint, "$Fare > F_2$" can be inferred based on the integrity constraint, "$F_1 > 0$", and the function "$Fare = F_1 + F_2$". Thus $travel : 4$ is a monotonic argument, i.e., $> (travel : 4)$. Query constraint "$Fare < 1500$" is consistent with the monotonic argument. It can be pushed into the recursive rule and terminates the exit-chain evaluation. Query constraint "$Fare > 800$" is not consistent with monotonic argument of $> (travel : 4)$ and thus cannot be pushed in. □

In general, an exit-chain evaluation terminates if there is a constraint consistent with a monotonic argument, and it is pushed in from the query end; whereas a chain-exit evaluation terminates if there is a constraint consistent with a monotonic argument, but it is pushed in from the exit end. In other words, to determine whether an evaluation plan can terminate, the constraints at the finish end of a chain need to be examined to see whether some of them can be used as a termination constraint.

2.4 Constraint propagation in multiple levels of recursions

The derivation of monotonic arguments and termination constraints relies on constraint propagation in logic programs. In a program r, the set of constraints held on r are either explicitly defined in r in the form of rule/query/exit constraints or integrity constraints, or implicitly represented: i.e., being inferred by inference rules or propagated from higher or lower level programs. Thus, it is necessary to study monotonicity detection and constraint propagation. Constraint propagation in a multiple

level program r is to derive useful constraints which are implied by a set of known constraints in r to aid monotonicity detection and constraint pushing in r.

2.4.1 Constraint propagation via invariant arguments and by argument shifting

Definition 2.3 *An argument $q : i$ of the recursive predicate defined in* (2.23) *is* invariant *if $X_i = Y_i$.*

If $q : i$ is an invariant argument, its value remains the same in every chain iteration, i.e., $q_{(1)} : i = q_{(2)} : i = \ldots = q_{(k)} : i$. A constraint (either query, rule or exit constraint) on the invariant arguments of a recursive rule can be propagated universally, i.e., it can be applied in query, in the body of a recursive rule or an exit rule. Such a propagation is useful in binding passing from a query to the exit rule for the detection of a monotonic argument and the termination of a recursion.

Example 2.5 Consider the program *mod* in Figure 2.2 with query "$? - mod(2, 4, Z)$, $Z > 0, Z < 10$". Program *mod* is a single chain recursion, with "$X \geq Y, X_1 = X - Y$" as the chain element. Query mod^{bbf} can be evaluated in the chain-exit evaluation because mod^{bbf} instantiates X and Y in rule (2.15), both "$X \geq Y$" and "$X_1 = X - Y$" are finitely evaluable, and the evaluation produces mod^{bbf}, which can be propagated further in the chain-exit direction until the final evaluation of the exit rule. The argument $mod : 3$ is invariant. Both constraints, "$Z > 0$" and "$Z < 10$", can be propagated into the exit rule to serve as exit constraints. Since there is a constraint "$X = Z$" in the exit rule, constraint "$X > 0$" and "$X < 10$" are inferred in the exit rule. The monotonic argument $mod : 1$ is consistent with "$X > 0$". Thus, "$X > 0$" can be pushed into rule (2.15), which terminates the chain-exit evaluation of *mod*. Notice that without pushing the constraint "$X > 0$", the evaluation cannot terminate because there is no assumption that $mod : 3$ be a positive integer. □

Given a recursive rule (2.23), if $X_i = Y_j$ (where $i \neq j$), then $q_{(k)} : i = q_{(k+1)} : j$, i.e., the value of the i-th argument in q at the k-th iteration is equal to the value of the j-th argument of q at the $(k+1)$-st iteration. This kind of variable connections in a recursive rule is called *argument shifting.* Since a constraint on X_i in the k-th iteration is also a constraint on Y_j in the $(k+1)$-st iteration, a rule constraint on Y_j is also a rule constraint for X_i for all the iterations except the first, where the value of X_i in the iteration depends on the query. The constraint which is defined on all iterations except the first is called *query-dependent constraint.* Similarly, a rule constraint on X_i may also be a rule constraint for Y_j except the last iteration where value of Y_j is determined in the exit rule, and such a constraint is called *exit-dependent constraint.* A query-dependent or exit-dependent constraint may terminate a monotonic argument if appropriate analysis on the query or exit rule is conducted to provide constraint in the missing iteration.

Example 2.6 Let's examine query "$? - gcd(6, Y, 2), Y < 20$" on the program *gcd* in Figure 2.2. The *gcd* program is a single chain recursion, with chain element of "$Y > 0, mod(X, Y, Z)$". Query gcd^{bfb} can be evaluated in the chain-exit evaluation, and the exit rule can be evaluated based on the bindings passed from the query via an invariant argument $gcd : 3$, then mod^{fbb} is to be evaluated. *mod* is a single chain recursion and the query mod^{fbb} can be evaluated by the exit-chain evaluation. Because the exit rule of *mod* is (finitely) evaluable through the bindings passed from the query via the invariant arguments of $mod : 2$ and $mod : 3$, which instantiate variables Y and Z, so both predicates "$X_1 = X - Y$" and "$X \geq Y$" are (finitely) evaluable. Since mod^{fbb} is evaluable, *gcd* is evaluable by exit-chain evaluation.

The constraint "$Y < 20$" in the query "$? - gcd(6, Y, 2), Y < 20$" can be pushed into (2.13). This is because $mod : 2$ and $mod : 3$ are both invariant arguments, and the exit constraint $Z < Y$ can be propagated to the query and to the rule *gcd.* Therefore, $gcd : 2$ is monotonic under "$>$" relation. Query constraint "$Y < 20$" is consistent with $> (gcd : 2)$ and is pushed in. Moreover, the rule constraint, "$Y > 0$", is propagated into *mod* because $mod : 2$ is an invariant argument. Together with "$X_1 = X - Y$", constraint "$X > X_1$" is inferred. So $mod : 1$ is monotonic, $> (mod : 1)$. A constraint

on X in (2.13) is needed to terminate *mod*. From rule constraint (pushed from query), "$Y < 20$", a query-dependent constraint of "$X < 20$" is inferred by argument shifting from $gcd : 2$ to $gcd : 1$. This constraint terminates $mod : 1$ in all the iterations of *gcd* except the first, where "$X = 6$" (query constraint) is transformed to "$X \leq 6$" and is then used to terminate *mod* in the first iteration. □

2.4.2 Constraint propagation by inference rules

The constraints discussed here are equality or inequality constraints of on data with a partial order. The transitivity property of partial order can be represented in the form of inference rules. Figure 2.6 presents some of the most useful inference rules for constraint propagation, where $\prec$ can be $>, <, \geq, \leq$.

$$X \prec Z \ :- \ X \prec Y \ \wedge \ Y \prec Z \tag{2.31}$$
$$X \prec Z \ :- \ X \prec Y \ \wedge \ Y = Z \tag{2.32}$$
$$X = Z \ :- \ X = Y \ \wedge \ Y = Z \tag{2.33}$$
$$X \prec Y \ :- \ X = Y + Z \ \wedge \ Z \prec 0 \tag{2.34}$$
$$Y \prec X \ :- \ X = Y - Z \ \wedge \ Z \prec 0 \tag{2.35}$$

Figure 2.6: Inference rules.

Given a set of known constraints $\mathcal{C}$ on program r, and a set of inference rules $\mathcal{I}$, there are many constraints implied by $\mathcal{C}$. Obviously, not all of these implied constraints are useful in helping monotonicity detection and constraint pushing, and it is expensive to derive the closure of all of the derivable constraints. A method is proposed here which tries to propagate only the useful constraints.

Based on our discussion, given a recursive program r with a set of query/exit constraints, only those constraints which are consistent with the corresponding monotonic arguments can be pushed into the program. Therefore, our task is to first look for the monotonicities that are consistent with the query/exit constraints by making necessary assumptions and verify the assumptions in r. If an assumption is verified to be

true, then the corresponding (possibly transformed) query or exit constraint(s) can be pushed into r. Such an assumption is called a *template goal* and is verified by a procedure *template_goal_verify* as shown in Algorithm 2.1. For example, given a query "$? - gcd(6, Y, 2), Y < 20$", to test whether "$Y < 20$" can be pushed into the body of the recursive rule, a template goal "$Y > Z$" in (2.13) should be verified to make sure that $gcd : 2$ is a monotonic argument, i.e., $> (gcd : 2)$.

A template goal is an assumed constraint on r. It can be an explicitly represented constraint in r or derivable from a set of constraints using inference rules. If a template goal g cannot be verified directly by applying inference rules but can be verified by first verifying an intermediate template goal g' and then use g' to verify g, g' is then set as a new template goal. This process is called *template goal propagation.*

For example, given a template goal "$X < Y$" and an inference rule "$X \prec Z :- X \prec Y \wedge Y \prec Z$", unification generates "$X < Y :- X < \$T \wedge \$T < Y$". If there is a constraint "$X < Z$" in the body of the rule, a new template goal "$Z < Y$" is generated. Otherwise, "$X < \$T$" is generated, where $\$T$ is a *meta-variable* which may match any literal in a subsequent unification. When an inference rule is used, template goals can be of the form "$X \prec Y$", "$X \prec c$" or "$X \prec \$T$", and a template goal may contain at most one meta-variable.

For a multiple level recursion, if g cannot be verified in r but there exists a lower level predicate s defining r which can unify with g, then g should be verified in every rule that defines s.

The template goal verification process is described in Algorithm 2.1. Given a set of constraints $\mathcal{C}$ on program r, it verifies whether there is a template goal g on r by first checking whether $\mathcal{C}$ contains g and then trying to unify g with the head of the inference rule to see whether g can be either inferred from the constraints, or there exists a new template goal g' such that verifying g is equivalent to verifying g'.

Algorithm 2.1 procedure *template_goal_verify*(g, r)

Input: A set of constraints $\mathcal{C}$, a program r, constraint template g.

Output: *True* if g is implied by $\mathcal{C}$ and r, *false* otherwise.

Method: The template goal verification process given below:

1. Collect rule constraints, integrity constraints, and constraints on functional predicates of r into a constraint set $\mathcal{C}$. Let $c \in \mathcal{C}$.
2. If g has no meta-variable and $\mathcal{C}$ contains g, g is verified.
3. If g has meta-variable(s), unify g with $c \in \mathcal{C}$. If there is a successful unification, g is verified.
4. Verify whether g can be inferred from $\mathcal{C}$ by an inference rule. For inference rule "$h : -b_1, b_2, \ldots, b_k$" such that g and h can be unified, try to unify each b_i (for $i = 1, \ldots, k$) with $c \in \mathcal{C}$ such that all the unifications are consistent.
5. If all the b's are successfully unified, g is verified. Apply the same unification on h, and h is an answer.
6. If a b is not successfully unified with $c \in \mathcal{C}$ but b has some argument(s) unified, b is a new template goal. If b can be verified in r or in every rule defining a lower level predicate, g is verified.
7. If the variable in g is an invariant argument, verify g in the exit rule or query constraint. If it is a shifting argument, verify g by shifting g to g' and verifying g'. □

Theorem 2.1 *Algorithm* 2.1 *for template_goal_verify*(g, r) *takes polynomial time to verify a goal in a nested linear recursion.*

Proof sketch. In the algorithm *template_goal_verify*(g, r), the constraint propagation using inference rules takes most of the time. The number of literals in each inference rule is at most two (as shown in our inference rule table Figure 2.6, an inference rule with more literals can always be broken into multiple inference rules with binary literals).

Let's first consider the verification of a goal in a single-level program by assuming r is a single-level logic program. Assume $\mathcal{C}$ is the set of constraints and functions in r, and $\mathcal{V}$ is the set of distinguished literals (variables or constants) in $\mathcal{C}$, and $\mathcal{I}$ the set of inference rules. Thus, the literals of any template goal propagated during *template_goal_verify*(g, r) is in $\mathcal{V}$, so the maximum number of possible template goals propagated (either verified to be true or false) is bounded by $|\mathcal{I}| * |\mathcal{V}|^2$ because there could be at most $|\mathcal{I}|$ different types of binary relationships inferred. Furthermore, because for any pair of constraints or functions there is at most one inference rule applicable, the maximum number of goals verified to be true is bounded by $|\mathcal{V}|^2$. Since a new template goal is propagated in the same recursive level if there is one predicate in the body of an inference rule unified with a known constraint or a function, the cost of verifying a goal g is the sum of the following costs: unifying the goal with a rule head predicate, unifying a body predicate with $\mathcal{C}$ to propagate a new template goal, and the cost of verifying g' in r. Applying the same cost formula to g', the number of subgoal propagated from g directly and indirectly is bounded by $|\mathcal{I}| * |\mathcal{V}|^2$, with at most 3 unifications of one goal/predicate and $\mathcal{C}$ associated with each goal propagated. Since unifying a goal/predicate with $\mathcal{C}$ takes at most $|\mathcal{C}|$ unifications, the cost of *template_goal_verify*(g, r) is bounded by $3 * |\mathcal{I}| * |\mathcal{V}|^2 * |\mathcal{C}|$ unifications, which is polynomial time.

Secondly, we examine the goal verification process in a multiple-level program. Suppose r has a lower level predicate. Assume $\mathcal{C}$ is the set of constraints and functions on all the programs nested in r. Then the same formula above holds on a nested program as well. In a nested program, a template goal can be verified not only in the current level program, but in a lower level program as well. Comparing template goal propagation in the nested program with the constraint set $\mathcal{C}$ to that in a single-level program with the same constraint set $\mathcal{C}$, the template goals propagated in the nested program is a subset of the template goals propagated in the single-level program, because in a nested program, $\mathcal{C}$ can be viewed as being partitioned by deduction levels. Therefore, the bound in a single-level program applies to a multiple-level program as well. □

Example 2.7 Given query "? $- travel(FnoList, vancouver, paris, Fare)$, $Fare < 1500$" on the program *travel* in Figure 2.5, to push query constraint "$Fare < 1500$", one needs to check whether there is a monotonic argument $> (travel : 4)$. A template goal "$Fare > F_2$" is set for (2.29), the recursive rule of *travel*. Initially, the constraint set $\mathcal{C}$ is $\{Fare = F_1 + F_2,\ F_1 > 0,\ F_2 > 0,\ cons(Fno, L, FnoList),\ length(L) < length(FnoList)\}$. The template goal is not found directly in $\mathcal{C}$. Thus the inference rules are applied. The inference rule "$X \prec Y : -X = Y + Z\ \wedge\ Z \prec 0$" generates subgoals "$Fare = F_2 + \$T,\ \$T > 0$". Since $\$T$ is unifiable with F_1, the template goal "$Fare > F_2$" is verified. □

2.4.3 Termination control and constraint pushing in functional programs

Based on the above discussion, the termination control and constraint pushing in a compiled single- or multiple-level linear recursion can be integrated in one algorithm as follows.

Algorithm 2.2 *Termination control and constraint pushing in a compiled single- or multiple-level linear recursion.*

Input: A query on a compiled linear recursion with function symbols, a set of query constraints, integrity constraints, and exit constraints.

Output: Determine whether the query evaluation terminates and, if it does, push the (transformed) query constraints into the recursive rule.

Method. 1. Identify invariant arguments and shifting arguments.

2. For each query constraint c_q, test whether c_q can be pushed into the rule body by verifying the template goal g which leads to a monotonic argument consistent with c_q. Do the same for each exit constraint c_e.

3. If there exists a chain with monotonic argument(s) restrained by query constraint(s) or exit constraint(s), push the corresponding transformed query constraint(s) into the rule to terminate the exit-chain evaluation. If there exists a chain with monotonic argument(s) restrained by exit constraint(s), push the corresponding transformed exit constraint(s) into the rule to terminate the chain-exit evaluation.
4. If the termination cannot be determined, detect if there exist other monotonic argument(s) by enumerating the remaining possible monotonicity on arguments of the recursive predicate, and verifying the corresponding template goals.
5. If an argument is found monotonic, set template goal of termination constraint which is consistent with the monotonic argument. If the template goal is verified in rule body (rule constraints imply termination constraint), both chain-exit and exit-chain evaluation terminate. If the template goal is verified using query constraint set (query constraints imply termination constraint), push template goal into rule to terminate exit-chain evaluation. If the template goal is verified using exit constraint (exit constraints imply termination constraint), push template goal into rule to terminate chain-exit evaluation.
6. If an argument of a chain predicate is found monotonic, and the variable in its position appears in an EDB predicate or a safe IDB predicate in the rule body, then both chain-exit and exit-chain evaluations terminate.
7. If a chain predicate is an EDB predicate which contains only acyclic data, the recursion terminates.
8. A multiple-chain recursion terminates if any of its chains terminates. □

Theorem 2.2 *Algorithm 2.2 correctly pushes constraints and terminates a compiled single- or multiple-level linear recursion.*

Proof sketch. According to algorithm 2.2, step 2 tests for each query constraint c_q, whether it can be pushed into the rule body. The test is based on the template goal verification algorithm proved before. Thus if there exists a query constraint which is consistent with a monotonic argument, the values of a monotonic argument

grow/shrink monotonically in the evaluation, which sooner or later will be bounded by the constraint. Thus the constraint can be pushed in and the chain evaluation terminates. Similar reasoning can be performed for pushing exit constraints and for steps 3-5.

For step 6, if a monotonic argument in a chain appears in an EDB predicate or a safe IDB predicate, the chain terminates. This is because in both cases, the predicates have a finite number of instances (tuples), the monotonic argument reaches the limit sooner or later, and thus the chain evaluation terminates.

Step 7 deals with the case that a chain has an acyclic EDB predicate as its chain predicate. Since the chain predicate links arguments on the same position of recursive predicates in the head and the body, the number of iterations in the evaluation must be finite, and thus the chain evaluation terminates.

Finally, in a multiple-chain recursion (step 8), since all of its chains are evaluated synchronously, its evaluation terminates if any of the chains terminates. □

Example 2.8 Query "? $- gcd(6, Y, 2), Y < 20$" on the *gcd* program of Figure 2.2 terminates and some constraints can be pushed in based on Algorithm 2.2 as shown below.

To check whether "$Y < 20$" can be pushed into the body of the recursive rule (2.13), a template goal "$Y > Z$" is set. Since it cannot be verified within (2.13), "$Y > Z$" should be verified in $mod(X, Y, Z)$. In the execution of *template_goal_verify*$(Y > Z, mod)$, the template goal is tested first in the exit rule (2.14) of *mod*, which infers "$Y > Z$". Since both Y and Z are invariant arguments in the rule (2.15), the constraint "$Y > Z$" holds in recursive rule as well. Thus, "$Y < 20$" can be pushed into (2.13), and the exit-chain evaluation of *gcd* terminates if *mod* is finite.

To verify whether $mod(X, Y, Z)$ returns finite results to the query, one needs to use the query constraints "$Y > 0$" and "$Y < 20$". Since $mod : 2$ is an invariant argument, one cannot judge whether *mod* is terminable by judging $mod : 2$ only, although both of these constraints are propagated into the body of rule (2.15). The monotonicity

of other arguments in the recursive predicate of *mod* needs to be examined. Since there is no mapping on $mod : 1$, only the template goal "$X \prec X_1$" needs be verified. The initial constraint set for *mod* is $\{X \geq Y,\ X_1 = X - Y,\ Y > 0,\ Y < 20\}$. The template goal "$X > X_1$" is verified by "$Y > 0$" and "$X_1 = X - Y$" according to the inference rule. So, $mod : 1$ is monotonically decreasing, i.e., $> (mod : 1)$. Because there is no termination constraint within *mod*, template goal "$X < \$1$" is verified in the *gcd* rule to find possible query constraint, which generates query-dependent constraint "$X < 20$" due to the argument shifting and "$X = 6$" in the query. So, the first iteration of *mod* program has a constraint "$X \leq 6$" pushed in, and the remaining iterations has "$X < 20$" pushed in. Therefore, the exit-chain evaluation of *mod* terminates.

Similarly, it can be shown that query "$? - gcd(X, 4, 2),\ X < 6$" terminates on *gcd*. Query constraint "$X < 6$" cannot be pushed in (2.13), but "$Y \leq 4$" is transformed from a query constant and is pushed into rule (2.13) due to the monotonic argument $> (gcd : 2)$, which terminates *gcd*. Whereas "$X \leq 4$" is obtained by argument shifting and pushed into rule (2.15) for the *mod* subquery in all *gcd* iterations except the first one, where "$X < 6$" of *gcd* is pushed in (2.15). Thus the evaluation of *mod* terminates. □

2.5 Search space reduction using monotonic list constraints

It is well known that the size of a list has monotonicity behavior (i.e., growing or shrinking monotonically) in many recursive programs. Interestingly, the values of list elements in a list may also have certain monotonicity behavior in many programs which can be used as an effective constraint for search space reduction in query evaluation.

Definition 2.4 *An empty list* [] *is a monotonic list, so is a list with a single element. A list* L *of* $[a_1, a_2, \ldots, a_n]$ *is a* monotonic list *if there exists a mapping* $\mathcal{M}$ *such that* $\mathcal{M}(a_i) \prec \mathcal{M}(a_{i+1})$ *for* $i = 1, \ldots, n-1$ *according to a partial order. It is denoted by a special built-in functional predicate,* $monolist(L, \prec_{\mathcal{M}})$*, in a compiled logic program, or simply as* $monolist(L, \prec)$.

Predicate *monolist* does not have to be built-in predicate. During query optimization, the original program can be rewritten to include the following program, such that the rewritten program is executed more efficiently.

$$
\begin{aligned}
monolist([], \prec_{\mathcal{M}}).& \\
monolist([X], \prec_{\mathcal{M}}).& \\
monolist([X|Y|L], \prec_{\mathcal{M}}) \quad :- \quad & monolist([Y|L], \prec_{\mathcal{M}}),\ \mathcal{M}(X) \prec \mathcal{M}(Y).
\end{aligned}
$$

Hence, *monolist* has a completely declarative semantics and are treated in the same way as ordinary predicate. Although *monolist* has higher order syntax (function symbols appear in argument position), it has first order semantics. Since the function $\prec$ are built-in in the program, and $monolist(L, \prec)$ can be transformed into the following *monolist_gt* program during query optimization if $\prec$ is bound to greater than function ($>$):

$$
\begin{aligned}
monolist\_gt([]).& \\
monolist\_gt([X]).& \\
monolist\_gt([X|Y|L]) \quad :- \quad & monolist\_gt([Y|L]),\ X > Y.
\end{aligned}
$$

Corollary 2.1 *If a list* $[X|L]$ *is a monotonic list, then* L *is a monotonic list. A list* $[X|L]$ *is monotonic if* L *is monotonic and there exists a mapping of* $\mathcal{M}$ *such that for the head element* Y *in* L, $\mathcal{M}(X) \prec \mathcal{M}(Y)$.

Proof. Let $X = a_0$ and $L = [a_1, a_2, \ldots, a_n]$. We first show that the first statement is true. Based on the definition, if $[X|L]$ is a monotonic list, there must exist a mapping

$\mathcal{M}$ such that $\mathcal{M}(a_i) \prec \mathcal{M}(a_{i+1})$ for $i = 0, \ldots, n-1$ according to a partial order. This should also be true for $i = 1, \ldots, n-1$, that is, L must be a monotonic list as well.

Then we show that the second statement is also true. Since Y is the head of L, $Y = a_1$. If L is monotonic, there must exist a mapping $\mathcal{M}$ such that $\mathcal{M}(a_i) \prec \mathcal{M}(a_{i+1})$ for $i = 1, \ldots, n-1$ according to a partial order. Since $\mathcal{M}(X) \prec \mathcal{M}(Y)$, that is, $\mathcal{M}(a_0) \prec \mathcal{M}(a_1)$. The monotonicity relationship $\mathcal{M}(a_i) \prec \mathcal{M}(a_{i+1})$ can be extended to $i = 0$ as well. Thus $[X|L]$ is monotonic. □

2.5.1 Derivation of monotonic list

Corollary 2.1 provides us with a technique of testing whether a list is a monotonic list.

Definition 2.5 *Given the recursive rule in* (2.23)*, argument* $q : i$ *is said to be a* list construction argument *if there is a variable* Z *(called a* list element variable*) such that list construction predicate* $cons(Z, Y_i, X_i)$ *is in the rule body.*

List construction arguments are common in recursive rules involving lists. For example, in the program *travel* defined in Figure 2.5, *travel* : 1 is a list construction argument, and *Fno* is its list element variable.

Corollary 2.2 *Given a recursion defined in* (2.23) *and* (2.24)*, argument* $q : i$ *is a monotonic list* $monolist(q : i, \prec)$ *if (1) it is a list construction argument in* (2.23) *and* $head\_X_i \prec head\_Y_i$ *where* $head\_X_i$ *is the first element in* X_i *and* $head\_Y_i$ *is the first element in* Y_i*, and (2)* $q : i$ *is a monotonic list* $monolist(q : i, \prec)$ *in the exit rule* (2.24) *as well.*

Example 2.9 Considering the recursion, *ordered*, in rules (2.21) and (2.22). Since *ordered* : 1 is a single element in the exit rule (2.21), it is a monotonic list. Moreover, the first element of $XYYs$ in the head predicate of the recursive rule (2.22) is X,

the first element of YYs in the body of the recursive predicate in (2.22) is Y, and "$X \leq Y$". Thus *ordered* : 1 is a monolist under relation "$\leq$" based on Corollary 2.2. □

2.5.2 Pushing monotonic list constraints

A query constraint of $monolist(X_i, \prec)$ in the recursive rule (2.23) can be pushed into the rule body if the implication, $monolist(X_i, \prec) \Longrightarrow monolist(Y_i, \prec)$, can be derived in the rule body. Obviously, *monolist* pushed into the body of a recursive rule (as a rule constraint) reduces more search space than serving as a query constraint in the query evaluation.

Example 2.10 Figure 2.7 is the rectified program of *sort* defined in Figure 2.3. Suppose a query constraint $monolist(YYs, \leq)$ (derived in Example 2.9) is enforced on the program *select*. Since argument *select* : 2 has YYs in the head and Ys in the body where Ys is the tail of YYs, we have $monolist(YYs, \leq) \Longrightarrow monolist(Ys, \leq)$. Thus argument *select* : 2 is a monotonic list, and $monolist(YYs, \leq)$ is pushed into the body of rule (2.39). □

$$sort(Xs, Ys) \ :- \ permutation(Ys, Xs),\ ordered(Ys). \quad (2.36)$$

$$permutation(Xs, ZZs) \ :- \ cons(Z, Zs, ZZs),\ select(Z, Xs, Ys),\ permutation(Ys, Zs). \quad (2.37)$$

$$permutation(Xs, ZZs) \ :- \ Xs = [],\ ZZs = []. \quad (2.38)$$

$$select(X, YYs, YZs) \ :- \ cons(Y, Ys, YYs),\ cons(Y, Zs, YZs),\ select(X, Ys, Zs). \quad (2.39)$$

$$select(X, YYs, YZs) \ :- \ cons(X, YZs, YYs). \quad (2.40)$$

Figure 2.7: Rectified permutation sort program.

Monotonic list and monotonic argument are tightly related as stated in the following corollaries. Their inter-derivability is important to derive implications between

monotonic list constraints in a recursion. In general, to obtain such implications, a known monotonic list constraint is first mapped to a monotonic argument via the list construction argument, then the target monotonic list constraint is derived through constraint propagation and the derivation of monotonic list constraint.

Corollary 2.3 *Given a recursion in* (2.23) *and* (2.24), *suppose argument* $q : i$ *is a list construction argument with* X *as its list element variable. If* $monolist(X_i, \prec)$ *holds, then (1)* X *is a monotonic argument of* $\prec$ (X), *(2) constraint* $monolist(q : i, \prec)$ *holds in the exit rule, (3) if* Z *is the first element of the variable of* $q : i$ *in the exit rule,* $X \prec Z$.

Proof sketch. Argument $q : i$ is a list construction argument, then $cons(X, Y_i, X_i)$ is in the body of (2.23). So the values of argument X, $X_{(1)}, X_{(2)}, \ldots, X_{(n)}$ correspond to the elements in X_i. Since $monolist(X_i, \prec)$, $X_{(i)} \prec X_{(i+1)}$ for $i = 1, \ldots, n-1$. With $cons(X, X_i, Y_i)$ in rule body, then $monolist(X_i, \prec) \Longrightarrow monolist(Y_i, \prec)$, therefore $monolist(Y_i, \prec)$ holds in body of (2.23). Therefore, $q : i$ in (2.24) is monotonic list, and if its first element is Z, then $X \prec Z$. □

Corollary 2.4 *Given a recursion in* (2.23) *and* (2.24), *if argument* $q : i$ *is a list construction argument and* X *is its list element variable, then* $q : i$ *is a monotonic list if (1)* X *is a variable of a monotonic under* $\prec$, *(2)* $q : i$ *in the exit rule is a monotonic list:* $monolist(q : i, \prec)$, *(3)* $X \prec Y$ *where* Y *is the first element of* $q : i$ *in the exit rule.*

Corollary 2.4 can be proven similarly as Corollary 2.3.

Example 2.11 With the above preparations, we examine how the program in Figure 2.4 is derived.

Consider program *sort* in Figure 2.7. First, $monolist(Ys, \leq)$ is derived from *ordered* program (in Example 2.9) and is appended to the body of rule (2.36). Then the program *permutation* has a query constraint, $monolist(Xs, \leq)$, in rule

(2.37). To push the *monolist* constraint into the rule body, one needs to verify that "$monolist(Xs, \leq) \Longrightarrow monolist(Ys, \leq)$" in rule (2.37), i.e., "$monolist(YYs, \leq) \Longrightarrow monolist(YZs, \leq)$", in rule (2.39). Both arguments $select : 2$ and $select : 3$ are list construction arguments with Y as their common list element variable in rule (2.39). Given $monolist(YYs, \leq)$, Y is a monotonic argument of $\leq$ (Y) and $Y \leq head\_Ys$ in rule (2.39), and $monolist(YYs, \leq)$ holds in the exit rule (2.40). Since YZs is the tail of YYs in rule (2.40), $monolist(YZs, \leq)$ and $head\_YZs \leq head\_YYs$ hold in rule (2.40) as well. Thus, $head\_Ys \leq head\_Zs$ in rule (2.39). and Zs is monotonic in rule (2.39) as well. Therefore, $monolist(Xs, \leq)$ can be pushed into the body of rule (2.37) as well, which can be further pushed into the body of recursive rule (2.39) as explained in Example 2.10. The push of the *monolist* constraint reduces the complexity of evaluation of the same query from $O(n!)$ to $O(n^2)$, where n is the number of elements to be sorted. □

Following is another example where *monolist* is successfully employed to reduce the search space from $O(n!)$ to $O(n^2)$ for the n-queens query.

Example 2.12 Consider query "$? - nqueens(N, [5, 3, 1, 4, 2])$" where the program is shown in Figure 2.1. This query can be processed by

1. passing binding $Qs = [5, 3, 1, 4, 2]$ from head of Rule (2.1) to Qs in predicate *queens*;
2. processing subquery "$? - queens(Ns, [], [5, 3, 1, 4, 2])$" in *queens* program shown in Rules (2.5) and (2.4);
3. processing subquery "$? - range(1, N, Ns)$", where Ns is instantiated from result in step 2.

Detailed analysis of evaluation of "$? - queens(Ns, [], [5, 3, 1, 4, 2])$" reveals that Ns contains a set of lists of all permutation of [5,3,1,4,2]. Which means, the complexity for processing queries such as "$nqueens(N, [5, 3, 1, 4, 2])$" is $O(n!)$, where n is the size of the list.

However, if constraint derivation and pushing are employed, the search space is vastly reduced to from $O(n!)$ to $O(n^2)$. From the *range* program, constraint $M < M_1$ can be inferred from function "M_1 *is* $M+1$" in rule (2.2). Therefore, the first argument of *range* is monotonically decreasing, $<(range:1)$. Since M is the list constructing argument for $range:3$, and $range:3$ in rule (2.3) has only one argument, it can be derived that $range:3$ is a monotonic list, thus constraint $monolist(Ns,<)$ is derived in rule (2.1).

To push constraint $monolist(Ns,<)$ into the body of rule (2.4), the relationship of $monolist(\ Unplaced,<) \Longrightarrow monolist(Unplaced1,<)$ needs to be established. Similar to the analysis in Example 2.11, from *select* program in rules (2.9) and (2.10), $monolist(select:2,<) \Longrightarrow monolist(select:3,<)$. Therefore, constraint $monolist(Ns,<)$ is pushed into the rule body and becomes $monolist(Unplaced1,<)$ in (2.4). Such constraint prunes all the intermeidate *queens* predicates which are not in ascending order. Finally, only one *queens* predicate *queens*([1, 2, 3, 4, 5], [], [5, 3, 1, 4, 2]) is derived instead of 5!=120 predicates. □

2.6 Discussion

This study of constraint pushing and termination control in multiple level linear recursive programs with function symbols has shown:

1. termination of multiple level program requires sophisticated analysis of constraints and monotonicity.
2. monotonicity can be caused by constraints and functions and can be extracted by inference.
3. monotonicity provides a vehicle for constraint pushing.
4. transformation of query/exit constraints is needed to satisfy the consistence requirement of constraint pushing.

5. derivation and pushing of monotonic list constraints may substantially reduce the search space.

Comparing with other approaches to incorporation of constraints into logic programs [140, 62, 70, 66, 71, 82, 100, 123, 124, 128, 134], our approach supports pushing of both query constraint and rule constraint (integrity constraint), and investigates constraint derivation and pushing in nested linear programs which poses more challenges. The chain-based compilation provides a good platform for the analysis of constraints and monotonicity, so that a good evaluation plan can be selected, which is flexible to best utilize the binding patterns and constraints, and to facilitate the interaction among constraints and monotonicity at different levels of programs. Although constraint pushing is better guided when an evaluation plan (or candidate plan) is available, our approach can be applied independently of evaluation schemes, because monotonicity detection and constraint pushing depend only on the availability of constraints and variable connections in the program.

Moreover, the types of constraints considered in the program are extended, which may involve any constraints of partial order under certain mapping. The introduction of *monolist* concept covers the monotonicity behavior of all the elements in a list, which opens a new route for constraint propagation in list functions for query optimization. Derivation and pushing of monotonic list constraint in multiple-level programs implies that constraints of one recursion can be extracted and applied to optimizing another recursion in a multiple-level recursive program, which may imply a new direction for optimization of declarative logic queries.

The effectiveness of constraint pushing is demonstrated in the following example, which compares the performance of different evaluation strategies for queries in *gcd* and *sort* programs.

Example 2.13 The query evaluation efficiency of four evaluation methods: (1) *Prolog*, (2) *magic*: magic sets method, (3) *chain w/o constraint*: chain-based evaluation without constraint pushing, and (4) *chain with constraint*: chain-based evaluation

with constraint pushing (including monotonic list constraint), are compared in the evaluation of the *gcd* and *sort* recursions with different query bindings.

The following four queries are used in the examination:

Q_1: "$? - gcd(4, 2, Z)$",

Q_2: "$? - gcd(X, 4, 2),\ X < 6$",

Q_3: "$? - gcd(4, Y, 2),\ Y < 6$", and

Q_4: "$? - sort([4, 3, 2, 1], Ys)$".

A simple cost model is constructed to facilitate the comparison of different methods. The following three kinds of *basic steps* are used in the cost estimation.

type-a: Each execution of one built-in arithmetic or comparison operator, such as "$M = N - 1$", counts as one type-a basic step.

type-b: Each execution of a primitive-level (non-arithmetic) predicate (including EDB), such as a call to "cons", counts as one type-b basic step.

type-c: Each execution of a call to an IDB predicate counts as one type-c basic step.

The total cost of query execution is expressed in the form: $\alpha a + \beta b + \gamma c$, where a, b, c are the unit cost for type-a, type-b, and type-c operations respectively. Such a cost model represents the sum of the numbers of basic steps in the three types respectively. This cost model is simple to construct and easy to compute. However, it reflects a reasonable approximation to the amount of work involved in the query evaluation. A method usually costs more in comparison with others if it involves more basic steps in the execution of the same query on the same recursion.

The four evaluation methods are compared based on the above programs, queries and the cost model. Their execution costs are presented in Table 2.1. Each slot is

Method\Query	Q_1	Q_2	Q_3	Q_4
Prolog	$16a + 10c$	N/A	N/A	$87a + 237b + 185c$
magic	N/A	N/A	N/A	N/A
chain w/o constraint pushing	$16a + 10c$	N/A	N/A	$87a + 237b + 185c$
chain with constraint pushing	$16a + 10c$	$53a + 19c$	$55a + 21c$	$11a + 71b + 43c$

Table 2.1: Performance comparison of different query evaluation methods.

filled up by the cost of query evaluation if the program is evaluated *correctly* by the method, or "*N/A* (i.e., not applicable)" otherwise.

The table reveals the following:

- The magic sets method cannot evaluate these queries on the *gcd* or *sort* programs. For *gcd*, the process of deriving magic predicates of *mod* cannot terminate due to the infiniteness of functions or built-in predicates. For *sort*, the bottom-up evaluation cannot pass sufficient bindings from *permutation* in the body to its head in Rule (2.17).

- Prolog cannot evaluate Q_2 and Q_3 on the *gcd* program due to its predicate/rule order dependency.

- With incorporation of constraint pushing, chain-based evaluation can successfully evaluate Q_2 and Q_3. The evaluation returns 3 tuples for each query.

- For Q_4 on *sort*, constraint pushing substantially reduces the evaluation cost. □

We conclude this chapter with two figures, Figure 2.8 and Figure 2.9 illustrate the effect of constraint pushing on reduction of search space for n-queens and permutation sort problems respectively, where size is number of queens to be placed in n-queens problem and the number of elements to be sorted in the permutation sort problem.

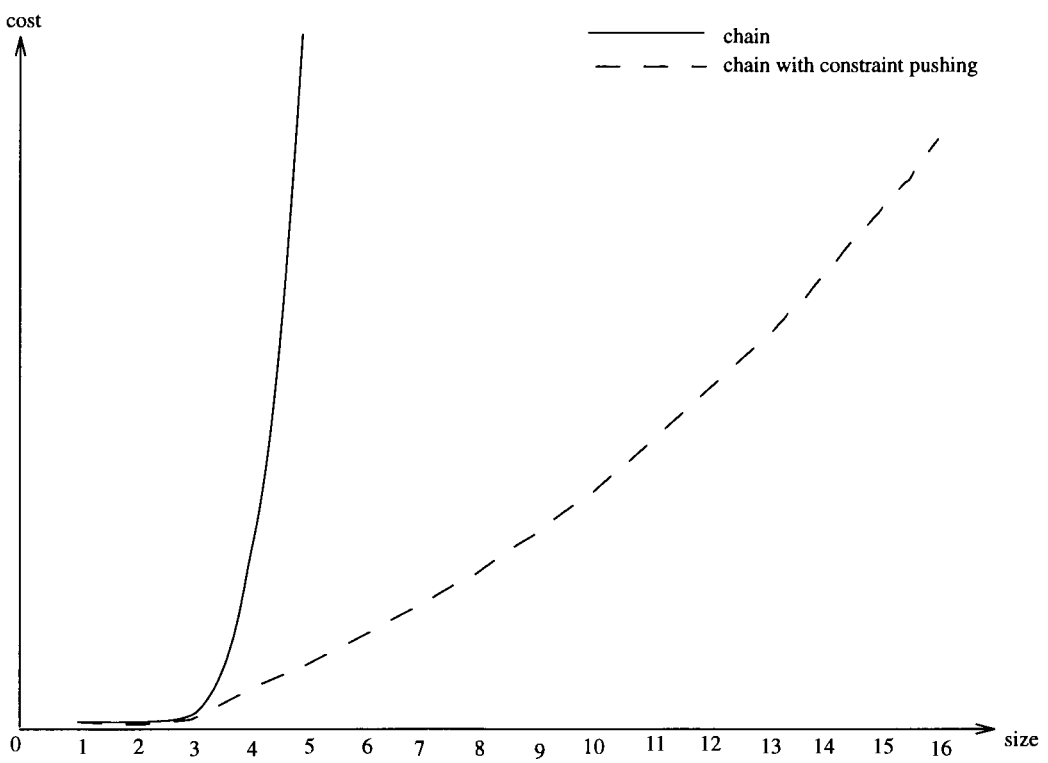


Figure 2.8: Effectiveness of constraint pushing for $nqueens^{fb}$.

