Top-Down beats Bottom-Up for Constraint Extensions of Datalog

David Toman
Dept. of Comp. and Info. Sci.
Kansas State University, Manhattan, KS–66506
david@cis.ksu.edu

Abstract

This paper proposes an efficient method for evaluating queries over constraint databases. The method is based on a combination of top-down resolution with memoing and closed form bottom-up evaluation. In this way top-down evaluation terminates for all queries for which the bottom-up evaluation also terminates. The main advantage of the proposed method is the direct use of partially instantiated queries without the need for rewriting of the original program. The evaluation algorithm automatically propagates the necessary constraints during the computation. In addition, top-down evaluation potentially allows the use of compilation techniques developed for compilers of logic programming languages, which can make query evaluation very efficient.

1 Introduction

We propose a new method for evaluating deductive queries over constraint databases (i.e., where the constraints are used to represent the information stored in *generalized relations* [7]). The evaluation of queries in constraint databases is different from the one used in standard database systems. The constraints are used as the actual representation of the data stored in the database rather than as mere restrictions of the contents of otherwise ground relations. Algorithms for the evaluation of queries over constraint databases should satisfy the following criteria:

1. The evaluation algorithm should terminate on all possible input queries.
2. The constraint language should try to encompass a wide class of constraints over particular domains.
3. Partially instantiated queries have to be evaluated efficiently.

The first requirement is especially difficult to achieve in the case of constraint databases because the extensions of constraint relations are usually infinite. There exist two main approaches to satisfy the above requirements in the case of Datalog. But neither of them seems to address all three requirements.

The first approach is based on the fixpoint (bottom-up) evaluation of the rules. Here the first condition is usually met (e.g., for Datalog [23], Datalog with integer constraints [14, 22], and sets [15]). However, the classes of allowed constraints are restricted to guarantee termination. Also, the evaluation process is not *goal-oriented* and thus the evaluation of partially instantiated queries is quite inefficient[1].

The other approach is based on resolution (top-down) method. Here the second and third conditions are usually met. However, termination for all queries is often sacrificed (an exception is [18], where no constraints are allowed) in order to improve expressiveness and efficiency. These methods can also take full advantage of compilation techniques developed for other logic programming languages, e.g.,

[1] Application of program transformation techniques does not solve this problem in general.

[19, 24]. This greatly improves the (practical) efficiency of query evaluation [18] in the case of Datalog. We show that similar results can be achieved in the case of constraint databases as well.

In this paper we try to combine the advantages of both approaches. We propose an evaluation method, *Constraint Memoing*, applicable to constraint-based extensions of Datalog (Datalog$^{\mathcal{C}}$), that has the following features:

- Termination. Constraint Memoing guarantees termination of queries for all classes of constraints that have a closed-form terminating bottom-up evaluation procedure. Also, the complexity bounds of the bottom-up procedure are preserved.
- The expressiveness of the language can be easily extended to accommodate various classes of constraints as long as every class of constraints is equipped with several elementary operations on the underlying representations of the constraints. This step is quite subtle if the termination of queries is to be preserved. In contrast to bottom-up methods, it is also possible to extend the query language to classes of constraints, where termination is not guaranteed[2].
- The use of a top-down method allows a fully goal-oriented evaluation (i.e., information present in the partially instantiated queries is used to prune the search space) of queries. The efficiency achieved by this method is better than the efficiency of comparable bottom-up methods including program rewriting techniques (e.g., Magic Set Transformation). Also, in the future, the top-down evaluation strategy will allow a direct use of compilation techniques developed in the area of Logic Programming.

Recently, there have been several other attempts to make query evaluation in the presence of constraints efficient. There are two main directions of this research:

1. The first direction has its roots in the (deductive) database community: In [9, 12, 16, 17] techniques for pushing constraints present in the query are proposed. However, the goal of these methods is to *preprocess* the query (i.e., the goal and the rules) with respect to the given constraints for subsequent bottom-up evaluation. We present a completely different evaluation strategy where the constraints are propagated dynamically without the need for the preprocessing of the query. Also, in the standard database approach, the constraints are considered only as *conditions* that restrict the otherwise ground answer. Constraint Memoing uses constraints as a tool for *representing* both the data computed by the queries and stored in the database itself (i.e., non-ground relations are allowed). This dramatically increases the expressive power of the query language while preserving termination and efficiency.
2. The other direction is pursued in the area of (general) Logic Programming: In [4, 6, 10] top-down evaluation for constraint logic programs is proposed. However, in all cases, general constraint solving procedures are used. Thus these methods are not directly useful for query evaluation in constraint databases as the termination can not be guaranteed. The closest to our work is [4]. However, the method proposed there allows only propagation of constants (i.e., constraints of the form $x = a$); the *constraint* part of the query is essentially computed bottom-up. Our approach allows full propagation of all possible constraints during the whole evaluation process (we compare the achieved efficiency of the two methods in section 5).

In [23] the bottom-up approach (equipped with a query transformation phase) is shown to be no worse to the top-down approach for restricted classes of Datalog programs over ground relations. We show that top-down approach is no worse than the bottom-up approach in the worst case, and in many empirical examples top-down evaluation is much faster than bottom-up evaluation of the same query.

[2]Even in cases when termination can not be guaranteed, the algorithm reduces the possibility of non-termination.

The rest of the paper is organized as follows: section 2 introduces a closed form bottom-up evaluation procedure for Datalog$^{\mathcal{C}}$ and also presents the main ideas behind the goal-oriented evaluation procedures. Section 3 describes the proposed evaluation method, Constraint Memoing, and possible optimization techniques, that are specific to processing constraint queries. Section 4 introduces a general Magic Set Transformation (MST$^{\mathcal{C}}$) for Datalog$^{\mathcal{C}}$ for comparison purposes. Section 5 studies both the analytical complexity of query evaluation using Constraint Memoing and gives results that provide empirical evidence of usefulness of the proposed evaluation method. Section 6 concludes the presentation with possibilities of further improvements and directions for further research.

2 Preliminaries

This section introduces the basic building blocks in terms of which the evaluation of Datalog$^{\mathcal{C}}$ queries is defined. Also, for reference, the standard bottom-up query evaluation procedure is introduced in terms of such building blocks.

Definition 2.1 *Let $\mathcal{C}_0$ be a set of satisfiable atomic constraints. We define $\mathcal{C}$ to be the least set of constraints defined by following rules:*

1. true $\in \mathcal{C}$
2. $\mathcal{C}_0 \subseteq \mathcal{C}$
3. *if $C_1, C_2 \in \mathcal{C}$ and $C_1 \wedge C_2$ is satisfiable then $C_1 \wedge C_2 \in \mathcal{C}$*
4. *if $C \in \mathcal{C}$ and $x \in FV(C)$ then there exists a quantifier free formula $C_1 \vee \ldots \vee C_k$ (in DNF) equivalent to $\exists x.C$ such that $C_i \in \mathcal{C}$ for every satisfiable C_i where $0 < i \leq k$.*
5. *if $C \in \mathcal{C}$ and θ is a renaming of variables then $C\theta \in \mathcal{C}$.*

This definition is similar to the definition of Constraint Domain [5]. However, $\mathcal{C}$ contains only satisfiable constraints. The elements of $\mathcal{C}$ are used as a finite representation of the (possibly infinite) relations in a constraint database. The query evaluation over such representation is based on the following operations:

Definition 2.2 (Constraint Class) *A* Constraint class *is a set of constraints $\mathcal{C}$ from Definition 2.1 equipped with the following (computable) operations:*

Constraint Conjunction $\wedge^{\mathcal{C}} : \mathcal{C} \times \mathcal{C} \to \mathcal{C} \cup \{\perp\}$ *that for every pair of constraints C_1, C_2 computes the* conjunction $C_1 \wedge C_2$ *if the conjunction is satisfiable; otherwise returns $\perp$.*

Constraint Projection $\exists^{\mathcal{C}} : V \times \mathcal{C} \to 2^{\mathcal{C}}$ *that for every constraint C and every set $V \subseteq FV(C)$ computes the set $\{C_i\}$ (a* projection *of C to V) that satisfies the condition*

$$\left(\bigvee_{C_i \in \exists^{\mathcal{C}}(V,C)} C_i \right) \equiv \exists x_1. \cdots \exists x_l.C$$

where $x_j \in FV(C) - V$ for $0 < j \leq l$. Note that by Definition 2.1 the function is well defined and always returns a finite *subset of $\mathcal{C}$.*