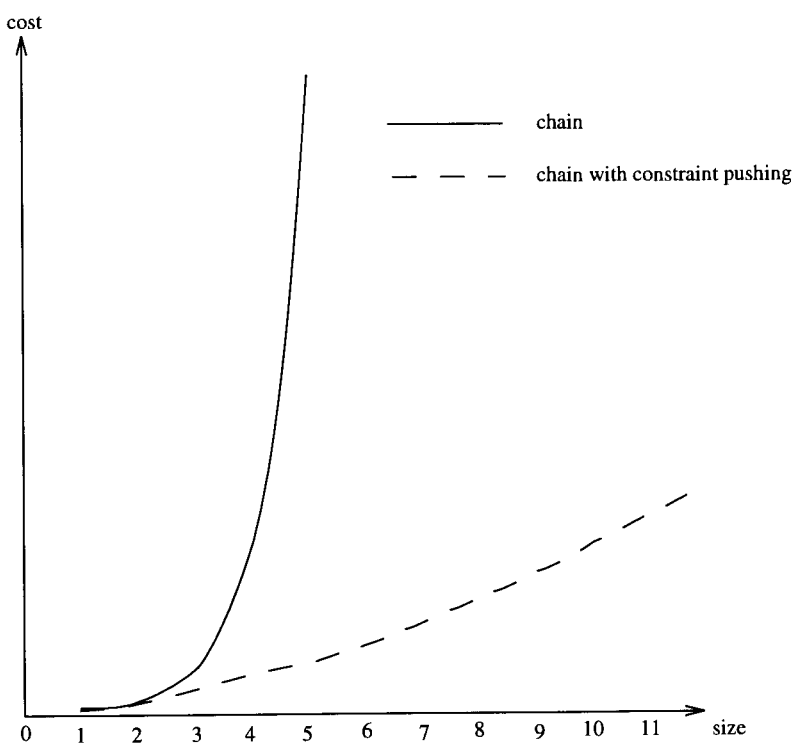


Figure 2.9: Effectiveness of constraint pushing for $permutation\_sort^{bf}$.

Chapter 3

Design and Implementation of LogicBase

3.1 Motivation

As an important extension to the relational approach, research on deductive database systems represents a direction towards declarative query processing, high-level database programming, and integration of logic programming and relational database technology [122]. Many deductive database systems or prototypes, such as LDL [28], Glue-NAIL! [87], CORAL++ [125], EKS-V1 [142], ADITI [135], XSB [118], have been developed and reported in recent years.

Efficient query evaluation in deductive databases is an essential issue in the realization of deductive database systems. Previous researches [10, 61, 65, 133, 28, 87, 105, 135] lead to two influential classes of deductive query evaluation methods: (1) bottom-up evaluation, represented by the magic sets computation and the semi-naive evaluation [10, 28, 87, 105, 135], and (2) top-down evaluation, represented by the query/subquery approach [142] and XSB [118]. These methods explore set-oriented evaluation, focus of the search on query relevant facts, with freedom of looping and

easy termination testing, and have achieved impressive results. However, because a recursion is more or less treated as a black box by these methods without a detailed analysis of its particular structure, it is difficult to capture the regularities of a particular recursion and maximally utilize the information about constraints and recursion structures in query evaluation.

The *LogicBase* project takes a different approach. *LogicBase* emphasizes efficient compilation and query evaluation of *application-oriented recursions* in deductive databases. It adopts *query-independent compilation* and *chain-based query evaluation*, where the former [58] transforms a set of deduction rules into highly regular compiled forms, which facilitates quantitative analysis of queries and efficient query evaluation; whereas the latter explores set-oriented evaluation of each compiled chain with appropriate constraint transformation and push, which reduces unnecessary or redundant computation and facilitates the judgement of termination. The method can be viewed as a natural extension to relational query evaluation methods and an integration of a top-down evaluation (by starting with the query as a goal) and a bottom-up evaluation (by set-at-a-time evaluation without infinite looping and repeated computation of subgoals).

The design goals for *LogicBase* are:

- *realization of pure declarative logic programming*: the evaluation of a program should be independent of the ordering of rules in the program and ordering of predicates in a rule.
- *handling of functional program*: function symbols should be allowed in a program, efficient and safe evaluation should be provided.
- *support of efficient evaluation*: set-oriented evaluation similar to that in the relational approach should be supported.
- *exploration of constraints*: to provide safety for evaluation of functional program and to reduce search space.

3.2 Major Features of *LogicBase*

Although recursions can be in complex forms, most recursions in *practical applications* can be compiled into *chain* or *chain*-like forms to which efficient query analysis and evaluation techniques can be explored [58, 57]. The design of *LogicBase* is based on this regularity of recursion and the strength of chain-based compilation and evaluation.

The *LogicBase* system has the following major features:

- *Query-independent compilation*, which captures the bindings that could be difficult to be captured otherwise and derives highly-regular and precise compiled chain programs for query analysis and evaluation.
- *Chain-based evaluation*, which includes a set of interesting techniques, such as chain-following, chain-split, constraint pushing, etc., explores query constraints, integrity constraints, recursion structures, and other features of the programs in query evaluation.
- *Efficient processing of logic programs with functions, lists and complex data structures.* The programs in *LogicBase* are declarative, independent of the ordering of predicates in a rule and the ordering of rules and facts in a program. Queries in different input/output mode combinations can be processed properly.

  For example, in the $nqueens(N, Qs)$ recursion shown in Figure 2.1, the predicates or rules in the program can be swapped randomly, and the queries, such as "$? - nqueens(8, Qs)$" (the 8-queens problem), and "$? - nqueens(N, [2, 4, 1, 3])$" (whether this is a valid n-queens chess board), can be answered correctly and efficiently [56, 54, 53].

3.2.1 Capture of more bindings in query binding propagation

LogicBase compiles complex recursions into highly regular chain forms. By such

compilation, the selection-pushing technique can capture more bindings in complex recursions than those using traditional rule rewriting techniques, such as the magic rule rewriting [9, 133, 10]. This is illustrated by the following example [58].

Example 3.1 Traditional rule rewriting techniques may encounter some difficulties in the propagation of bindings in some recursive rules [58], which is demonstrated in the analysis of the following recursion.

Suppose that query "$? - r(c, c_1, Y)$" is posed on a linear recursion defined by Rules (3.1) and (3.2), where c and c_1 are constants, X's and Y's are variables, and r is a recursive predicate defined by EDB predicates a, b and e.

$$r(X_1, X, Y) \ :- \ e(X_1, X, Y). \tag{3.1}$$

$$r(X_1, X, Y) \ :- \ a(X, Y),\ r(X_2, X_1, Y_1),\ b(X_2, Y_1). \tag{3.2}$$

Following the binding propagation rules [10, 133], the bindings in the adorned goal, r^{bbf}, are propagated to the subgoal r in the body of Rule (3.2), resulting in an adorned subgoal, r^{fbf}, as shown in Rule (3.3), which are in turn propagated to the next expansion, resulting in r^{fff}, as shown in Rule (3.4), which cannot propagate any bindings further to subsequent expansions, and the binding propagation terminates.

$$r^{bbf}(X_1, X, Y) \ :- \ a^{bf}(X, Y),\ r^{fbf}(X_2, X_1, Y_1),\ b^{bb}(X_2, Y_1). \tag{3.3}$$

$$r^{fbf}(X_2, X_1, Y_1) \ :- \ a^{bf}(X_1, Y_1),\ r^{fff}(X_3, X_2, Y_2),\ b^{bb}(X_3, Y_2). \tag{3.4}$$

This kind of binding propagation relies on the *backward* binding propagation only, in the sense that the bindings are propagated from the head to the body in a rule and from the IDB subgoal in the body of a rule to the head of the rule which unifies it. For this recursion, the propagation cannot reduce the set of data to be examined in the semi-naive evaluation because the derived magic set contains the entire data relations. Furthermore, it is easy to verify that reordering of the subgoals cannot improve the evaluation efficiency.

For such recursions, bindings should also be propagated *forward* from the body to the head in a rule and from the rule unifying the IDB subgoal to the corresponding

IDB subgoal in the body of the original rule. Such a propagation cannot be caught by the traditional approaches but can be captured by the compilation (or normalization) of linear recursions [58].

For this recursion, because the second and the third arguments of predicate r are defined only by non-recursive predicate a in Rule (3.2), a new predicate t can be introduced to define the first argument of r, such that r is defined as following:

$$\begin{aligned} r(X_1, X, Y) &:- e(X_1, X, Y). \\ r(X_1, X, Y) &:- a(X, Y), t(X_1). \end{aligned}$$

where $t(X_1)$ is intended to replace the remaining part in Rule (3.2): $r(X_2, X_1, Y_1)$, $b(X_2, Y_1)$. Hence t is defined as following:

$$\begin{aligned} t(X_1) &:- a(X_1, U), b(X_2, U), t(X_2). \\ t(X_1) &:- e(X_2, X_1, Y_1), b(X_2, Y_1). \end{aligned}$$

The above definition for r is the normalized, equivalent form of the original program. Obviously, the bindings of the query r^{bbf} can be propagated to any expansions in the normalized recursion. □

This example shows that a complex linear recursion can be compiled (normalized) into highly regular chain forms for efficient query analysis and evaluation. Many other complex recursions are also compiled successfully by *LogicBase* and generate highly regular chain recursions.

Although function-free recursions cover an interesting class of recursions in deductive databases, many recursions in practical applications contain function symbols, such as structured data objects, arithmetic functions, and recursive data structures (lists, trees, sets, etc.). By transforming functions into functional predicates, the compilation and evaluation techniques developed for function-free recursions can be extended to functional ones [47]. Furthermore, the method can be generalized to logic programs containing modularly stratified negation [105] and those with higher-order syntax and first-order semantics [26]. Therefore, compilation of recursions into chain

and pseudo-chain forms [50] represents a powerful program transformation technique which transforms recursion into simple, easily-analyzable forms and facilitates the application of efficient evaluation methods.

In general, deduction rule compilation in *LogicBase* consists of two major units: (1) *classification* (classification and simplification of recursions), and (2) *compilation* (compilation and normalization of recursions).

The classification unit takes a complex recursive program as input, rectifies it, eliminates mutual recursions when possible, simplifies the recursion when appropriate, and identifies the class of recursions to which the program belongs [57]. By this processing, a recursion is classified into one of the following classes: (1) (single) linear recursion, (2) nested linear recursion, (3) multiple linear recursion, (4) regular nonlinear recursion, and (5) irregular recursion [57].

The compilation unit (based on [58, 57]) takes the preprocessed recursion and compiles (normalizes) it into a chain program, when possible, based on a compilation (normalization) algorithm described in Section 3.3.2. Furthermore, algebraic simplification is performed on the compiled expressions. The compiled recursion is fed to query analysis and evaluation.

3.2.2 Chain-following and chain-split evaluation

Since many recursions can be compiled into chain forms, *chain-based evaluation* should be explored on the compiled recursions. Chain-based evaluation can be viewed as an extension to relational database query analysis and optimization techniques, because a compiled chain consists of an infinite set of highly regular relational expressions. The compilation makes explicit the regularity of the operation sequences in a recursion, on which quantitative analysis and optimization can be explored systematically. Such a quantitative analysis, similar to the access path selection and query plan generation for relational queries [119], can be performed based on the characteristics of the compiled chains, query instantiations, inquiries, integrity constraints, and database statistics of

EDB relations [47]. Notice that quantitative analysis has been incorporated in many other recursion handling methods to generate different query evaluation plans as well.

In general, the chain-based query evaluation method consists of *chain-following, chain-split*, and *constraint-based evaluation* techniques.

The simplest chain-based evaluation is *chain-following evaluation*, which starts with a highly selective end of a chain (called the *start end*) and proceeds towards the other end of the chain (called the *finish end*) and then possibly to other chains. It simulates partial transitive closure processing in the case of single chain recursion [65, 61] and the counting method [9, 44] in the case of multiple chain recursion.

Example 3.2 The following recursion *length* defined by Rules (3.5) and (3.6) can be compiled into a double-chain recursion.

$$length([], 0). \tag{3.5}$$

$$length([X|L], N+1) \ :- \ length(L, N). \tag{3.6}$$

Rule (3.7) shows the rectified recursive rule.

$$length(XL, N_1) :- cons(X, L, XL),\ plus(N, 1, N_1),\ length(L, N). \tag{3.7}$$

One chain has a chain element "$cons(X, L, XL)$", the other chain contains an element "$plus(N, 1, N_1)$". Query "$? - length([a, b, c], 3)$" can be evaluated by a typical chain following evaluation. Both chain are instantiated at query end. At each iteration, the third argument of *cons* and the second and the third arguments of the *plus* functional predicate are instantiated, and both predicates are evaluated. The instantiation is then passed to the next iteration via shared variables. Therefore, both chains are evaluated in the chain-exit direction. At last, the exit rule is satisfied, and the query is returned with answer *true*. If the query is "$? - length([a, b, c], N)$", it is evaluated by the chain following as well. The *cons* chain is evaluated in the chain-exit direction, but the *plus* chain in the exit-chain direction. In this case the chain-following method is the same as the counting method because synchronization is needed between two chains. □

Depending on the available query bindings, some functional predicates in a chain element may not be immediately finitely evaluable, or the iterative evaluation may generate a huge intermediate relation. In these cases, a chain can be partitioned into two portions: *immediately evaluable portion* and *buffered portion.* The former is evaluated at the begining but the latter is buffered until the exit rule is evaluated. Then the evaluation proceeds in a way similar to the evaluation of a multi-chain recursion, except that the corresponding buffered values should be patched in the latter evaluation. Such an evaluation technique is called *chain-split evaluation* [47].

Example 3.3 Rules (3.8) and (3.9) define a recursion *append*, which can be compiled into a single-chain recursion. For the query "$? - append(U, V, [a, b])$" whose adorned predicate is $append^{ffb}$, the adorned normalized rule set is shown in Rules (3.10) and (3.11).

$$append([], L, L). \tag{3.8}$$

$$append([X|L_1], L_2, [X|L_3]) : - \ append(L_1, L_2, L_3). \tag{3.9}$$

$$append^{ffb}(U, V, W) : - \ U =^{fb} [], \ V =^{fb} W. \tag{3.10}$$

$$append^{ffb}(U, V, W) : - \ cons^{ffb}(X_1, W_1, W), \\ append^{ffb}(U_1, V, W_1), \ cons^{bbf}(X_1, U_1, U). \tag{3.11}$$

Since the chain element "$cons(X_1, U_1, U), \ cons(X_1, W_1, W)$" cannot be finitely evaluated as a whole based on the only available binding on W, the chain-split evaluation technique should be applied in the evaluation. That is, the chain should be split into two portions: (1) the *immediately evaluable predicate* "$cons(X_1, W_1, W)$", and (2) the *buffered predicate* "$cons(X_1, U_1, U)$".

The evaluation proceeds as follows. The evaluation of the exit rule (3.10) derives the first set of answers: "$U = []$" and "$V = [a, b]$". The evaluation of the recursive rule (3.11) proceeds along the immediately evaluable predicate "$cons(X_1, W_1, W)$", which derives "$W_1 = [b]$" and "$X_1 = a$" from "$W = [a, b]$". Then X_1 is buffered, and W_1 is passed to the exit rule, making "$V = [b]$" and "$U_1 = []$". Then the buffered predicate

becomes evaluable since X_1 and U_1 are available. The evaluation derives "$U = [a]$". Thus, the second set of answer is $\{U = [a], V = [b]\}$. Similarly, the evaluation may proceed along the immediately evaluable predicate "$cons(X_1, W_1, W)$" further, which derives the third set of the answer: $\{U = [a, b], V = []\}$. □

3.2.3 Constraint-based query evaluation

Besides the distinction of chain-following vs. chain-split evaluation, another important strength of the method is the systematic analysis and exploration of available constraints [47].

Taking the evaluation of a single-chain recursion as an example, we examine how to push query constraints (or instantiations) at both ends of a compiled chain. The processing should start at a more restrictive end (the *start end*) and proceeds to a less restrictive end (the *finish end*). It is straightforward to push query constraints at the start end of the chain. However, care should be taken when pushing query constraints at the finish end.

Example 3.4 An IDB predicate, $travel(FnoList, Dep, Arr, Fare)$, defined by Rules (3.12) and (3.13), represents a sequence of connected flights with the initial departure city Dep, the final arrival city Arr, and the total fare $Fare$, where edb_flight is an EDB predicate representing the stored flight information.

$$travel([Fno], Dep, Arr, Fare) :-$$
$$\quad edb\_flight(Fno, Dep, Arr, Fare). \tag{3.12}$$
$$travel([Fno|FnoList], Dep, Arr, Fare) :-$$
$$\quad edb\_flight(Fno, Dep, Int, F_1),$$
$$\quad travel(FnoList, Int, Arr, F_2),\ Fare = F_1 + F_2. \tag{3.13}$$

The recursion can be compiled into a single-chain recursion shown in Rules (3.14)

and (3.15).

$$travel(L, D, A, F) :- edb\_flight(Fno, D, A, F),$$
$$cons(Fno, [], L),\ sum(F, 0, F). \tag{3.14}$$
$$travel(L, D, A, F) :-$$
$$edb\_flight(Fno, D, I, F_1),\ sum(F_1, S_1, F),$$
$$cons(Fno, L_1, L),\ travel(L_1, I, A, S_1). \tag{3.15}$$

The query is to "find a set of (connecting) flights from Vancouver to Zurich, with at most 4 hops and with the total fare between \$500 to \$800", that is,

$$? - travel(FnoList, vancouver, zurich, F),$$
$$F \geq 500,\ F \leq 800,\ length(FnoList, N),\ N \leq 4.$$

According to the compiled form, D, L and F are located at the query end of the chain; whereas A, L_1 and S_1 are at the exit end of the chain. The information at the query end is, (i) $D =$ "$vancouver$", (ii) $500 \leq F \leq 800$, and (iii) $FnoList = L, length(FnoList, N), N \leq 4$; whereas that at the exit end is, (i) $A =$ "$zurich$", (ii) $L_1 = []$, and (iii) $S_1 = 0$.

Since the information at the exit end is more selective than that at the query end, the exit end is chosen as the *start end* for chain evaluation (query end as the *finish end*). Thus, all the query constraints at exit end are pushed into the chain for efficient processing.

The query constraints associated with the finish end cannot be pushed into the chain in iterative evaluation without additional information. For example, pushing the constraint, "$Fare \geq 500$", into the chain will cut off a promising connection whose first hop costs less than 500. On the other hand, it is clearly beneficial to push the constraint, "$Fare \leq 800$", into the chain to cut off the hopeless connections when the accumulative fare is already beyond 800. However, a constraint like "$Fare = 800$" cannot be pushed into the chain directly, but a transformed constraint, "$Fare \leq 800$", can be pushed in for iterative evaluation.

A systematic way to push query constraints at the finish end can be derived by examining the interactions between query constraints and monotonicity constraints [47]. If the value (or the mapped value) of an argument in the recursive predicate monotonically increases but does not converge to a limit during the evaluation, a query constraint which blocks such an increase is useful at reducing the search space in iterative evaluation.

Based on the monotonicity constraint of the argument $Fare$, a *termination restraint template*, "$Fare \not> C$", is set up, where C is a variable which can be instantiated by a *consistent* query constraint. For example, a constraint, "$Fare \leq 800$", or "$Fare = 800$", instantiates the template to a *concrete termination restraint*, "$Fare \not> 800$". However, the constraint, "$Fare \geq 500$", is not consistent with the termination restraint template. Thus, it cannot instantiate a termination restraint. An instantiated termination restraint can be pushed into the chain for efficient processing.

Similarly, a constraint, "$Dep =$ '*vancouver*'", can be used for constraint pushing if we have the airport location information and a constraint: *same flight direction* (a monotonic constraint on flight direction). A concrete termination restraint, such as "$longitude(Dep) \not> longitude(vancouver)$", can be derived from the analysis of the query constraints and monotonicity constraints of the recursion, and the tuples generated at any iteration with the departure airports located to the west of $Vancouver$ is pruned in the chain processing. Also, the constraint, "$length(FnoList, N)$, $N \leq 4$", can be pushed into the chain in the iterative evaluation. □

Because of the availability of compiled chains and their precise connection information, it is much easier to perform a detailed analysis of the monotonicity behavior of each chain and perform appropriate constraint transformation and constraint pushing for efficient evaluation.

3.2.4 Chain-based evaluation of complex classes of recursions

Since a (single) linear recursion can be compiled into a chain form or a bounded recursion [58], chain-based evaluation can be applied to this class of recursion. Similarly, a *nested linear recursion* can also be so compiled and evaluated. The evaluation of nested linear recursion can be illustrated in the following.

Example 3.5 A typical n-queens recursion defined in Figure 2.1 is a nested linear recursion whose query analysis can be performed as follows.

For a query, "? $- nqueens(4, Qs)$", the binding pattern for predicate *nqueens* is $nqueens^{bf}(N, Qs)$. The bf binding of *nqueens* leads to $range^{bbf}$ and $queens^{bbf}$ if *range* is evaluated first. Notice that it is unreasonable to evaluate *queens* first since it is easy to verify that the binding $queens^{fbf}$ cannot lead to a finite set of answers. Thus, the adorned program for *nqueens* is as follows.

$$nqueens^{bf}(N, Qs) : -range^{bbf}(1, N, Ns),\ queens^{bbf}(Ns, [], Qs).$$

Similarly, the adorned program for the remaining program is presented below. Notice that "$M_1 = M +^{fbb} 1$" denotes the bindings of the three arguments of the plus predicate.

$$range^{bbf}(M, N, MNs) : - M <^{bb} N, M_1 = M +^{fbb} 1,$$
$$range^{bbf}(M_1, N, Ns),\ cons^{bbf}(M, Ns, MNs). \quad (3.16)$$
$$range^{bbf}(M, N, MNs) : - M =^{bb} N,\ cons^{bbf}(N, [], MNs). \quad (3.17)$$

$$queens^{bbf}(U, S, Qs) : - select^{fbf}(Q, U, U_1),\ not\ attack^{bb}(Q, S),$$
$$cons^{bbf}(Q, S, S_1),\ queens^{bbf}(U_1, S_1, Qs). \quad (3.18)$$
$$queens^{bbf}(U, S, Qs) : - U =^{bb} [],\ S =^{bf} Qs. \quad (3.19)$$

$$select^{fbf}(X, YYs, YZs) : - cons^{ffb}(X, YZs, YYs). \quad (3.20)$$
$$select^{fbf}(X, YYs, YZs) : - cons^{ffb}(Y, Ys, YYs),$$

$$select^{fbf}(X, Ys, Zs),\ cons^{bfb}(Y, Zs, YZs). \tag{3.21}$$

$$attack^{bb}(X, Xs) : - attk^{bbb}(X, 1, Xs). \tag{3.22}$$

$$attk^{bbb}(X, N, YYs) : -$$
$$(X = Y +^{bfb} N;\ X = Y -^{bfb} N),$$
$$cons^{bfb}(Y, Ys, YYs). \tag{3.23}$$

$$attk^{bbb}(X, N, YYs) : - cons^{bfb}(Y, Ys, YYs),$$
$$N_1 = N +^{fbb} 1,\ attk^{bbb}(X, N_1, Ys). \tag{3.24}$$

The binding propagation analysis determines both the appropriate query evaluation strategy and the predicate evaluation order. For example, the analysis on adorned program $range^{bbf}$ indicates that chain-split evaluation should be performed on $range^{bbf}(M,\ N,\ MNs)$ because the compiled chain "$M < N,\ M_1 = M + 1,\ cons(M,\ Ns,\ MNs)$" must be split into two portions: "$M < N,\ M_1 = M + 1$" and "$cons(M,\ Ns,\ MNs)$", in the evaluation (in order to guarantee finite evaluation) according to the binding propagation ordering shown in (3.16). Similarly, chain-following evaluation should be performed on $queens^{bbf}(U, S, Qs)$, chain-split evaluation on $select^{fbf}(X,\ YYs,\ YZs)$, and existence-checking evaluation [44] on $attk^{bbb}(X, N, YYs)$.

Queries with other adornments can be analyzed and evaluated similarly. For query "$? - nqueens(N, [2, 4, 1, 3])$", the predicate $queens$ should be evaluated first. Otherwise, it is unsafe to evaluate $range^{bff}$. Thus, the adorned program becomes,

$$nqueens^{fb}(N, Qs) : -$$
$$queens^{fbb}(Ns, [], Qs),\ range^{bfb}(1, N, Ns). \tag{3.25}$$

$$queens^{fbb}(U, S, Qs) : -$$
$$queens^{ffb}(U_1, S_1, Qs),\ cons^{fbb}(Q, S, S_1),$$
$$not\ attack^{bb}(Q, S),\ select^{bfb}(Q, U, U_1). \tag{3.26}$$

$$queens^{ffb}(U, S, Qs) : -$$
$$queens^{ffb}(U_1, S_1, Qs),\ cons^{ffb}(Q, S, S_1),$$

$$not\ attack^{bb}(Q,S),\ select^{bfb}(Q,U,U_1). \tag{3.27}$$

$$queens^{ffb}(U,S,Qs) :- U =^{fb} [],\ S =^{fb} Qs. \tag{3.28}$$

$$select^{bfb}(X,YYs,YZs) :- cons^{bbf}(X,YZs,YYs). \tag{3.29}$$

$$select^{bfb}(X,YYs,YZs) :- cons^{bfb}(Y,Zs,YZs),$$
$$select^{bfb}(X,Ys,Zs),\ cons^{bbf}(Y,Ys,YYs). \tag{3.30}$$

$$range^{bfb}(M,N,MNs) :-$$
$$cons^{bfb}(M,Ns,MNs),\ M_1 = M +^{fbb} 1,$$
$$range^{bfb}(M_1,N,Ns),\ M <^{bb} N. \tag{3.31}$$

$$range^{bfb}(M,N,MNs) :- M =^{bf} N,\ cons^{bbb}(N,[],MNs). \tag{3.32}$$

Binding propagation analysis derives the predicate evaluation ordering: first evaluate $queens^{fbb}(Ns,[],Qs)$ and then $range^{bfb}(1,N,Ns)$. Similarly, $queens^{fbb}(Ns,[],Qs)$ implies that $queens(U_1,S_1,Qs)$ (the recursive predicate in the body) should be evaluated first. The bindings in the head $queens^{fbb}$ are propagated to the body as shown in Rule (3.26), which results in $queens^{ffb}(U_1,S_1,Qs)$. The further propagation in the recursive rule leads to the same binding pattern in the recursive predicate in the body, as shown in Rule (3.27). This evaluation order can be naturally viewed as evaluating first the exit rule portion and then the chain portion in the chain-based evaluation. Based on such binding analysis, appropriate query evaluation plans can be generated and queries can be processed efficiently. □

3.3 Implementation of *LogicBase*

In this section, the design principles and implementation details for the *LogicBae* system are presented. *LogicBase* consists of three phases: compilation of logic programs, generation of an evaluation plan and execution of the plan. The overall picture of *LogicBase* is introduced first, followed by algorithms for each phase.

3.3.1 *LogicBase* system architecture

Figure 3.1 presents an overview of *LogicBase*.

1. Compilation of a program: Classification, normalization and compilation of a program.
2. Plan generation: Analysis of binding passing, determining evaluation direction for each chain, determining termination for the plan, generating an evaluation plan and optimizing the query plan.
3. Plan execution: Execution of relational operations according to the generated plan.

Figure 3.1: Overview of *LogicBase*.

A more detailed description of the three phases in *LogicBase* is as follows:

1. *Compilation*: Recursive programs are first classified into the following classes: (1) (single) linear recursion, (2) nested linear recursion, (3) multiple linear recursion, (4) regular nonlinear recursion and (5) irregular recursion. Then they (except irregular nonlinear recursion) are compiled and normalized into chain or chain-like forms to facilitate query processing. Linear mutual recursions are compiled into one or a set of linear recursions or pseudo-linear recursions which can be evaluated in the same way as normal linear recursions. The compilation is query-independent.
2. *Query plan generation and query optimization*: A systematic and quantitative analysis is performed to determine an appropriate evaluation strategy and to generate a query processing plan. *LogicBase* takes the binding passing on the compiled chain forms and evaluation cost into consideration to select a query processing strategy. For each chain, bindings can be propagated in either directions: (1) *chain-exit*: where bindings are first passed from the query to the chain (via the query end of the chain), then through the chain (via shared variables

among successive chain elements), finally to the exit rules (via the exit end of the chain); (2) *exit-chain*: where bindings in the query are passed to the exit rule via shared variables between the head predicate and the recursive predicates in the body, then from the exit rule into the chain (via the exit end of the chain) and passed through the chain (via shared variables in successive chain elements), finally bindings are passed back to the query (via the query end of the chain). According to the binding propagation, query processing strategies are categorized into the following:

- *non-recursive processing*, which handles a query on a non-recursive program.
- *all-up chain following evaluation*, where all the chains are evaluated in the chain-exit direction. It is similar to the top-down evaluation with respect to the binding passing.
- *all-down chain following evaluation*, where all chains are evaluated in the exit-chain direction. It is similar to the bottom-up evaluation with respect to the way bindings are passed.
- *up-down chain following evaluation*, where some chains are evaluated in the chain-exit direction and the rest are evaluated in the exit-chain direction. It is similar to the counting method.
- chain-split evaluation, where a chain is split into two portions: *immediately evaluable portion* and *buffered portion* such that the immediately evaluable portion is evaluated first in the chain-exit direction whereas the buffered portion is buffered and its evaluation is delayed until all of the immediately evaluable portion of the chain is fully evaluated and the bindings from the chain-exit evaluation are passed to the exit rule, such bindings are then passed to the buffered portion to enable evaluation on the buffered portion.

Termination analysis and constraint pushing are also important tasks done at this stage. Termination on evaluation of function-free program is guaranteed by the chain-based evaluation. However, for a functional program, analysis on

monotonic constraints is employed to ensure termination of chain-based evaluation. In *LogicBase*, an evaluation plan for a functional program is considered to be safe only if a constraint is found in the program to restrain an argument whose values monotonically increase or decrease. If no such constraint(s) can be located, *LogicBase* strives to push other constraints found in the query or the exit rules if possible. For an evaluation plan of functional program without such constraints, the plan is abandoned and another alternative plan is examined.

Besides plan generation, plan optimization is performed in this stage. Given a valid and safe evaluation plan, statistics on EDB relations is used to give an estimated cost for the query evaluation under the plan. When multiple strategies are applicable, the one with the least estimated cost is taken as the evaluation plan. The query plan can be further optimized by rearranging the order of predicates within the plan, which can be accomplished by making use of query optimization techniques in the relational database, because evaluation in chain-based approach is an extension to the relational database query processing.

3. *Plan execution*: The query evaluation plan, generated at the last phase, is executed to obtain answers to the query. A query plan is represented in an internal data structure which specifies what needs to be done in plan execution. Various relational operations such as relational join, selection, projection, set difference, are performed according to the plan.

The following subsections will introduce techniques and algorithms used in these three phases in detail.

3.3.2 Compilation of linear recursive programs

The methods and algorithms presented here for the automatic compilation of linear recursive programs are from Han and Zeng [58]. They are included here to present a complete view for the *LogicBase* system.

A variable connection graph-matrix, *V-matrix* is used to register the structure of a linear recursive program, and also to discover the minimal necessary expansions in the compilation of complex linear recursions. Furthermore, the compiled forms of a linear program can be generated automatically through the expansion operation on V-matrix.

The compiled form of a recursive program is generated by expanding recursive rules and identifying regularity of the expansion. The expansion behavior of a recursion is closely related to the variable connections among its predicates.

The following concepts are used to introduce the necessary algorithms for automatic generation of chain form, to which the compilation portion of the *LogicBase* system is implemented accordingly.

Definition 3.1 *Two predicates in the body of a rule are* connected *if they share variable(s) with each other or with a set of connected predicates. Two nonrecursive predicates in the body of a rule are* U-connected *if they share variable(s) with each other or with a set of U-connected predicates. A set of variables are* U-connected *if they are in the same nonrecursive predicate or in the same set of U-connected (nonrecursive) predicates.*

Definition 3.2 *The variable connection graph-matrix, V-matrix, for a linear recursive rule of arity n (the arity of the head predicate) consists of a sequence of rows. Each row consists of n columns with the i-th one corresponding to the i-th argument position of the recursive predicate. Moreover, there are possibly U-connection edges between some columns in a row.*

The contents of an initial V-matrix reflect the variable connection information in the corresponding arguments in the original recursive rule. Its expansions reflect similar information in the expanded recursive rules. The compilation procedure in *LogicBase* expands recursion by manipulating V-matrix.

The initial V-matrix, which consists of the first two rows (row [0] and row [1]) of the V-matrix, is constructed according to the following V-matrix initialization rules shown as follows, while the remaining rows, if any, are constructed based on the V-matrix expansion rules to be presented later.

A V-matrix is initialized in the following four steps:

1. Partition the variables in the rule according to the U-connections (and each partition is called a *U-connected set*);
2. Copy the variables in the recursive predicate in the head and the body to the corresponding columns in row [0] and row [1] respectively;
3. Replace the variable at each column of row [1], say x, by the set of distinguished variables U-connected with x, if any; and
4. Set up a U-connection edge between each pair of columns in the corresponding row if the pair of columns are in row [0] and contain U-connected distinguished variables, or if they are in row [1] and contain U-connected nondistinguished variables.

A V-matrix can be partitioned into one or more *unit V-matrices* based on the connections among matrix columns.

Definition 3.3 *Two columns of a V-matrix are* connected *if the two columns in the initial V-matrix share a variable or a set of U-connection edges with each other or with a set of connected columns. A set of connected columns form a* unit V-matrix. *A linear recursive rule whose V-matrix consists of only one unit is a* single-unit rule*; otherwise, it is a* multiple-unit rule.

New rows of a V-matrix can be generated from its initial V-matrix by a set of V-matrix expansion rules, and the generated rows reflect the U-connectivities of the corresponding expanded recursive rules.

Definition 3.4 *A variable y is a* derivative *of a distinguished variable x in a V-matrix if y is derived by x, that is, y and x are at the same column in the V-matrix but y's row number is x's row number + 1.*

In general, the V-matrix expansion rules can be summarized as follows, where the row NewRow (= LastRow + 1) is generated from the row LastRow of the V-matrix:

1. *Row generation*: for each distinguished variable x in V-matrix [LastRow, i], add x's derivatives to V-matrix [NewRow, i].

2. *U-connection Propagation*: the U-connection edges are copied from LastRow to NewRow and then from LastRow -1 to LastRow. If such copying makes a distinguished variable x U-connected to the set of variables in V-matrix [NewRow, i], x is added to the set of variables in V-matrix [NewRow, i].

It is proved in [58] that each row of the V-matrix generated by following the above V-matrix expansion rules correctly registers the set of distinguished variables U-connected to each column of the recursive predicate in the body at each expansion. Furthermore, after certain number of expansions (*stable level*), newly generated rows start to repeat rows generated some number of expansion (*period*) before in the V-matrix. Such stable level is no greater than the arity of the head predicate.

Equality of two rows in a V-matrix is defined in the following.

Definition 3.5 *The* DV-set *of a column is the set of all the distinguished variables U-connected to the variable(s) in the column. Two rows, row [i] and row [j], in a V-matrix are* identical *(denoted as row[i] == row[j]) if each pair of their corresponding columns has the same DV-set.*

Based on such definition of identicality, for a single-unit recursive rule of arity n, it can be proved that there exists integers S (stable level) and T (period) such that for any k-th expansion with $k > S$, it is identical to $(k-T)$-th expansion in its V-matrix, and $S + T \leq n$.

Definition 3.6 *If starting at row S, there exists a T such that the row of a single-unit V-matrix repeats at every T more expansions, that is, row $[S + k \times T] == row\ [S]$ for all $k > 0$, then S is called the* stable level *and the smallest T the* period *of the V-matrix. If row [S] contains no distinguished variables, T is defined as 0.*

The following algorithm is used in *LogicBase* for the expansion of a single-unit V-matrix and the derivation of its stable level S and the period T.

Algorithm 3.1 *The expansion of a single-unit V-matrix and the derivation of its stable level S and the period T.*

Input: An initial single-unit V-matrix.

Output: An expanded V-matrix, the stable level S and the period T.

Method: **begin**

```
LastRow := 0; CurrentRow := 1;
while not RowRepeating (CurrentRow, ExistingRow)
   LastRow := CurrentRow; CurrentRow := CurrentRow + 1;
   /* Generate the contents of the CurrentRow. */
   for each column i /* Every column in CurrentRow is initially empty. */
      for each distinguished variable x in V-matrix[LastRow, i]
         Add x's derivatives to V-matrix[CurrentRow, i];
   /* U-connection Propagation. */
   Copy the U-connections from LastRow to CurrentRow;
   Copy the U-connections from LastRow - 1 to LastRow;
   for each column i
      for each x in V-matrix[CurrentRow, i]
         if x is U-connected to a distinguished variable y which is not already in
            V-matrix[CurrentRow, i]
         then Add y to V-matrix[CurrentRow, i] and remove, if any,
            nondistinguished variables there;
S := ExistingRow;
```

if there is no distinguished variable in CurrentRow
then $T := 0$;
else $T :=$ CurrentRow - ExistingRow;
end. □

Notice the *RowRepeating* is a Boolean function which returns true if the there is a row in the V-matrix called ExistingRow identical to the CurrentRow. Such function is used to identify the stable level S and the period T and terminate the expansion of the V-matrix.

The generation of chain-predicates for nonnull chains from the recursive rules in the $(S + T)$-th expansion consists of the following three steps:

1. Take the set of nonrecursive predicates generated from the $(S + 1)$-st expansion to the $(S + T)$-th expansion as the candidate set of the chain predicates.

2. Replace the predicates in the candidate set which are not U-connected to any set of distinguished variables or which corresponds to a distinct DV-set but are not U-connected together by their corresponding predicates in the previous expansions to make the predicates corresponds to a DV-set U-connected together. Each chain is a set of predicates in the replaced set corresponding to a distinct DV-set.

3. Rename and index the variables in the $(S+T)$-th expansion. Ignore the variables not shared with any predicate outside of the chain. Let the set of variables in the recursive predicate at the S-th and the $(S + T)$-th expansions be *S-set* and *ST-set*, respectively. For the remaining variables in the chain predicate, rename and index them when necessary to make each variable in the ST-set have the same name as the corresponding variable in the S-set but with the index increased by one. If a variable in the ST-set appears also in the S-set, the same variable should be the same (name and index) in the new set of variables. Renaming and indexing of a variable should be performed consistently for every occurrence of the variable.

Algorithm 3.2 *Generation of the compiled form for a single-unit linear recursion.*

Input: A linear recursion R, its stable level S and period T.

Output: The compiled form of the recursion.

Method. Case 1: $T = 0$. The recursion is bounded and the compiled form is the union of the expanded exit rules from 0-th to S-th expansions. That is,

$$R(x_1, x_2, \ldots, x_n) = E_0(x_1, x_2, \ldots, x_n) \cup E_1(x_1, x_2, \ldots, x_n) \cup \ldots, \cup E_S(x_1, x_2, \ldots, x_n).$$

Case 2: $T \neq 0$. The compiled form for the recursion can be generated as follows:

If a recursion contains only null chain predicates, it is bounded and its compiled form is the union of the k-th expanded exit rules for $0 \leq k \leq S + T - 1$. That is:

$$R(x_1, x_2, \ldots, x_n) = E_0(x_1, x_2, \ldots, x_n) \cup E_1(x_1, x_2, \ldots, x_n) \cup, \ldots, \cup E_{S+T-1}(x_1, x_2, \ldots, x_n).$$

Otherwise, the recursion is a single- or multiple- chain recursion with the following compiled form:

$$R = SS \cup (\bigcup(MM, CC_i, TT)),$$

which consists of four portions: (i) prestable exit rule portion (SS), (ii) miscellaneous portion (MM), (iii) chain-portion (CC), and (iv) stable exit rule portion (TT).

The SS-portion represents the rule expansion before reaching its stable stage, which is the union of E_k for k from 0 to $S - 1$ if $S > 0$ or empty otherwise. That is:

$$SS = \bigcup_{i=0}^{S-1} E_i(x_1, x_2, \ldots, x_n).$$

The TT-portion consists of the bodies of the exit rules contributing to the period of the recursion, which is formed by the union from E_0 to E_{T-1}. That is:

$$TT = \bigcup_{j=0}^{T-1} E_j(\ldots, u_i, \ldots, x_i, \ldots).$$

The chain-portion CC consists of a set of nonnull chains in the exponential form with the same exponent i. Each nonnull chain predicate is generated following the chain-generating rules and is in the form of $A(x_{i-1}, x_i)$, where A is the chain predicate, and x_{i-1} and x_i are connection variable vectors. The formula consists of a set of unions starting from $i = 0$ to infinity. The variable indices outside of the chain predicates should be set accordingingly based on $i = 1$.

Finally, the miscellaneous portion, MM, if any, is composed of the predicate(s) left in the $(S+T)$-th expansion, i.e., those not used in the formation of the chain predicate(s). □

The V-matrix expansion and compilation of chains on single unit linear recursions are generalized to multiple unit ones. Because each unit reaches its own stable level independent of other units, the stability for the whole can be determined by examining each unit separately. Thus the stable level for multiple level recursion is the maximum of the stable level of each unit.

Algorithm 3.3 *The expansions of a multiple-unit V-matrix.*

Input. An initial V-matrix V which is partitioned into k unit V-matrices, $V_1, \ldots, V_k$.

Output. A stable level S of the V-matrix and the period T_i $(1 \leq i \leq k)$ for each unit V-matrix V_i.

Method. For each unit V-matrix V_i, derive its S_i and T_i based on Algorithm 3.1. Then $S = maximum(S_1, \ldots, S_k)$, and each unit V_i maintains its own period T_i.

□

The compiled form for multiple unit recursion can be derived by merging independent compiled form of each unit. In order to generate a combined compiled form, all independent compiled forms should be aligned to a common S, and T should be the least common multiplier of the nonzero T's of each unit. The following algorithm describes how to generate compiled chain form for multiple unit recursions.

Algorithm 3.4 *Generation of the compiled form for a multi-unit linear recursion.*

Input: A multiple-unit linear recursion R, its stable level S and the period T_i ($1 \leq i \leq k$) for each unit V_i.

Output: The compiled form of the recursion.

Method. Generation of compiled form consists three steps:

1. For each unit V-matrix V_i, generate its compiled form R_i according to Algorithm 3.2;
2. Generate the aligned compiled form for each unit V_i based on the common stable level S and the common period T, where $S = max(S_1, \ldots, S_k)$ and $T = lcm(T_1, \ldots, T_k)$;
3. Merge the multiple aligned compiled forms into one combined compiled form in which:
   (a) the SS-portion consists of the union of E_0 to E_{S-1} if $S > 0$ or empty otherwise;
   (b) the TT-portion consists of the union of E_i's for i from 0 to $T - 1$;
   (c) the chain-portion consists of all the nonnull chains, with each chain predicate determined within its unit and then aligned up for merging. All the chain predicates are in the exponential form with the same exponent i, and each variable connected to the set of distinguished variables is in the form of x_{i-1} for a distinct x, and that connected to the set of variables in the recursive predicate is in the form of x_i; and

(d) the MM-portion consists of the predicates at the $(S+T)$-th expansion which does not participate the chain predicates. □

3.3.3 Plan generation

The second phase of query evaluation in *LogicBase* is the plan generation, in which a compiled chain form of a recursion is passed from the compilation phase, and a plan generator analyzes whether a given query can be answered on the compiled form and determines how bindings should be propagated along each chain. Due to the presence of functions and multiple level recursions, several issues need to be addressed for the plan generator phase:

- *binding propagation*, to consider how the binding from the query instantiation can be propagated in the chain. Bindings can be passed in either chain-exit or exit-chain direction. The plan generator determines the evaluation direction for each chain to provide sufficient binding passing and evaluation efficiency.

- *safety of plan*, to make sure that an evaluation will terminate for functional programs. Given a tentative evaluation plan for a functional program, if there exists a constraint or base relation bounding an monotonic argument, the evaluation under the plan is safe.

- *query optimization by constraint pushing*, in some cases even though a program does not contain a constraint, an external constraint in its query or exit rules can be pushed into the program and can be served as a bound to a monotonic argument, therefore the program can be safely evaluated. Another benefit of such query optimization is reduced search space for some query evaluation.

- *nested recursive program*, where queries and answers are exchanged between rules at different deduction levels. During query evaluation in a higher level rule, an IDB predicate appeared in the rule body is queried with instantiation from the rule. Such query is processed in a program with lower deduction

level and the answer is used in query evaluation in the higher level rule. Such query processing is accomplished by recursively calling the query processor in *LogicBase*. Besides the exchange of query and answers, constraints need to be exchanged as discussed in chapter 2.

Following is the algorithm for query plan generation in *LogicBase*, the actual plan is stored in an internal data structure which is filled during plan generation phase and accessed during plan execution phase.

Algorithm 3.5 *Plan generation.*

Input: query predicate q, binding b.

Output: true if evaluation plan for q is generated, false otherwise. Evaluation plan is stored in a structure *Plan*.

Method: the algorithm is as following:

```
procedure plan_generator(q, b)
begin
  Plan := ∅
  if q is an EDB predicate
  then
    register q in Plan
    return true
  else if q is a functional symbol
  then
    if q is evaluable under b
    then
      register q in Plan
      return true
    else
      return false
  else if q is a non-recursive IDB predicate defined by rule r
```

then

store in I the instantiated variables in the r's head under binding b

$Uneval$:= set of all predicates in the r's body

return *predicate_set_evaluable*($Uneval, I$)

else if q is a recursive IDB predicate defined by a recursive rule r and an exit rule *exit*

then

if *chain_exit_plan*($r, exit, b$)=true

/* strategy is chain-exit chain following */

then return true

else if *exit_chain_plan*($r, exit, b$)=true

/* strategy is exit-chain chain following */

then return true

else if *chain_split_plan*($r, exit, b$)=true

then return true

else return false /* none of the strategies works */

end □

It should be noticed that *plan_generator* is a recursive algorithm to deal with nested program. The algorithm for function *predicate_set_evaluable* is shown in algorithm 3.6. It accepts a set of predicates as input and determines their ordering to ensure proper binding propagation.

Algorithm 3.6 *Determine the evaluation order for a set of predicates.*

Input: a set of predicates $Uneval$, a set of instantiated variables I.

Output: if the set of predicates are evaluable, return *true* and the order, other return *false*.

Method: the algorithm is as following:

procedure *predicate_set_evaluable*($Uneval, Instantiated$)

begin

while $Uneval \neq \emptyset$
  if (there exists p, such that ($p \in Uneval$) **and**
    (b' is the binding of p under I) **and**
    ($plan\_generator(p, b')$ = true))
  then
    $Uneval := Uneval - p$
    register p in $Plan$
    $I := I \cup$ set of all variables in p
    else /* no predicate in $Uneval$ is evaluable */
      return false
  return true /* $Plan$ contains predicates should be evaluated */
end □

LogicBase has employed four evaluation strategies so far: chain following in the *chain-exit* direction; chain following in the *exit-chain* direction; chain-split and counting. The plan generation for chain-split and counting is merged as a single one. These plan generation algorithms are given in algorithm 3.7, 3.8 and 3.9 respectively.

3.3.4 Chain-based evaluation

Chain-based evaluation accepts compiled chain form of a recursive program as input, classifies the program according to the binding passing patterns and evaluates the query using either chain-following, chain-split or counting method.

3.3.4.1 Chain following evaluation: chain-exit direction

In the chain following evaluation in chain-exit direction (*chain-exit* evaluation for short), all chains in the program are evaluated from the query end of the chain to the exit end. Assume that the chain predicates for a recursive program defining r are a and b (either single chain or multiple chain), the chain expansion form is:

$$r : -(a_{(1)}b_{(1)})(a_{(1)}b_{(2)}) \ldots (a_{(k)}b_{(k)})exit. \tag{3.33}$$

where $(a_{(i)}b_{(i)})$ is the i-th iteration of the chain and *exit* is (set of) the exit rule body predicate(s). The chain-exit evaluation passes binding from the query to the first chain iteration $a_{(1)}b_{(1)}$, which becomes evaluable from the query instantiation. Then binding in $a_{(1)}b_{(1)}$ is passed to $a_{(2)}b_{(2)}$, which becomes evaluable in turn. Therefore, all iterations of the chain are evaluable and finally *exit* becomes evaluable, thus the query is answered.

Plan generation for chain-exit evaluation needs to verify the successful binding passing from query to the first chain iteration, from the i-th chain iteration to the $(i+1)$-st, and from the last chain iteration to the exit rule. Algorithm 3.7 gives the algorithm of plan generation for chain-exit evaluation.

Algorithm 3.7 *Plan generation for the chain-exit evaluation.*

Input : compiled recursive rule r, exit rule *exit*, query binding b.

Output : *true* if query can be evaluated by chain-exit evaluation, *false* otherwise.

Method : the algorithm is as following:

```
procedure chain_exit_plan(r, exit, b)
begin
    /* verify the evaluability of the first chain iteration */
  I := set of instantiated variables in r's head under binding b;
  for each chain in r
    Uneval := all predicates in the chain;
    if predicate_set_evaluable(Uneval, I)=false
    then
      return false;
    /* verify evaluability of the following chain iteration */
  I' := all variables in the recursive predicate in r's body;
  for each chain in r        /* passing binding from one iteration to the next */
    Uneval' := all predicates in the chain;
    if predicate_set_evaluable(Uneval', I')=false
    then
```

```
        return false; /* unevaluable chain */
      /* pass binding from r to exit */
    I'' := all variables in the exit's head predicate;
    Uneval'' := all predicates in the body of exit;
    if predicate_set_evaluable(Uneval'', I'')=false
    then return false;
    else return true;
end.
```

□

3.3.4.2 Chain-following: exit-chain evaluation

In the chain following evaluation in exit-chain direction (*chain-exit* evaluation for short), all chains in the program are evaluated from the exit end to the query end. For the chain expansion shown in Formula (3.33), the exit-chain evaluation passes binding from the query to the exit rule by shared variables in the same argument position between the head predicate of the exit rule and the recursive predicate in the recursive rule body. The exit rule is then evaluated, which provides binding passing from the exit rule to the last chain iteration. The binding is then passed from one chain iteration backward to the previous chain iteration, and finally to the first iteration. Thus all chain iterations are evaluated and the query is answered.

Plan generation for the exit-chain evaluation needs to verify successful binding passing from query to exit rule, binding passing from one chain iteration to the preceding chain iteration. Algorithm 3.8 gives the algorithm for the plan generation of the exit-chain evaluation.

Algorithm 3.8 *Chain-exit evaluation plan generation.*

Input: A compiled recursive program r, its exit rule *exit* and the query binding b.

Output: *true* if the query can be evaluated by the chain-exit evaluation, *false* otherwise.

Method: the algorithm is as following:

procedure *chain_exit_plan*$(r, exit, binding)$
begin
/* passing binding from query to exit rule */
$Instantiated_r$:= set of the instantiated variables in head predicate of r under b;
I_r := those variable in $Instantiated_r$ s.t. they appear at the same argument positions in r's head predicate and the recursive predicate in r's body;
b_{exit} := binding of $exit$'s head predicate under instantiation I_r;
/* evaluability of $exit$ under $binding_{exit}$ */
I_{exit} := set of the instantiated variables in $exit$'s head predicate under b_{exit};
$Uneval_{exit}$:= all predicates in $exit$'s body;
if *predicate_set_evaluable*$(Uneval_{exit}, I_{exit})$=false
then /* insufficient binding to evaluate exit rule */
return false;
/* pass binding from $exit$ to r's chain */
I_r := set of all variables in the recursive predicate in r's body;
for each chain in r
$Uneval_r$:= set of all predicates in the chain;
if *predicate_set_evaluable*$(Uneval_r, I_r)$=false
/* last chain iteration unevaluable */
then
return false;
/* pass binding from one chain iteration to the previous */
I_r := set of variables in the r's head;
for each chain in r
$Uneval_r$:= set of predicates in the chain;
if *predicate_set_evaluable*$(Uneval_r, I_r)$=false
then /* insufficient binding to propagate evaluation in chain */
return false; /* unevaluable chain */
return true;

end. □

3.3.4.3 Counting for multiple chains

For a multiple chain program, not all chains can be evaluated in the same direction, some are evaluated in the chain-exit direction, whereas the rest in the exit-chain direction. This is similar to the counting method approach, because synchronization is needed between the various evaluations in two directions. Assume in Formula (3.33), predicates a, b belong to two different chains. Query instantiation enables a-chain to be evaluated in the chain-exit direction, but not the b-chain, because the query instantiation is not sufficient for the evaluation of b-chain. However, after evaluation of the exit rule, new bindings can be passed to the last iteration of b-chain, which enables the evaluation of b-chain in the exit-chain direction. Synchronization between these two chains by counting levels ensures proper termination of b-chain.

3.3.4.4 Chain-split evaluation

The chain-split evaluation is similar to the counting method, in that the chain-exit and the exit-chain evaluation are employed to process a query. Instead of two chains in different evaluation directions, a chain can be split into two parts which are evaluated in different directions in the chain-split evaluation. A portion of the chain, *immediately evaluable portion* (IMP), can be evaluated in the chain-exit direction, the rest of the chain, *buffered portion* (BP), has to be evaluated in the exit-chain direction. BP needs the bindings passed from the IMP and the exit rule to evaluate. Synchronization between IMP and BP is needed via counting levels. The difference between the chain-split evaluation and the counting evaluation for multiple chains is that there are shared variables between IMP and BP in the chain-split evaluation, whose value have to be stored in a buffer during the evaluation of IMP, whereas in the counting evaluation for multiple chains, there is no shared variable between chains, thus no buffer is needed.

The plan generation algorithm shown in Algorithm 3.9 works for both counting

and chain-split evaluations, where IMP represents the immediately evaluable portion in case of the chain-split evaluation, or chains evaluated in the chain-exit direction in case of counting, and BP stands for the buffered portion in chain-split or chains evaluated in the exit-chain direction in counting. Information about creating and maintaining a buffer is recorded in the plan for the chain-split evaluation.

In Algorithm 3.9, analysis of the following binding passing is carried out: binding passing from the query to the first iteration of IMP; from one iteration of IMP to the next; from the last iteration of IMP to the exit rule; from the exit rule to the last iteration of BP; from one iteration of BP to its preceding one; and from one iteration of IMP to the same iteration of BP.

Algorithm 3.9 *Plan generation for the chain-split and the counting evaluation.*

Input: a compiled recursive rule r, its exit rule *exit* and a query binding b.

Output: *true* if the query can be evaluated either by the chain-split or the counting method, *false* otherwise.

Method: the algorithm is as following:

procedure chain_split_plan$(r, exit, b)$
begin
  I_r := set of instantiated variables in r's head predicate under b;
  $IMP := \emptyset$; $BP := \emptyset$;
  for each chain in r
    $Uneval$:= set of predicates in the chain;
    $I' := I_r$;
    while $Uneval \neq \emptyset$
      if there exists a predicate $p \in Uneval$ s.t. b_p is binding of p under I'
        and *plan_generator*(p, b_p)=true
      then /* p is evaluable */
        insert p into IMP;
        $Uneval := Uneval - p$;

$I' := I' \cup$ set of variables in p;
else /* there is no evaluable predicate in $Uneval$ */
$BP := BP \cup Uneval$; /* BP is the set of buffered predicates */
$Uneval := \emptyset$;
if $IMP = \emptyset$ /* not suitable for chain-split */
then return false;
/* verify binding passing from IMP part of chain to exit rule */
I_{IMP} := set of variables in all predicates of IMP;
b_{exit} := binding of the recursive predicate in r's body under I_{IMP};
I_{exit} := set of instantiated variables in $exit$'s head predicate under b_{exit};
$Uneval_{exit}$:= set of the predicates in $exit$'s body;
if $predicate\_set\_evaluable(Uneval_{exit}, I_{exit}) = false$
then /* insufficient binding for $exit$ */
return false; /* verify binding passing from $exit$ to BP in r */
I_{BP} := set of variables in the recursive predicate in r's body;
$I_{BP} := I_{BP} \cup$ set of all variables in IMP;
$Uneval_{BP}$:= set of predicates in BP;
if $predicate\_set\_evaluable(Uneval_{BP}, I_{BP})$=false
then /* insufficient binding to evaluate buffered portion */
return false;
/* verifying binding passing from one iteration of BP to the previous one */
/* is the same as binding passing verification from $exit$ to BP */
return true;
end. □

3.4 Plan execution

An internal structure stores the evaluation plan. *LogicBase* adopts an unified approach towards plan execution. Every evaluation can be viewed as going through the

following evaluation stages, the plan executor takes appropriate actions according to the instructions specified in the evaluation plan structure:

1. *chain-exit stage*: where the query instantiation is passed to the chain(s) and each chain (whole or part) is evaluated in the chain-exit direction (if possible).
2. *exit-rule stage*: instantiation is passed to the exit rule from either a chain or the query, and exit rule is evaluated.
3. *exit-chain stage*: instantiation is passed from the exit rule to the chain(s) and each chain is evaluated in the exit-chain direction.

Each query evaluation strategy consists of a sequence of these stages. For the chain following method in *chain-exit* direction, only stage 1 and 2 are involved; whereas stage 2 and 3 are needed for the chain following in *exit-chain* direction. Counting and chain-split need all stages. A single plan executor is implemented to perform all actions. It would have costed more implementation effort if a dedicated plan executor had been implemented for each method in *LogicBase*.

3.5 Other implementation issues

Issues of termination control and constraint pushing are discussed in Chapter 2, cyclic counting method for function-free program is discussed in Chapter 5, and processing of multiple linear rules is presented in Chapter 4. We discuss some other implementation related issues in this section.

3.5.1 User interface

A simple graphical user interface and a terminal-oriented user interface are provided. *LogicBase* reads in definitions for EDB, IDB and query, from either interfaces and performs syntax analysis using YACC and LEX to transfer them into an internal

data representation. The specification of the syntax for *LogicBase* can be found in Appendix A.

3.5.2 Data structure

Static data structures and dynamic structures are blended in *LogicBase* to provide efficiency and flexibility for query evaluation. Static structures are adopted where frequent access is needed, such as the table for schema information, structures for constant and variable arguments, and the internal representation for the query evaluation plans. Dynamic structures are employed to support complex structures such as list and functional term, which can be nested each other to any level. The basic argument type can be one of the following types: integer, constant, variable, list or functional term. A list has a head of argument type and a tail of list type. A functional term has a number of elements, each of argument type.

Such implementation is flexible and efficient. Most work for query processing is efficiently done on the static structures to reduce overhead for dynamic data structure access and maintenance, whereas representation and manipulation of complex objects is achieved.

3.5.3 Negation

Stratified negation is supported in *LogicBase*. In the compilation phase, negative literals are treated in the same way as positive literals. During the plan generation phase, evaluation of a negative predicate is scheduled only after all of its variables are instantiated. In the plan execution phase, a negated subgoal is computed by first obtaining its corresponding positive subgoal, then performing a relational difference operation.

3.5.4 Query and evaluation plan optimization

LogicBase utilizes various query optimization strategies. Search space reduction by constraint derivation and pushing discussed in Chapter 2 is one kind of optimization. Compilation of a recursive program into bounded form is another kind of query optimization, where recursive program is compiled into an equivalent non-recursive one.

Furthermore, like optimization in relational query processing, optimization can be performed on query evaluation plan using statistical information available in EDB relations. Since in *LogicBase*, query evaluation is not carried out until a thorough evaluation plan is devised on the compiled program, a detailed analysis is possible to facilitate query optimization similar to that in relational databases, such as pushing a selection deeply into relational expression and optimization on join operations.

3.5.5 Variable naming

It is necessary to distinguish two sets of variables: (1) external variables, which are defined in an IDB predicate and/or queried by the user; (2) internal variables, which are internal representations of arguments during query processing and are created during the compilation phase.

It is noted that one external variable in a logic rule is independent of another external variable with the same name in a different logic rule, even if both rules may define the same IDB predicate. For example, suppose the following program is defined in *LogicBase*:

$$r(X,Y) \ :- \ a(X,Z), r(Z,Y). \tag{3.34}$$

$$r(X,Y) \ :- \ b(X,Z), c(Z,Y). \tag{3.35}$$

the second argument of predicate a in rule (3.34) has nothing to do with the second argument of predicate b in rule (3.35), although variable Z is used in both places.

However, if they are replaced by the same internal variable, query evaluation may lead to error because an extra equality relationship is mistakenly added to this program.

A variable naming mechanism in *LogicBase* is responsible to prevent such an undesirable situation. The internal variables in one logic rule are independent of the internal variables in other logic rules, and are independent of the external variables as well. The only exception is that the external variables in the query are actually used during query processing to facilitate query answer extraction.

3.5.6 Handling of functions

Inclusion of functions in a deductive database results in more effort to ensure a safe and finite evaluation. In *LogicBase*, a built-in function is transformed into a new equivalent form which contains an extra argument as the returned value for the function. For example, a predicate p with a list argument "$p([X|L])$" is transformed into "$p(X\_L), cons(X, L, X\_L)$" where "$cons(X, L, X\_L)$" is the equivalent functional predicate. For each built-in function, the binding patterns under which it can be evaluated and the way it should be evaluated are supplied internally in *LogicBase*. A table is used to store information about the evaluable bindings, which is consulted during plan generation to determine whether the function is evaluable. During plan execution, the function is actually evaluated by built-in routines.

3.5.7 Handling of functional terms

A functional term is a dynamic data type. It is considered to be a constant if every argument is instantiated, e.g., "$person(johnson, birth\_date(10, Jan, 1970))$". It is considered to be a variable if some of its arguments are variables, e.g., the following *person* predicate:

$$person(johnson, birth\_date(Day, Month, Year)) \tag{3.36}$$

The constant functional term is transformed into an internal dynamic structure. Similar to the transformation of functions, a variable functional term is transformed into a special predicate which contains all the arguments in the functional term and an extra argument designating the value of the functional term. For example, the above *person* predicate in (3.36) is transformed into the following:

$$person(johnson, X), birth\_date\_p(Day, Month, Year, X),$$

where *birth_date_p* is a special predicate for the functional term *birth_date*.

Such functional term $birth\_date\_p(Day, Month, Year, X)$ is evaluable if: (1) X is instantiated, then Day, $Month$ and $Year$ can be derived; or (2) Day, $Month$ and $Year$ are all instantiated, then X becomes instantiated.

Chapter 4

Evaluation of Multiple Linear Recursions

So far, query processing in single linear recursion using chain-based method has been investigated. In this chapter, efficient query processing in multiple linear recursion is discussed.

4.1 Introduction

The efficient evaluation of function-free linear recursions has been studied extensively in deductive database research [10, 9, 48, 59, 94],Ullm88. Most studies on linear recursions assume that a linear recursion consists of one linear recursive rule and one or more nonrecursive rules. We call such kind of recursions *single linear (SL) recursions* in contrast with the recursions to be studied here, *multiple linear (ML) recursions.*

A multiple linear (ML) recursion is a recursion which consists of multiple linear recursive rules and one or more nonrecursive rules. ML recursions occur in many applications. For example, *sg* (same generation cousin) can be defined by more than

one linear recursive rule as shown in the Figure 4.1. Moreover, ML recursions can be generated by the compilation of some mutual recursions or multiple levels of recursions [57]. Since SL recursions have been studied extensively, it is natural to extend the domain of study to ML recursions.

$$
\begin{aligned}
sg(X,Y) &:- parent(X,W),\ parent(Y,V),\ sg(W,V).\\
sg(X,Y) &:- child(W,X),\ child(V,Y),\ sg(W,V).\\
sg(X,Y) &:- cousin(X,W),\ sg(W,Y).\\
sg(X,Y) &:- sibling(X,Y).
\end{aligned}
$$

Figure 4.1: A recursion with multiple linear recursive rules.

There have been some interesting studies on the evaluation of ML recursions. Henschen and Naqvi [59] presented a formula derived by the expansions of an ML recursion and proposed a technique similar to their evaluation of SL recursions. Beeri and Ramakrishnan [13] developed Generalized Counting and Generalized Magic Sets methods which are applicable to the evaluation of ML recursions. Han and Henschen [48] presented a side-relation unioned compilation technique for a special class of ML recursions where each recursive rule is a *one-sided* recursive rule. Naughton [94] performed a detailed study on such kind of ML recursions, which he called *separable recursions*, and showed that an efficient algorithm similar to a transitive closure query processing algorithm is applicable. Naughton, Ramakrishnan, Sagiv and Ullman [96] further extended the technique to right-linear, left-linear and multi-linear rules.

This study provides a systematic study on different kinds of ML recursions and their query evaluation techniques. We classify ML recursions into side-coherent and non-side-coherent ML recursions, while the former is further classified into three types. Efficient query evaluation techniques are developed for side-coherent ML recursions, which integrate side-relation unioned processing [48] with the transitive closure algorithms [10, 48], the Magic Sets method [9], and the Counting method [9]. Moreover, flexible methods can be applied to the evaluation of queries with complex instantiations and inquiries on ML recursions.

4.2 A Classification of ML Recursion

We assume without loss of generality that all the rules in a recursion are rectified, where the rules for a predicate r are *rectified* [132] if all the heads of its rules are identical and in the form of $r(X_1, \ldots, X_n)$ for distinct variables $X_1, \ldots, X_n$. We also assume that there is exactly one *default* nonrecursive (exit) rule in an ML recursion in the form of

$$r(X_1, \ldots, X_n) :- e(X_1, \ldots, X_n).$$

where n is the arity of r.

We first introduce the concepts of a *k-sided recursive rule* and *side-coherency.*

Definition 4.1 *A* **k-sided recursive rule** *is a linear recursive rule in which there are $k + l$ variable vectors (where $k > 0$ and $l \geq 0$) in the head predicate, where each of the k variable vectors is connected to the same argument position of the recursive predicate in the body via a nonrecursive predicate (called a* **side-relation***) and each of the remaining l variables retains the same argument position of the recursive predicate in the body. Notice that different side-relations in a k-sided rule do not share variables. It is called* **one-sided** *when $k = 1$ or* multiple-sided *when $k > 1$. Each of such k variable vectors is called a* **side-vector**, *and each of the remaining l variables in the head predicate is called an* exit variable.

For example, the following rule is a k-sided recursive rule,

$$r(X_0, X_1, \ldots, X_k) :- p_1(X_1, W_1), \ldots, p_k(X_k, W_k),\ r(X_0, W_1, \ldots, W_k).$$

where X_0 is an exit variable and each X_i, for $1 \leq i \leq k$ is a side-vector connected to its corresponding argument position of the recursive predicate in the body via a nonrecursive predicate p_i.

Definition 4.2 *An ML recursion is* **side-coherent** *if each of its recursive rules is one- or multiple-sided and each side-vector of every recursive rule is either a*

side-vector or an exit variable vector of every other recursive rule. Otherwise, it is **non-side-coherent**. *Furthermore, a side-coherent ML recursion is* **strongly side-coherent** *if each side-vector of its every recursive rule is exactly one side-vector of its every other recursive rule. In a side-coherent recursion, a* **side-vector** *of the recursion is a side-vector of* at least *one recursive rule, and the* **exit-vector** *of the recursion is the set of exit variables shared by all of the recursive rules.*

$$
\begin{aligned}
r(X_1, X_2, Y_1, Y_2) &:- p_1(X_1, X_2, W_1, W_2),\ r(W_1, W_2, Y_1, Y_2).\\
r(X_1, X_2, Y_1, Y_2) &:- p_1(X_1, Y_1, W_1, W_2),\ r(W_1, X_2, W_2, Y_2).
\end{aligned}
$$

Figure 4.2: A non-side-coherent ML recursion.

$$
\begin{aligned}
r(X, Y, Z) &:- p_1(X, W),\ r(W, Y, Z).\\
r(X, Y, Z) &:- p_2(X, W),\ r(W, Y, Z).\\
r(X, Y, Z) &:- r(X, Y, U),\ q_1(U, Z).\\
r(X, Y, Z) &:- r(X, Y, U),\ q_2(U, Z).
\end{aligned}
$$

Figure 4.3: A side-coherent ML recursion where all of the recursive rules are one-sided (Type I).

Example 4.1 The recursion which consists of one default nonrecursive rule and two one-sided recursive rules shown in Figure 4.2 is non-side-coherent because the side-vector of the first rule is $< X_1, X_2 >$ while that of the second one is $< X_1, Y_1 >$.

The recursion which consists of one default nonrecursive rule and four one-sided recursive rules shown in Figure 4.3 is side-coherent. It has two side-vectors, X and Z, and one exit-vector Y.

The recursion which consists of one default nonrecursive rule and two two-sided recursive rules shown in Figure 4.4 is strongly side-coherent. It has two side-vectors, X and Z, and one exit-vector Y. Notice that all of the recursive rules in a strongly side-coherent recursion have the same number of sides. □

$$r(X,Y,Z) \ :- \ p_1(X,W),\ r(W,Y,U),\ q_1(U,Z).$$
$$r(X,Y,Z) \ :- \ p_2(X,W),\ r(W,Y,U),\ q_2(U,Z).$$

Figure 4.4: A strongly side-coherent (Type II) ML recursion.

To facilitate the development of efficient query processing methods, we further classify side-coherent ML recursions into the following three types:

- **Type I: multiple one-sided.** A side-coherent ML recursion is in Type I if all of its recursive rules are one-sided, e.g., Figure 4.3.
- **Type II: multiple balanced k-sided.** A side-coherent ML recursion is in Type II if it is strongly side-coherent and all of its recursive rules are multiple-sided, e.g., Figure 4.4.
- **Type III: multiple mixed k-sided.** A side-coherent ML recursion is in Type III if it does not belong to the above two types, e.g., Figure 4.5 and 4.6.

$$r(X,Y,Z) \ :- \ p_1(X,W),\ r(W,Y,U),\ q_1(U,Z).$$
$$r(X,Y,Z) \ :- \ p_2(X,W),\ r(W,Y,Z).$$

Figure 4.5: A Type III ML recursion where the recursive rules have different sides.

$$r(X,Y,Z) \ :- \ p_1(X,W),\ r(W,Y,U),\ q_1(U,Z).$$
$$r(X,Y,Z) \ :- \ p_2(X,W),\ r(W,U,Z),\ c_2(U,Y).$$

Figure 4.6: A Type III ML recursion where the recursive rules have the same number of sides but are not strongly side-coherent.

Conceptually, non-side-coherent ML recursions cover a large set of ML recursions. However, many of such ML recursions are transformable to side-coherent ML recursions or SL recursions by compilation and/or variable vectorization [43].

$$r(X,Y,Z) \ :- \ p(X,W),\ r(Y,W,Z).$$
$$r(X,Y,Z) \ :- \ q(X,Y),\ c(X_1,Y_1),\ r(X_1,Y_1,Z).$$

Figure 4.7: An ML recursion which is compilable to a single linear recursion.

$$r(X,Y,Z,U,W) \ :- \ p(X,Y),\ q(X_1,Y,Y_1),\ c(Z,W,W_1,Z_1),\ r(X_1,Y_1,Z_1,U,W_1).$$
$$r(X,Y,Z,U,W) \ :- \ f(X_1,Y,X,Y_1),\ g(U_1,U),\ r(X_1,Y_1,Z,U_1,W).$$

Figure 4.8: A complex ML recursion.

Example 4.2 The ML recursion shown in Fig 4.7 is transformable to a two-sided SL recursion because (i) the first recursive rule becomes a two-sided recursive rule after one more expansion on itself, (ii) the second recursive rule is a *bounded recursive rule* which, together with a nonrecursive rule, forms a bounded recursion [97], and (iii) further expansions of the second recursive rule on itself or on the first recursive rule are absorbed by the existing rules and thus treated as part of the nonrecursive rule set [43, 95].

The ML recursion shown in Fig 4.8, though quite complex, is a side-coherent recursion because it becomes a typical Type III side-coherent ML recursion by taking "$pq(X,Y,X_1,Y_1) :- p(X,Y),\ q(X_1,Y,Y_1)$." and treating $< X,Y >$, $< X_1,Y_1 >$, $< Z,W >$, and $< Z_1,W_1 >$ as vectors V, V_1, T and T_1 respectively, as shown in Fig 4.9. Such a variable vectorization process reduces the arity of the recursive predicate and simplifies the computation. A similar technique of reducing the arity

$$r(V,T,U) \ :- \ pq(V,V_1),\ c(T,T_1),\ r(V_1,T_1,U).$$
$$r(V,T,U) \ :- \ f(V,V_1),\ g(U_1,U),\ r(V_1,T,U_1).$$

Figure 4.9: The recursion in the previous figure is a Type III ML recursion by variable vectorization.

of a recursive predicate is studied recently in [96]. □

Some non-side-coherent ML recursions cannot be transformed to SL recursions or side-coherent ML recursions. The recursion shown in Fig 4.2 is one such example. However, it is difficult to find appropriate semantic interpretations and application models for such kind of ML recursions. Therefore, our study of efficient evaluation of ML recursions is confined to side-coherent ML recursions.

4.3 Evaluation of Single-Probe Queries in ML Recursions

A *single-probe query* is a query in which one variable of the recursive predicate is instantiated by one or a set of constants. A typical such query is

$$? - r(a, \_, Z). \tag{4.1}$$

where the predicate $r(X, Y, Z)$ is a ternary predicate in which Y is the exit-vector and X and Z are two side-vectors, "a" indicates that X is instantiated by one or a set of constants, "_" indicates that Y is irrelevant to the query, and "Z" indicates that the third argument of r is inquired. Notice that if "a" denotes a set of constants, the semantics of the query is to find the set of all Z's such that a tuple "$(a, \_, Z)$" exists. If we want to find the set of corresponding Z's for each X_0 in a set of constants satisfying a predicate s, the query should be written as, "$? - s(X_0),\ r(X_0, \_, Z)$", and a *binary algorithm* should be used, when necessary, to trace the (*source, sink*) pairs in the derivation [42].

In this section, we study the efficient evaluation of the single-probe query (4.1) on different side-coherent ML recursions.

4.3.1 Side-Relation Unioned Processing of Type I ML Recursions

We first examine the compilation of Fig 4.3, a typical Type I recursion. An expansion of the recursion may either (i) add an p_1 or p_2 to the left side of r, or (ii) add a q_1 or q_2 to the right side of r, or (iii) change r to e to generate an expanded formula, where e, the body of the exit rule, is called the *exit expression* of the recursion. Therefore, a possible expanded formula (with the variables inside the predicates omitted) should be

$$(p_1 \cup p_2)^i \; e \; (q_1 \cup q_2)^j$$

where $i, j \geq 0$. If we define

$$p(X,Y) := p_1(X,Y) \cup p_2(X,Y)$$

and

$$q(X,Y) := q_1(X,Y) \cup q_2(X,Y)$$

the *compiled formula* (the EDB expressions generated by all the possible expansions of the recursion) of Fig 4.3 should be

$$r(X_0,Y_0,Z_0) = \bigcup_{i=0}^{\infty} \bigcup_{j=0}^{\infty} (p^i(X_{i-1},X_i),\ e(X_i,Y_0,Z_j),\ q^j(Z_j,Z_{j-1}))$$

where the notation $p^i(X_{i-1},X_i)$ is defined as (i) a tautology when $i = 0$, and (ii) a sequence of compositions of i p's when $i > 0$, that is,

$$p^i(X_{i-1},X_i) = p(X_0,X_1),\ldots,p(X_{i-1},X_i).$$

Moreover, if the variables inside the predicates are omitted, we have

$$R = p^* \; e \; q^*.$$

Obviously, the processing of query (4.1) on such recursions should be similar to the processing of two transitive closure queries. It proceeds as follows. First, the single probe a is used to derive a unary query-relevant transitive closure of p. Then it joins

with the exit expression e, and the derived set of elements is used as the probe to derive a unary query-relevant transitive closure of q. The set of Z's so derived should be the answer set to the query.

Here, the evaluation adopts a **side-relation unioned processing** technique [48], where iterative processing is performed on the *union* of the side-relations at the same side of multiple recursive rules. In comparison with the *side-relation separate processing* (the processing performed on each side-relation), the side-relation unioned processing has the following advantages:

1. *It saves the interleaved accessing of multiple side-relations.* A relation is usually stored in a B-tree or a hash table. If there are k side-relations at one side, side-relation separate processing requires the interleaved accessing of k file structures while the side-relation unioned processing needs to access only one file structure (the unioned relation) at each iteration.

2. *It saves redundant processing of overlapped tuples.* An *overlapped tuple* is a tuple shared by more than one side-relation. In side-relation separate processing, overlapped tuples will be stored in more than one side-relation and be accessed more than once. In side-relation unioned processing, overlapped tuples are combined into one, which will not only reduce the storage space but also save the accessing cost.

3. *Side-relation unioned technique may even benefit relation partitioned processing.* Relation partitioning techniques have been popularly used in distributed database query processing [132]. At the first glance, it seems that the side-relation unioned processing contradicts the philosophy of relation partitioning. However, by first performing union on several side-relations at the same side and then partitioning and distributing the unioned relation according to its accessing structure, for example, index range, only the drivers which match the specified accessing structure will be transmitted to the corresponding site. Thus both message transmission and accessing cost will be saved in comparison with the side-relation separate processing.

The side-relation unioned processing technique is applicable to any Type I ML recursion which consists of k side-vectors, for $k \geq 1$. We consider a special case where there exists a unary nonrecursive predicate associated with an exit variable in some k-sided recursive rule. For example, if we change the first recursive rule in Fig 4.3 from

$$r(X, Y, Z) : -p_1(X, W),\ r(W, Y, Z)$$

to

$$r(X, Y, Z) : -p_1(X, W),\ r(W, Y, Z),\ c_1(Y)$$

the evaluation plan needs some slight modification as follows. For each element derived via at least one tuple in p which is originally from p_1 only, the evaluation of the exit expression should be on "$e(W, Y, Z), c_1(Y)$" instead of on "$e(W, Y, Z)$". This can be considered as the special case of removal of recursively redundant literals studied by Naughton [95]. Similar modifications of the evaluation plans should be adopted in Type II and Type III ML recursions as well.