Constraint Subsumption $\leq^{\mathcal{C}} : \mathcal{C} \times \mathcal{C} \to$ bool *that satisfies following condition:*

$$C_1 \leq^{\mathcal{C}} C_2 \textit{ implies } C_1 \supset C_2$$

The first two operations are, in the world of constraints, the equivalents of relational algebra join and projection. However, while in the case of ground tuples (represented using equality constraints) constraint projection returns always only one constraint (tuple); in the case of more general constraints (e.g., [22]) constraint projection can return a set with more than one element. The last operation, constraint subsumption, is used for elimination of duplicate answers, and in the case of Datalog reduces to checking for equality. Note, that the $\leq^{\mathcal{C}}$ is not unique by definition: the $\supset$ does not have to imply $\leq^{\mathcal{C}}$. However, a *better approximation* of $\supset$ relation by the $\leq^{\mathcal{C}}$ operation reduces the number of possible duplicate answers and improves the efficiency of the evaluation methods. In the following text we omit the superscripts $^{\mathcal{C}}$. We also use a *strict*[3] version of $\wedge^{\mathcal{C}}$. The following definition states a fundamental property of constraint classes on which the termination proofs of bottom-up query evaluation procedures are implicitly based [14, 15, 22, 23].

Definition 2.3 (Constraint-compact Class of Constraints) *Let $\mathcal{C}$ be a class of constraints. If for every finite set of variables V and for every set $\mathcal{D} \subseteq \mathcal{C}$ such that $\forall C \in \mathcal{D}.FV(C) \subseteq V$ there is a finite subset $\mathcal{D}_{fin} \subseteq \mathcal{D}$ such that $\forall C \in \mathcal{D}.\exists C' \in \mathcal{D}_{fin}.C \leq^{\mathcal{C}} C'$. (i.e., $\mathcal{D}_{fin}$ covers $\mathcal{D}$ with respect to $\leq^{\mathcal{C}}$) then $\mathcal{C}$ is called* constraint-compact.

This property plays a central role in the termination proof of the top-down query evaluation procedure developed in section 3.

Example 2.4 (Common Constraint Classes) The standard Datalog can be defined using the class of constraints generated from the set $\{x = a : a \in A\}$ where A is the set of all constants in the Datalog program.

Allowing general equality may cause problems to the standard evaluation strategies (rules may be non-range-restricted). However, in our case we simply generate the appropriate class of constraints from the set $\{x = a : a \in A\} \cup \{x = y\}$. The evaluation remains unchanged as we use more general evaluation mechanism.

Incorporation of more interesting constraints, e.g., constraints over integers (Z) is also easy: the *gap-order* constraints [14] are generated from the set $\mathcal{C}_{<Z} = \{x < u : u \in A\} \cup \{u < x : u \in A\} \cup \{x + c < y : c \in Z^+\}$. Similarly the *periodicity* constraints are generated from $\mathcal{C}_{\equiv Z} = \{x \equiv_k c : c \in A\}$. In [22] a closed form bottom-up evaluation procedure for $\mathcal{C}_{\equiv,<Z} = \mathcal{C}_{<Z} \cup \mathcal{C}_{\equiv Z}$ was developed including all the constraint operations as in Definition 2.2. We can also incorporate the *dense order* constraints over Q by a slight modification of constraint operations defined in [7]. All the above constraint classes are constraint-compact. However, there are also constraint classes where all the constraint operations are defined, but which are *not* constraint-compact, e.g., the class generated from the set $\{x + c < y : c \in Z\}$ (gap-order constraints with possibly negative size of the gap [14]) or the linear arithmetic constraints [7]. For such constraint classes termination can not be guaranteed for any query evaluation method. However, Constraint Memoing tries to eliminate the number of non-terminating queries in this case.

Definition 2.5 (Datalog$^{\mathcal{C}}$ Program) *Let $\mathcal{C}$ be a class of constraints. A* atom *is a predicate symbol with distinct variables as its arguments. A* Datalog$^{\mathcal{C}}$ *is a set of clauses of the form*

$$A \leftarrow D, B_1, \ldots, B_k$$

where A and B_i are atoms and $D \in \mathcal{C}$.