The method discussed above applies equally well to those Type III recursions whose compiled formulas are the same as Type I recursions. For example, if the side-relation at each side of every k-sided recursive rule ($k > 1$) of a Type III recursion is a subset of the union of the side-relations of its one-sided recursive rules, e.g., adding "$r : -p_1, r, q_1$." to the recursion of Fig 4.3, each such k-sided recursive rule is redundant and can be eliminated from the recursion, and the processing should be the same as a Type I recursion.

4.3.2 Evaluation of Type II ML Recursions

Similar to a multiple-sided single linear recursion [10], a Type II ML recursion requires the synchronization of its different sides in the processing. This can be seen from the expansions of the recursion of Figure 4.4 as below:

$$p_{i_1} p_{i_2} \cdots p_{i_n} e\ q_{i_n} \cdots q_{i_2} q_{i_1}$$

where $i_j (1 \leq j \leq n)$ is either 1 or 2 [59], and p_{i_j} and q_{i_j} are symmetric to e.

Such a recursion can be evaluated based on a technique proposed by Henschen and Naqvi [59], which registers the accessed *paths* (predicate sequences) in the evaluation of *up-relations* (p-part) and matches in reverse sequences in the evaluation of *down-relations* (q-part). It can also be evaluated by the generalized counting method [13], where the side-relation information is encoded in the counting sets.

However, there are two difficulties in these methods. First, the interleaved accessing of multiple relations at the same side is costly as shown in the last section. Secondly, it is difficult to handle cyclic data on ML recursions. This is because synchronization is not only on the length of the paths but also on the corresponding side-relations along the paths, and the time complexity of such synchronized processing is exponential to the length of the paths on ML recursions [40].

4.3.2.1 First Attempt: Side-Relation Unioned Path-Tracing Method

To solve the first problem, we propose an improvement of the above methods using the side-relation unioned processing technique. At the first glance, it seems difficult to apply side-relation unioned processing because the union of the two side-relations of Figure 4.4 forms a rule "$r : -p, r, q.$", which is not equivalent to the original recursion. However, if (i) each tuple in a unioned relation is associated with appropriate information to indicate its *origin*, that is, from which side-relation(s), as shown in Algorithm 4.1, and (ii) each derived value is associated with an *origin-path* (a sequence of origins) to register its accessing history, correct synchronization can still be achieved without suffering interleaved accesses.

Algorithm 4.1 Side-information associated union *(notion: $\dot{\cup}$) of relations at the same side.*

Input : A set of relations $p_1, \ldots, p_k$.

Output : The side-information associated union relation $p := \dot{\cup}\ (p_1, \ldots, p_k)$.

Method: Perform union of the participating relations $p_1, \ldots,$ and p_k, with each tuple of the unioned relation p associated with a k-bit *origin*, in which each bit represents one side-relation participating the union. The corresponding bit is "1" if the tuple is from that side-relation, or "0" otherwise. □

For example, for $p := \dot{\cup}(p_1, p_2)$, the origin of a tuple t in p is "01" if it is from the relation p_2 only, "10" if from p_1 only, and "11" if from both p_1 and p_2.

Then we present an algorithm similar to the Counting method [9]:

Algorithm 4.2 Side-relation unioned path-tracing evaluation of Type II recursions.

Input : A Type II ML recursion (Figure 4.4) and a single-probe query (4.1).

Output : The set of answers to the query.

Method : First, perform side-information associated union of the relations at the same side, that is, $p := \dot{\cup}(p_1, p_2)$ and $q := \dot{\cup}(q_1, q_2)$. Then:

1. *Perform up-relation processing*, which is similar to the up-relation processing in Counting [9] except that each derived (unary) element is associated with an *origin-path* which inherits its driver's origin-path and appends the *origin* of the currently accessed tuple in relation p. The resulting closure (*up-closure*) consists of the set of derived elements each associated with its origin-path.
2. *Perform flat-relation processing*, that is, join the up-closure with the exit expression e, which is similar to the flat processing in Counting except that each derived element inherits the origin-path of its driver.
3. *Perform down-relation processing,* which is similar to the down-relation processing in Counting except that it performs an *origin-matching test* in accessing each tuple of q to test the match of the tail in the origin-path of a driver and the *origin* of the currently accessible tuple. The test is

performed by *first bitwise-anding the two origins and then oring all of the resulting bits.* The test is passed if the result is 1 and failed otherwise. Each derived element inherits the origin-path of its driver but with the tail of the origin-path removed. The derivation from an element discontinues if the element carries an empty origin-path.

4. The *answers* to the query are the derived elements with empty origin-paths.

□

Theorem 4.1 *Algorithm* 4.2 *terminates and derives all of the answers to the query for Type II ML recursions on acyclic (unioned) relations.*

Proof:

First, we show that every element so derived is in the answer set. Assume that an element c_x is derived by accessing the unioned side-relation n times in the up-relation processing. Thus c_x must carry an origin-path of length n, equivalent to the accessing of a sequence of separate side-relations, $p_{i_1}, p_{i_2}, \ldots, p_{i_n}$. In the flat-relation processing, d_x is derived by accessing a tuple (c_x, d_x) in e and the origin-path of c_x is passed to d_x. In the down-relation processing, the origin-matching test is performed, which tests the match of the origins between the tail of the origin-path and the current tuple in the side-unioned relation q. Therefore, if there is a sequence of n accesses of the down-relation from d_x, it must be equivalent to the accessing of the side-separate down-relations in the sequence of $q_{i_n}, \ldots, q_{i_2}, q_{i_1}$. Such an accessing sequence is equivalent to the compiled formula of the recursion and makes the origin-path empty. Therefore, each element so derived is in the answer set.

Secondly, every answer to the query is derivable by the algorithm. Suppose an element d is in the answer set. Then d must satisfy the compiled formula of the recursion. That is, it should be derivable by starting with a, passing an equivalent length of side-relation strings at each side, and matching the corresponding side-relations. This follows the algorithm exactly. Thus, d should be derivable by executing the algorithm.

Thirdly, we show that the process terminates on acyclic unioned relations. In the up-relation processing, since every derivation path must be finite in an acyclic relation, the length of the origin-path of each derived element is finite. In the down-relation processing, the length of the origin-path associated with each derived element decreases at each iteration and its derivation cannot proceed when its origin-path decreases down to 0. Thus the process terminates. □

An obvious benefit of the method is that a driver accesses only one unioned relation at each iteration instead of k relations where k is the number of recursive rules in the recursion, which reduces the database accessing cost. Similar to the analysis of the counting method on single linear recursions [84], the worst case complexity of the algorithm on acyclic databases is $O(ne)$, where n and e are the number of nodes and the number of edges of the two unioned relations, respectively.

The algorithm can be refined by associating one element with a set of origin-paths if the element is derived via several paths [48]. Thus one DB access using such an element is equivalent to several accesses of an element associated with a single origin-path. When an element is associated with a set of origin-paths, a new origin should be appended to each path in the set in the up-relation processing, and the origin-matching test should be performed on each origin-path in the down-relation processing. A path which failed the test should be dropped from the origin-path set of the derived element since the element cannot be derived via that path.

The side-relation unioned processing reduces the interleaved accessing of multiple side-relations. However, it may lead to another inefficiency problem, *the growth of the associated origin-paths.* For example, if the up-relation processing involves 1000 iterations, the length of each origin-path of the derived element may grow to 1000 as well. Moreover, similar to the Counting method, the algorithm cannot terminate on cyclic databases. Notice that a database usually contains more cycles in an ML recursion than in an SL recursion because cycles can also be formed by interleaved traversing of multiple side-relations. Although such cycles can be detected by examining a unioned side-relation, it is difficult to perform path synchronization involving

cycles in ML recursions, as we discussed in the analysis of the proposal of Henschen and Naqvi [59]. Therefore, the side-relation unioned path-tracing method should not be considered as a general evaluation technique.

4.3.2.2 Side-Relation Unioned Magic Sets Method

Since the magic sets method handles both cyclic and acyclic data uniformly for SL recursions, it is promising to apply the method to Type II ML recursions. According to [10, 9], three magic rules for the single-probe query (4.1) on the Figure 4.4 recursion can be generated as shown in Figure 4.10 which forms a Type I ML recursion.

$$
\begin{array}{lll}
magic(a). & & \\
magic(Y) & :- & p_1(X,Y),\ magic(X). \\
magic(Y) & :- & p_2(X,Y),\ magic(X).
\end{array}
$$

Figure 4.10: Magic rules for the ML recursion.

According to the above discussion on Type I recursions, the magic set is the probe-relevant transitive closure of p, where "$p :=\dot{\cup}\ (p_1, p_2)$". Then p', the portion of p relevant to the query, can be derived easily from, "$p'(X,Y) :- p(X,Y), magic(X)$.". A *side-matched semi-naive evaluation* can be performed on p', e and the side-unioned relation q. We describe the method as follows.

Algorithm 4.3 *Side-relation unioned magic sets evaluation of Type II ML recursions.*

Input and **Output**: the same as Algorithm 4.2.

Method: 1. Derive (i) the magic set based on the rule set of Figure 4.10, which can be implemented using the query-relevant transitive closure techniques on p, where $p :=\dot{\cup}\ (p_1, p_2)$, and (ii) p', the query-relevant portion of p, based on "$p'(X,Y) :- p(X,Y),\ magic(X)$.".

2. Perform *side-matched-semi-naive-evaluation* (p', e', q), where

$$e'(X, Z) = \Pi_{X,Z}(magic(X) \bowtie e(X, Y, Z))$$

is a projection of the query relevant portion of e on the relevant attributes. It is performed similar to the semi-naive evaluation [132] as follows:

$\Delta := e'(X, Z)$;
closure := Δ;
repeat
  join_result := side-matched-join(p', Δ, q);
  new_Δ := join_result - closure;
  closure := closure $\cup$ new_Δ;
  Δ := new_Δ;
until $\Delta = \emptyset$.

Notice that the side-matched join results in a binary relation which is the join results of the Δ with relations p' and q projected onto the two non-join attributes, where the join results are those joinable tuples which have passed the origin-matching test. □

Theorem 4.2 *Algorithm* 4.3 *generates all of the answers to the query* (4.1) *and terminates on all kinds of EDBs.*

Proof: First, the magic rule set computed by the algorithm is the same one as that obtained by the rule rewriting technique in the magic sets method (according to the rules for the generation of magic rules [10]). Thus the relation p' so obtained collects the set of query-relevant facts in p_1 and p_2 (with side-information associated).

Secondly, the second step of the algorithm performs side-matched semi-naive evaluation on (i) p', the query-relevant portion of p, (ii) e', the query-relevant portion of e, and (iii) q, the side-information associated union of q_1 and q_2. Based on the correctness and termination of the semi-naive evaluation [132], the evaluation terminates on all kinds of data, and any answer set derivable from the query must also be derivable

by such an evaluation. Moreover, the side-matched join in the semi-naive evaluation collects only those joinable tuples which are from relations p' and q respectively and have passed the origin-matching test. Thus only those tuples which match both sides of the corresponding recursive rules can be derived in the evaluation. Therefore, all of the results generated by the algorithm belong to the answer set. □

Similar to the analysis of the magic sets method on single linear recursions [84], we can derive that the algorithm works on all kinds of data with the worst case complexity of $O(edge^2)$, where *edge* is the number of edges in the relations p', e' and q respectively.

4.3.2.3 Refinements: Superset Counting and Superset Transitive Closures

The side-relation unioned magic sets method can be refined by restricting the portions of the relations at the uninstantiated side to be enclosed in the side-matched semi-naive evaluation, which results in two alternatives to the side-relation unioned magic sets method: a superset counting method and a superset transitive closures method.

The *superset counting method* is performed as follows. First, it applies the counting method (without enforcing side-information matching in its evaluation) to evaluate the same query "$? - r(a, \_, Z)$" on the recursion "$r :- p, r, q.$", where p and q are the two unioned side-relations, that is, "$p :=\dot{\cup} (p_1, p_2)$", and "$q :=\dot{\cup} (q_1, q_2)$". The evaluation derives q' (similarly p'), a smaller q-side relation which collects the tuples in relation q accessed in the Counting evaluation. The q' so obtained is essentially the query-relevant portion of q on the recursion formed by Fig 4.4 recursion plus two more rules:

$$r(X, Y, Z) \ :- \ p_1(X, W),\ r(W, Y, U),\ q_2(U, Z).$$
$$r(X, Y, Z) \ :- \ p_2(X, W),\ r(W, Y, U),\ q_1(U, Z).$$

Clearly, q' is a subset of q, but a superset of the portions of q_1 and q_2 *truly* relevant

to the query (since side-matching is not considered in the Counting). Then the side-matched semi-naive evaluation is performed on such a q'. The algorithm is presented below.

Algorithm 4.4 *Superset counting evaluation of Type II ML recursions.*

Input and **Output**: the same as Algorithm 4.2.

Method :

1. Perform side-information associated union, that is, "$p :=\dot{\cup}\ (p_1, p_2)$" and "$q :=\dot{\cup}\ (q_1, q_2)$". Applying the counting method (or the cyclic counting method ([48, 40, 81]) if the unioned relation contains cycles), evaluate query (4.1) on the SL recursion:

$$r \ :- \ p,\ r,\ q.$$
$$r \ :- \ e.$$

   It derives p', q', and e', (Note: p' and e' are the same as Algorithm 4.3), the sets of query-relevant facts of relation p, q, and e respectively.

2. Similar to Step 2 of Algorithm 4.3, *side-matched-semi-naive-evaluation* (p', e', q') is performed, where e', p' and q' are the query-relevant portion of e, p and q determined in Step 1. □

Since counting on acyclic databases and cyclic counting on general databases terminates for single linear recursions [9, 40], Step 1 terminates on all kinds of databases. Furthermore, according to the proof in Theorem 4.2, side-matched semi-naive evaluation in Step 2 derives all of the correct answers and terminates on all kinds of EDBs [132]. Therefore, we can easily prove that Algorithm 4.4 generates all of the answers to the query (4.1) and terminates on all kinds of EDBs.

We then examine the worst-case time complexity of Algorithm 4.4. The worst-case time complexity in Step 1 is $O(n * edge)$ where n and $edge$ ($edge$ is used here

to not confuse with relation e) are the number of nodes and the number of edges in the database graph of the original database respectively [40, 84]. The worst-case time complexity of Step 2 is $O(edge'^2 + edge_{e'})$, where $edge'$ is the sum of the number of edges of the unioned relations, p' and q', that is, $edge' = edge_{p'} + edge_{q'}$, and $edge_{e'}$ is the number of edges of e'. Since the relation q' is derived based on query instantiations, it is usually substantially smaller than q. Therefore, the method in general results in substantial savings in comparison with the side-relation unioned magic sets method.

Another technique can also be used to restrict the size of the q-side relations. Based on the expansion formula:

$$p_{i_1}, p_{i_2}, \ldots, p_{i_n}, e, q_{i_n}, \ldots, q_{i_2}, q_{i_1}$$

where i_j $(1 \leq j \leq n)$ is either 1 or 2 [59], we can easily find another formula:

$$R = p^* \; e \; q^*$$

where "$p :=\dot{\cup}\, (p_1, p_2)$", and "$q :=\dot{\cup}\, (q_1, q_2)$", to compute the restricted supersets. This leads to another version of the refinement of Algorithm 4.3, the *superset transitive closures method.*

The superset transitive closures method applies two transitive closure operations to derive the portions of each side relation relevant to the query "$? - r(a, \_, Z)$". The evaluation derives (i) p', the query relevant portion of p by computing the magic set, and (ii) q', the superset of the query relevant portion of q, by first joining the magic sets with e to obtain *source drivers* of q and then using them to compute the partial transitive closure of q, the union of the two side-relations q_1 and q_2, relevant to those source drivers. The algorithm is presented as follows:

Algorithm 4.5 *A superset transitive closures method for the evaluation of Type II ML recursions.*

Input and **Output**: the same as Algorithm 4.2.

Method :

1. Perform side-information associated union, that is, "$p :=\dot{\cup}\ (p_1, p_2)$" and "$q :=\dot{\cup}\ (q_1, q_2)$". Then derive the transitive closure, $magicp(X)$, on relation p using the query constant a. Join $magicp(X)$ and $e(X, \_, Z)$ to obtain the set of Z's (the source drivers for q) and use them to compute the probe-relevant transitive closure of q, $magicq(Z_1)$. Finally, $p'(X, X_1)$ and $q'(Z_1, Z)$ are obtained by

$$p'(X, X_1) : -magicp(X),\ p(X, X_1).$$
$$q'(Z_1, Z) : -magicq(Z_1),\ q(Z_1, Z).$$

2. Similar to Step 2 of Algorithm 4.3, *side-matched-semi-naive-evaluation* (p', e', q') is performed, where e', p' and q' are the query relevant portion of e, p and q determined in Step 1. □

Similarly, we can prove that Algorithm 4.5 generates all of the answers to the query (4.1) and terminates on all kinds of EDBs. The difference between algorithms 4.4 and 4.5 is the way to reduce the portion of q for semi-naive evaluation. The former applies Counting while the latter applies transitive closure operations. Transitive closure is easier to implement than Counting. The worst-case time complexity of Counting is $O(ne)$, while that of the transitive closures method is $O(e)$, where n and e are the number of nodes and the number of edges in the database graph respectively. However, Counting enforces more restrictions than the transitive closures method on the relevant portion of q and thus makes the semi-naive evaluation more efficient.

4.4 Evaluation of Type III ML Recursions

Since a Type III recursion may not have balanced side-relations, it requires more complex synchronization than a Type II recursion. We first examine the recursion shown in Figure 4.5 which is essentially:

$$r\ :-\ p_1,\ r,\ q_1.$$
$$r\ :-\ p_2,\ r.$$

Its compiled formula is:

$$r = \bigcup_{i=0}^{\infty} (p_2^*(p_1 p_2^*)^i e\ q_1^i).$$

The superset counting method does not work properly for this recursion because we have to either treat the p-side as $p_1 p_2^*$, a huge relation, or derive a query-relevant transitive closure of p_2 for each element derived in the accessing of p_1 at each iteration. Similarly, it is difficult to apply the path-tracing method to this recursion. However, the side-relation unioned magic sets method, if modified appropriately, is still applicable to such kind of recursions.

A novel technique in our implementation is to view the Type III recursion as a Type II one with a faked (pseudo-) side-relation q_2. The faked side-relation q_2 consists of only one pseudo-tuple (X, X), which plays the role of passing all of the values through this missing side. However, since the pseudo-tuple carries the corresponding origin of q_2, it will not mistakenly pass through any value of q_1 in the side-relation unioned processing. A similar pseudo-tuple can be added to a corresponding p-side relation to make it a Type II one if an p-side relation is missing in a Type III recursion. We present the algorithm as follows:

Algorithm 4.6 *A side-relation unioned magic sets method for the evaluation of a Type III ML recursion.*

Input : A Type III ML recursion (Figure 4.5) and a single-probe query (4.1).

Output : The set of answers to the query.

Method: 1. (The same as Step 1 of Algorithm 4.3.) Derive the magic set and p' by evaluating the partial transitive closure of p relevant to the query constant a, where "$p := \dot{\cup}\ (p_1, p_2)$.", and "$p'(X, Y) :- p(X, Y),\ magic(X)$.".

2. Perform *side-matched-semi-naive-evaluation* (p', e', q), where p' and e' are the same as Algorithm 4.3, q is the side-information associated union of q_1 and the pseudo-tuple (X, X) which carries an origin with only the bit

corresponding to the missing q_2 set to 1. Notice that the pseudo-tuple matches any value of a driver and passes it to the second column as long as the driver passes the origin-matching test. □

Theorem 4.3 *Algorithm* 4.6 *generates all of the answers to the query* (4.1) *and terminates on all kinds of EDBs.*

Proof: We only need to verify the modified portion of the algorithm since the remaining is the same as Algorithm 4.3. In the *side-matched semi-naive evaluation*, by setting a pseudo-tuple for the missing relation q_2, the recursion becomes a Type II recursion. Since the pseudo-tuple (X, X) does not carry the origin bit for q_1, the side-matched semi-naive evaluation of the rule "$r :- p_1, r, q_1$." will not be influenced by the pseudo-tuple. Moreover, since it carries the origin bit for q_2 and passes the value of a driver from the first column to the second one as long as the driver passes the origin-matching test, it derives exactly the results derivable from the rule "$r :- p_2, r$.". Therefore, based on the proof of Theorem4.2, the correctness and termination of the algorithm can be verified. □

In comparison with the magic sets method, the algorithm takes an advantage of side-relation unioned processing both in the derivation of magic sets and in the semi-naive evaluation. The worst-case time complexity of the algorithm should be the same as Algorithm 4.3.

The algorithm can also be refined by superset transitive closure processing, which is similar to that discussed in the evaluation of Type II recursions and thus omitted in our discussion.

If we add to the recursion one more one-sided rule as below:

$$r(X, Y, Z) : -r(X, Y, Z_1),\ q_3(Z_1, Z)$$

the recursion is still in Type III, as shown below:

$$r\ :-\ p_1,\ r,\ q_1.$$

$$r \ :- \ p_2, \ r.$$
$$r \ :- \ r, \ q_3.$$

The recursion can be evaluated in a similar way as Algorithm 4.7. Notice that one pseudo-tuple should be added to each unbalanced side to represent the corresponding missing relation. That is, the pseudo-tuple (X, X) and (Z, Z) should be added, with the corresponding origin bit set to represent the missing relation p_3 and q_2 respectively. The evaluation algorithm can then be derived accordingly.

4.4.1 Generalized Side-Relation Unioned Magic Set Method

We generalize the above discussion to the recursions which consist of multiple side-vectors. We assume that a recursion of Fig 4.11 consists of n recursive rules, each having *at most* $m + k$ side-vectors. The recursion is in Type II if no side-relation is missing. It is in Type I if there is only one side-relation left (that is, all the other side-relations are missing) in each rule. Otherwise, it is a Type III ML recursion. Since a Type I recursion can be easily handled by a transitive closure query processing method, we assume that it is either a Type II or a Type III recursion.

$$\begin{aligned}
r(X_1, \ldots, X_m, Y, Z_1, \ldots, Z_k) \ :- \ & p_{11}(X_1, W_1), \ldots, p_{1m}(X_m, W_m), \\
& r(W_1, \ldots, W_m, Y, V_1, \ldots, V_k), \\
& q_{11}(V_1, Z_1), \ldots, q_{1k}(V_k, \ Z_k). \\
\ldots\ldots\ldots\ldots\ldots\ldots & \\
r(X_1, \ldots, X_m, Y, Z_1, \ldots, Z_k) \ :- \ & p_{n1}(X_1, W_1), \ldots, p_{nm}(X_m, W_m), \\
& r(W_1, \ldots, W_m, Y, V_1, \ldots, V_k), \\
& q_{n1}(V_1, Z_1), \ldots, q_{nk}(V_k, Z_k).
\end{aligned}$$

Figure 4.11: A general side-coherent ML recursion.

We examine a query of the form (4.2) which provides highly selective instantiations

on m side-vectors and inquires the information for the remaining k side-vectors.

$$? - r(a_1, \ldots, a_m, \_, Z_1, \ldots, Z_k). \tag{4.2}$$

Since the side-relation unioned magic sets method has been proved to be applicable to both Type II and Type III ML recursions on both acyclic and cyclic databases, we present our algorithm based on this method.

Algorithm 4.7 *A generalized side-relation unioned magic sets method for Type II and Type III recursions.*

Input : The recursion Fig 4.11 and the query (4.2).

Output : The set of answers to the query.

Method :

1. Perform side-information associated union of the side-relations at each side, that is,

   $$p_i(X_i, W_i) = \dot{\cup} (p_{1i}(X_i, W_i), \ldots, p_{ni}(X_i, W_i))$$

   and

   $$q_j(v_j, Z_j) = \dot{\cup} (q_{1j}(v_j, Z_j), \ldots, q_{nj}(v_j, Z_j))$$

   where $1 \leq i \leq m$, and $1 \leq j \leq k$. Each tuple in every unioned relation carries an n-bit vector (*origin*) where n equals to the number of recursive rules in the recursion. Notice if p'_{li} is missing in the l-th rule, a pseudo-tuple in the form of (X_{li}, X_{li}) is added to the relation p'_i with only the l-th bit set in its origin. Similar treatment is performed for the missing side-relations at the q-side.

2. Derive the magic set, *magic*, based on the following magic rule set,

   $$magic(a_1, \ldots, a_m).$$
   $$magic(W_1, \ldots, W_m) \ :- \ p_{11}(X_1, W_1), \ldots, p_{1m}(X_m, W_m),$$

$$
\begin{array}{lll}
 & & magic(X_1, \ldots, X_m). \\
 & \ldots\ldots\ldots\ldots\ldots & \\
magic(W_1, \ldots, W_m) & :- & p_{n1}(X_1, W_1), \ldots, p_{nm}(X_m, W_m), \\
 & & magic(X_1, \ldots, X_m).
\end{array}
$$

The magic set is the side-relation matched (partial) transitive closure of "$p_1(X_1, W_1), \ldots, p_m(X_m, W_m)$" relevant to the query probe $(a_1, \ldots, a_m)$. It is derived by starting with the query probe, iteratively evaluating the side-relation unioned relations $p_1, \ldots, p_m$ with the side-matching test for all the m sides. (The side-matching test for multiple sides is described in detail in Step 3). The evaluation terminates when no new tuple can be added to *magic*. Then *magic* is used to derive p_i' $(1 \leq i \leq m)$, the query relevant portion of p_i, by:

$$p_i'(X_i, W_i) : -p_i(X_i, W_i),\ magic(X_1, \ldots, X_m).$$

3. Perform the side-matched semi-naive evaluation as follows. Assume that a driver is of the form $t(W_1, \ldots, W_m, V_1, \ldots, V_k)$ (Y is dropped since it is irrelevant to the query), and a derived tuple is of the form $t(X_1, \ldots, X_m, Z_1, \ldots, Z_k)$. Each driver examines $m + k$ side-relations, p_i' $(1 \leq i \leq m)$ and q_j $(1 \leq j \leq k)$, with a total of $m + k$ n-bit vectors as shown in Fig 4.12. The side-matched-semi-naive evaluation accesses each tuple in the side-relation p_i in the form of $p_i'(X_i, W_i)$ using W_i, and extracts the origin (n-bit vector) from each accessed tuple. A similar operation is performed at the q_j side. The origin-matching test is performed on each obtained tuple by bit-oring of the n-bit vector obtained by bitwise anding of all the $m + k$ origin-vectors. The test is passed if the following holds:

$$\vee(\wedge(\vee_{i=1}^{m} t_{p_i'}.origin, \wedge_{j=1}^{k} t_{q_j}.origin)) = 1$$

where "$\wedge$" represents *bitwise-and*, and "$\vee$" represents *bit-or*. The newly derived tuples are appended to the closure and are used as the drivers at the next iteration. The evaluation terminates when no new tuple can be

derived at an iteration. The final answers to the query are those obtained by performing selection on the closure using the query constants. □

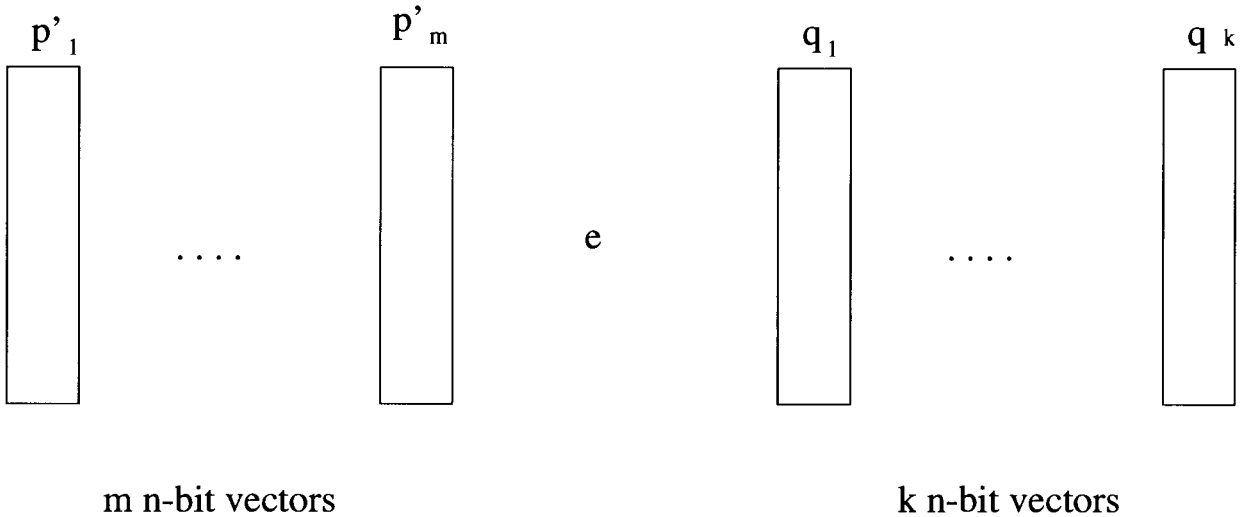


Figure 4.12: Side-matching test on $m + k$ n-bit vectors.

Theorem 4.4 *Algorithm* 4.7 *generates all the answers to the query* (4.2) *and terminates on all kinds of EDBs.*

Proof: First, the magic rule set is the same as those derived by rule rewriting in the magic sets method [10]. The side-relation matched transitive closure computation terminates (based on the termination of the corresponding transitive closure algorithm) and derives all of the query-relevant facts at the p_i side for $1 \leq i \leq m$. The correctness of the origin-matching test can be verified in the same way as that in the side-matched semi-naive evaluation.

Secondly, the side-matched semi-naive evaluation is similar to that of Algorithm 4.7. We only need to verify the correctness of the origin-matching test. Since the corresponding side of different recursive rules is represented by the same bit in each n-bit vector, the n-bit vector obtained by *bit-wise anding* of all the $m + k$ n-bit vectors of the joinable tuples in the relations, p'_i and q_j (for $1 \leq i \leq m$ and $1 \leq j \leq k$), should

have at least one bit set to 1 if the evaluation passes all the participating sides for at least one recursive rule. Therefore, it is correct to perform the *bit-or* on the result of the *bit-wise and* of the n-bit vectors of all the participating sides. □

In comparison with the magic sets method, the algorithm reduces the cost of interleaved accessing of separate side-relations by side-relation unioned processing. The worst-case time complexity of the algorithm should be the same as that of Algorithm 4.7, in which the calculation of the total number of edges should include all the relations involved in the computation.

Notice that another way to compute the set of query-relevant facts by magic sets is to derive one separate magic set at each side, that is, to derive m unary magic sets, $magic_i$, for $1 \leq i \leq m$, which is the partial transitive closure of p_i relevant to the query probe a_i. The computation of m unary magic sets does not take advantage of side-matching test which eliminates the side-unmatched facts, thus it may result in deriving a larger set of query-relevant facts, $p'_1, \ldots, p'_m$ than the computation of one m-ary magic set. However, it avoids the computation of the combinations of the values from the m sides. In the worst-case, the size of the *magic* relation derived by such possible combinations may reach $\prod_{i=1}^{m} |p_i|$, where $|p_i|$ is the number of the distinct values in relation p_i. However, the total size of m unary magic sets by computing m side-separate transitive closures is $\sum_{i=1}^{m} |p_i|$ in the worst-case. Obviously, the choice between the two variations depends on the characteristics of data. For example, if the derivation starting at the constants a_i, where $1 \leq i \leq m$, terminates at only a small number of iterations, it is unnecessary to compute the probe-relevant transitive closures for other relations beyond this iteration in the computation of one m-ary magic set. In this case, computing one m-ary magic set is more efficient than computing m unary magic sets. However, the situation is reversed if cyclic data are dominant in each of the m-sides. For example, if a_1 is in a cycle of 1000, and a_2 is in a cycle of 1001, the binary magic set (suppose $m = 2$) will contain 1,001,000 binary elements while the two unary magic sets have only 2001 unary elements.

Similar to the previous discussion, Algorithm 4.7 can be refined by the superset

transitive closure method. The refinement is performed as follows. First, one m-ary magic set or m unary magic sets are derived based on the query constants. Then the result is joined with the exit relation e to derive the instantiated driver set for each unioned side-relation q_j, where $1 \leq j \leq k$. After that, a partial transitive closure operation can be performed on each q_j to reduce the size of the q_j relations to be participated in the semi-naive evaluation.

4.5 Evaluation of Complex Queries in ML Recursions

The previous section has discussed the evaluation of single-probe queries on ML recursions. It is natural to examine whether the technique can be applied to complex recursive queries. The evaluation of complex queries on SL recursions has been studied in [42], which shows that flexible strategies should be applied to the evaluation of complex queries, and the selection of appropriate processing strategies should be determined based on the kinds of recursions, query instantiations and inquiries, and EDB statistics. Since the evaluation of side-coherent ML recursions follows the framework of the evaluation of SL recursions as shown in the previous discussion, the techniques obtained in the study of SL recursions can be transferred to side-coherent ML recursions.

Here we examine the evaluation of complex queries on Type II ML recursions only. Methods for other types of ML recursions can be derived accordingly.

First, query processing may proceed in different processing direction combinations, which is mainly determined by the selectivities of query constants. We suggest three major processing direction combinations: *up-down*, *all-down*, and *all-up*. The terms *up* and *down* are from Bancilhon and Ramakrishnan [10].

1. *up-down*: The *up-down* processing starts at some side-vectors (*driving sides*), climbs up to the center (the exit-expression), and then steps down to the set

of the remaining side-vectors (*driven sides*). It should be used when the query provides highly selective information at driving sides, such as queries (4.1) and (4.2) on the recursion of Fig 4.5.

2. *all-down*: The *all-down* processing starts at the center and steps down towards all the side-vectors. It should be used when the exit-vector carries the highly selective information.

3. *all-up*: The *all-up* processing starts at all of the side-vectors and proceeds towards the center. It should be used when the query provides highly selective information at all side-vectors.

The *up-down* processing has been studied in the last section. Here we present some queries which need other processing direction combinations.

Example 4.3 We examine the evaluation of the query

$$? - f(a, Y), r(X, Y, Z)$$

on the same ML recursion of Fig 4.5.

Suppose f is an EDB predicate and $f(a, Y)$ provides highly selective instantiation for r. Since only the exit-vector provides highly selective information, the *all-down* processing is appropriate. Similar to the previous discussion, side-relation unioned processing can be adopted by first performing side-information associated union of the side-relations at each side, which derives p and q, and then performing *side-matched semi-naive evaluation* on p, e' and q, where "$e'(X, Z) = \Pi_{X,Z}(\sigma_{Y=a}e(X, Y, Z))$."

In comparison with the *up-down* processing, the *all-down* processing performs semi-naive evaluation without derivation of magic sets. This is because the instantiation at the exit vector makes both sides directly and synchronously evaluable.

Similarly, if $f(a, X)$ and $g(b, Z)$ provide highly selective instantiations,

$$? - f(a, X),\ r(X, Y, Z),\ g(b, Z)$$

the query above should be evaluated by *all-up* processing which derives a binary magic set by starting with both sides. Here the semi-naive evaluation is unnecessary since the join of the query-relevant portion of the up-relation with e derives the required answer set. □

Clearly, both *all-down* processing and *all-up* processing can be viewed as special cases of Algorithm 4.7, in which the former (that is, the *all-down* processing) omits the computation of magic sets while the latter omits the semi-naive evaluation.

Although many queries require to derive query-relevant closures (using the *query closure strategy*), some queries may require different evaluation strategies, such as *nonrecursive, total closure*, and *existence checking* [42]. Moreover, although *binary algorithms* may be necessary for some queries to register (*source, sink*) pairs in the processing, *unary algorithms*, which do not trace (*source, sink*) pairs, may be sufficient for other queries. The use of unary algorithms, when possible, may substantially improve the processing efficiency according to the performance study by Bancihon and Ramakrishnan [10]. Among the algorithms developed in this paper, the magic set-based algorithms are binary algorithms while the counting or transitive closure-based algorithms are unary ones. Furthermore, quantitative analysis based on the selectivities of query instantiations and the sizes and join selectivities of side-relations play an important role in the determination of evaluation directions and algorithms. The selection of appropriate processing strategies for complex queries on single linear recursions has been studied in [42]. The principles derived from the study of SL recursions are applicable to the evaluation of ML recursions as well.

4.6 Summary

We have developed some efficient query evaluation techniques for side-coherent multiple linear recursions by an integration of side-relation unioned processing with transitive closure algorithms, the Counting method, and the Magic Sets method. Therefore, the processing of side-coherent multiple linear recursions is mapped to the framework

of the processing of single linear recursions. Most of the evaluation techniques developed in the study of single linear recursion can be applied to multiple linear recursions.

Our study of efficient evaluation of side-coherent multiple linear recursions can be extended to the evaluation of other kinds of multiple linear, nonlinear and mutual recursions. Although such kind of recursions can be evaluated by Generalized Magic Sets method [13], it is often beneficial to integrate side-relation unioned processing with the Generalized Magic Sets or other evaluation techniques. A detailed study of the compilation and optimization of recursions containing other kinds of multiple recursive rules is an interesting topic for the future research.

Chapter 5

Compressed Counting Method

In this chapter, the evaluation of function-free recursion by counting method in cyclic base relations is explored. Counting method is one of the query processing strategies used in *LogicBase* system, however, if base relation contains cycles, termination is not guaranteed by the counting method. Thus extension of the counting method (cyclic counting) is made to handle cycles and to retain efficiency of counting method.

We propose a counting method called *compressed counting* which combines the merits of several proposed cyclic counting algorithms and processes linear recursive queries in both cyclic and acyclic databases as efficiently as the counting method does in acyclic databases. The method precompiles database digraphs, compresses each strongly connected component (SCC) into a single node, and reduces the database digraph into a small DAG for guidance of query processing. Thus, query processing involving cyclic paths at both sides is simplified to the propagation and transformation of the precomputed offset-period information. Moreover, further optimization is performed on the computation involving both acyclic and cyclic paths. The derived algorithm uniformly handles both cyclic and acyclic data and facilitates parallel processing of queries in deductive databases.

5.1 Introduction

5.1.1 Background and motivation

Counting and Magic Sets are two well-known methods [9] for the efficient processing of queries on (single) linear recursions in deductive databases. Both complexity analysis and performance studies on a set of interesting linear recursive query processing algorithms [10, 11, 84] have shown that Counting has the time complexity of $O(ne)$ on acyclic databases, where n is the number of nodes and e is the number of edges in a database digraph, that is more efficient than Magic Sets which has the time complexity of $O(e^2)$. Unfortunately, Counting encounters termination problems when the database digraph contains cycles.

$$
\begin{array}{lll}
r(X,Y) & :- & up(X,U),\ r(U,V),\ down(V,Y). \\
r(X,Y) & :- & flat(X,Y). \\
?- & r(a,Y). &
\end{array}
$$

Figure 5.1: A typical linear recursion and its query.

The problem can be easily shown using a typical linear recursion problem defined in Figure 5.1 (essentially, the *same-generation recursion* [10, 132]), where R is recursive predicate, *up*, *down* and $flat$ are base relations [10], a is a query constant and Y is an inquired variable. The first rule in Figure 5.1 is a linear recursive rule, the second one is a nonrecursive (*exit*) rule, and the last one is a query on the recursive predicate R. The recursion can be compiled into the form $\bigcup_{k=0}^{\infty}(up^k\ flat\ down^k)$, which indicates that the answer set to the query should be those starting at a, traversing k times of the *up* relation, passing the $flat$ relation, and then traversing k times of *down* relation. This is the spirit of Counting. When *up* and *down* relations contain cyclic data, Counting cannot terminate since these relations can be traversed infinite number of times.

Much efforts have been paid on studying *cyclic counting* technique [3, 17, 39, 49,

41, 59, 115, 147] and several interesting algorithms have been proposed, which includes Sacca and Zaniolo's Magic Counting [115], Haddad and Naughton's Cyclic Counting [41], Han and Henschen's Level-Cycle Merging [49], Aly and Ozsoyoglu's Synchronized Counting [3], etc. Unfortunately, these proposals suffer from either efficiency or complex implementation problems. For example, Cyclic Counting [41] is elegant at handling cyclic paths in both *up* and *down* relations, but it cannot uniformly handle the mixture of acyclic and cyclic data, and moreover, all the information relevant to an SCC (strongly connected component) must be recomputed when different source nodes entering the same SCC in the query processing. Level-Cycle Merging [49] represents the counting-level by a level-cycle set and processes a query by level-cycle merging along paths. However, the derivation of pre-stable level set involves complicated computation and relatively large storage overhead during pre-compilation. Synchronized Counting [3] traverses cyclic database until the intermediate result becomes stable, that may take up to $O(n^2)$ semi-join operations over data relations.

This motivates our further study on cyclic counting algorithms. The study leads to the development of a *compressed counting* method that combines the merits of several counting algorithms, especially, Cyclic Counting, Level-Cycle Merging and Synchronized Counting, and processes queries in acyclic data, cyclic data and their mixtures as efficiently as the counting method does in acyclic databases. The method precompiles database digraphs, compresses each (maximal) strongly connected component (SCC) into a single node, and reduces the database digraph into a small directed acyclic graph (DAG) for guidance of processing. Query processing involving cyclic paths at both sides is simplified to the propagation and transformation of the precomputed information about relative distance set designated by *offset-period* pair. Moreover, further optimization is performed on the computation involving both acyclic and cyclic paths by a complementary counting technique. The derived algorithm uniformly handles both cyclic and acyclic data and facilitates the development of highly parallel processing algorithms.

5.1.2 Overview of compressed counting

In compressed counting, the answer set R to the query constitutes two parts: R_{acyc} and R_{cyc}. R_{acyc} is a set of nodes in (the digraph of) *down* that can be reached from the query constant(s) by a path of length l in *up*, then an edge in $flat$, and a path of length l in *down*, and at least one path in *up* or *down* is acyclic. R_{cyc} corresponds to a similar set of nodes but can be reached by cyclic paths in both *up* and *down*. If the acyclic counting algorithm [9] is applied until the counting level equal to the length of the longest acyclic path in *up* and *down*, R_{acyc} is obtained. Therefore, the major challenge is to find an efficient approach to deriving R_{cyc}. Based on the previous studies [41, 49], an infinite set of path lengths can be mapped to a finite set of periodic measurement for the derivation of R_{cyc}. Regardless of the absolute cyclic path length, two nodes linked by a cyclic path can be characterized by the period of the cyclic path and the offset of the path length to that period. Thus, an offset-period (OP) pair is used in the compilation and representation of cyclic data, where a cyclic path length is measured by the period of the path and the offset of the length to the period.

To efficiently derive OP pairs for nodes in data digraphs, *up* and *down* digraphs are precompiled into two compressed graphs, CG_{up} and CG_{down}, each of which is a small DAG that consists of a set of nodes and edges. Each node represents a maximal strongly connected components (SCC), i.e., a cluster of interconnected cycles. Each inter-SCC edge reflects how offset values change as one traverses from one SCC to another. Using such a compressed graph, OP's are propagated among SCCs, and each data node in an SCC gets its corresponding OP readily.

Figure 5.2 outlines the paradigm of the compressed counting method.

Example 5.1 Before a systematic presentation of the method, a tiny example is presented in Figure 5.3 to illustrate the idea of the technique. Edges in *up* and *down* (digraphs) are illustrated with solid lines whereas that in $flat$ with dash line. Each data node is associated with an offset-period pair in the form of [offset set, period]. In *up*, nodes b, c, d and e form an SCC with a period of 4. The distance from a to

- **Precompilation Phase:**
  Precompilation of G_{up} and G_{down}: (1) partition each digraph (G_{up}/G_{down}) into a set of SCCs, (2) derive internal OP's for the nodes in each SCC, and (3) construct compressed graphs, CG_{up} and CG_{down}.

- **Query Processing Phase:**

  1. **up-Processing:** compute distance-OP's for all data nodes reachable from query constants in G_{up}. This includes the computation of counting levels for acyclic node from query constants in G_{up}, initialization of OP's of SCCs reachable from the constants in G_{up}, derivation of distance-OP's for all SCC nodes in CG_{up} by merging OP's in topological order, and computation of OP's for the reachable nodes.
  2. **down-Processing and R_{cyc} Extraction:** It contains 3 steps: (1) instantiate nodes in G_{down} via *flat* and pass the OP's computed in G_{up}; (2) compute difference-OP's for all data nodes in G_{down}: (i) obtain counting levels for acyclic nodes from instantiated nodes in G_{down}, (ii) initialize difference-OP's of SCCs in CG_{down} reachable from instantiated nodes, and (iii) derive difference-OP's for all SCCs and the nodes along in G_{down} by merging difference-OP's in topological order in CG_{down}; and (3) extract R_{cyc} by detecting whether the difference-OP of a node in G_{down} contains 0.
  3. **Complementary Counting for Extraction of R_{acyc}.**

Figure 5.2: Overview of compressed counting method.

them can be measured as [1,4], [2,4], [3,4], [0,4] respectively, where the first number in a pair is the offset and second is the period for a cyclic distance. In *down*, f and g form a cycle with period of 2 and distance from f measured as [0,2] and [1,2]. The difference-OP for a node in *down* is the subtraction of the inherited OP from its own distance-OP. Since f has an OP of [3, 4] inherited from d in *up* relation, f's difference-OP is [1, 2]. Similarly, g's difference-OP is [0, 2]. Since only g's difference-OP contains 0, g is the only answer to the query. □

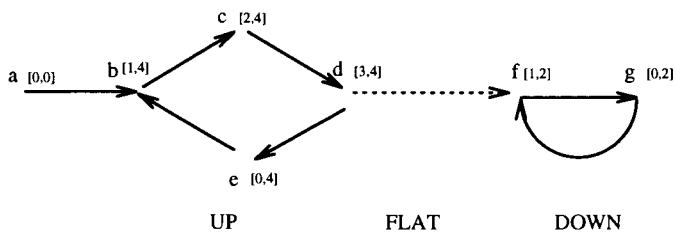


Figure 5.3: A tiny example database.

5.2 Principles of Compressed Counting

The theoretical foundation for the derivation of R_{cyc} is presented in this section, which includes representation of and operations on offset-period pairs. The correctness and completeness of the method is proven here, the actual compressed counting method is presented in next section.

In this section, it is shown that a distance set can be replaced by its partial periodic subset to derived R_{cyc}; and this subset can be represented by OP; thus the derivation of the distance set and difference set is accomplished by the derivation of OP; and finally the query answer extraction becomes 0-containment test on difference set, which is actually done on the corresponding OP set.

5.2.1 Distance set and difference set

The concepts of distance set and difference set are introduced to formalize query answering.

Without loss of generality, EDB relations are assumed to be binary relations. A binary relation A can be represented as a digraph (directed graph) $G(V, E)$, where $a \in V$, $b \in V$, and $(a, b) \in E$ if and only if there is a tuple $(a, b) \in A$. G_{up} and G_{down} denote digraphs for *up* and *down* relations.

Distance set defined below is to represent the length of paths from one node to another in a digraph. The length of each edge in the digraph is 1, and the length of

a path is represented by an integer.

Definition 5.1 *For nodes $c, x \in V$ in digraph G,* **distance set** $D(c,x) = \{l : \exists$ *a path $c \rightarrow x$ of length l in $G\}$.*

Distance set $D(c,x)$ contains an infinite set of integers when there exists a cyclic path between c and x. If there is no path from c to x, $D(c,x) = \emptyset$. $D(x,x) = \{0\}$ when there is no cycle passing through x. The answer for the recursive query in Figure 5.1 can be expressed by distance sets: $\{z : z \in V_{down}$ *and* $\exists x \in V_{up}$ *and* $\exists y \in V_{down}$ *s.t.* $(x,y) \in flat$ *and* $D(a,x) \cap D(y,z) \neq \emptyset\}$. That is, there exist paths from a to x in G_{up} and from y to z in G_{down} of the same length and joined by an edge (x,y) in G_{flat}.

The addition and subtraction operations on distance sets are defined as follows:

Definition 5.2 *For integer set C_1 and C_2,* addition $C_1 \oplus C_2 = \{c_1 + c_2 : c_1 \in C_1$ *and* $c_2 \in C_2\}$. *For integer i,* $C_1 \oplus i = \{c + i : c \in C_1\}$.

Definition 5.3 *For integer set C_1 and C_2,* subtraction $C_1 \ominus C_2 = \{c_1 - c_2 : c_1 \in C_1$ *and* $c_2 \in C_2\}$. *For integer i,* $C_1 \ominus i = \{c - i : d \in C_1\}$.

$C_1 \oplus C_2$ and $C_1 \ominus C_2$ are $\emptyset$ if either C_1 or C_2 is $\emptyset$. Addition and subtraction on integer sets are commutative.

The answer to the recursive query can be represented as the following set: $\{z : z \in V_{down}$ *and* $\exists x \in V_{up}$ *and* $\exists y \in V_{down}$ *s.t.* $(x,y) \in flat$ *and* $0 \in (D(y,z) \ominus D(a,x))\}$.

Modulus operation on integer set is defined as following:

Definition 5.4 *For integer set C and integer i,* $C \bmod i = \{c \bmod i : c \in C\}$.

Difference set is defined to denote the subtraction of two distance sets, particularly in this paper, the subtraction of distance sets in G_{down} and G_{up}.

Definition 5.5 *For $a \in V_{up}$ and $z \in V_{down}$,* **difference set** $Diff(a,z) = \{l : \exists x \in V_{up}$ *and* $\exists y \in V_{down}$ *s.t.* $(x,y) \in flat$ *and* $l \in (D(y,z) \ominus D(a,x))\}$.

Lemma 5.1 states that distance set can be derived progressively through addition operation.

Lemma 5.1 *For* $x, z \in V$, $D(x,z) = \bigcup_{y \in V}(D(x,y) \oplus D(y,z))$.

Proof: If $d \in D(x,z)$, then there exists a node y' such that path $x \rightarrow z$ passing through y' (y' could even be x or z). So there is a path $x \rightarrow y'$ with length d_1 and a path $y' \rightarrow z$ with length d_2 such that $d = d_1 + d_2$. Since $d_1 \in D(x,y')$ and $d_2 \in D(y',z)$, $d \in D(x,y') \oplus D(y',z)$. Hence $d \in \bigcup_{y \in V}(D(x,y) \oplus D(y,z))$. So we have $D(x,z) \subseteq \bigcup_{y \in V}(D(x,y) \oplus D(y,z))$.

If $d \in \bigcup_{y \in V}(D(x,y) \oplus D(y,z))$, there exists a $y' \in V$ such that $d \in D(x,y') \oplus D(y',z)$. There exist $d_1 \in D(x,y')$ and $d_2 \in D(y',z)$ such that $d = d_1 + d_2$. So there is a path from x to y' of length d_1 and there is a path from y' to z of length d_2. So there is a path from x to z via y' of length $d_1 + d_2$. Hence $d \in D(x,z)$. So $\bigcup_{y \in V}(D(x,y) \oplus D(y,z)) \subseteq D(x,z)$.

Thus, $D(x,z) = \bigcup_{y \in V}(D(x,y) \oplus D(y,z))$. □

Lemma 5.2 states that although difference set is defined by difference of distance sets, it can be derived through addition of difference set and distance set. Which ensures that the distance set in G_{up} and difference set in G_{down} are treated in the same way.

Lemma 5.2 *For* $a \in V_{up}$ *and* $z \in V_{down}$, $Diff(a,z) = \bigcup_{y \in N_{down}}(Diff(a,y) \oplus D(y,z))$.

Proof: We first prove $(\bigcup_{i=1}^{k} D_i) \ominus D = \bigcup_{i}^{k}(D_i \ominus D)$. If $d \in (\bigcup_{i=1}^{k} D_i) \ominus D$, then there exist $d_1 \in \bigcup_{i=1}^{k} D_i$ and $d_2 \in D$ such that $d = d_1 - d_2$. Hence $d_1 \in D_j (0 < j \leq k)$ and $d \in (D_j \ominus D)$. So $d \in \bigcup_{i=1}^{k}(D_i \ominus D)$. That is, $(\bigcup_{i=1}^{k} D_i) \ominus D \subseteq \bigcup_{i=1}^{k}(D_i \ominus D)$.

On the other hand, if $d \in \bigcup_{i=1}^{k}(D_i \ominus D)$, then there exists $0 < j \leq k$, such that $d \in D_j \ominus D$. So there exist $d_1 \in D_j$ and $d_2 \in D$ such that $d = d_1 - d_2$. Hence $d_1 \in \bigcup_{i=1} kD_i$ and $d_1 - d_2 \in (\bigcup_{i=1} kD_i) \ominus D$. So, $\bigcup_{i=1} k(D_i \ominus D) \subseteq (\bigcup_{i=1} kD_i) \ominus D$. Thus, $\bigcup_{i=1} k(D_i \ominus D) = (\bigcup_{i=1} kD_i) \ominus D$.

Now we prove the lemma. From the definition of difference set, we have:

$$Diff(a,z) = \bigcup_{(x,y)\in flat} (D(y,z) \ominus D(a,x))$$

From Lemma 5.1:

$$Diff(a,z) = \bigcup_{(x,y)\in flat} (\bigcup_{w\in V_{down}} (D(y,w) \oplus D(w,z)) \ominus D(a,x))$$

which equals to $\bigcup_{(x,y)\in flat} (\bigcup_{w\in V_{down}}((D(y,w) \ominus D(a,x)) \oplus D(w,z)))$, which is:

$$\bigcup_{w\in V_{down}} (\bigcup_{(x,y)\in flat} (D(y,w) \ominus D(a,x)) \oplus D(w,z)) = \bigcup_{w\in V_{down}} (Diff(a,w) \oplus D(w,z))$$

□

Answer set for recursive query in Figure 5.1 can be rewritten by difference set as $\{z : z \in V_{down}\ s.t.\ 0 \in Diff(a,z)\}$. The principle of the compressed counting method is to find those nodes in G_{down} whose corresponding difference sets contain 0.

5.2.2 Offset-period representation

The distance set is infinite if there is a cycle in data digraph, which makes handling of distance set difficult. In this section, a special representation of distance set is presented to catch the regularity of distance set and map the infinite sets into finite ones. Both distance set and difference set can be represented by the offset-period (OP) representation to derive R_{cyc}.

It is first proved in the following lemma that if two distance sets for cyclic paths intersect, the intersection is an infinite set.

Lemma 5.3 *If paths $a \rightarrow x$ and $y \rightarrow z$ are cyclic in up and down relations respectively and $D(a,x) \cap D(y,z) \neq \emptyset$, then $D(a,x) \cap D(y,z)$ is an infinite set.*

Proof: Since $D(a,x) \cap D(y,z) \neq \emptyset$, we assume $d \in D(a,x) \cap D(y,z)$ and $c_1 \in D(x',x')$ where x' is a node on the path $a \rightarrow x$ and $c_2 \in D(y',y')$ where y' is a node on the path $y \rightarrow z$ ($c_1, c_2 > 0$). We have $d + c_1 \times i \in D(a,x)$ and $d + c_2 \times i \in D(y,z)$ for $i = 0, 1, \ldots$. Hence, $d + c_1 \times c_2 \times i \in D(a,x)$ and $d + c_1 \times c_2 \times i \in D(y,z)$ for $i = 0, 1, \ldots$. So $D(a,x) \cap D(y,z)$ is infinite. □

The concept of strongly connected component is employed to help analyze the cyclic behavior in digraph. A strongly connected component (SCC) in a digraph is a subgraph in which there is a path between any pair of nodes. In the following context, we assume that SCC refers to maximal SCC. A directed cyclic graph is composed of several SCCs connected by acyclic paths.

Definition 5.6 *The* period *of an SCC is the greatest common divisor of the lengths of all the cycles in the SCC.*

An $O(e_{SCC})$ method is presented in [49, 41] to calculate the period of an SCC, where e_{SCC} is the number of edges in the SCC.

Lemma 5.4 states that in an SCC, all paths between two nodes have the same offset to the SCC period, which ensures the uniqueness of offset to the period of the SCC, and path length pattern (captured by OP) between two nodes are independent on the actual path.

Lemma 5.4 *Suppose SCC $S(V,E)$ has the period of p and $x, y \in V$. Then $D(x,y)$* mod *p has only one integer in the result set.*

Proof: We prove that any two paths from x to y will have the same value after $D(x,y)$ mod p. Suppose there are two paths from x to y with lengths l_1 and l_2 respectively, and there is a path from y to x with length k. Then, there are two cycles

in SCC with the lengths of $l_1 + k$ and $l_2 + k$. According to the definition of the SCC period, $l_1 + k = n_1 \times p$, $l_2 + k = n_2 \times p$, where n_1, n_2 are positive integers. Hence we have $l_1 - n_1 \times p = l_2 - n_2 \times p$, and $(l_1 - n_1 \times p) \bmod p = (l_2 - n_2 \times p) \bmod p$. Thus, $l_1 \bmod p = l_2 \bmod p$. ◻

Lemma 5.7 states that self cycle in an SCC becomes periodic after certain levels. Lemma 5.5 and Lemma 5.6 are two auxiliary lemmas helping prove Lemma 5.7.

Lemma 5.5 *For k positive integers $n_1, n_2, \ldots, n_k$ such that $gcd(n_1, n_2, \ldots, n_k) = 1$, there exists an integer N_0 such that for any integer $N > N_0$, $N = \lambda_1 n_1 + \lambda_2 n_2 + \cdots + \lambda_k n_k$ where $\lambda_1, \lambda_2, \ldots, \lambda_k \in \mathcal{N}_0$, where $\mathcal{N}_0$ denotes the set of integers greater than or equal to 0.*

Proof: Without loss of generality, it is assumed that $n_1 > n_2 > \cdots > n_k > 0$, and that $n_i = \alpha_i n_{i+1} + \gamma_i$ where $\alpha_i, \gamma_i \in \mathcal{N}_0$ for $i = 1, 2, \ldots, k-1$. Since the set of γ_i $(i = 1 \ldots k-1)$ and n_k have the same *gcd* as the set of n_i, $gcd(\gamma_1, \gamma_2, \ldots, \gamma_{k-1}, n_k) = 1$. This means that we have a set of smaller numbers with *gcd* equals 1. By sorting this new set of numbers and performing the above procedure finite times, we will have a γ value of 1, which is the arithmetic combination of the original n_i numbers. In other words, $1 = \beta_1 n_1 + \beta_2 n_2 + \cdots + \beta_k n_k$ where β_i are integers for $i = 1, 2, \ldots, k$. Now we set $N_0 = |n_k \beta_1| n_1 + |n_k \beta_2| n_2 + \cdots + |n_k \beta_k| n_k$. For any $N > N_0$, assume $N - N_0$ is $mn_k + \gamma$ where $m, \gamma \in \mathcal{N}_0$ and $\gamma < n_k$. N can be represented as $N_0 + mn_k + \gamma$. Hence $N = N_0 + mn_k + \gamma(\beta_1 n_1 + \beta_2 n_2 + \cdots + \beta_k n_k)$ which is $\Sigma_{i=1}^{k-1}((|n_k \beta_i| + \gamma\beta_i) n_i) + (|n_k \beta_k| + \gamma\beta_k + m) n_k$. Since $\gamma < n_k$, so $|n_k \beta_i| + \gamma\beta_i > 0$ and $|n_k \beta_k| + \gamma\beta_k + m > 0$, so N is rewritten as $\lambda_1 n_1 + \lambda_2 n_2 + \cdots + \lambda_k n_k$, where $\lambda_i = |n_k \beta_i| + \gamma\beta_i$ for $1 \leq i \geq k-1$, and $\lambda_k = |n_k \beta_k| + \gamma\beta_k + m$, which proves the lemma. □

Lemma 5.6 $gcd(a_1, \ldots, a_k, a_1 + c_1, \ldots, a_k + c_k) = gcd(a_1, \ldots, a_k, c_1, \ldots, c_k)$.

Proof: Assume $gcd(a_1, \ldots, a_k, a_1 + c_1, \ldots, a_k + c_k) = p$, and $gcd(a_1, \ldots, a_k, c_1, \ldots, c_k) = q$, where $p, q \geq 1$. Hence it can be assumed that $a_i = l_i \times q, c_i = m_i \times q$. We have

$a_i + c_i = (l_i + m_i) \times q$. So q is a common divisor of $a_1, \ldots, a_k, a_1 + c_1, \ldots, a_k + c_k$. Then $p \bmod q = 0$.

On the other hand, since p is the *gcd* of set of a_i and $a_i + c_i$, it can be assumed that $a_i = n_i \times p$ and $a_i + c_i = o_i \times p$, so $c_i = (o_i - n_i) \times p$. Hence p is a common divisor of $a_1, \ldots, a_k, c_1, \ldots, c_k$. So $q \bmod p = 0$. So, we have $p = q$. □

Lemma 5.7 *For any node x in SCC, there exists an integer n_0 such that for $n \geq n_0, n \times p \in D(x,x)$, p is the period of SCC.*

Proof: Although the period of SCC is defined as *gcd* of all the cycles in SCC, there exist a finite set of cycles $\{cycle_1, \ldots, cycle_k\}$ with lengths of $c_1, \ldots, c_k$ such that $p = gcd(c_1, c_2, \ldots, c_k)$. For any node x, there exist self cycles $\{scycle_1, \ldots, scycle_k\}$ with the lengths of $sc_1, \ldots, sc_k$, such that $scycle_i$ starts at x, reaches a node on $cycle_i$, and returns x. Hence $c_i + sc_i$ is the length of a self cycle of x, that is, $c_i + sc_i \in D(x,x)$.

Since period is the *gcd* of all cycles in SCC, $gcd(c_1, \ldots, c_k, sc_1, \ldots, sc_k) = p$. From Lemma 5.6, we have $gcd(c_1 + sc_1, \ldots, c_k + sc_k) = p$. Assume $c_i + sc_i = n_i \times p (i = 1, \ldots, k)$, we have $gcd(n_1, n_2, \ldots, n_k) = 1$. Based on Lemma 5.5, there exists n_0 such that for $n \geq n_0, n = \lambda_1 n_1 + \lambda_2 n_2 + \cdots + \lambda_k n_k$. In other words, $n \times p = \lambda_1 \times n_1 \times p + \lambda_2 \times n_2 \times p + \cdots + \lambda_k \times n_k \times p$. Hence $n \times p = \lambda_1 \times (c_1 + sc_1) + \lambda_2 \times (c_2 + sc_2) + \cdots + \lambda_k \times (c_k + sc_k)$. This is equivalent to the path length of cycle passing through corresponding self cycles of length $c_i + sc_i$ λ_i times respectively. So, for $n \geq n_0$, $n \times p \in D(x,x)$. □

Since period p of an SCC is the *gcd* of all cycles, if $d \in D(x,x)$ in SCC, then $d = n \times p$. Hence we have integer set $\{n \times p : n = n_0, n_0 + 1, \ldots\}$ which is equivalent to the subset of $D(x,x)$ of $\{d : d \in D(x,x)$ *and* $d \geq n_0 \times p\}$. Such integer set is called *asymptotically equivalent* to $D(x,x)$. The distance set of cyclic path in an SCC is measured by its asymptotically equivalent set.

Definition 5.7 *Integer set C_1 and C_2 are* **asymptotically equivalent** *if there exists an integer n_0 such that if $c \in C_1$ and $c \geq n_0$ then $c \in C_2$; if $c \in C_2$ and $c \geq n_0$ then*

$c \in C_1$. $C_1 \Leftrightarrow C_2$ *denotes asymptotical equivalence.* n_0 *is called the* stable level *of* C_1 *and* C_2.

Asympototic equivalence of a distance set intends to represent the cyclic portion of the distance set and difference set with a regular formula of $np + c$, $n \geq n_0$, where $[c, p]$ forms the OP representation for the distance set and difference set.

The above lemmas suggest a simple representation of $D(x, y)$ in an SCC. Since only synchronized cyclic paths in the *up* and *down* digraphs may contribute to the answer set R_{cyc}, and the distance set in the cyclic portion becomes periodic after certain level, instead of comparing the whole distance sets in *up* and *down* relations, the periodic partial sets are adequate for that purpose. Therefore, the distance set $D(x, y)$ in an SCC can be simply expressed as $[c, p]$, where p is the period of the SCC, and c is the distance from x to y moduled by p (called the *internal offset* of y in terms of x in SCC). Notice that there is only one c value according to Lemma 5.4. This offset-period pair represents an integer set which is asymptotically equivalent to $D(x, y)$ in an SCC. It captures the regularity of distances set when the distance becomes greater than the stable level. OP[c,p] (indicating [offset, period]) is used to denote the representation.

In each SCC, one node r is assumed to be the *reference node* with 0 as its offset. Thus, the *internal OP* (OP represent distance set within an SCC) of any other node q in the SCC is represented by a unique OP tuple $[c, p]$, where p is the period of SCC, and $c = D(r, q) \bmod p$, the relative distance from the reference node. The distance set between any two nodes x and y in the SCC can be inferred from OP of x to the reference node and OP of reference node to y. Furthermore, the internal offset c of a node in SCC is relative to the reference node such that if a new offset c' is assigned to the reference node, every node in SCC with the offest c will have the same offset "drifting" with new offset value of $(c + c') \bmod p$.

The offset-period representation of distance set in SCC can be extended to representing distance set in general cyclic data relations, where the digraph $G(V, E)$ is composed of several SCCs. For $x, y \in V$, there might be many paths from x to y, and

each path might traverse through different SCCs.

Theorem 5.1 states that the distance set of a cyclic path becomes periodic over the greatest common divisor of all the SCC periods along the path.

Theorem 5.1 *For cyclic path $x \rightarrow y$ in G passing through SCC_1, SCC_2, ..., SCC_k with periods of $p_1, p_2, \ldots, p_k$, there exists an integer d_0 such that if $d \in D(x,y)$ and $d \geq d_0$, then $d + p \in D(x,y)$ where $p = gcd(p_1, p_2, \ldots, p_k)$.*

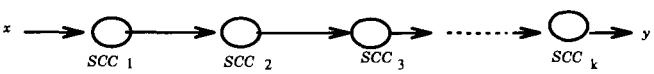


Figure 5.4: A path passing through SCCs.

Figure 5.4 illustrates the case, in which the distance from x to y is periodic on p when the distance is greater than d_0.

Proof: Suppose that a path x to y passes through $SCC_1, SCC_2, \ldots, SCC_k$ via acyclic paths. Its length can be represented as $l + n_1 \times p_1 + n_2 \times p_2 + \cdots + n_k \times p_k$ where $n_i \in \mathcal{N}_0$, $n_i \times p_i$ represents the length of self cycles within SCC_i, and l represents the accumulated acyclic path length from x to y. From Lemma 5.7, for SCC_i there exists n_{0i}, such that self cycle in SCC_i becomes periodic after level n_{0i}.

Assume that $p_i = \alpha_i \times p$ for $i = 1, \ldots, k$, we then have $gcd(\alpha_1, \alpha_2, \ldots, \alpha_k) = 1$. From Lemma 5.5, there exists $\alpha_0 \in \mathcal{N}_0$ such that for any $\alpha \geq \alpha_0, \alpha = \beta_1 \times \alpha_1 + \beta_2 \times \alpha_2 + \ldots + \beta_k \times \alpha_k$ for $\beta_i \in \mathcal{N}_0$.

We may now set $d_0 = \alpha_0 \times p + \Sigma_{i=1}^{k} n_{0i} \times p_i + e$, where e is the number of edges in data digraph. If $d \geq d_0$ and $d \in D(x,y)$, it can be assumed that $d = l + n_1 \times p_1 + n_2 \times p_2 + \ldots + n_k \times p_k$. Because $d \geq d_0$, d may be represented as $d = d_0 + p'$, which is $l + \Sigma_{i=1}^{k} n_{0i} \times p_i + p'$, where $p' \geq 0$ and $p' \bmod p = 0$. Hence $d \geq d_0$ is equivalent to $l + \Sigma_{i=1}^{k} n_{0i} \times p_i + p' \geq \alpha_0 \times p + \Sigma_{i=1}^{k} n_{0i} \times p_i + e$. Since $l < e$, we have $p' > \alpha_0 p$. Since $p' \bmod p = 0$, p' can be represented as $\Sigma_{i=1}^{k} \beta_i \times \alpha_i \times p$, which is $\Sigma_{i=1}^{k} \beta_i \times p_i$. Hence $d + p = l + \Sigma_{i=1}^{k} n_{0i} \times p + p' = l + \Sigma_{i=1}^{k} (n_{0i} + \beta_i) \times p_i$. Since each $(n_{0i} + \beta_i) \times p_i$ is the length of a self cycle in SCC_i, $l + \Sigma_{i=1}^{n} (n_{0i} + \beta_i) p_i$ is the length of path $x \rightarrow y$. So $d + p \in D(x,y)$. □

Definition 5.8 *The* period of a cyclic path *in a digraph is defined as the greatest common divisor of the periods of the SCCs along the path.*

To extend OP representation of distance set from a single SCC to a digraph of multiple SCCs, we first assume that all the paths from one node to another traverse the same set of SCCs in the same order. This gives us the same period for all paths, the presence of multiple acyclic paths joining SCCs results in multiple offsets to the period, and hence the OP representation should contain a set of offsets, such as $OP[C, p]$ where C is a set of offsets, called (*external*) *offsets* in contrast to internal offset within an SCC. Now assume that different paths traverse different SCCs and/or in different order. There are multiple OP's corresponding to each set of paths traversing the same set of SCCs in the same order, namely $[C_1, p_1], \ldots, [C_k, p_k]$. Since the distance set is the union of distance set along each path, the OP representation of the distance set should be able to express unioned set of integers expressed by OP's along each path, which is obtained by expanding each OP from $[C_i, p_i]$ to $[C_i', p]$ where $p = lcm(p_1, \ldots, p_k)$ and $C_i' = \{c, c + p_i, \ldots, c + (\lambda - 1)p_i : c \in C_i \text{ and } p = \lambda p_i\}$; and then unioning all these C_i'. Thus, the OP representation for distance set is $[\bigcup_{i=1}^{k} C_i', p]$. This is proved in a later section.

Now consider the presence of a reference node in an SCC. Since the distance set is represented in the OP form, it is necessary to know the OP set for each node of an SCC in order to answer a recursive query. However, we will show later that it is adequate to derive the OP set only for a reference node in each SCC. The OP set of every other node in the SCC can be inferred from their internal offset and the OP of the reference node.

The offset-period representation of distance sets and the corresponding operations lay the foundation for the discussion of recursive query processing in cyclic databases using counting method.

It should be noted that there could be multiple OP representations with different C sets and/or p for the same distance set, and each of them can be used to determine the answer to the query. However, it is desirable to use the normal form of OP.

Two OP's are equivalent if they denote the same integer set, although their representation may not be the same.

Definition 5.9 *$OP[C,p]$ is in* normal form *if for $c \in C$, $0 \leq c < p$, and for any $OP[C',p']$ which is equivalent to $OP[C,p]$, $p' \geq p$.*

Normalization operation denoted as $Norm(OP[C,p])$ transforms an OP representation of the distance set into a normal form OP representation by simplifying OP[C,p] into an equivalent OP set of OP'$[C',p']$ with the smallest p'. For example, $OP[(1,2,3,7,8,9),12]$ can be replaced by $OP[(1,2,3),6]$.

Offset-Period representation of the distance set maps an infinite set into a pair: period and a small set of offsets. As shown in the following sections, the operations defined on an OP map the derivation and comparison of distance sets to the operations on OP sets. Although OP cannot represent a complete difference set, it will be shown that OP representation is perfect for difference set to facilitate efficient test of 0 containment. Therefore, it is adequate to derive R_{cyc} using OP representation.

5.2.3 Derivation of OP sets

In this section, it is shown that operations on distance set and difference set can be replaced by operations on OP.

Addition of $OP[C_1,p_1]$ and $OP[C_2,p_2]$, denoted as $OP[C_1,p_1] \oplus OP[C_2,p_2]$, is an integer set: $\{(c_1 + k_1 \times p_1) + (c_2 + k_2 \times p_2) : c_1 \in C_1 \text{ and } c_2 \in C_2 \text{ and } k_1, k_2 \in \mathcal{N}_0\}$. Since $(c_1 + k_1 \times p_1) + (c_2 + k_2 \times p_2)i$ equals to $c_1 + c_2 + k \times gcd(p_1,p_2)$ which is $(c_1+c_2) \bmod gcd(p_1,p_2) + k' \times gcd(p_1,p_2)$, $OP[C_1,p_1] \oplus OP[C_2,p_2]$ can be represented as $OP[(C_1 \oplus C_2) \bmod p, p]$ where $p = gcd(p_1,p_2)$.

Definition 5.10 *$OP[C_1,p_1] \oplus OP[C_2,p_2]$ is $OP[(C_1 \oplus C_2) \bmod p, p]$, where $p = gcd(p_1, p_2)$. $OP[C_1,p_1] \oplus C_2$ is $[(C_1 \oplus C_2), p_1]$.*

Theorems 5.2 and 5.3 state that the addition and union of distance sets can be obtained through the addition of OP sets.

Theorem 5.2 *Let $OP[C_1, p_1]$ and $OP[C_2, p_2]$ be the OP representation for distance sets D_1 and D_2 respectively. $OP[C_1, p_1] \oplus OP[C_2, p_2]$ is the OP representation for $D_1 \oplus D_2$.*

Proof: This is to prove that $OP[C_1, p_1] \oplus OP[C_2, p_2]$ is asymptotically equivalent to $D_1 \oplus D_2$, given that $OP[C_1, p_1]$ and $OP[C_2, p_2]$ are asymptotically equivalent to D_1 and D_2 respectively.

Assume that n_{01}, n_{02} are the stable levels for D_1, D_2 respectively. Let $N = max(n_{01}, n_{02})$, and $N_0 = 4 \times N$.

If $d \in D_1 \oplus D_2$ and $d \geq N_0$, then $d = d_1 + d_2$ for $d_1 \in D_1, d_2 \in D_2$. If $d_1 \geq N$ and $d_2 \geq N$, then $d_1 \in OP[C_1, p_1]$ and $d_2 \in OP[C_2, p_2]$. So, $d \in OP[C_1, p_1] \oplus OP[C_2, p_2]$. Suppose that $d_1 < N$ ($d_2 < N$ case is proved in the same way). It can be assumed that on the path corresponding to D_1, there is a self cycle with length l, which is a multiple of p_1 from the definition of SCC period, and without loss of generality l is assumed to be less than N. Then there exists an integer $\lambda > 0$ such that $N \leq d_1 + \lambda \times l < N + l$. Further, d can be rewritten as $d_1 + \lambda \times l + d_2 - \lambda \times l$. It is clear that $d_1 + \lambda \times l$ is a path length, so $d_1 + \lambda \times l \in D_1$, and can be rewritten as $c_1 + k_1 \times p_1$ for $c_1 \in C_1$. Since $d_1 + d_2 \geq 4N$ and $\lambda \times l < 2N$, $d_2 - \lambda \times l > N$. Thus d_2 can be written as $c_2 + k_2 \times p_2$ for $c_2 \in C_2$ since $d_2 \in OP[C_2, p_2]$; and l as $a \times p_1$ since l is the length of self cycle. So $d = c_1 + k_1 \times p_1 + c_2 + k_2 \times p_2 - a \times p_1 = c_1 + c_2 + k \times gcd(p_1, p_2)$, that is, $d \in OP_1 \oplus OP_2$.

If $d \in OP_1 \oplus OP_2$ and $d \geq N_0$, $d = c_1 + c_2 + k \times gcd(p_1, p_2)$. Thus d can be written as sum of $c_1 + k_1 \times gcd(p_1, p_2) > N$ and $c_2 + k_2 \times gcd(p_1, p_2) > N$ for $k_1 + k_2 = k$. Since $c_1 + k_1 \times gcd(p_1, p_2) \in D_1$ and $c_2 + k_2 \times gcd(p_1, p_2) \in D_2$, $d \in D_1 \oplus D_2$. It has been proved that $OP[C_1, p_1] \oplus OP[C_2, p_2] \Leftrightarrow D_1 \oplus D_2$. □

The union of $OP[C_1, p_1]$ and $OP[C_2, p_2]$, denoted as $OP[C_1, p_1] \cup OP[C_2, p_2]$ is defined below. The result of the union should be $\{c_1 + k \times p_1, c_2 + k \times p_2 : c_1 \in$

C_1 *and* $c_2 \in C_2$ *and* $k \in \mathcal{N}_0\}$. If p_1 and p_2 are the same, an OP set can be constructed with $C_1 \cup C_2$ as its offset set and p_1 as its period. Otherwise, each OP can be transformed into an OP with its period equivalent to the least common multiplier of p_1 and p_2.

Definition 5.11 Expansion operation, *denoted as* $Exp(OP[C,p],\lambda p) = OP[C',\lambda p]$, *transforms* $OP[C,p]$ *into an equivalent OP with a larger period, where* $OP[C,p]$ *is normalized and* λ *is a positive integer,* $C' = \{c+kp : c \in C \text{ and } k = 0,1,\ldots,(\lambda-1)\}$.

Definition 5.12 Union operation $OP[C_1,p_1] \cup OP[C_2,p_2]$ *is* $[(C_1' \cup C_2'),p]$, *where* $p = lcm(p_1,p_2)$ *and* $[C_1',p] = Exp([C_1,p_1],p), [C_2',p_2] = Exp([C_2,p_2],p)$.

The following theorem states the union of distance sets can be obtained by union of OP.

Theorem 5.3 *Suppose* $OP[C_1,p_1]$ *and* $OP[C_2,p_2]$ *are the OP representation for distance sets* D_1 *and* D_2 *respectively, then* $OP[C_1,p_1] \cup OP[C_2,p_2]$ *is the OP representation for* $D_1 \cup D_2$.

Proof: It is to prove that $OP[C_1,p_1] \cup OP[C_2,p_2]$ and $D_1 \cup D_2$ are asymptotically equivalent given that $OP[C_1,p_1]$ and $OP[C_2,p_2]$ are asymptotically equivalent to D_1, D_2 respectively.

Assume n_{01}, n_{02} are the stable levels for D_1 and D_2. Let $N = max(n_{01}, n_{02})$.

If $d \in D_1 \cup D_2$ and $d \geq N$, then $d \in D_1$ or $d \in D_2$. Since $d \geq max(n_{01}, n_{02})$, $d \in OP[C_1,p_1]$ or $d \in OP[C_2,p_2]$. Hence $d \in OP[C_1,p_1] \cup OP[C_2,p_2]$.

If $d \in OP[C_1,p_1] \cup OP[C_2,p_2]$ and $d \geq N$, then $d \in OP[C_1,p_1]$ or $d \in OP[C_2,p_2]$. Since $d \geq max(n_{01}, n_{02})$, $d \in D_1$ or $d \in D_2$. Hence $d \in D_1 \cup D_2$. Thus the theorem is proved. □

The above theorems map the derivation of distance sets to that of OP sets. Since addition and union on distance sets are commutative, the addition and union operations on OP's are commutative.