We assume that the extensional database (EDB) is represented by a set of unit clauses $A \leftarrow D$ as a part of the constraint database. A query over such database is a tuple containing an atom and a constraint the returned tuples have to satisfy. The database itself is included as the last component of the query.

Definition 2.6 (Query) *Let P be a Datalog$^{\mathcal{C}}$ program, G an atom, and $C \in \mathcal{C}$. We call the tuple (G, C, P) a* query. *The* answer *to the query (G, C, P) is a set of valuations θ such that $P \models (G \wedge C)\theta$. A* query evaluation procedure *is an algorithm that computes an answer to the query. A query is* partially instantiated *if the constraint C is nontrivial (i.e., $C \neq$ true).*

A query (G, C, P) could be written as a SQL query `select * from` G `where` C that is evalueted in the database P.

[3] I.e., $\perp$-preserving.

2.1 Closed-form Bottom-up Evaluation

The usual approach to query evaluation for Datalog$^{\mathcal{C}}$ is a variation on the bottom-up evaluation algorithm [23]. In its simplest form a bottom-up evaluation algorithm is defined as follows:

Definition 2.7 (Interpretation) *Let $R(x_1,\ldots,x_k)$ be an atom and $C \in \mathcal{C}$ a constraint such that $FV(C) \subseteq FV(R)$. A pair (R,C) is a* named constraint*. A $\mathcal{C}$-*interpretation *is a set of named constraints.*

Named constraints play the role of ground atoms (tuples) stored in a standard relational system. The definition of the TP operator is now similar to the definition of the operator on ground atoms. However, in this case all the operations in the definition of TP are defined with respect to the chosen class of constraints $\mathcal{C}$ (see Definition 2.2).

Definition 2.8 (Immediate Consequence Operator) *Let I be a $\mathcal{C}$-interpretation and P a Datalog$^{\mathcal{C}}$ program. We define*

$$\begin{aligned}\mathrm{TP}_{\mathcal{C}}(I) = I \cup \{(A,C') :\ & A \leftarrow D, B_1,\ldots,B_k \in P,\\ & (B_i,C_i) \in I \textit{ for all } 0 < i \leq k, D \in \mathcal{C},\\ & C = D \wedge C_1 \wedge \ldots \wedge C_k \textit{ exists}, C' \in \exists_A.C,\\ & \textit{and } C' \not\leq C'' \textit{ for any } (A,C'') \in I\}\end{aligned}$$

where $\exists_A$ is a shorthand for $\lambda x.\exists^{\mathcal{C}}(FV(A),x)$. The variables in the constraints are renamed using the variable names in the associated atoms of the clause.

The bottom-up evaluation algorithm remains unchanged (i.e., all the modifications needed for the evaluation of *constraint* queries are encapsulated in the definition of the TP operator).

Algorithm 2.9 (Naive Bottom-up Evaluation) *Let (G,C,P) be a query.*

$I := \emptyset$
repeat $J, I := I, \mathrm{TP}_{\mathcal{C}}(I)$ *while* $J \neq I$
return $\{C \wedge D : (G,D) \in I\}$

This arrangement also shows how other evaluation procedures based on TP can be utilized for constraint query evaluation (e.g., the semi-naive bottom-up evaluation [23]). Algorithm 2.9 was shown to be sound and complete for Datalog [23] and for Datalog$^{<,\equiv Z}$ [14, 22]. A simple generalization of the proofs in [22] shows soundness and completeness for a general class $\mathcal{C}$.

Theorem 2.10 (Termination) *Let $\mathcal{C}$ be a constraint-compact class of constraints. Then Algorithm 2.9 terminates for every query.*

All the constraint classes in Example 2.4 have a closed-form terminating bottom-up evaluation procedure (based on definitions 2.8 and 2.9).