We now prove that OP (either distance or difference OP) of a node in an SCC can be inferred from its internal offset and the reference node OP, so that an SCC can be collapsed into a single node in simplied data digraph.

Lemma 5.8 *If node $u, y, x \in SCC$ and there is a cyclic path from node a to the SCC, $D(a,u) \oplus D(u,x)$ is asymptotically equivalent to $D(a,y) \oplus D(y,x)$.*

Proof: $D(a,u) \oplus D(u,x)$ and $D(a,y) \oplus D(y,x)$ represent the lengths of the paths $a \rightarrow u \rightarrow x$ and $a \rightarrow y \rightarrow x$. Since u, y are in the same SCC, $D(a,u) \oplus D(u,x)$ and $D(a,y) \oplus D(y,x)$ have the same period p_0. Assume $N = max$(stable levels of $D(a,u) \oplus D(u,x)$, $D(a,y) \oplus D(y,x)$, $D(a,u)$). Since $D(u,x)$ is within SCC, $D(u,x) \Leftrightarrow OP[c_x, p]$ where p is the period of the SCC. Since period of $D(a,u)$ is the gcd of SCC periods, $p \bmod p_0 = 0$ and $p_0 \leq p$. So, $D(a,u)$ may have OP representation of $OP[C_u, p]$ with period of p instead of p_0. So, $D(a,u) \oplus D(u,x) \Leftrightarrow OP[C_u, p] \oplus OP[c_x, p]$, which is $D(a,u) \oplus D(u,x) \Leftrightarrow OP[C_u \oplus c_x, p]$.

If $d \in D(a,u) \oplus D(u,x)$ and $d \geq N + p$, then $d \in OP[C_u \oplus c_x, p]$, $d = c_u + c_x + k \times p$. So $d - c_x = c_u + k \times p \in OP[C_u, p]$. Since $d \geq N + p$ and $d - c_x > N$, $d - c_x \in D(a,u)$. Since u, y are nodes in SCC, it is assumed $D(u,y) \Leftrightarrow OP[c_y, p]$. A path from u to y has the length of $c_y + k' \times p$. Hence $d - c_x + c_y + k' \times p$ is the length of a path $a \rightarrow u \rightarrow y$, and $d - c_x + c_y + k' \times p \in D(a,u) \oplus D(u,y) \subseteq D(a,y)$. Assume $D(y,x) \Leftrightarrow OP[c_{yx}, p]$. Since path $u \rightarrow y \rightarrow x \rightarrow u$ is a self cycle within SCC, and $D(x,u) \Leftrightarrow OP[(p - c_x), p]$, $c_y + c_{yx} + (p - c_x) \bmod p = 0$. So $D(y,x) \Leftrightarrow OP[(c_x - c_y), p]$. So there is a path $y \rightarrow x$ with length of $c_x - c_y + k'' \times p$. So $d - c_x + c_y + k' \times p + c_x - c_y + k'' \times p$ is the length of path $a \rightarrow y \rightarrow x$, in other word, $d + (k' + k'') \times p \in D(a,y) \oplus D(y,x)$. Assume that $D(a,y) \oplus D(y,x) \Leftrightarrow OP[C_y, p]$. So, $d + (k' + k'') \times p = c' + k''' \times p$ for $c' \in C_y$. $d = c' + (k''' - k'' - k') \times p \in OP[C_y, p]$. Since $d > N$, we have $d \in D(a,y) \oplus D(y,x)$.

It can be proved in the same way that if $d \in D(a,y) \oplus D(y,x)$ and $d \geq N + p$,

$d \in D(a,u) \oplus D(u,x)$. □

If the equivalence relationship is relaxed to asymptotical equivalence, the following theorem states that $D(a,x)$ can be obtained from $D(a,u) \oplus D(u,x)$, where u is any node in SCC. Thus a node in SCC can be designated as a reference node, the distance set from a to the rest of nodes in SCC can be derived from the reference node.

Theorem 5.4 *For nodes $u, x \in SCC$, $D(a,x)$ is asymptotically equivalent to $D(a,u) \oplus D(u,x)$.*

Proof: Since every path from a to x will involve at least a node in SCC, and based on lemma 5.1, $D(a,x) = \bigcup_{y \in SCC}(D(a,y) \oplus D(y,x))$. Since for any $y \in SCC$, $D(a,y) \oplus D(y,x) \Leftrightarrow D(a,u) \oplus D(u,x)$ (lemma 5.8), $\bigcup_{y \in SCC}(D(a,y) \oplus D(y,x)) = D(a,u) \oplus D(u,x)$. So $D(a,x) \Leftrightarrow D(a,u) \oplus D(u,x)$. □

Corollary 5.1 *If there is a cyclic path from a to an SCC, the distance OP for any node x in SCC is the addition of OP for the reference node and the internal offset of x.*

5.2.4 Derivation of distance set and difference set

Since SCCs are the source of cycles in digraph, a cyclic path can be studied by decomposing a digraph into DAG of SCCs, deriving period for each SCC and extracting interconnections among SCCs. A digraph may consist of a set of SCCs connected by directed edges. Connection among SCCs can be either serial or parallel as shown in Figure 5.5 and Figure 5.6. Serial connection occurs when an SCC has only one precedent SCC, while parallel connection occurs when an SCC has multiple precedent SCCs. In the context of OP derivation, two SCCs joining at a node followed by an SCC is equivalent to two SCCs joined at the third SCC.

The following theorems state how to derive distance set.

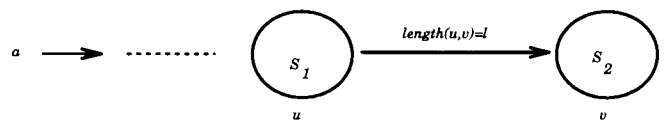


Figure 5.5: Serial merge.

Theorem 5.5 *For SCC S_1, S_2 connected in serial with u, v as their reference nodes as in Figure 5.5, $D(a,v)$ is asymptotically equivalent to $D(a,u) \oplus D(u,v)$.*

Proof: First it is assumed that there is only one acyclic path connecting S_1 and S_2, node $x \in S_1$ is the its start node. Then all the paths from a to v pass x. So $D(a,v) = D(a,x) \oplus D(x,v)$ and $D(u,v) = D(u,x) \oplus D(x,v)$. From above theorem, $D(a,x) \Leftrightarrow D(a,u) \oplus D(u,x)$. Hence $D(a,v) \Leftrightarrow (D(a,u) \oplus D(u,x)) \oplus D(x,v) = D(a,u) \oplus (D(u,x) \oplus D(x,v)) = D(a,u_1) \oplus D(u,v)$.

Now consider there are multiple path connecting S_1 and S_2. Let $x_1, \ldots x_k$ be their starting nodes. $D(a,v) = \bigcup_{i=1}^{k}(D(a,x_i) \oplus D(x_i,v))$, and $D(u,v) = \bigcup_{i=1}^{k}(D(u,x_i) \oplus D(x_i,v))$. Since $D(a,x_i) \Leftrightarrow D(a,u) \oplus D(u,x_i)$, then $D(a,v) \Leftrightarrow \bigcup_{i=1}^{k}(D(a,u) \oplus D(u,x_i) \oplus D(x_i,v)) = D(a,u) \oplus (\bigcup_{i=1}^{k}(D(u,x_i) \oplus D(x_i,v)))$. So, $D(a,v) \Leftrightarrow D(a,u) \oplus D(u,v)$. □

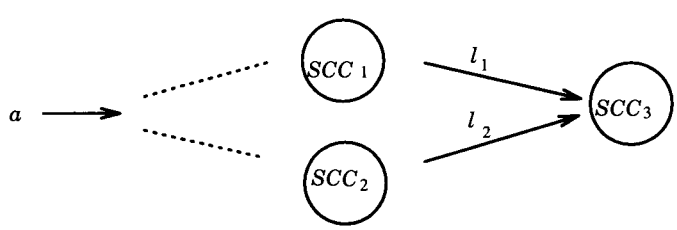


Figure 5.6: Parallel merge.

Theorem 5.6 *If SCC_1, SCC_2, SCC_3 are connected in parallel as shown in Figure 5.6 with u, v, w as their reference nodes, $D(a,w)$ is asymptotically equivalent to $(D(a,u) \oplus D(u,w)) \cup (D(a,v) \oplus D(v,w))$.*

Proof: Paths from a to w consist of two parts: paths via SCC_1 and paths via SCC_2. Since these two parts do not mix up, $D(a,w)$ is the union of the path lengths of the

two. From the above theorem, the length of the path via SCC_1 is asymptotically equivalent to $D(a,u) \oplus D(u,w)$, and that via SCC_2 is asymptotically equivalent to $D(a,v) \oplus D(v,w)$. Hence $D(a,w) \Leftrightarrow (D(a,u) \oplus D(u,w)) \cup (D(a,v) \oplus D(v,w))$. □

Based on the above theorems, when SCCs are connected in serial, the distance OP for S_2 is the addition of distance OP for S_1 and $S_1 \rightarrow S_2$. It is called the *serial merge* of OP's (of SCCs). When SCCs are connected in parallel, the distance OP for SCC_3 is the union of two additions: SCC_1 and $SCC_1 \rightarrow SCC_3$, SCC_2 and $SCC_2 \rightarrow SCC_3$. It is called the *parallel merge* of OP's (of SCCs).

We now discuss how to derive difference set in G_{down}. Suppose in G_{up} $D(a,x)$ is known, and in G_{down} $D(y,z)$ is known and $(x,y) \in flat$. Then z is an answer to the query if $Diff(a,x) = D(y,z) \ominus D(a,x)$ contains 0. Although $Diff(a,z)$ is an infinite set, it is possible to use offset-period pair to represent it. All the following derivation of difference set follows the principle $Diff(a,z) = \bigcup_{w \in G_{down}} (Diff(a,w) \oplus D(w,z))$.

Based on the above analysis, subtraction of OP's can be defined accordingly.

Definition 5.13 *$OP[C_1,p_1] \ominus OP[C_2,p_2]$ is defined as $OP[(C_1 \ominus C_2) \bmod p, p]$, where $p = gcd(p_1,p_2)$.*

The following theorem transforms the problem of distance set comparison into that of 0 containment test of difference set. It guarantees the correctness of OP representation for the difference set in terms of 0 containment test. It also guarantees the correctness of derivation method of difference set.

Theorem 5.7 *If $D(a,x) \Leftrightarrow OP[C_1,p_1]$ and $D(y,z) \Leftrightarrow OP[C_2,p_2]$, then $0 \in D(y,z) \ominus D(a,x)$ is equivalent to $0 \in OP[C_2,p_2] \ominus OP[C_1,p_1]$.*

Proof: Assume $N = max(\text{the stable levels of } D(a,x), D(y,z))$.

If $0 \in D(y,z) \ominus D(a,x)$, then from lemma 5.3 and its proof, there exists $d \geq N$ and $d \in D(y,z), d \in D(a,x)$. So $d \in OP[C_1,p_1]$ and $d \in OP[C_2,p_2]$, and d can be written as $c_1 + k_1 \times p_1$ and $c_2 + k_2 \times p_2$ for $c_1 \in C_1, c_2 \in C_2$ and $k_1, k_2 \in \mathcal{N}_0$.

So, $c_2 - c_1 = k_1 \times p_1 - k_2 \times p_2$, and $(c_2 - c_1) \bmod gcd(p_1, p_2) = 0$. Hence, $0 \in (C_2 \ominus C_1) \bmod gcd(p_1, p_2)$, and $0 \in OP[C_2, p_2] \ominus OP[C_1, p_1]$.

If $0 \in OP[C_2, p_2] \ominus OP[C_1, p_1]$, there exist $c_1 \in C_1, c_2 \in C_2$ such that $(c_2 - c_1) \bmod gcd(p_1, p_2) = 0$. $c_2 - c_1 = k \times gcd(p_1, p_2)$ for integer k. Assume $p_1 = p_1' \times gcd(p_1, p_2), p_2 = p_2' \times gcd(p_1, p_2)$, then $gcd(p_1', p_2') = 1$. From lemma 5.5, there exists N_0 such when $N \geq N_0$ $N = \lambda_1 \times p_1' + \lambda_2 \times p_2'$, and $N + 1 = \lambda_3 \times p_1' + \lambda_4 \times p_2'$. $1 = (\lambda_3 - \lambda_1) \times p_1' + (\lambda_4 - \lambda_2) \times p_2'$. It can be denoted as $1 = \beta \times p_2' - \alpha \times p_1'$ for $\alpha, \beta \in \mathcal{N}_0$ without loss of generality. Multiplying $gcd(p_1, p_2)$ at both sides of the equation, $gcd(p_1, p_2) = \beta \times p_2 - \alpha \times p_1$. Hence $c_2 - c_1 = k \times \beta \times p_2 - k \times \alpha \times p_1$. So, $c_2 + k_2 \times p_2 = c_1 + k_1 \times p_1$. Algebraic manipulation like multiplying $\beta \times p_2'$ with $1 = (\beta \times p_2' - \alpha \times p_1')$ gives $(\beta^2 \times p_2)p_2 - (\alpha \times \beta \times p_2')p_1' = 1$, that means a new set of coefficients is found which are larger. So, there exist $k_1, k_2 \in \mathcal{N}_0$, such that $c_1 + k_1 \times p_1 = c_2 + k_2 \times p_2 > N$. Hence, $c_1 + k_1 \times p_1 \in D(a, x)$ and $c_2 + k_1 \times p_2 \in D(y, z)$. Therefore, $0 \in D(y, z) \ominus D(a, x)$. □

As a summary for this section: it has been discussed how to derive R_{cyc}. First, distance sets are replaced by its partial periodic subset. Second, periodic partial distance set is captured by the offset-period representation. Third, derivation of distance sets is transformed into that of OP sets. Fourth, for the purpose of 0 containment test, difference set is managed in the same way as distance set.

The derivation of R_{cyc} is as follows. In *up* relation, OP's for the distance set (called distance OP) are derived. The *flat* relation passes those distance OP to the *down* relation, where OP's for the difference set (called difference OP) are initialized. Difference OP's in *down* are derived in the same way as the distance OP are in *up*. A query answer is obtained if the difference set contains 0.

5.3 Compressed Counting Method

The compressed counting method computes both R_{cyc} and R_{acyc}. The former is studied in this section; whereas the latter in the next one. In this section, we discuss (1)

precompilation of *up* and *down* digraphs, and (2) query processing for the derivation of R_{cyc}, which consists of (i) the derivation of distance-OP in *up*, (ii) the derivation of difference-OP in *down* and (iii) answer extraction.

5.3.1 Precompilation of Data Relations

Since a database digraph is composed of a set of SCCs and their connections, precompilation of a database digraph before query processing will avoid rederivation of the SCC information when multiple queries are posed to the same data relation or the same SCC is used many times during query processing.

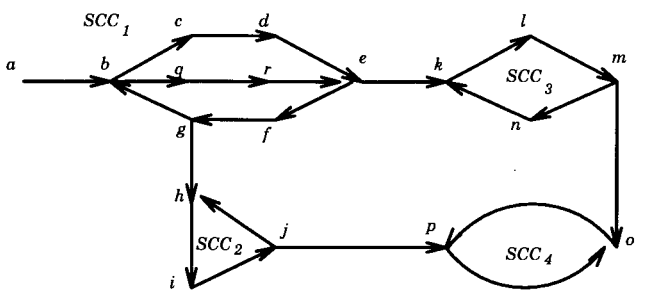
Precompilation of a digraph (*up* or *down* relation) consists of three steps: (1) partition the digraph into a set of SCCs, (2) derive the internal OP's for the nodes in each SCC, and (3) construct a compressed graph by extraction of inter-SCC connections.

Studies in [49, 41, 129] contribute to a simple algorithm that derives SCCs and their periods in a digraph G in O(e) time, where e is the number of edges in G. At the same time, the nodes in an SCC of period p can be partitioned into p equivalence classes. Let one class be the *reference class* (labeled 0) in which every node is a *reference node* with internal offset 0. Other classes are labeled from 1 to $(p-1)$ according to their *distance* (shortest traversal length) from the reference class. Every node in class j has internal offset j. A node in class j of SCC_k with period p is registered as $(k, [j, p])$, where k is the SCC-id, j the internal offset, p the period, and $[j, p]$ is the internal OP of the node.

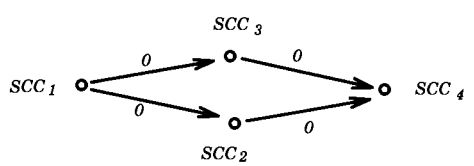
The compressed graph (CG, or CG_{up}/CG_{down} for the up/down relation) is a directed acyclic graph (DAG) in which a node represents an SCC, and an edge between two nodes (called an *inter-SCC edge*) represents the inter-SCC connection(s) between two connected SCCs. An inter-SCC edge from SCC_1 to SCC_2, denoted as $SCC_1 \rightarrow SCC_2$, is labeled by the offset of the distance OP for $SCC_1 \rightarrow SCC_2$, which is an integer or a set of integers, obtained by a set of path lengths $l_1, l_2, \ldots, l_k$ moduled by $gcd(p_1, p_2)$, where p_1 and p_2 are periods of SCC_1 and SCC_2, and l_i (for $1 \leq i \leq k$)

is the length of the i-th inter-SCC connection path, calculated by the length between two SCC reference nodes in the digraph.

Example 5.2 Figure 5.7 shows how the precompilation is done on a small digraph. The graph is partitioned into four SCCs. The period of each SCC and the internal OP of the nodes in each SCC are computed. Finally, the graph is compressed into a compressed graph CG with four nodes and four labeled inter-SCC edges. □



Data Digraph



Compressed Digraph

SCC period	SCC 1 =6								SCC 2 =3			SCC 3 =4				SCC 4 =2	
data node	b	c	d	e	f	g	q	r	h	i	j	k	l	m	n	o	p
internal offset	0	1	2	3	4	5	1	2	0	1	2	0	1	2	3	1	0

Internal Offset of Data Nodes in SCCs

Figure 5.7: Precompilation of data relation.

A digraph of a data relation usually contains only a small number of SCCs. Thus, a compressed graph (CG) is a small DAG. The mapping of a large data digraph into a tiny CG which contains rich information about SCCs and their interconnections will save the recomputation of periods and internal offsets of the nodes in each SCC and guide efficient query evaluation. This will be discussed in the query processing phase.

5.3.2 *up* processing

When a query is posed to the system, the *up* relation processing starts, which derives the distance-OP for every data node reachable from the query constant(s).

If query constant a is not in any SCC, the distance-OP of a is initialized to [0,0]. A breadth-first search (join operation) starting from a is performed in G_{up} to find out all the SCCs reachable from query constant a via acyclic path(s) only. The offset l keeps increasing along the acyclic path with the OP of the derived node assigned to be $[l, 0]$ until it reaches an SCC. When the derivation reaches an SCC S at the node n with internal OP $= [c, p]$, the distance-OP of S (which is the distance OP of S's reference node) is initialized to $Norm([l - c, p])$, where l is the current offset. If there are multiple paths from a to the same SCC, the distance-OP for the SCC is the union of the distance-OP derived from each path. If query constant a is in an SCC, the SCC is initialized in the same way as above except the offset l is 0 and the node n is node a itself.

As proved in the previous section, OP representing distance set for an SCC in CG_{up} is derived by adding and unioning distance OP's of its preceding SCCs. So once an SCC is instantiated, all the SCCs in CG_{up} reachable from it can be instantiated immediately using the compressed graph CG_{up} following the topological order.

In a digraph, the distance-OP of every node in each reachable SCC is inferred from distance-OP of the SCC (which is the distance-OP for the reference node of the SCC) and the internal offset of the node. For node $n \in$ SCC S_i with $OP[C_i, p_i]$ and internal offset of m, the distance-OP for n is $NORM(OP[C_i, p_i] \oplus m)$.

Example 5.3 Suppose *up* relation is shown in Figure 5.7. Since $length(a \rightarrow b) = 1$, and the internal OP of the reference node b is [0, 6] (0 is the internal offset, 6 is the period of SCC_1), the distance-OP for SCC_1 is initialized to [1, 6]. OP propagation in the compressed graph proceeds as follows. Since $label(SCC_1 \rightarrow SCC_2) = 0$, and SCC_1 is the only predecessor of SCC_2, OP of SCC_2 can be derived by adding OP of SCC_1 with $SCC_1 \rightarrow SCC_2$, that is, $NORM([1, 6] \oplus [0, gcd(6, 3)]) = [1, 3]$. Similarly,

the OP for SCC_3 is [1, 2]. Since SCC_4 is connected with other SCCs in parallel, the OP for SCC_4 is obtained by adding OP's for SCC_3, $SCC_3 \rightarrow SCC_4$, adding OP's for SCC_2, $SCC_2 \rightarrow SCC_4$ and unioning the two addition results, which is $NORM(([1,2] \oplus [0, gcd(4,2)]) \cup ([1,3] \oplus [0, gcd(3,2)]))$, or [0, 1]. Once the OP for each SCC is known, OP's for data nodes are inferred easily. The result is shown in Figure 5.8. □

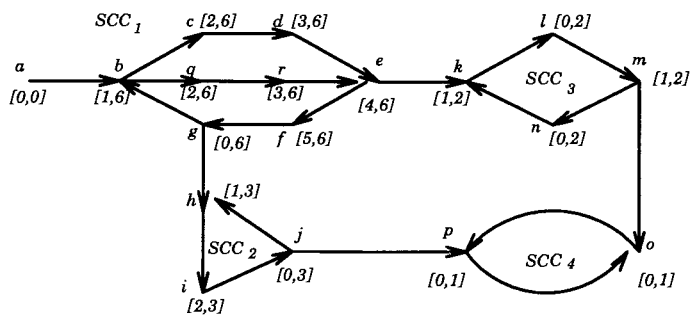


Figure 5.8: OP for data nodes.

Notice that distance-OP's should be derived for non-SCC nodes along an inter-SCC path. The nodes following an SCC carries the same merged period of the SCC, and the offset set carried is incremented by one for each traversal along the path. A parallel merge is performed when the paths following two or more SCCs merge at a non-SCC node. Such distance-OP propagation proceeds until it reaches a node on another SCC.

An implementation issue concerning OP representation should be mentioned here. It has been assumed that all OP is represented in $[C, p]$ pair so far, which can be referred to as explicit OP representation. However, it is not necessary to represent OP explicitly in OP derivation. An implicit OP representation is available which represents the same information with much less computation and storage overhead. For SCC node, the pair of SCC-id and internal offset is the implicit OP representation, since the explicit OP representation can be inferred from it. Let's assume that for non-SCC node, the SCC that derives it is its driver SCC. Then for non-SCC node the pair of its driver SCC-id and the distance from the driver SCC reference node is

the implicit OP representation. The explicit OP representation is only required when necessary, i.e., when passing of distance OP to *down* relation, OP initialization and the answer extraction. An SCC OP buffer associating SCC-id with SCC OP is needed to transform the implicit OP representation into the explicit one.

5.3.3 *down* processing and answer extraction

The nodes reachable from the query constant a in the *up* digraph are joined with the *flat* relation, which results in a set of instantiated nodes for *down* processing. The *distance-OP* of participating node(s) in G_{up} is passed as an *inherited OP* to the corresponding instantiated node(s) of G_{down}.

Derivation of the difference-OP in G_{down} is very similar to the distance-OP derivation in G_{up}, with some difference in the OP initialization.

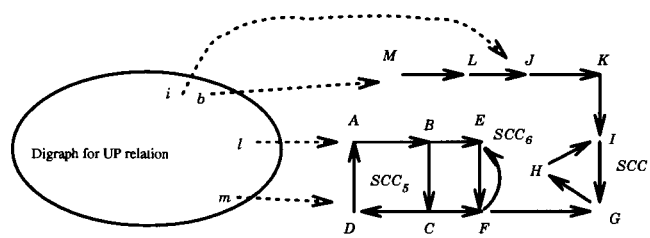
The initialization for difference-OP in G_{down} starts from the instantiated nodes. Suppose an instantiated node has an inherited OP of $[C, p_1]$. If it is on an SCC and has an internal offset of c and the period of p_2, the initial difference-OP for the SCC is $NORM([p_2 - c, p_2] \ominus [C, p_1])$. If it is not on any SCC but has a path of length l to node n on SCC S with period p_2, and the internal offset for n is c, then the difference OP for S is initialized to $NORM([l + p_2 - c, p_2] \ominus [C, p_1])$. If an SCC has more than one path from one or more instantiated nodes, its initial difference-OP is the union of the individually initialized difference-OP's.

Once the initialization of difference-OP has finished, the derivation of difference-OP for the remaining SCCs and the nodes in the SCCs in *down* relation resembles the OP derivation in *up* relation.

Finally, for answer extraction, it is easy to assert that R_{cyc} is the set of data nodes in G_{down} whose difference-OP contains 0, that is, the offset set of the difference-OP contains 0.

5.3.4 An example

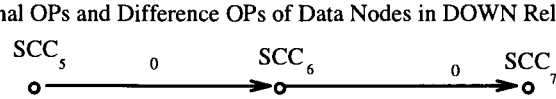
Example 5.4 Figure 5.9 is an example showing how compressed counting method derives R_{cyc}, where the *up* relation is shown in Figure 5.8, *flat* relation is shown here with dash lines and data nodes in *down* digraph are denoted with upper case letters. Since distance-OP's for node l and m are [0, 2] and [1, 2], difference-OP for SCC_5 is initialized to $[0,4] \ominus [0,2] \cup [3,4] \ominus [1,2]$, which is [0, 2]. Since the inherited distance-OP for node J and M are [2, 3] and [1, 6] respectively, the difference OP for SCC_7 is initialized to $([2, 3] \oplus 2 \ominus [2, 3]) \cup ([2, 3] \oplus 4 \ominus [1, 6])$, which is [2, 3]. Consider the topological order in compressed graph, OP for SCC_6 is [0, 2] and OP for SCC_7 is [0, 2] $\cup$ [1, 3], which is [{0,1,2,4}, 6] in its normal form. The OP for each data node is shown in the table. The answer nodes for cyclic portion are A, C, E, G, I since their difference-OP contain 0. □



Data Digraph

SCC period	SCC_5 =4				SCC_6 =2		SCC_7 =3		
data node	A	B	C	D	E	F	G	H	I
internal offset	0	1	2	3	0	1	0	1	2
difference OP	[0,2]	[1,2]	[0,2]	[1,2]	[0,2]	[1,2]	[{0,1,2,4},6]	[{1,2,3,5},6]	[{0,2,3,4},6]

Internal OPs and Difference OPs of Data Nodes in DOWN Relation



Compressed Digraph for DOWN Relation

Figure 5.9: Example of compressed counting method.

5.4 Complementary counting: optimizations

5.4.1 Dealing with acyclic paths

In principle, the derivation of R_{acyc} where the derivation path in either G_{up} or G_{down} is acyclic, can be performed by simply applying the acyclic Counting method [9] to count the acyclic path length starting from a in G_{up} and from instantiated nodes in G_{down}. The counting method terminates when the counting level reaches the longest acyclic path length.

However, a complete separation of the derivations of R_{cyc} and R_{acyc} may not lead to good performance because it is easy to observe that much of counting in both G_{up} and G_{down} are performed separately but redundantly in the two derivations. Therefore, it is desirable to devise an integrated approach in the processing of both cyclic and acyclic data. As an integral part of compressed counting, we propose a *complementary counting* technique for the derivation of R_{acyc}.

Assume path $a \rightarrow x$ in G_{up}, edge $(x, y) \in flat$, and path $y \rightarrow z$ in G_{down}. Let l_{bc} denote the length of path $b \rightarrow c$ in G_{up} (or G_{down}). There are three cases to be considered: (1) $a \rightarrow x$ is acyclic, (2) both $a \rightarrow x$ and $y \rightarrow z$ are cyclic, and (3) $a \rightarrow x$ is cyclic but $y \rightarrow z$ is acyclic.

The second case is the derivation of R_{cyc}, where z is an answer if $0 \in$ difference-OP of z, which has been studied in the last section.

In the first case, the counting level of node x is registered in *up* and passed to y in *down*. The counting level decreases as the paths from y are traversed in *down* until the level reaches 0 or the paths terminate, no matter whether the paths are cyclic or acyclic. Node z is an answer if it has counting level of 0.

A node x in G_{up} with an acyclic path from a has the counting level for x, l_{ax}, derived during distance-OP initialization, so do nodes with acyclic path from an instantiated node y in G_{down} during difference-OP initialization. The acyclic length

(counting level) is represented by the same offset-period notation, where an acyclic node has 0 as the period and the actual length as the offset. This kind of OP can be referred to as acyclic OP in contrast to cyclic OP as discussed before.

This implies that the computation of R_{cyc} can be minorly modified to incorporate the computation in the first case. The modification is presented as follows. In difference-OP initialization in G_{down}, if an instantiated node y has an acyclic inherited OP of $[c, 0]$, the OP of node y becomes $0 \ominus [c, 0]$, i.e., $[-c, 0]$. The path(s) (both cyclic and acyclic) starting from y is traversed in G_{down}, and the counting level (offest) is increased by one and passed to the next node along the path at each step. This process proceeds until either the path terminates or the counting level reaches 0, and at this point the corresponding data node is registered in the answer set.

In the third case, since the compressed counting derives only distance-OP for x instead of the lengths from a to x, a traversal from a to x is necessary to obtain all the counting levels of x up to level of l_{yz} to facilitate the length matching of l_{ax} and l_{yz}. Such a process can be optimized by a complementary counting technique presented below.

5.4.2 Complementary counting optimization

To facilitate the improved processing of acyclic data, two extra pieces of information need to be stored in the *up* processing during the R_{cyc} computation. The first is *flipping node*s, where a flipping node is a node in an SCC reached via acyclic path(s) from a query constant, i.e., a node whose internal period is nonzero but is derived by a zero-period driver in the processing, *e.g.* node x in Figure 5.10. Suppose x_1 is a flipping node reached from a query constant a at counting level l_1. Then the flipping node (a, x_1, l_1) is stored in a *flipping node buffer* during the R_{cyc} computation. Flipping nodes are the starting points when a retraversal of *up* relation becomes necessary. The second is *driver SCC*(s) which should be registered for every data node not in any SCC but derived by some SCC in the computation. If a non-SCC node x is derived by an SCC S via an acyclic path, S is a driver-SCC of node x. Driver-SCC information

is used to prune the irrelevant SCCs in the retraversal of the *up* relation.

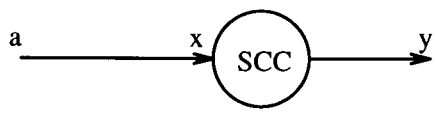


Figure 5.10: A flipping node is shown.

For the same purpose, *candidate nodes* need to be stored in the down processing during R_{cyc} computation, where a candidate node is a node reached via acyclic path(s) only from an instantiated node in *down*, and $0 \in ((l \ominus C) \bmod p)$, where $[C, p]$ is the inherited-OP and l is the acyclic path length from an instantiated node to the current node. Suppose z_1 is a candidate node reached from an instantiated node y_1 (driven from the query constant a) at counting level m_1. The candidate node (a, y_1, z_1, m_1) is stored in the candidate node buffer. Candidate nodes are those in the *down* relation that might be the answer to the query. Notice that $0 \in ((l \ominus C) \bmod p)$ is a necessary condition for the node to be an answer.

After the computation of R_{cyc}, and the extraction of the answers for those traversing via acyclic paths only in the *up* processing (Case 1), a complementary counting process starts to derive those answers derivable by traversing cyclic path(s) in *up* and acyclic path(s) in *down*.

Complementary counting strives for efficiency in four aspects: (1) traversal in *up* restarts at the flipping nodes rather than at the query constants; (2) search only the subgraphs of the compressed graph CG_{up} which may contribute to the derivation of answers; (3) the search terminates immediately when the counting level reaches the *corresponding* maximum acyclic counting level in *down* (since further processing will be fruitless to match levels in *down*); and (4) answers are extracted directly from the candidate node buffer rather than retraversal of the *down* graph (that is, there is no acyclic path in *down* to be retraversed in complementary counting).

We analyze how to realize these four aspects. Aspect 1 is realized easily by starting the complementary counting at the flipping nodes. The stored flipping node information (a, x, l) indicates that there is an acyclic path $a \rightarrow x$ which reaches the node x

at the counting level l.

The key to Aspect 2 is to know via which set of SCCs, a flipping node $(a, x_1, \_)$ may derive an instantiated node y_1 in *down* for a candidate node $(a, y_1, \_, \_)$. Since the SCC-id's of x_1 and x_1' (where $(x_1', y_1) \in flat$) are known in *up* processing, the relevant SCCs can be obtained by analysis of the compressed graph CG_{up}. Only those SCCs in CG_{up} which are along the path from x_1 to x_1' are traversed.

Realization of Aspect 3 requires to find the *corresponding* maximum acyclic counting level in *down*. A simple solution is to take the maximum counting level m for query constant a found among the candidate nodes driven by a, that is, $m = maximum(m_1, \ldots, m_i)$ for all the candidate nodes in the form of $(a, \_, \_, m_i)$. An improved solution is to associate with the subgraph (found in Aspect 2) only those corresponding maximum counting level which are derivable from the subgraph. Such maximum counting level can be easily found from the candidate nodes in the form of $(a, y, \_, m_i)$, where y is in the set of instantiated nodes derivable by the subgraph.

Aspect 4, the direct extract of answer from the candidate node buffer, can be easily implemented since a node z with a cyclic path in *up* and an acyclic one in *down* is in the answer set if and only if there exists (i) a node x derived from a with counting level $= l$ in *up*, (ii) $(x, y) \in flat$, and (iii) (a, y, z, l) in the candidate node buffer.

Thus, we have the following algorithm for complementary counting: (1) for each query constant a and its instantiated node y, find the maximum (down) counting level $maxlevel(a, y)$ from among all the candidate nodes in the form of (a, y, z, m); (2) the maximum (up) counting level $maxlevel(a, x') = maxlevel(a, y)$ if $(x', y) \in flat$; (3) starting at S', the SCC associated with x', or the driver SCC of x' if x' is not on any SCC, propagate backward to each SCC in CG_{up} three pieces of information: (i) query constant a, (ii) $maxlevel(a, x')$, and (iii) a set of SCC-id's along the path; (4) starting at each SCC containing the flipping nodes of a, traverse G_{up} forward and increment the counting level until it reaches $maxlevel(a, x')$, and notice that a new SCC will be explored only if a is associated with the SCC and the SCC-id is in the associated SCC-id set, and (5) perform join of (a, x', l) with $(x', y) \in flat$, and with (a, y, z, l)

in the candidate node buffer, the result set of z's is the set of answers derived by complementary counting.

5.5 Discussion

5.5.1 Complexity Analysis

The compressed counting method has the worst case time complexity of $O(ne)$ where n and e are the number of nodes and the number of edges respectively in the database digraph. This is analyzed for both precompilation and query processing.

Precompilation consists of the following four steps : (1) partition a digraph into a set of SCCs, (2) compute the period of each SCC, (3) partition nodes in each SCC into equivalence classes, and (4) map the digraph into a compressed graph CG. According to the studies in [49, 41], steps 1 to 3 takes $O(e)$ time. The worst case time complexity for graph compression (step 4) is $O(ne)$ because it takes at most n joins to derive all the inter-SCC connections. Such a complexity makes precompilation feasible for a relatively large database, even considering data updates. Moreover, partial recompilation can be explored if the updated data influences only one or a small number of SCCs.

The cost for storage of precompiled results is minimal. Each node in an SCC needs to store only two integers: an SCC-id and an internal offset. Since the number of SCCs is much less than the number of data nodes in a digraph, the cost for storage of the compressed graph is negligible in most databases.

Theorem 5.8 *The time complexity for query processing in compressed counting method is $O(ne)$, where n, e are the number of nodes and edges in digraph G_{up} and G_{down}.*

Proof. For query processing, initialization of OP and traversal of inter-SCC paths can be done with each edge in acyclic paths being traversed once, it takes at most

$O(ne)$ time. The computation of R_{acyc} is bounded by the worst case time of acyclic counting, which takes at most $O(l_{acyc}e)$ time where l_{acyc} is the length of the longest acyclic path in *up* and *down*, so it is bounded by $O(ne)$.

The following paragraph is a proof of time complexity for OP derivation. The OP derivation in query processing consists of two parts: OP derivation for SCC nodes and OP derivation for non-SCC nodes.

OP derivation for SCC nodes accesses each data node in SCC only once, hence the I/O time is $O(n)$. OP merging is by arithmetic computation which takes only CPU time. If SCC_1, SCC_2 with periods of p_1, p_2 are connected in serial, the result OP has the period of $gcd(p_1, p_2) \leq min(n_1, n_2)$, where n_i is the number of nodes in SCC_i. Since it takes $O(p_1 p_2)$ for serial merge of SCC_1 and SCC_2, it is bounded by $O(n_1 n_2)$. If SCC_1, SCC_2 and SCC_3 with periods of p_1, p_2, p_3 are connected in parallel as shown in Figure 5.6, then the result OP has period of $lcm(gcd(p_1, p_3), gcd(p_2, p3))$, which is equivalent to $gcd(lcm(p_1, p_2), p_3)$, hence the result period is less than n_3. It takes $O(p_1 p_3 + p_2 p_3 + p_3)$ time for parallel merge, or it is bounded by $O(n_1 n_3 + n_2 n_3 + n_3)$. It can be stated that the period of an SCC OP (either distance-OP or difference-OP) is less than n_{SCC}. In other words, cost of an SCC by OP merging is bounded by the $n(n_1 + \ldots + n_k)$ where n is the number of nodes in the SCC, and $n_1, \ldots, n_k$ are the number of nodes in the preceding SCCs. Hence, the cost for OP derivation in compressed graph is $\Sigma_{SCC_i \rightarrow SCC_j \in CG}(n_i n_j)$. Inferring OP for a node on SCC_i takes $O(n_i)$, so the cost for inferring OP for all SCC nodes takes $\Sigma_i n_i^2$. The total cost of OP derivation for SCC nodes is $\Sigma_i n_i^2 + \Sigma_{SCC_i \rightarrow SCC_j \in CG}(n_i n_j)$, which is less than $(\Sigma_i n_i)^2$. n_i is the number data nodes in SCC_i, so $\Sigma_i n_i < n$, the cost of OP derivation for SCC nodes is $O(n^2)$.

Using the implicit representation of OP, the derivation of OP for non-SCC nodes resembles the counting level derivation in acyclic counting and takes $O(ne)$. The transformation from an implicit representation into the corresponding explicit one requires one or more $[C_i, p_i] \oplus l$ operation, which takes $O(n_i)$ each. Hence one transformation takes $O(n)$, and the transformation for all non-SCC nodes takes $O(n^2)$. So,

the cost of OP derivation for non-SCC nodes is $O(ne)$.

Test for 0 containment in difference OP takes $O(1)$ for each node in *down* assuming offset set in difference OP is sorted or $O(n_{SCC})$ if it is not sorted. So answer extraction takes at most $O(n^2)$.

Hence, the worst case time complexity for query processing is bounded by $O(ne)$. □

5.5.2 Extension to multiple source

So far, we have assumed that there is only one node in *up* instantiated from the query. Nevertheless, if there are multiple nodes in G_{up} instantiated, it is straightforward to extend the single source approach.

The precompilation phase remains the same for multiple source nodes. In the query processing phase, each OP carries a source origin designating which source node it is derived from. For any operations on OP such as addition, subtraction and merge, only two OP's with the same origin are allowed to be operated on. The query answer is extracted from G_{down} whose difference OP contains 0, hence the origin of its OP establishs which query instantiation derives such answer node.