2.2 Goal-oriented Evaluation Strategies

There are several standard improvements to the naive bottom-up evaluation algorithm, e.g., the semi-naive algorithm [23]. However, such strategies fail to take into account the information contained in a partially instantiated query (i.e., they are not goal-oriented). There are two different approaches to solving this problem in the framework of standard Datalog:

1. Rewrite the program using Magic Set Transformation (MST) [1, 12] and subsequently evaluate the transformed program bottom-up, and

2. Adopt a top-down evaluation strategy [3, 20] that is based on the resolution principle [11].

In this paper we consider mainly top-down resolution-based methods. However, the MST optimization for constraint deductive query languages will also be introduced for comparison purposes. It is well known that the standard top-down strategies, e.g., SLD-resolution [3], despite their efficiency have a major drawback as query evaluation procedures: they lead to nontermination even in situations where bottom-up algorithms are guaranteed to terminate. Note also that breadth-first traversal of a SLD-tree does not guarantee termination in general.

The drawback is caused by the occurrence of infinite paths in SLD search trees. This has been observed in several papers, e.g., [20, 21] and an alternative to SLD-resolution was proposed (under various names). The main idea consists of *remembering* answers for already resolved subgoals. Such method also guarantees termination in the case of function-free logic programs [18]. We extend this method to *constraint* deductive queries while preserving termination and complexity bounds of bottom-up evaluation.

3 Top-down evaluation for Datalog$^{\mathcal{C}}$

In the last section, bottom-up evaluation of Datalog$^{\mathcal{C}}$ was defined in terms of elementary operations over a specific class of constraints. This section shows, how a top-down evaluation procedure (SLG-resolution [20]) can be refined using the same operations to handle constraint queries. This approach allows us to build a very efficient top-down evaluation procedure for every class of constraints that has a closed form bottom-up evaluation procedure. Moreover the termination property of the bottom-up algorithm is preserved.

The modification of SLG resolution for Constraint Memoing (SLG$^{\mathcal{C}}$ resolution) is defined by the following set of rewriting rules:

Definition 3.1 (SLG$^{\mathcal{C}}$ rewriting rules)

Action/Node	Children	Conditions
$\boxed{\text{Clause resolution}}$ $\mathsf{root}(G;C)$	$\begin{cases}\mathsf{body}(G;B_1^1,\ldots,B_{k_1}^1;C\wedge D^1)\\ \vdots \\ \mathsf{body}(G;B_1^l,\ldots,B_{k_1}^l;C\wedge D^l)\end{cases}$	for all $0<i\le l$ such that $G\leftarrow D^i,B_1^i,\ldots,B_{k_i}^i\in P$ and $C\wedge D^i$ is satisfiable
$\boxed{\text{Query projection}}$ $\mathsf{body}(G;B_1,\ldots,B_k;C)$	$\begin{cases}\mathsf{goal}(G;B_1,C_1;B_2,\ldots,B_k;C)\\ \vdots \\ \mathsf{goal}(G;B_1,C_l;B_2,\ldots,B_k;C)\end{cases}$	for all $C_i\in\exists_{B_1}.C$
$\boxed{\text{Answer propagation}}$ $\mathsf{goal}(G;B,C';B_2,\ldots,B_k;C)$	$\begin{cases}\mathsf{body}(G;B_2,\ldots,B_k;C\wedge A_1)\\ \vdots \\ \mathsf{body}(G;B_2,\ldots,B_k;C\wedge A_l)\end{cases}$	for all $A_i\in\mathrm{ans}(B,C'')$ where $C'\le_{\mathcal{C}}C''$ and $C\wedge A_i$ satisfiable
$\boxed{\text{Answer projection}}$ $\mathsf{body}(G;\ ;C)$	$\begin{cases}\mathsf{ans}(G;A_1)\\ \vdots \\ \mathsf{ans}(G;A_l)\end{cases}$	for all $A_i\in\exists_G.C$

where G, B_i^j are atoms, $A_i, C, C', C'', C_i, D^i \in \mathcal{C}$, and $\mathrm{ans}(B,C'')$ is introduced in Notation 3.4.

A SLG-tree *is a tree built from a node $\mathsf{root}(G;C)$ by finite application of the above rules. A* SLG-forest *is a set of SLG-trees.*

Note that the *Answer propagation* and *Answer projection* rules have to "cooperate": every time a new answer[4] is produced in a specific SLG-tree, it is propagated to all the nodes that have already been resolved using answers from this particular tree. Also, the *Answer propagation* rule is responsible for creating new SLG-trees in the SLG-forest when no tree with root node that subsumes the goal to be resolved can be found (see Fig. 1).

The main difference between SLG and SLG$^{\mathcal{C}}$ is in two additional rules: Query Projection and Answer Projection. The Query Projection rule is responsible for determining what are the goals to be resolved by answer resolution (i.e., what is the goal-constraint pair to be looked for among the already computed answers), and the Answer Projection is responsible for storing the computed answers for a given goal for subsequent lookup. Note the essential use of constraint projection which allows to determine the relevant constraint for every atom.

Also, SLG-resolution can handle negation (by use of additional rules). However, presence of negation together with constraints leads often to non-termination [14]. Thus our current proposal allows only positive programs. Adding negation is briefly discussed in section 6.

The SLG$^{\mathcal{C}}$ rewriting rules are used for the evaluation of queries as follows:

Definition 3.2 *Let* $\mathrm{SLG}(G, C)$ *be the SLG-forest that is generated from the query* (G, C, P) *as follows:*

1. *create a SLG-forest containing a single tree* $\{\mathsf{root}(G; C)\}$.
2. *expand the leftmost*[5] *node using the rules in Definition 3.1 as long as they can be applied.*
3. *return* $\mathrm{ans}(G, C)$ *as the answer for the query.*

Let $\mathrm{slg}(G, C)$ *be a SLG-tree rooted by the node* $\mathsf{root}(G; C)$.

We also order the answers (i.e., the nodes $\mathsf{ans}(G; C)$) in the SLG-forest according to time when a particular answer is derived:

Definition 3.3 *Let* $\mathsf{ans}(G; A)$ *and* $\mathsf{ans}(G'; A')$ *be leaves in the SLG-forest generated by the SLG-rules (cf. Definition 3.1). Then we say that* $\mathsf{ans}(G; A)$ *is* older than $\mathsf{ans}(G'; A')$ *if the node* $\mathsf{ans}(G; A)$ *is generated before the node* $\mathsf{ans}(G'; A')$.

Notation 3.4 *Let* $\mathrm{ans}(G, C)$ *be the set of all* A *such that* $\mathsf{ans}(G; A) \in \mathrm{slg}(G, C)$ *and if an older* $\mathsf{ans}(G; A') \in \mathrm{slg}(G, C)$ *then* $A \not\leq A'$.

Lemma 3.5 *Let* $\mathsf{ans}(G; A)$ *be a leaf of the SLG-tree* $\mathrm{slg}(G, C)$. *Then for every application of the answer propagation rule along the path* $\mathsf{root}(G; C) \to \ldots \to \mathsf{ans}(G; A)$ *if* $\mathsf{ans}(G'; A')$ *was the answer propagated by the rule then* $\mathsf{ans}(G'; A')$ *is older than* $\mathsf{ans}(G; A)$.

Soundness and completeness of Constraint Memoing is proven by a reduction to soundness and completeness of bottom-up evaluation (Algorithm 2.9).

Notation 3.6 *Let* $S \subseteq \mathcal{C}$. *We denote* $\|S\|$ *the set of valuations* θ *such that* $\theta \models C$ *for some* $C \in S$.

Note that the set $\mathrm{ans}(G, C)$ may not be unique (depending on the order in which the nodes $\mathsf{ans}(G; A)$ are generated). However, it is sufficient for our purposes that the set of valuations $\|\mathrm{ans}(G, C)\|$ is unique.

Lemma 3.7 *Let* $\mathrm{slg}(G, C)$ *be a SLG-tree and* ϑ *a valuation. Let* $\vartheta \in \|\mathrm{ans}(G, C)\|$. *Then* $\vartheta \models C$.

[4] I.e., an answer not subsumed by any answer in the same tree generated earlier.

[5] Assuming the SLG-trees are ordered in the same order they were created. However, the choice of selection strategy does not affect the soundness/completeness of the algorithm.

To prove correctness of the algorithm, we show that all the derived answers are also derived in the bottom-up computation:

Lemma 3.8 *Let (G_0, C_0, P) be a query. Then for every* $\mathrm{slg}(G, C) \in \mathrm{SLG}(G_0, C_0)$ *and every valuation* ϑ

$$\vartheta \in \| \operatorname{ans}(G, C) \| \implies G\vartheta \in \mathrm{TP}^{\omega}(\emptyset)$$

P r o o f: Induction on the "age" of answers.