If there are k nodes instantiated in G_{up}, the worst case time complexity for compressed counting is O(kne).

5.5.3 Strength of compressed counting

Compressed counting has absorbed the major ideas from the previous studies on cyclic counting. Separation of the computation of R_{cyc} and R_{acyc} is from Magic Counting [115] and Cyclic Counting [41]. SCC extraction and equivalence class derivation is

from [3, 49, 41]. Precompilation of database digraph is from [49]. However, compressed counting integrates the strength of these studies, develops a graph compression technique and an integrated cyclic and acyclic data processing technique and provides a simple and efficient solution to the problem.

In comparison with the previous studies on cyclic counting, the compressed counting has the following advantages.

First, precompilation of *up* and *down* relations saves the recomputation of periods and offsets for nodes and SCCs not only among different queries but also within the same query but with multiple SCC entry points. Most of the computation for OP merging and transformation are performed at the SCC level. Only offset adjustment are performed on individual nodes in an SCC, which requires minimum computation.

Secondly, the use of compressed graphs facilitates computation, propagation and merge of offset-period information in group mode. It also facilitates the analysis of data flow in the computation of both R_{cyc} and R_{acyc}.

Thirdly, the algorithm is highly parallel in nature. When one SCC is reached in the *up* or *down* processing, merge and propagation of OP's are performed in the small compressed graph. Then the computation of all the data nodes in relevant SCCs can be performed in parallel.

Fourthly, both acyclic and cyclic data are computed in one uniform algorithm, which reduces redundant processing in the derivation of R_{cyc} and R_{acyc}.

Some optimization techniques explored in other studies, such as the golden-cycle optimization [49], can be adopted naturally in the compressed counting as well.

Finally, it is worth mentioning that the counting technique is not confined to the evaluation of queries in "well-formed" (single) linear recursions as shown in Figure 5.1. It can be applied to the evaluation of more general classes of recursions, such as compiled general linear recursions [49] and certain classes of multiple linear recursions [51, 52].

5.6 Summary

The compressed counting method has been studied for the extension of acyclic counting method to general databases. The method integrates the merits of several proposed cyclic counting algorithms and provides a uniform handling of both acyclic and cyclic data in the processing of linear recursive queries. The method precompiles and compresses database digraphs into a small compressed graph which guides the efficient query evaluation and optimization. The method processes linear recursive queries in both cyclic and acyclic databases as efficiently as the counting method does in acyclic databases. Also, it facilitates parallel processing and the exploration of many optimization techniques.

Chapter 6

Discussions and Conclusions

The author's contribution to the problem of efficient recursive query processing in deductive databases has been discussed in the previous chapters, including the design and implementation aspects on *LogicBase*; how constraints and monotonicity are employed to ensure the safety of query evaluation and increase its efficiency; extensions to the chain-based evaluation to deal with multiple linear recursions and cyclic databases in the counting method. The application scope, strength and limitation and performance evaluation in comparison with other popular approaches are discussed in this chapter.

The analysis of the evaluation of different query instantiations for the *nqueens* recursion (in section 3.2.4) discloses an interesting fact: a logic program can be executed declaratively, independently of query modes and rule/predicate ordering. Moreover, it derives the complete set of answers and terminates properly. Obviously, this is quite different from the implementations of Prolog [126] which perform no systematic rule compilation and query analysis, and therefore, cannot judge termination, enforce sophisticated constraints, determine appropriate rule/predicate ordering, or derive efficient query evaluation plans. Recent studies on constraint logic programming, such as [140, 83], enforce more constraints than Prolog but still mainly confine the program evaluation ordering to those given by programmers.

The question is how far we can push this methodology towards declarative programming in general logic programs.

6.1 Applicable domains of the methodology

The chain-based query evaluation method is based on the compilation of each recursion in a program into a highly regular chain form on which the systematic query analysis can be performed to determine an efficient query evaluation plan and the termination of query evaluation. Based on the studies in [47, 58], linear and nested linear recursions can be compiled into highly regular chain forms.

Many complex recursions, though they cannot be compiled into highly regular chains, may still have interesting regularities among the variable connections in the recursive rules. For example, the recursion *tower_of_hanoi* shown in Example 6.1 with the head predicate "$hanoi(N, A, B, C, Moves)$" is a typical nonlinear recursion which cannot be compiled into highly regular chain forms. However, because of the regularity of its binding passing across two recursive subgoals in the recursive rule, the expansions of the recursive rule still demonstrate certain chain-like regularity and the portion in front of or behind each recursive subgoal in subsequent expansions can be treated as a pseudo-chain in the query analysis. Thus, the chain-based query evaluation method can still be applied to such recursions, and queries such as "? – $hanoi(3, a, b, c, Moves)$" or "? – $hanoi(N, a, b, c, [a\ to\ b,\ a\ to\ c,\ b\ to\ c,\ a\ to\ b,\ c\ to\ a,\ c\ to\ b,\ a\ to\ b])$" can still be analyzed systematically and evaluated efficiently [50].

Example 6.1 The recursion *hanoi*, defined by Rules (6.1) and (6.2), is a functional nonlinear recursion. It defines the *Towers of Hanoi* puzzle [126], that is, moving N discs from peg A to peg B using peg C as an intermediary.

$$hanoi(1, A, B, C, [A\ to\ B]). \tag{6.1}$$

$$hanoi(succ(N), A, B, C, Moves) :-$$
$$\quad hanoi(N, A, C, B, Ms_1),\ hanoi(N, C, B, A, Ms_2),$$

$$append(Ms_1, [A\ to\ B|Ms_2], Moves). \tag{6.2}$$

□

However, this does not imply that chain-based evaluation can be applied effectively to all kinds of recursions. This is because some recursions may not have regular variable passing patterns and cannot be compiled into chain or even pseudo-chain forms. For example, the nonlinear recursion r, defined by Rules (6.3) and (6.4), belongs to this class.

$$r(X, X_1, Y) :- a(X, Y),$$
$$\quad r(X_1, X_2, Y_1),\ r(X_2, X_3, Y_2),\ b(X_3, Y_1, Y_2). \tag{6.3}$$
$$r(X, X_1, Y) :- e(X, X_1, Y). \tag{6.4}$$

Therefore, a major limitation of the chain-based evaluation method is its limited applicability to complex classes of irregular recursions.

Nevertheless, according to our survey and experience, most practically interesting recursions are in relatively simple forms or are compilable to highly regular forms to which the query analysis and evaluation techniques studied here are applicable. Theoretically, the recursions solvable by our method cover only a subset of all the possible recursions. However, it is difficult to find semantically meaningful recursions to which the method cannot be applied. One possible explanation of this fact could be based on the simplicity and regularity of human reasoning processes which guides the writing of recursive programs. It seems that a recursive program with no obvious expansion regularities is difficult for human to comprehend. There should exist certain regularities (such as connections between the corresponding argument positions in the head predicate and the same recursive predicate in the body) in a meaningful recursive program, and such regularity should be characterizable by their semantic linkages (i.e., either by shared variables or by connected predicates). That is the intuition behind our claim that the techniques discussed here are applicable to a large class of interesting recursions. This is also the basis of our design of *LogicBase*, which takes chain-based evaluation as a major evaluation technique and leaves a more general technique, such

as the generalized magic sets method, as an assistant one to be applied only when chain-based evaluation cannot derive efficient query evaluation plans.

6.2 A comparison with other logic program implementation techniques

6.2.1 Comparison

To the best of our knowledge, current implementations of logic programming languages cannot evaluate logic programs independently of the order of rules or predicates or query mode to find the complete set of answers and terminate properly.

Recent studies on deductive databases have developed bottom-up query evaluation methods, such as the magic sets method and its variations, for efficient evaluation of recursions [28, 105, 135, 133]. These methods apply set-oriented processing, confine their search to the portion of the database relevant to a query, and evaluate order-independent and query mode-independent *function-free* logic programs completely and correctly. Several deductive database system prototypes, such as LDL [28], ADITI [135], NAIL! [133], CORAL [105], are constructed based on this approach. However, without normalizing recursions and performing a detailed analysis of the behavior of compiled recursions, it is difficult to fully explore query constraints and behavior properties of a particular recursion or a particular query in the evaluation, which may encounter difficulties when evaluating sophisticated function-bearing logic programs. For example, the magic sets method cannot evaluate the *nqueens* recursion in predicate order-independent and query-mode independent fashion [105]. Similar comments can be applied to the EKS-V1 system [142] which adopts the query-subquery evaluation approach.

Prolog represents an effort toward declarative computing from the logic programming community. However, it does not have the ability to deal with a large amount

of data, or the declarativeness found in *LogicBase*.

The limitation of the chain-based query evaluation technique is in that the compilation method is confined to the recursions that can be compiled into highly regular forms [58]. In contrast, the magic sets method is applicable to general function-free recursions. However, the chain-based method facilitates quantitative analysis of compiled recursions and, therefore, can reduce search space more accurately than the magic sets method. Thus, it represents an interesting direction towards sophisticated query analysis and evaluation of complex, declarative logic programs.

6.2.2 Comparison of evaluation costs

6.2.2.1 Cost model

We briefly compare the cost to evaluate queries using the top-down (Prolog), the magic sets and the chain-based evaluation approaches. Prolog adopts depth-first state space search with backtracking strategy, thus it processes one tuple at a time. To derive the complete set of answers, Prolog has to traverse essentially the whole state space. A top-down set-oriented processing is taken as the method to be compared instead of Prolog itself, which should traverse the same number of states for all solutions but with less overhead of accessing EDB or relations for intermediate results. From our observation, the cost for compilation of a recursive program is negligible comparing that of query processing. Therefore, only the query processing cost is considered.

The same cost model as in Chapter 2 is used to do performance comparison for various query evaluation strategies independent of system implementation. Cost for each evaluation is based on three aspects: cost for evaluating arithmetic and "*cons*" functions (type a); cost for accessing EDB relations (type b); and cost for processing IDB rules (type c). Of these three types of costs, accessing an EDB relation is the most expensive, and cost for evaluating a function is the least expensive. For example, calculating "$X + Y$" only needs a few machine instructions for each X, Y tuple whereas a selection in a relation needs thousands of machine instructions, which can

be translated into a few dozens of machine instructions for each query instance. To provide a reasonable and measurable basis for performance comparison, it is assumed that on average each arithmetic function incurs one unit of cost; each "*cons*" function incurs 3 units of cost; accessing EDB relations incurs 10 units of cost for each tuple and processing IDB rules incurs 5 units of cost for each tuple.

The magic sets and the top-down evaluation method are implemented under the same environment as where *LogicBase* is implemented. For query evaluation using the magic sets method, the complete set of rewritten rules (including rules generating the magic predicates) is fed into the *LogicBase* system and the semi-naive evaluation is applied to the magic rules to derive the magic predicates, and then to rewritten rules to obtain answers to the original query. For example, Figure 6.1 shows the set of rewritten rules for the *ancestor* program, which is first defined in Figure 1.1 in Chapter 1 using Prolog syntax. In Figure 6.1, *m_ancestor* is the magic predicate, and *edb_parent* is an EDB predicate.

$$
\begin{aligned}
& m\_ancestor(john). \\
& \quad m\_ancestor(Z) \ :- \ m\_ancestor(X),\ edb\_parent(X, Z). \\
& \quad ancestor(X, Y) \ :- \ m\_ancestor(X),\ edb\_parent(X, Y). \\
& \quad ancestor(X, Y) \ :- \ ancestor(Z, Y),\ m\_ancestor(X),\ edb\_parent(X, Z). \\
& ?- ancestor(john, Y).
\end{aligned}
$$

Figure 6.1: Rewritten rules and magic rules for $ancestor^{bf}$.

The top-down method can be viewed as a set-at-a-time Prolog evaluation method. It follows the same unification process as in Prolog, but adopts set-at-a-time data accessing strategy. Therefore, no backtracking is needed. It treats each predicate in a rule body as a new subgoal, when a query is made against an IDB rule, query instantiation is passed to the rule body by unification between the query and the rule head. Each predicate in the rule body becomes a subgoal (or a new query) and is evaluated by the same top-down approach. The original query is processed if all its subgoals are evaluated. Figure 6.2 shows the program for query $ancestor^{bf}$ using the

top-down method, in which the ordering of rules and predicates has to be carefully specified to ensure safe query processing.

$$
\begin{aligned}
ancestor(X,Y) \ &:- \ edb\_parent(X,Y). \\
ancestor(X,Y) \ &:- \ edb\_parent(X,Z),\ ancestor(Z,Y). \\
?- ancestor(john,Y).&
\end{aligned}
$$

Figure 6.2: $ancestor^{bf}$ program for the top-down evaluation.

Programs used in cost comparison for the top-down and the magic sets methods can be found in Appendix B.

For simplicity, chain, magic and top-down are used to refer to the chain-based evaluation method, the top-down set-oriented evaluation method and the magic sets evaluation method, respectively.

6.2.2.2 Cost comparison

Figure 6.3 shows the cost for query $nqueens^{bf}$, where a placement of N queens on an $N \times N$ chess board is searched so that none of the queens attacks each other.

Figure 6.3 illustrates that top-down and chain methods have very close costs, whereas query processing using magic costs significantly more for the same type of query. For example, evaluating query "$?- nqueens(4, Qs)$" costs 3385, 3490 and 7668 units for top-down, chain and magic, respectively. Magic costs more than top-down and chain because in this program the magic predicate cannot reduce the size of an EDB relation. The program for the magic method is shown in Figure B.3 and Figure B.4 in Appendix B.

Figure 6.3 also indicates finding all valid chess board placements for N queens has a very high complexity. The size of search space is $O(n!)$.

Figure 6.4 shows the cost for query $queens^{fb}$, such as query "$?- nqueens(N,$ [3, 1, 4, 2])", where given a particular $N \times N$ chess board it is verified whether the queens on

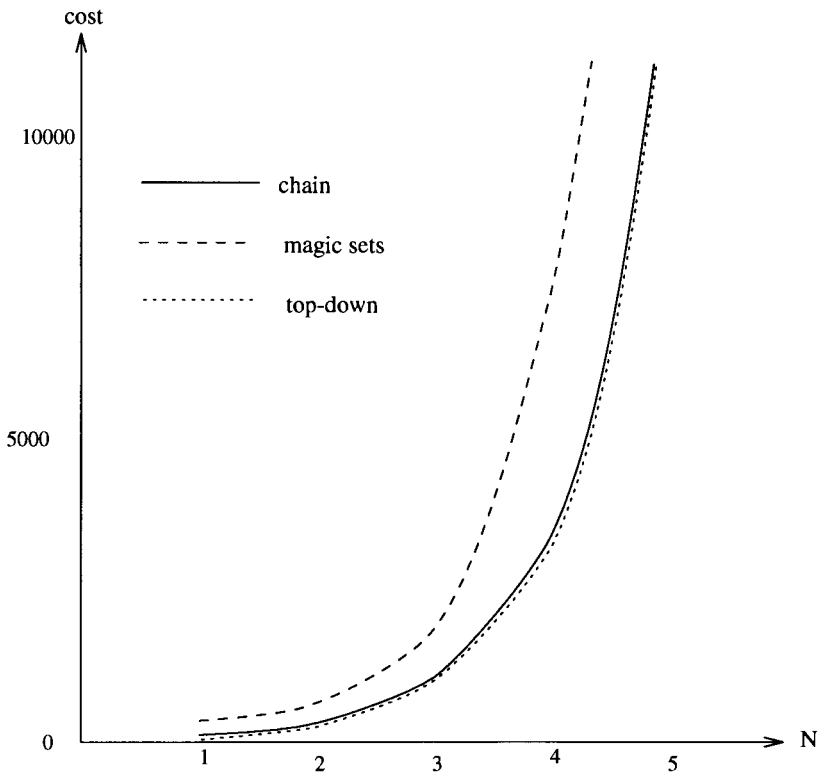


Figure 6.3: Cost for $nqueens^{bf}$.

the board attack each other, and if they do not, N is returned. Top-down and magic are unable to evaluate such kind of query no matter how the predicates or the rules in the program are reordered. Therefore, only the cost using chain is shown, with and without constraint pushing as discussed in Chapter 2. It is shown that the complexity for $queens^{fb}$ is substantially reduced, from $O(n!)$ to $O(n^2)$, when constraint derivation and pushing is employed. Notice that the chain method uses the same program in processing $nqueens^{bf}$ and $nqueens^{fb}$ queries, thus a greater level of declarativeness is achieved.

Figure 6.5 shows the costs for sorting a list using the permutation sort method. Corresponding programs are shown in Appendix B. Top-down and chain have the same costs, whereas magic costs more. However, when equipped with constraint pushing, chain has a much lower cost than that of top-down. Constraint pushing in the chain method reduces complexity from $O(n!)$ to $O(n^2)$ in this case.

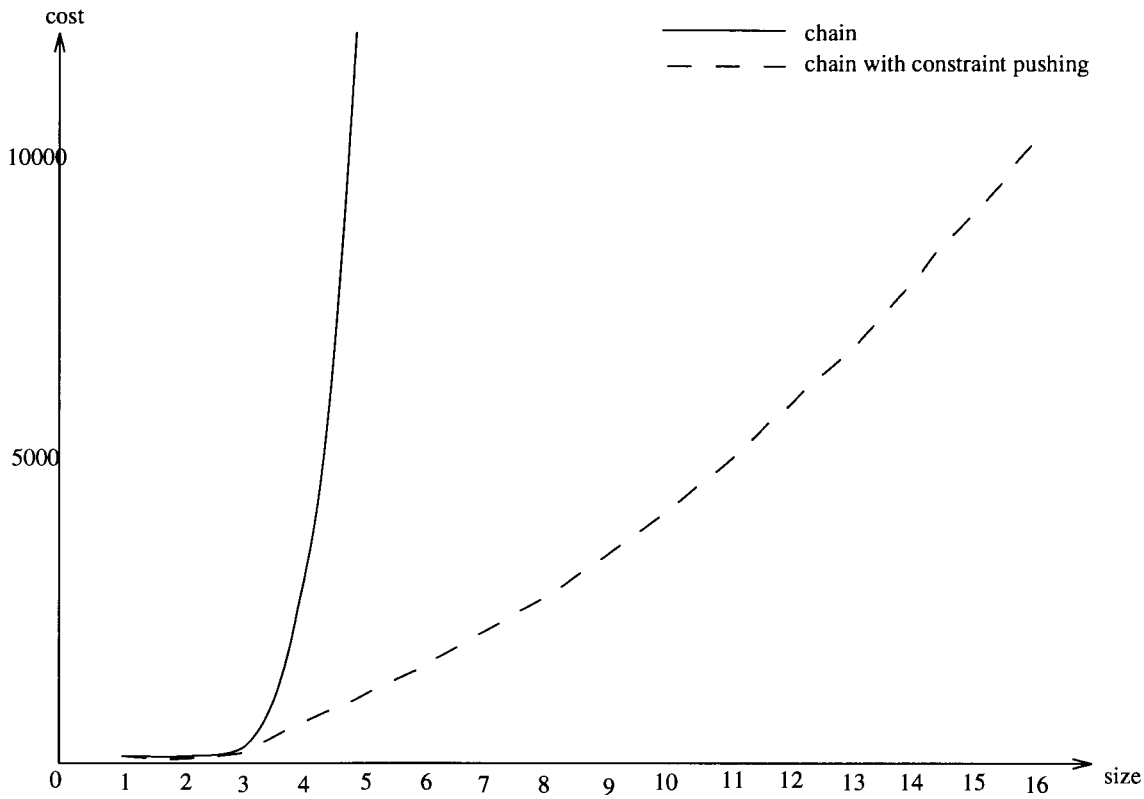


Figure 6.4: Cost for $nqueens^{fb}$.

For the insertion sort program shown in Figure 6.6 and Figure B.6 in Appendix B, chain has a slightly higher cost than top-down, whereas magic costs significantly more than both, as shown in Figure 6.7.

Figure 6.9 illustrates the costs for reversing a list. The programs are defined in Figure 6.8 and Figure B.7. The cost curves have similar shape as those in Figure 6.7.

In Figure 6.10, processing of query $ancestor^{bf}$ is considered, whose programs are shown in Figure 6.2 and Figure 6.1 for its version of the magic sets method. This program contains no function, and the cost for accessing EDB relations dominates the overall cost. To offset the effect that different data entry points have on the cost, the average cost for queries of type $ancestor^{bf}$ with entry point from each data node is shown. To maintain consistency among EDB relations of different size, all EDB relations have the same shape (tree or inverted-tree) and the same branch factor.

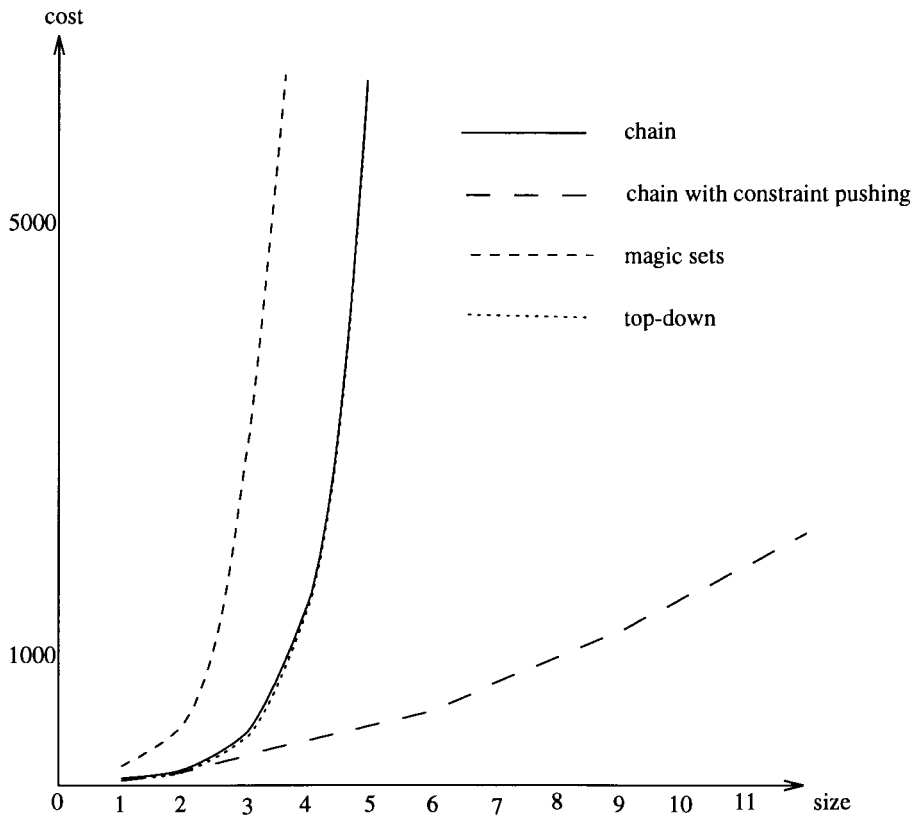


Figure 6.5: Cost for $permutation\_sort^{bf}$.

The costs for query $ancestor^{fb}$ is shown in Figure 6.11. Magic and chain have significant lower cost than the top-down method.

Figure 6.12 shows the cost for evaluating queries of sg^{bf} for the same generation program.

6.3 Further development of *LogicBase*

The current *LogicBase* implementation is a prototype system. The following major features are being incorporated in the development, with the incorporation of the recent research results in deductive databases [90, 111, 125].

$$
\begin{aligned}
&insertion\_sort([],[]).\\
&insertion\_sort([X|Xs],Ys) :- \\
&\quad insertion\_sort(Xs,Zs),\ insert(X,Zs,Ys).\\
&insert(X,[],[X]).\\
&insert(X,[Y|Ys],[X,Y|Ys]) :- X \leq Y.\\
&insert(X,[Y|Ys],[Y|Zs]) :- \\
&\quad X > Y,\ insert(X,Ys,Zs).\\
&?- insertion\_sort([4,3,2,1],X).
\end{aligned}
$$

Figure 6.6: Insertion sort program.

- *Aggregation and modularly stratified negation*:

  Stratified aggregation can be easily incorporated into *LogicBase*. Once the syntax for aggregation is parsed, the compilation phase may treat aggregate predicate as an ordinary predicate. During the plan generation phase, the evaluation of an aggregate predicate should be scheduled only after the predicate being aggregated has been available.

  To support modularly stratified negation, some kind of delaying mechanism is needed during the evaluation of the negated predicate so that all the data within each strongly connected component can be evaluated together. More study is needed to gain an insight about how negation can be accommodated in the chain-based compilation.

- *Secondary storage management*:

  Current *LogicBase* prototype implementation is a main memory based system, appropriate interface is needed to hook up with a storage manager or a relational database system to provide full-fledged database management system features. Such extension will surely expand the application domain of *LogicBase*.

- *Towards a deductive and object-oriented database system*:

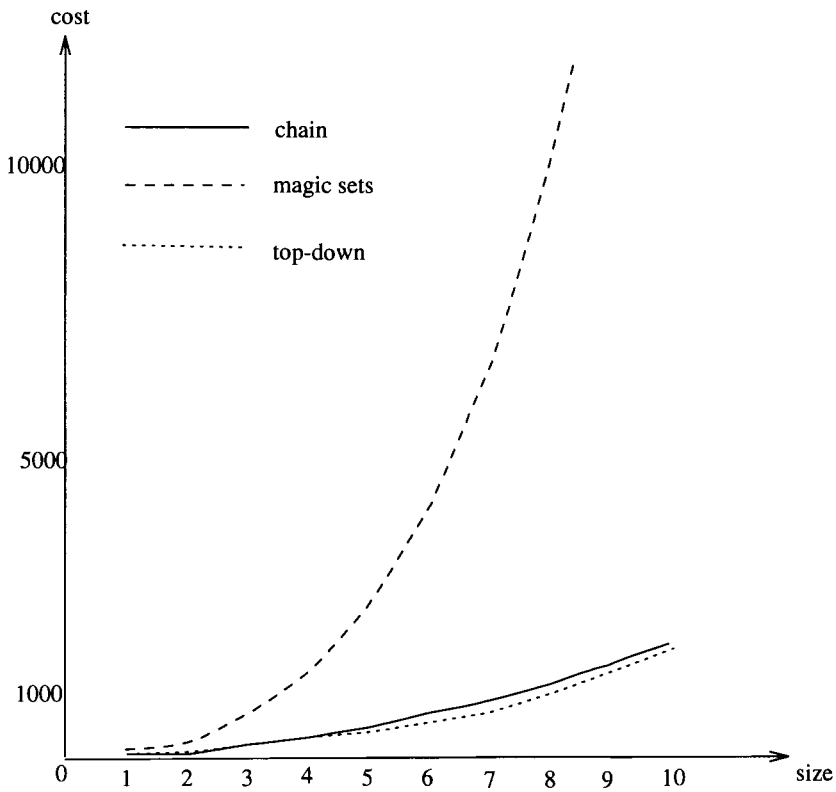


Figure 6.7: Cost for $insertion\_sort^{bf}$.

Merging of deductive and object-oriented data models in *LogicBase* presents an important direction and exciting challenge. A possible solution to incorporate object-oriented features into *LogicBase* is to adopt part of F-logic [74] as the logic representation and to extend the chain-based compilation and evaluation methods to support F-logic. Initial study indicates that in principle the introduction of F-logic will not interfere with the compilation of recursive [148], but the chain-based evaluation needs appropriate modification.

6.4 Conclusions

The *LogicBase* system has been implemented with a focus on the compilation and query evaluation of *application-oriented recursions*. The performance simulation among

$$
\begin{aligned}
&reverse([],[]).\\
&reverse([X|Xs],Zs) :- \\
&\quad reverse(Xs,Ys),\ append(Ys,[X],Zs).\\
&append([],L,L).\\
&append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).\\
&?- reverse([1,2,3],Y).
\end{aligned}
$$

Figure 6.8: Program to reverse a list.

different deductive query evaluation strategies demonstrates the potential of *LogicBase* as a declarative and efficient deductive database system. The system identifies different classes of recursions, and compiles recursions into regular chain forms when appropriate. Queries posed to the compiled recursions are analyzed systematically and efficient query evaluation plans are generated. Queries are executed mainly by chain-based evaluation, together with several other query evaluation methods, such as the generalized magic-sets method [13], etc. The system has been tested and demonstrated on some interesting deductive database and logic programming programs, with satisfactory results and good performance.

Based on our experimentation, it is felt that the *LogicBase* system prototype may represent an interesting alternative direction to efficient query evaluation in deductive databases and logic programming systems and may be worth further examination and development in the deductive database research.

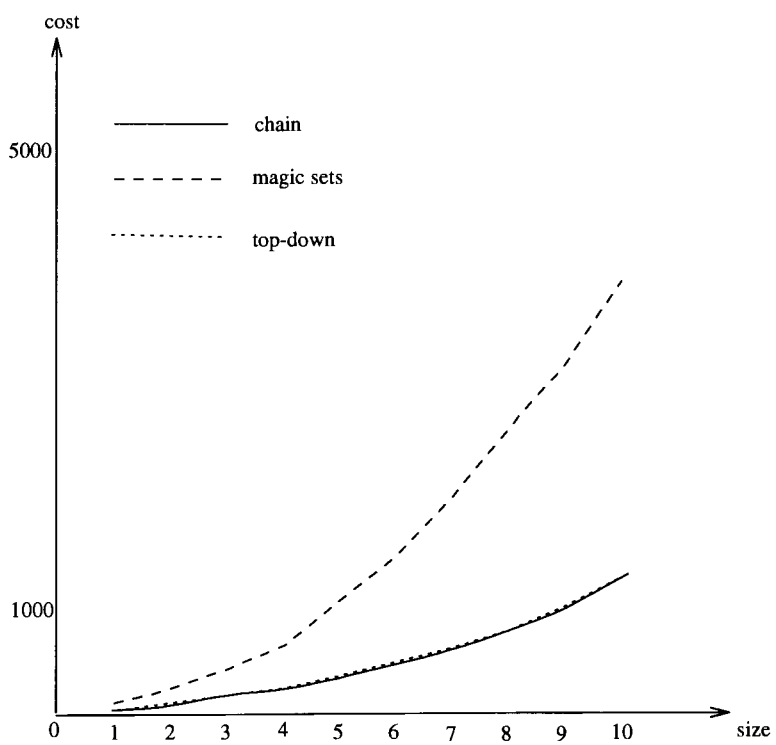


Figure 6.9: Cost for $reverse^{bf}$.

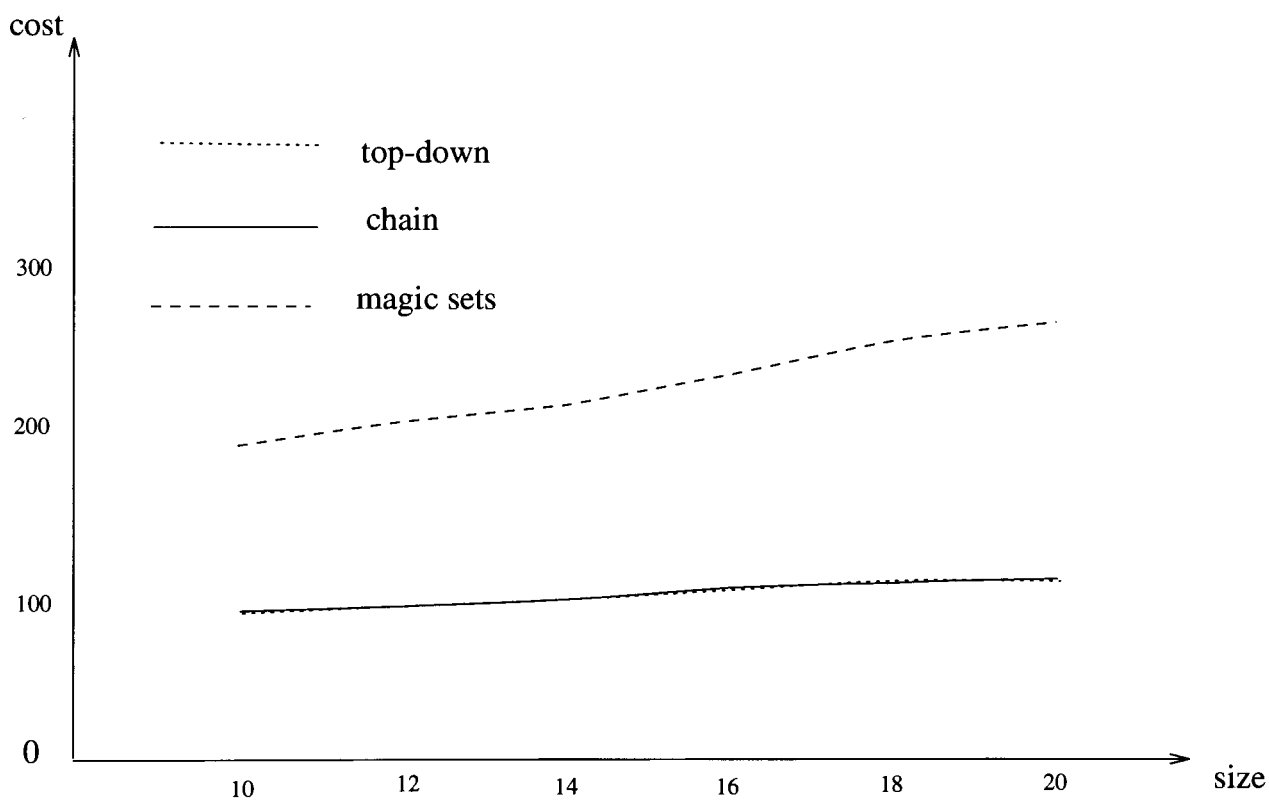


Figure 6.10: Average cost for $ancestor^{bf}$ with tree-shaped *edb_parent*.

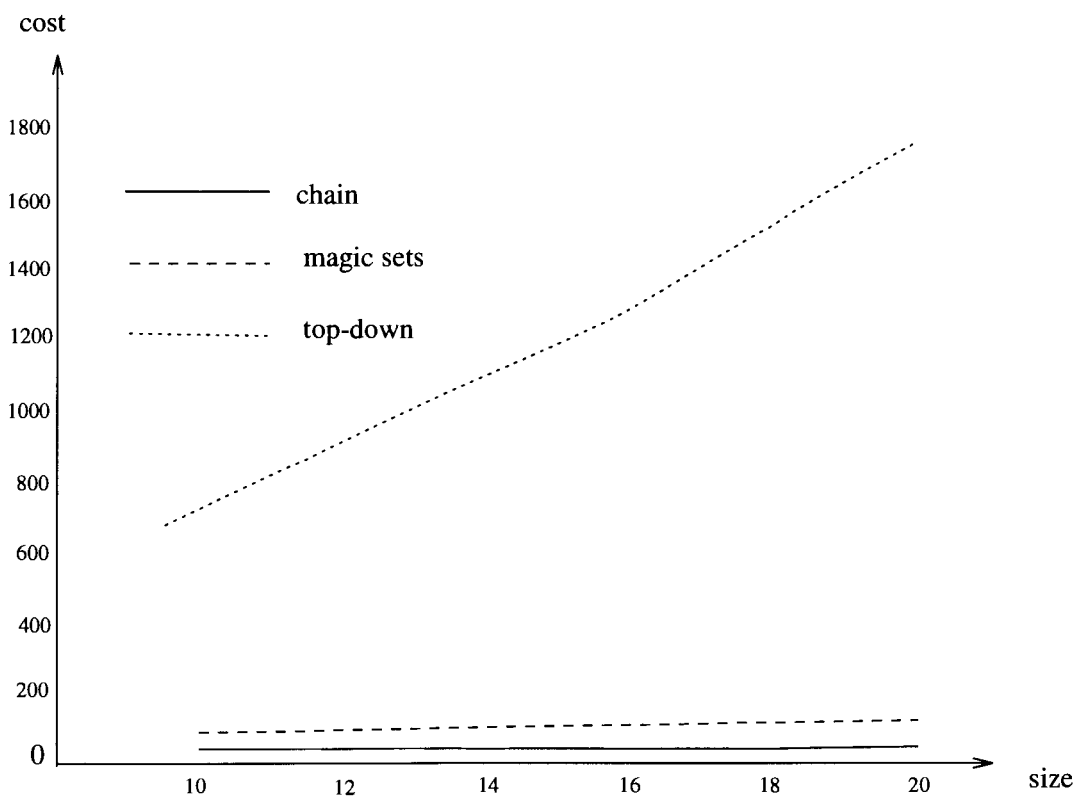


Figure 6.11: Average cost for $ancestor^{fb}$ with tree-shaped *edb_parent*.

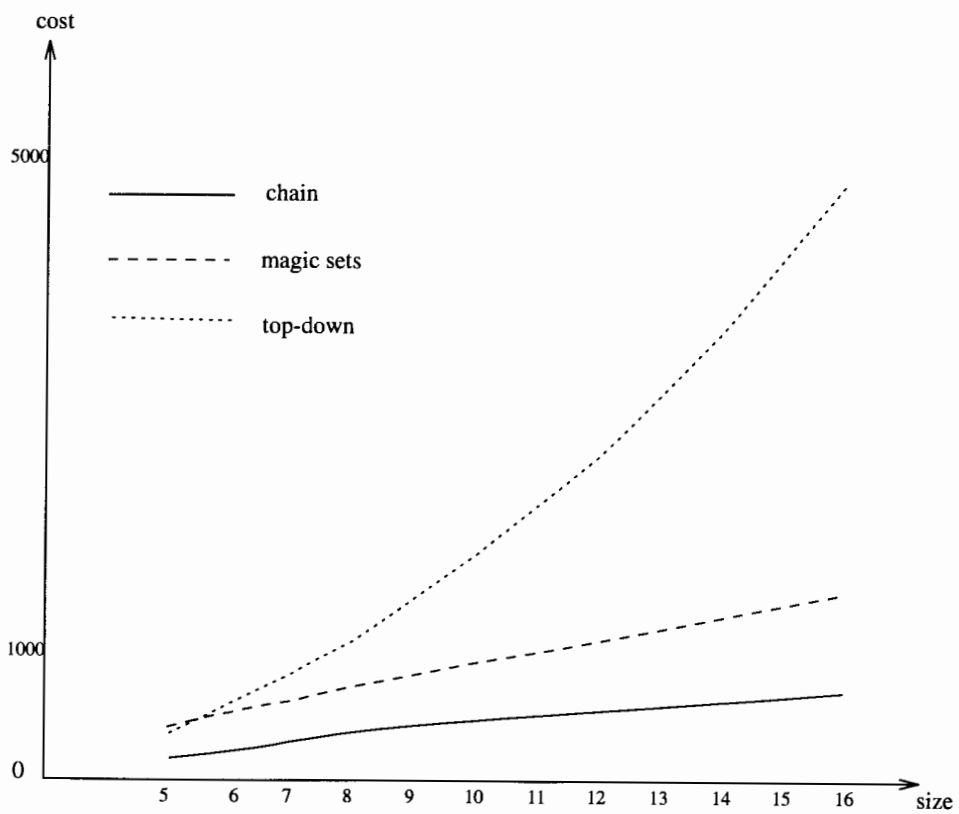


Figure 6.12: Cost for the same generation query sg^{bf}.