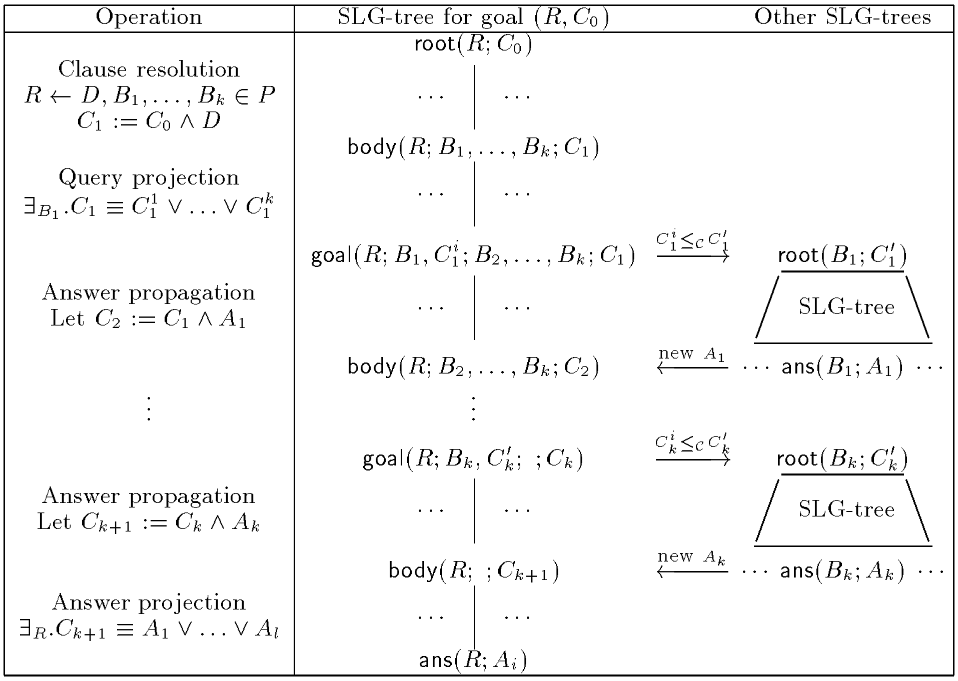

Figure 1: $\mathrm{SLG}^{\mathcal{C}}$ Evaluation of goal R w.r.t. constraint C_0.

Thus all answers—not only for the main query, but also for all subqueries represented by the remaining SLG-trees in the SLG forest generated from the main query—are sound.

Lemma 3.9 *Let G be a atom and $C_1, C_2 \in \mathcal{C}$ be constraints. Then*

$$C_1 \leq C_2 \implies \| \operatorname{ans}(G, C_1) \| \subseteq \| \operatorname{ans}(G, C_2) \|$$

The next thing to show is that the algorithm computes all the answers to the given query. This is a little bit more complicated, because the algorithm does not compute all the answers to an *uninstantiated* query as the bottom-up evaluation does. However,

Lemma 3.10 *Let (G_0, C_0, P) be a query. Then for every* $\mathrm{slg}(G, C) \in \mathrm{SLG}(G_0, C_0)$ *and every valuation* ϑ

$$G\vartheta \in \mathrm{TP}^{\omega}(\emptyset) \wedge \vartheta \models C \implies \vartheta \in \| \operatorname{ans}(G, C) \|$$

P r o o f: Induction on stages of TP. $G\vartheta \in \operatorname{ans}(G, C)$.

By composing the previous Lemmas we have:

Theorem 3.11 (Soundness and Completeness) *Let (G, C, P) be a query tuple. Then for all valuations ϑ*

$$G\vartheta \in \mathrm{TP}^{\omega}(\emptyset) \wedge \vartheta \models C \iff \vartheta \in \|\operatorname{ans}(G, C)\|$$

The soundness and completeness proof is based on the reduction to fixpoint computation on *ground* instances. However, to prove termination (in both bottom-up and top-down cases) a finite encoding of a potentially infinite result of the evaluation is needed [14, 22].