Appendix A

Syntax of LogicBase Input

Syntax for *LogicBase* input (specified in BNF):

program ::= definitions actions.

definitions ::= [edb] idb

actions ::= compilation_only | { query }

edb ::= 'define_edb' '{' edb_tables '}'

edb_ables ::= { an_edb_table }

an_edb_table ::= 'table' ':' constant ':' int ';' 'heading' ':' attr_pair ';' 'tuples' ':' tuple_set '.'

attr_pairs ::= one_attr_pair | attr_pairs ',' one_attr_pair

one_attr_pair ::= constant ':' int

tuple_set ::= a_tuple | tuple_set ';' a_tuple

a_tuple ::= arg | a_tuple ',' arg

idb ::= 'define_idb' constant '{' rules '}'

rules ::= { a_rule }

a_rule ::= a_pred '.' | a_pred '< −' preds '.'

preds ::= s_a_pred | preds ',' s_a_pred

s_a_pred ::= a_pred | 'not' ',' s_a_pred

a_pred ::= constant '(' arglist ')' | arg 'is' arg math_op arg | arg relation_op arg

arglist ::= arg | arglist ',' arg
arg ::= item_arg | list_arg | a_funct
a_function ::= constant '(' arglist ')'

item_arg ::= string | var | int | constant
list_arg ::= '[' ']' | '[' head ']' | '[' head '|' tail ']'
head ::= arg | arg ',' head
tail ::= var | list_arg
math_op ::= '+' | '−'
relation_op ::= '>' | '<' | '=' | '>=' | '<='
list_op ::= 'cons'
op ::= math_op | relation_op | list_op

compilation_only ::= 'compile' constant

query ::= '?-' a_query '.'
a_query ::= preds

low_case_letter ::= 'a' | 'b' | .... |'z'
upper_case_letter ::= 'A' | 'B' | .... | 'Z'
digit ::= '0' | '1' | '2' | .... | '9'
letter ::= lower_case_letter | upper_case_letter | digit
int ::= { digit }
constant ::= lower_case_letter [letter]
var ::= upper_case_letter [letter]

Appendix B

Programs For Cost Comparison

Programs used in cost comparison in chapter 6 are listed here.

Figure B.1 shows the N-queen program using the chain-based evaluation approach for both query $nqueens^{bf}$ and query $nqueens^{fb}$.

Figure B.2 shows the N-queen program using top-down approach for query $nqueens^{bf}$.

Figure B.3 and Figure B.4 together show the rewritten rule set for query $nqueens^{bf}$ of N-queen program using the magic sets method.

Figure B.5 gives the magic sets program for permutation sort.

Figure B.6 is the insertion sort program using the magic sets method.

Figure B.7 is the program for reverse a list using the magic sets method.

Figure B.8 shows the rewritten *ancestor* programs for the queries $ancestor^{fb}$ using the magic sets method.

Figure B.9 contains the rewritten same generation program for the query sg^{bf} for the magic sets method. The data in EDB relations satisfy following rules:

1. $r(a_i, a_{i+1})$ for $1 \leq i < n$.
2. $r(a_1, a_i)$ for $3 \leq i \leq n$.

$nqueens(N, Qs) : -range(1, N, Ns),\ queens(Ns, [], Qs).$
$range(M, N, [M|Ns]) : -M < N,\ M1\ is\ M + 1,\ range(M1, N, Ns).$
$range(N, N, [N]).$
$queens([], Qs, Qs).$
$queens(Unplaced, Safe, Qs) : -$
$\quad select(Q, Unplaced, Unplaced1),\ not\ attack(Q, Safe),$
$\quad queens(Unplaced1, [Q|Safe], Qs).$
$attack(X, Xs) : -attack1(X, 1, Xs).$
$attack1(X, N, [Y|Ys]) : -X\ is\ Y + N.$
$attack1(X, N, [Y|Ys]) : -X\ is\ Y - N.$
$attack1(X, N, [Y|Ys]) : -N1\ is\ N + 1,\ attack1(X, N1, Ys).$
$select(X, [X|Xs], Xs).$
$select(X, [Y|Ys], [Y|Zs]) : -select(X, Ys, Zs).$
$? - nqueens(4, Qs).$
$? - nqueens(N, [3, 1, 4, 2]).$

Figure B.1: N-queens program using the chain-based evaluation method.

$nqueens(N, Qs) : -range(1, N, Ns),\ queens(Ns, [], Qs).$
$range(M, N, MNs) : -$
$\quad M < N,\ M1\ is\ M + 1,\ range(M1, N, Ns),\ cons(M, Ns, MNs).$
$range(M, N, MNs) : -M = N,\ cons(N, [], MNs).$
$queens(Unplaced, Safe, Qs) : -Unplaced = [],\ Safe = Qs.$
$queens(Unplaced, Safe, Qs) : -select(Q, Unplaced, Unplaced1),$
$\quad not\ attack(Q, Safe),\ cons(Q, Safe, QSafe),$
$\quad queens(Unplaced1, QSafe, Qs).$
$attack(X, Xs) : -attack1(X, 1, Xs).$
$attack1(X, N, YYs) : -cons(Y, Ys, YYs),\ X\ is\ Y + N.$
$attack1(X, N, YYs) : -cons(Y, Ys, YYs),\ X\ is\ Y - N.$
$attack1(X, N, YYs) : -cons(Y, Ys, YYs),$
$\quad N1\ is\ N + 1,\ attack1(X, N1, Ys).$
$select(X, XXs, Xs) : -cons(X, Xs, XXs).$
$select(X, YYs, YZs) : -cons(Y, Ys, YYs),$
$\quad select(X, Ys, Zs),\ cons(Y, Zs, YZs).$
$? - nqueens(4, Qs).$

Figure B.2: N-queens program using the top-down approach for query $nqueens^{bf}$.

$nqueens(N, Qs) : -range(1, N, Ns),\ queens(Ns, [], Qs).$
$m\_range(M, N) : -range(M, N, MNs).$
$m\_range(M1, N) : -m\_range(M, N),\ M < N,\ M1\ is\ M + 1.$
$range(M, N, MNs) : -m\_range(M, N),\ M = N,\ cons(N, [], MNs).$
$range(M, N, MNs) : -range(M1, N, Ns),\ m\_range(M, N),$
$M < N,\ M1\ is\ M + 1,\ cons(M, Ns, MNs).$
$m\_queens(Unplaced, Safe) : -queens(Unplaced, Safe, Qs).$
$m\_queens(Unplaced1, QSafe) : -m\_queens(Unplaced, Safe),$
$select^{fbf}(Q, Unplaced, Unplaced1),\ not\ attack(Q, Safe),$
$cons(Q, Safe, QSafe).$
$queens(Unplaced, Safe, Qs) : -m\_queens(Unplaced, Safe),$
$Unplaced = [], Safe = Qs.$
$queens(Unplaced, Safe, Qs) : -queens(Unplaced1, QSafe, Qs),$
$cons(Q, Safe, QSafe), m\_queens(Unplaced, Safe),$
$notattack(Q, Safe), select^{bbb}(Q, Unplaced, Unplaced1).$

Figure B.3: N-queen program $nqueens^{bf}$ using the magic sets method, part I.

$attack(X, Xs) : -attack1(X, 1, Xs).$
$m\_attack1(X, N, YYs) : -attack1(X, N, YYs).$
$m\_attack1(X, N1, Ys) : -m\_attack1(X, N, YYs), N1\ is\ N + 1, cons(Y, Ys, YYs).$
$attack1(X, N, YYs) : -m\_attack1(X, N, YYs), cons(Y, Ys, YYs), X\ is\ Y + N.$
$attack1(X, N, YYs) : -m\_attack1(X, N, YYs), cons(Y, Ys, YYs), X\ is\ Y - N.$
$attack1(X, N, YYs) : -attack1(X, N1, Ys),\ m\_attack1(X, N, YYs),$
$\quad N1\ is\ N + 1,\ cons(Y, Ys, YYs).$
$m\_select^{bbb}(X, YYs, YZs) : -select^{bbb}(X, YYs, YZs).$
$m\_select^{bbb}(X, Ys, Zs) : -m\_select^{bbb}(X, YYs, YZs),$
$\quad cons(Y, Ys, YYs), cons(Y, Zs, YZs).$
$select^{bbb}(X, YYs, YZs) : -m\_select^{bbb}(X, YYs, YZs), cons(X, YZs, YYs).$
$select^{bbb}(X, YYs, YZs) : -select^{bbb}(X, Ys, Zs),$
$\quad m\_select^{bbb}(X, YYs, YZs),\ cons(Y, Ys, YYs),\ cons(Y, Zs, YZs).$
$m\_select^{fbf}(XXs) : -select^{fbf}(X, XXs, Xs).$
$m\_select^{fbf}(Xs) : -m\_select^{fbf}(XXs),\ cons(X, Xs, XXs).$
$select^{fbf}(X, XXs, Xs) : -m\_select^{fbf}(XXs),\ cons(X, Xs, XXs).$
$select^{fbf}(X, YYs, YZs) : -select^{fbf}(X, Ys, Zs),$
$\quad m\_select^{fbf}(YYs),\ cons(Y, Ys, YYs),\ cons(Y, Zs, YZs).$
$? - nqueens(1, Qs).$

Figure B.4: N-queen program $nqueens^{bf}$ using the magic sets method, part II.

$permutation\_sort(Xs, Ys) : -permutation(Ys, Xs), ordered(Ys).$
$m\_permutation(ZZs) : -permutation(Xs, ZZs).$
$m\_permutation(Zs) : -m\_permutation(ZZs),\ cons(Z, Zs, ZZs).$
$permutation(Xs, ZZs) : -m\_permutation(ZZs),\ Xs = [],\ ZZs = [].$
$permutation(Xs, ZZs) : -permutation(Ys, Zs),\ m\_permutation(ZZs),$
$\quad cons(Z, Zs, ZZs),\ select(Z, Xs, Ys).$
$m\_select(X, YZs) : -select(X, YYs, YZs).$
$m\_select(X, Zs) : -m\_select(X, YZs),\ cons(Y, Zs, YZs).$
$select(X, XXs, Xs) : -m\_select(X, Xs),\ cons(X, Xs, XXs).$
$select(X, YYs, YZs) : -select(X, Ys, Zs),$
$\quad m\_select(X, YZs),\ cons(Y, Zs, YZs),\ cons(Y, Ys, YYs).$
$m\_ordered(Y) : -ordered(Y).$
$m\_ordered(YYs) : -m\_ordered(XYYs),\ cons(X, YYs, XYYs).$
$ordered(Y) : -m\_ordered(Y),\ cons(X, [], Y).$
$ordered(XYYs) : -ordered(YYs),\ m\_ordered(XYYs),$
$\quad cons(X, YYs, XYYs),\ cons(Y, Ys, YYs),\ X <= Y.$
$? - permutation\_sort([5, 4, 3, 2, 1], Ys).$

Figure B.5: Magic sets program for $permutation\_sort^{bf}$.

$m\_insertion\_sort^{bf}(XXs) : -insertion\_sort(XXs, Ys).$
$m\_insertion\_sort^{bf}(Xs) : -m\_insertion\_sort^{bf}(XXs),\ cons(X, Xs, XXs).$
$insertion\_sort^{bf}([], []).$
$insertion\_sort^{bf}(XXs, Ys) : -insertion\_sort^{bf}(Xs, Zs),$
$\quad m\_insertion\_sort^{bf}(XXs),\ cons(X, Xs, XXs),\ insert^{bbf}(X, Zs, Ys).$
$m\_insert^{bbf}(X, YYs) : -insert^{bbf}(X, YYs, YZs).$
$m\_insert^{bbf}(X, Ys) : -m\_insert^{bbf}(X, YYs),\ cons(Y, Ys, YYs).$
$insert^{bbf}(X, YYs, YZs) : -m\_insert^{bbf}(X, YYs),$
$\quad cons(Y, Ys, YYs),\ X <= Y,\ cons(X, YYs, YZs).$
$insert^{bbf}(X, YYs, YZs) : -m\_insert^{bbf}(X, Ys),$
$\quad YYs = [], cons(X, [], YZs).$
$insert^{bbf}(X, YYs, YZs) : -insert^{bbf}(X, Ys, Zs),$
$\quad m\_insert^{bbf}(X, YYs), cons(Y, Ys, YYs),\ X > Y,\ cons(Y, Zs, YZs).$
$? - insertion\_sort^{bf}([5, 4, 3, 2, 1], X).$

Figure B.6: Insertion sort program for the magic sets method.

$m\_reverse(XXs) : -reverse(XXs, Zs).$
$m\_reverse(Xs) : -m\_reverse(XXs), cons(X, Xs, XXs).$
$reverse([], []).$
$reverse(XXs, Zs) : -reverse(Xs, Ys),$
$\quad m\_reverse(XXs), cons(X, Xs, XXs),$
$\quad cons(X, [], X1), append^{bbf}(Ys, X1, Zs).$
$m\_append^{bbf}(XL1, L2) : -append^{bbf}(XL1, L2, XL3).$
$m\_append^{bbf}(L1, L2) : -m\_append^{bbf}(XL1, L2), cons(X, L1, XL1).$
$append^{bbf}(XL1, L2, XL3) : -m\_append^{bbf}(XL1, L2), XL1 = [], L2 = XL3.$
$append^{bbf}(XL1, L2, XL3) : -append^{bbf}(L1, L2, L3),$
$\quad m\_append^{bbf}(XL1, L2), cons(X, L1, XL1), cons(X, L3, XL3).$
$? - reverse([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], Y).$

Figure B.7: Rewritten program for the magic sets method for query $reverse^{bf}$.

$$
\begin{aligned}
&m\_ancestor^{fb}(jack).\\
&ancestor^{fb}(X,Y) : -m\_ancestor^{fb}(Y), edb\_parent(X,Y).\\
&ancestor^{fb}(X,Y) : -ancestor^{fb}(Z,Y),\\
&\quad m\_ancestor^{fb}(Y), edb\_parent(X,Z).\\
&?- ancestor^{fb}(X, jack).
\end{aligned}
$$

Figure B.8: Rewritten rules for query $ancestor^{fb}$ using the magic sets method.

$$
\begin{aligned}
&m\_sg(X) : -sg(X,Y).\\
&m\_sg(Xp) : -m\_sg(X),\ r(X,Xp).\\
&sg(X,Y) : -m\_sg(X),\ q(X,Y).\\
&sg(X,Y) : -sg(Xp,Yp),\\
&\quad m\_sg(X),\ r(X,Xp),\ s(Y,Yp).\\
&?- sg(a1,Y).
\end{aligned}
$$

Figure B.9: Rewritten rules for sg^{bf} for the magic sets method

3. $q(a_n, b_n)$.

4. $s(b_i, b_{i-1})$ for $2 \leq i \leq n$.

Bibliography

[1] S. Abiteboul and S. Grumbach. A rule-based language with functions and sets. *ACM Transations on Database Systems*, 16:1–30, 1991.

[2] F. Afrati, C. Papadimitriou, G. Papageorgiou, A.R. Roussou, Y. Sagiv, and J.D. Ullman. On the convergence of query evaluation. *J. Computer and System Sciences*, 38:314–359, 1989.

[3] H. Aly and Z. M. Ozsoyoglu. Synchronized counting method. In *Proc. 5th Int. Conf. Data Engineering*, pages 366–373, Los Angeles, CA, February 1989.

[4] K.R. Apt, H.A. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, 1987.

[5] K.R. Apt and M.H. Van Emden. Contributions to the theory of logic programming. *J. ACM*, 29:841–862, 1982.

[6] I. Balbin, G.S. Port, K. Ramamohanaraon, and K. Meenakshi. Efficient bottom-up compuation of queries on stratified databases. *J. Logic Programming*, 11:295–344, 1991.

[7] I. Balbin and K. Ramamohanarao. A generalization of the differential approach to recursive query evaluation. *J. Logic Programming*, 4:259–262, 1987.

[8] F. Bancilhon. Naive evaluation of recursively defined relations. In M. Brodie and J. Mylopoulos, editors, *On Knowledge Base Management Systems*, pages 165–178. Springer-Verlag, 1986.

[9] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proc. 5th ACM Symp. Principles of Database Systems*, pages 1–15, Cambridge, MA, March 1986.

[10] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Proc. 1986 ACM-SIGMOD Int. Conf. Management of Data*, pages 16–52, Washington, DC, May 1986.

[11] F. Bancilhon and R. Ramakrishnan. Performance evaluation of data intensive logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 439–517. Morgan Kaufmann, 1988.

[12] C. Beeri, S. Naqvi, R. Ramakrishnan, O. Shmueli, and S. Tsur. Sets and negation in a logic database language. In *Proc. of the ACM Symposium on Principles of Database systems*, pages 21–37, San Diego, CA, March 1987.

[13] C. Beeri and R. Ramakrishnan. On the power of magic. In *Proc. 6th ACM Symp. Principles of Database Systems*, pages 269–283, San Diego, CA, March 1987.

[14] C. Beeri, R. Ramakrishnan, D. Srivastava, and S. Sudarshan. The valid model semantics for logic programs. In *Proc. of the 11th ACM Symp. on Principles of Database Systems*, 1992.

[15] J. Bocca. Educe: a marriage of convenience: Prolog and a relational database. In *Symp. on Logic Programming*, pages 36–45, IEEE, New York, 1986.

[16] B. Breitag, H. Schutz, and G. Specht. Lola – a logic language for deductive databases and its implementation. In *Proc. of 2nd Int'l Symp. on Database Systems for Advanced Applications*, 1991.

[17] D. A. Briggs. A correction of the termination conditions of the Henschen-Naqvi technique. *J. ACM*, 37:712–719, 1990.

[18] A. Brodsky and Y. Sagiv. On termination of datalog programs. In *Proc. 1st Int. Conf. Deductive and Object-Oriented Databases (DOOD'89)*, pages 95–112, Kyoto, Japan, December 1989.

[19] A. Brodsky and Y. Sagiv. Inference of inequality constraints in logic programs. In *Proc. 10th ACM Symp. Principles of Database Systems*, pages 227–240, Denver, CO, May 1991.

[20] F. Bry, Decker, and R. Manthey. A uniform approach to constraint satisfaction and constraint satisfiability in deductive databases. In *Proc. of the Int'l Conf. on Extending Database Technology*, 1988.

[21] F. Cacace, S. Ceri, S. Crespi-Reghizzi, L. Tanca, and R. Zicari. Integrating object-oriendted data modeling with a rule-baesd programming paradigm. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, 1990.

[22] M. Carey, D. DeWitt, G. Graefe, D. Haight, J. Richardson, D. Schuh, E. Shekita, and S. Vandenberg. The EXODUS extensible dbms project: An overview. In *Readings in Object-Oriented Databases*, pages 474–499, Morgan-Kaufman, 1990.

[23] S. Ceri, G. Gottlob, and G. Wiederhold. Interfacing relational databases and Prolog efficiently. In L. Kerschberg, editor, *Expert Database Systems*, pages 207–223. Benjamin-Cummings, Menlo Park, CA, 1987.

[24] A.K. Chandra and D. Harel. Structure and complexity of relational queries. *J. Computer and System Sciences*, 25:99–128, 1982.

[25] A.K. Chandra and D. Harel. Horn clause queries and generalizations. *J. Logic Programming*, 2:1–15, April 1985.

[26] W. Chen, M. Kifer, and D. S. Warren. Hilog: A foundation for higher-order logic programming. *J. Logic Programming*, 15:187–230, 1993.

[27] W. Chen and D. S. Warren. Query evaluation under the well-founded semantics. In *Proc. 12th ACM Symp. Principles of Database Systems*, pages 168–179, Washington, D.C., 1993.

[28] D. Chimenti, R. Gamboa, R. Krishnamurthy, S. Naqvi, S. Tsur, and C. Zaniolo. The LDL system prototype. *IEEE Trans. Knowledge and Data Engineering*, 2:76–90, 1990.

[29] R.L. Cole and G. Graefe. Optimization of dynamic query evaluation plans. In *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, pages 150–160, Minneapolis, MN, May, 1994.

[30] M. Consens, I. Cruz, and A. Mendelzon. Visualizing queries and querying visualization. In *ACM SIGMOD Record*, pages 39–46, May 1992.

[31] M.P. Consens and A.O. Mendelzon. Low complexity aggregation in graphlog and datalog. In *Proc. of the Int'l Conf. on Database Theory*, Paris, 1990.

[32] I.F. Cruz and T.S. Norvell. Aggregative closure: An extension of transitive closure. In *Proc. IEEE 5th Int'l Conf. Data Engineering*, pages 384–389, Los Angeles, CA, 1989.

[33] S. W. Dietrich. Extension tables: Memo relations in logic programming. In *Proc. 1987 Symp. Logic Programming*, pages 264–272, San Francisco, CA, 1987.

[34] M.H. Van Emden and R.A. Kowalski. The semantics of predicate logic as a programming language. *J. ACM*, 23:733–742, 1976.

[35] S. Ganguly, S. Greco, and C. Zaniolo. Minimum and maximum predicates in logic programming. In *Proc. of the 10th ACM Symposium on Principles of Databases Systems*, pages 154–163, Denver, CO, 1991.

[36] S. Ganguly, S. Greco, and C. Zaniolo. Greedy by choice. In *Proc. of the 11th ACM Symposium on Principles of Database Systems*, San Diego, CA, 1992.

[37] M. Gelfond and V.Lifschitz. The stable model semantics for logic programming. In *Proc. of the Fifth Int'l Conf./Symposium on Logic Programming*, pages 1070–1080, Seattle, WA, Sept. 1988.

[38] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25:73–170, 1993.

[39] S. Greco and C. Zaniolo. Optimization of linear programs using counting methods. In *Pro. 3rd Int'l Conf. on Extending Database Technology*, pages 72–87, Vienna, Austria, March 1992.

[40] R. W. Haddad and J. F. Naughton. Counting methods for cyclic relations. In *Proc. 7th ACM Symp. Principles of Database Systems*, pages 333–340, Austin, TX, March 1988.

[41] R. W. Haddad and J. F. Naughton. A counting algorithm for a cyclic binary query. *J. Computer and System Sciences*, 43:145–169, 1991.

[42] J. Han. Selection of processing strategies for different recursive queries. In *Proc. 3rd Int. Conf. Data and Knowledge Bases*, pages 59–68, Jerusalem, Israel, June 1988.

[43] J. Han. Compiling general linear recursions by variable connection graph analysis. *Computational Intelligence*, 5:12–31, 1989.

[44] J. Han. Multi-way counting method. *Information Systems*, 14:219–229, 1989.

[45] J. Han. Constraint-based reasoning in deductive databases. In *Proc. 7th Int. Conf. Data Engineering*, pages 257–265, Kobe, Japan, April 1991.

[46] J. Han. On the power of query-independent compilation. *Int'l J. Software Engineering and Knowledge Engineering*, 2:277–292, 1992.

[47] J. Han. Constraint-based query evaluation in deductive databases. *IEEE Trans. Knowledge and Data Engineering*, 6:96–107, 1994.

[48] J. Han and L. J. Henschen. Handling redundancy in the processing of recursive database queries. In *Proc. 1987 ACM-SIGMOD Int. Conf. Management of Data*, pages 73–81, San Francisco, CA, May 1987.

[49] J. Han and L. J. Henschen. The level-cycle merging method. In W. Kim, J.-M. Nicolas, and S. Nishio, editors, *Deductive and Object-Oriented Databases*, pages 65–82. Elsevier Science, 1990.

[50] J. Han and L. V. S. Lakshmanan. Evaluation of regular nonlinear recursions by deductive database techniques. In *J. Information Systmes*, to appear, 1995.

[51] J. Han and L. Liu. Processing multiple linear recursions. In *Proc. 1989 North American Conf. Logic Programming*, pages 816–830, Cleveland, OH, October 1989.

[52] J. Han and L. Liu. Efficient evaluation of multiple linear recursions. *IEEE Trans. Software Engineering*, 17:1241–1252, 1991.

[53] J. Han, L. Liu, and T. Lu. Evaluation of declarative n-queens recursion: A deductive database approach. In *(submitted to) J. Intelligent Information System*, 1994.

[54] J. Han, L. Liu, and Z. Xie. LogicBase: A system prototype for deductive query evaluation. In *Proc. ILPS'93 Workshop on Programming with Logic Databases*, pages 146–160, Vancouver, Canada, October 1993.

[55] J. Han, L. Liu, and Z. Xie. Logicbase: A deductive database system prototype. In *Proc. of Conf. of Information and Knowledge Management*, pages 226–233, Gaithersburg, Maryland, Nov. 1994.

[56] J. Han and T. Lu. N-queens problem revisited: A deductive database approach. In *Proc. 1992 IJCSLP Workshop on Deductive Databases*, pages 48–55, Washinton D.C., Nov. 1992.

[57] J. Han and W. Lu. Asynchronous chain recursions. *IEEE Trans. Knowledge and Data Engineering*, 1:185–195, 1989.

[58] J. Han and K. Zeng. Automatic generation of compiled forms for linear recursions. *Information Systems*, 17:299–322, 1992.

[59] L. J. Henschen and S. Naqvi. On compiling queries in recursive first-order databases. *J. ACM*, 31:47–85, 1984.

[60] Y.E. Ioanidis, J. Chen, M.A. Friedman, and M.M. Tsangaris. Bermuda – an architectural perspective on interfacing Prolog to a database machine. In L. Kerschberg, editor, *Proc. 2nd Int'l Conf. on Expert Database Systems*, pages 229–255. Benjamin-Cummings, Menlo Park, CA, 1989.

[61] Y. E. Ioannidis and R. Ramakrishnan. Efficient transitive closure algorithms. In *Proc. 14th Int. Conf. Very Large Data Bases*, pages 382–394, Long Beach, CA, August 1988.

[62] J. Jaffar and J-L. Lassez. Constraint logic programming. In *Proc. 14th ACM Symp. Principles of Programming Languages*, pages 111–119, Munich, Germany, 1987.

[63] M. Jarke and J. Koch. Query optimization in database systems. *ACM Comput. Surv.*, 16:111–152, 1984.

[64] M. Jeusfeld and M. Staudt. Query optimization in deductive object bases. In G. Vossen, J.C. Greytag, and D. Maier, editors, *Query Processing for Advanced Database Applications*. Morgan-Kaufmann, 1993.

[65] B. Jiang. A suitable algorithm for computing partial transitive closures. In *Proc. 6th Int. Conf. Data Engineering*, pages 264–271, Los Angeles, CA, February 1990.

[66] P.C. Kanellakis, G.M. Kuper, and P.Z. Revesz. Constraint query language. In *Proc. 9th ACM Symp. on Principle of Database Systems*, pages 299–313, Nashville, TN, 1990.

[67] C. Kellogg, A. O'Hare, and L. Travis. Optimizing the rule-data interface in a knowledge management system. In *Proc. 12th Int. Conf. Very Large Data Bases*, Kyoto, Japan, August 1986.

[68] D. Kemp, D. Srivastava, and P. Stuckey. Magic sets and bottom-up evaluation of well-founded models. In *Proc. of the Int'l Logic Programming Symposium*, pages 337–351, San Diego, CA, October, 1991.

[69] D. Kemp and P. Stuckey. Semantics of logic programs with aggregates. In *Proc. of the international Logic Programming symp.*, pages 387–401, San Diego, CA, October, 1991.

[70] D. Kemp and P. Stuckey. Analysis based constraint query optimization. In *Proc. of Int'l Conf. of Logic Programming*, Montreal, Canada, 1993.

[71] D. B. Kemp, K. Ramamohanarao, I. Balbin, and K. Meenakshi. Propagating constraints in recursive deductive databases. In *Proc. 1989 North American Conf. Logic Programming*, pages 981–998, Cleveland, OH, October 1989.

[72] J.M. Kerisit and J.M. Pugin. Efficient query answering on stratified database. In *Proc. of the Int'l Conf. on Fifth Generation Computer Systems*, pages 719–725, Tokyo, Japan, November 1988.

[73] M. Kifer and G. Lausen. F-Logic: A higher order language for reasoning about objects, inheritance, and scheme. In *Proc. 1989 ACM-SIGMOD Int. Conf. Management of Data*, pages 134–146, Portland, Oregon, June 1989.

[74] M. Kifer, G. Lausen, and J. Wu. Logical foundations for object-oriented and frame-based languages. In *J. ACM*, to appear, 1994.

[75] M. Kifer and E.L. Lozinski. SYGRAF–implementing logic programs in a database style. *IEEE Trans. on Software Eng.*, 14:922–935, July 1988.

[76] W. Kim, D. S. Reiner, and D. S. Batory. *Query Processing in Database Systems.* Springer-Verlag, 1985.

[77] R.A. Kowalski and D. Kuehner. Linear resolution with selection function. In *Artificial Intelligence*, 227-260, 1971.

[78] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In *Proc. 12th Int. Conf. Very Large Data Bases*, pages 128–137, Kyoto, Japan, Aug. 1986.

[79] L. V. S. Lakshmanan, F. Sadri, and I. N. Subramanian. On the logic foundations of schema integration and evolution in heterogeneous database systems. In S. Ceri and et. al., editors, *Deductive and Object-Oriented Databases (DOOD'93) [Lecture Notes in Computer Science 760]*, pages 81–100. Springer Verlag, 1993.

[80] A. Lefebvre. Towards an efficient evaluation of recursive aggregates in deductive databases. In *Proc. Int. Conf. Fifth Generation Computer Systems*, pages 915–925, Tokyo, Japan, June 1992.

[81] L. Liu and J. Han. Compressed counting method. In *Proc. Int'l Conf./Symp. on Logic Programming Workshop on Deductive Database*, pages 76–85, Washington D.C., Nov. 1992.

[82] A. K. Mackworth. Constraint satisfaction. In S. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 285–293. John Wiley and Sons, Inc., New York, 1992.

[83] M. J. Maher and P. J. Stuckey. Expanding query power in constraint logic programming languages. In *Proc. 1989 North American Conf. Logic Programming*, pages 20–36, Cleveland, OH, Oct. 1989.

[84] A. Marchetti-Spaccamela, A. Pelaggi, and D. Sacca. Worst-case complexity analysis of methods for logic query implementation. In *Proc. 6th ACM Symp. Principles of Database Systems*, pages 294–301, San Diego, CA, March 1987.

[85] D. McKay and S. Shapiro. Using active connection graphs for reasoning with recursive rules. In *Proc. 7th Int. Joint Conf. Artificial Intelligence*, pages 368–374, Vancouver, Canada, 1981.

[86] D.S. Moffat and P.M.D. Gray. Interfacing Prolog to a persistent data store. In *Proc. Third Int'l Conf. on Logic Programming*, pages 577–584, MIT Press, Cambridge, MA, 1986.

[87] S. Morishita, M. Derr, and G. Phipps. Design and implementation of the Glue-Nail database system. In *Proc. 1993 ACM-SIGMOD Conf. Management of Data*, pages 147–156, Washington, DC, May 1993.

[88] K. Morris, J.F. Naughton, Y. Saraiya, J.D. Ullman, and A. van Gelder. YAWN! (yet another window on NAIL!). In C. Zaniolo, editor, *Data Engineering 10(4)*, pages 28–43. 1987.

[89] I.S. Mumick, S.J. Finkelstein, H. Pirahesh, and R. Ramakrishnan. Magic conditions. In *Proc. of 9th ACM Symp. on Principle of Database Systems*, pages 314–330, Nashville, TN, 1990.

[90] I.S. Mumick, H. Pirahesh, and R. Ramakrishnan. The magic of duplicates and aggregates. In *Proc. 16th Int. Conf. Very Large Data Bases*, pages 264–277, Brisbane, Australia, August 1990.

[91] B. Napheys and D. Herkimer. A look at loosely-coupled Prolog/database systems. In *Proc. 2nd Int'l Conf. on Expert Database Systems*, pages 107–115, 1988.

[92] S. Naqvi. Negative queries in horn databases. In *Proc. 1st Int. Conf. Expert Database Systems*, pages 227–236, Charleston, SC, April 1986.

[93] S. Naqvi and S. Tsur. *A Logic Language for Data and Knowledge bases.* Computer Science Press, Orckville, MD, 1988.

[94] J. F. Naughton. Compiling separable recursions. In *Proc. 1988 ACM-SIGMOD Int. Conf. Management of Data*, pages 312–319, Chicago, IL, June 1988.

[95] J. F. Naughton. Minimizing function-free recursive inference rules. *J. ACM*, 36:69–91, 1989.

[96] J. F. Naughton, R. Ramakrishnan, Y. Sagiv, and J. D. Ullman. Efficient evaluation of right-, left-, and multi-linear rules. In *Proc. 1989 ACM-SIGMOD Int'l Conf. Management of Data*, pages 235–242, Portland, Oregon, June 1989.

[97] J. F. Naughton and Y. Sagiv. A decidable class of bounded recursions. In *Proc. 6th ACM Symp. Principles of Database Systems*, pages 214–226, San Diego, CA, March 1987.

[98] F.C.N. Pereira and D.H.D. Warren. Parsing as deduction. In *Proc. Twenty-first Annl. Meeting of the Assn. for Computational Linguistics*, pages 137–144, 1983.

[99] G. Phipps, M.A. Derr, and K. Ross. Glue-NAIL!: A deductive database system. In *Proc. 1991 ACM-SIGMOD Int. Conf. Management of Data*, pages 308–317, Denver, CO, June 1991.

[100] L. Plümer. Termination proofs for logic programs based on predicate inequalities. In *Proc. 7th Int. Conf. on Logic Programming*, pages 634–648, Jerusalem, 1990.

[101] H. Przymusinska and T.C. Przymusinski. Weakly perfect model semantics for logic programs. In *Proc. of the Fifth Int'l Conf./Symposium on Logic Programming*, pages 1106–1120, Seattle, WA, Sept. 1988.

[102] T.C. Przymusinski. On the declarative semantics of stratified deductive databases. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, 1988.

[103] R. Ramakrishnan. Magic templates: A spellbinding approach to logic programs. *J. Logic Programming*, 11:189–216, 1991.

[104] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Controlling the search in bottom-up evaluation. In *Proc. of the Joint Int'l Conf. and Symposium on Logic Programming*, pages 273–287, MIT Press, Cambridge, MA, 1992.

[105] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Coral - control, relations and logic. In *Proc. 18th Int. Conf. Very Large Data Bases*, pages 547–559., Vancouver, Canada, August 1992.

[106] R. Ramakrishnan, D. Srivastava, S. Sudarshan, and P. Seshadri. Implementation of the CORAL deductive database system. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 147–156, Washington, DC, May 1993.

[107] R. Ramakrishnan and J. D. Ullman. A survey of research on deductive database systems. In *J. Logic Programming*, to appear, 1994.

[108] J. Robinson. A machine oriented logic based on the resolution principle. *J. ACM*, 12:23–41, 1965.

[109] J. Rohmer, R. Lescoeur, and J. M. Kerisit. The Alexander method: A technique for the processing of recursive queries in deductive databases. *New Generation Computing*, 4:273–285, 1986.

[110] K. Ross. Modular stratification and magic sets for datalog programs with negation. In *Proc. of the ACM Symposium on Principles of Database Systems*, pages 161–171, Nashville, TN, 1990.

[111] K. Ross and Y. Sagiv. Monotonic aggregation in deductive databases. In *Proc. 11th ACM Symp. Principles of Database Systems*, pages 114–126, San Diego, CA, June 1992.

[112] D. Sacca and C. Zaniolo. The generalized counting method for recursive queries. In *Proc. 1st Int. Conf. Database Theory*, pages 31–53, Rome, Italy, 1986.

[113] D. Sacca and C. Zaniolo. On the implementation of a simple class of logic queries for databases. In *Proc. Fifth ACM Symp. on Principles of Database Systems*, pages 16–23, Cambridge, MA, 1986.

[114] D. Sacca and C. Zaniolo. Implementation of recursive queries for a data language based on pure Horn logic. In *Proc. Fourth Int'l Conf. on Logic Programming*, pages 104–135, MIT Press, Cambridge, 1987.

[115] D. Sacca and C. Zaniolo. Magic counting methods. In *Proc. 1987 ACM-SIGMOD Int. Conf. Management of Data*, pages 49–59, San Francisco, CA, May 1987.

[116] D. Sacca and C. Zaniolo. Differential fixpoint methods and stratification of logic programs. In *Proc. 3rd Int'l Conf. Data and Knowledge Bases*, pages 49–58, Jerusalem, Israel, June 1988.

[117] D. Sacca and C. Zaniolo. Stable models and non-determinism in logic programs with negation. In *Proc. of the 9th ACM Symposium on Principles of Database Systems*, Nashville, TN, 1990.

[118] K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In *Proc. ACM SIGMOD Int. Conf. Management of Data*, pages 442–453, Minneapolis, MN, 1994.

[119] P. Selinger, D. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access path selection in a relational database management system. In *Proc. 1979 ACM-SIGMOD Int. Conf. Management of Data*, pages 23–34, Boston, MA, May 1979.

[120] S. Shaw, L. Foggiato-Bish, I. Garcia, G. Tillman, D. Tryon, W. Wood, and C. Zaniolo. Improving data quality via LDL++. In *Proc. of the ILPS'93 Workshop on Programming with Logic Databases*, pages 60–73, Vancouver, 1993.

[121] O. Shmueli. Decidability and expressiveness aspects of logic queries. In *Proc. 6th ACM Symp. on Principles of Database Systems*, pages 237–249, San Diego, 1987.

[122] A. Silberschatz, M. Stonebraker, and J. D. Ullman. Database systems: Achievements and opportunities. *Comm. ACM*, 34:94–109, 1991.

[123] K. Sohn and A. van Gelder. Termination detection in logic programs using argument sizes. In *Proc. 10th ACM Symp. Principles of Database Systems*, pages 216–226, Denver, CO, May 1991.

[124] D. Srivastava and R. Ramakrishnan. Pushing constraint selections. *J. Logic Programming*, 16:361–414, 1993.

[125] D. Srivastava, R. Ramakrishnan, P. Seshadri, and S. Sudarshan. Coral++: Adding object-orientation to a logic database language. In *Proc. 19th Int. Conf. Very Large Data Bases*, pages 158–170, Dublin, Ireland, August 1993.

[126] L. Sterling and E. Shapiro. *The Art of Prolog.* The MIT Press, 1986.

[127] M. Stonebraker. Implementation of integrity constraints and views by query modification. In *Proc. 1975 ACM-SIGMOD Int. Conf. Management of Data*, pages 65–78, 1975.

[128] P. Stuckey and S. Sudarshan. Compiling query constraints. In *Proc. Symp. on Principles of Database Systems*, pages 56–67, Minneapolis, MN, May, 1994.

[129] R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM J. Computing*, 1:146–160, 1972.

[130] S. Tsur and C. Zaniolo. LDL: A logic-based data-language. In *Proc. 12th Int. Conf. Very Large Data Bases*, pages 33–41, Kyoto, Japan, Aug. 1986.

[131] J. D. Ullman. Implementation of logical query languages for databases. *ACM Trans. Database Syst.*, 10:289–321, 1985.

[132] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Vol. 1.* Computer Science Press, 1988.

[133] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Vol. 2.* Computer Science Press, 1989.

[134] J. D. Ullman and A. van Gelder. Efficient tests for top-down termination of logical rules. *J. ACM*, 35:345–373, 1988.

[135] J. Vaghani, K. Ramamohanarao, D. Kemp, Z. Somogyi, and P. Stuckey. An introduction to the ADITI deductive database system. *Australian Computer Journal*, 23:37–52, 1991.

[136] A. van Gelder. A message-passing framework for logical query evaluation. In *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, pages 155–165, Washington, DC, 1986.

[137] A. van Gelder. Negation as failure using tight derivations for general logic programs. In *Proc. of the Symposium on Logic Programming*, pages 127–139, Salt Lake City, Utah, 1986.

[138] A. van Gelder. The well-founded semantics of aggregation. In *Proc. of the ACM Symposium on Principles of Database Systems*, pages 127–138, San Diego, CA, 1992.

[139] A. van Gelder, K. Ross, and J.S. Schlipf. Unfounded sets and well-founded semantics for general logic programs. *J. ACM*, 38:620–650, 1991.

[140] P. van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.