Theorem 3.12 (Termination) *If the class $\mathcal{C}$ is constraint-compact then the $SLG^{\mathcal{C}}$ evaluation terminates for all queries (G, C, P).*
P r o o f: Let $\mathcal{C}$ be a constraint-compact class of constraints. Then

1. The number of trees in the SLG-forest $\mathrm{SLG}(G, C)$ is finite, as there are only finitely many predicate symbols $G' \in P$ and for every predicate symbol the set $\{C_i : \mathsf{root}(G', C_i) \in \mathrm{SLG}(G, C)\}$ is finite by Definitions 2.3 and 3.1.
2. Every root node has finitely many children, as there are only finitely many clauses in P.
3. Every body node has finitely many children, as the set $\exists_G.C$ is finite for any $C \in \mathcal{C}$.
4. Every goal node as only finitely many children, as there are only finitely many elements in the set $\operatorname{ans}(G', C')$ for any atom G' and $C' \in \mathcal{C}$ by Definition 2.3.
5. Every $\operatorname{slg}(G, C)$ has finite depth, because of finite number of subgoals in the bodies of each clause in P.

Together, the rules in the Definition 3.1 can be applied only finitely many times.

Thus the termination for Constraint Memoing is guaranteed in all cases when bottom-up algorithm terminates computing a finite interpretation $\mathrm{TP}^{\omega}_{\mathcal{C}}(\emptyset)$. Moreover, it is usually easy to *decompose* the original bottom-up evaluation procedure and extract in this way the elementary operations on constraints needed for Constraint Memoing (Definition 2.2).

3.1 Optimizations

To reduce the overhead introduced by $\mathrm{SLG}^{\mathcal{C}}$ resolution (in comparison to standard SLD resolution) we explore several possibilities:

1. Solving more general goals than necessary:

Action/Node	Children	Conditions
$\boxed{\text{Query projection}}$		
$\mathsf{body}(G; B_1, \ldots, B_k; C)$	$\left\{\begin{array}{l}\mathsf{goal}(G; B_1, C_1; B_2, \ldots, B_k; C)\\ \vdots \\ \mathsf{goal}(G; B_1, C_l; B_2, \ldots, B_k; C)\end{array}\right.$	for $\exists_{B_1}.C \supset C_1 \vee \ldots \vee C_l$ for some $C_1, \ldots, C_l$

This modification may reduce the number of SLG-trees in the SLG-forest (if $|\exists_{B_1}.C| > l$). However, the propagation of constraints *at the time of goal resolution* is reduced. The soundness and completeness properties are preserved by Lemma 3.9. The termination is guaranteed similarly to Theorem 3.12.

In [4] the following version of this modification was presented:

Action/Node	Children	Conditions
$\boxed{\text{Query projection}}$		
$\mathsf{body}(G; B_1, \ldots, B_k; C) \rightarrow$	$\mathsf{goal}(G; B_1, \mathrm{true}; B_2, \ldots, B_k; C)$	none

In this case, there is only one SLG-tree per predicate symbol. On the other hand, no constraints are propagated at the time of goal resolution (the constraints are used merely to restrict the returned answers). Thus the computation essentially computes all answers to an *uninstantiated* query (similarly to bottom-up algorithm) and the performance suffers[6].

2. Mixed SLG and SLD resolution (by memoing only subset of predicate symbols present in the program).

Action/Node	Children	Conditions
$\boxed{\text{Non-tabled resolution}}$ $\mathsf{body}(G; B_1, \ldots, B_k; C)$	$\begin{cases} \mathsf{body}(G; B_1^1, \ldots, B_k; C \wedge D^1) \\ \vdots \\ \mathsf{body}(G; B_1^l, \ldots, B_k; C \wedge D^l) \end{cases}$	for B_1 not tabled goal $B_1 \leftarrow D^i, B_1^i, \ldots, B_{k_i}^i \in P$ and $C \wedge D^i$ satisfiable

This is a different way of reducing the number of SLG-trees generated by the algorithm: SLG-trees are generated only for a subset of the predicate symbols in P. The remaining symbols are resolved using program clauses (similarly to SLD-resolution). Again, soundness and completeness are preserved (by simple modification of Theorem 3.11). Termination is guaranteed iff at least one predicate is tabled for every cycle in the dependency graph of P (this follows by an easy extension of Theorem 3.12). Otherwise, an infinite branch may appear in the some of the SLG-trees. This may lead to non-termination similarly as in the case of SLD-resolution.

Also, as there is only a bounded number of SLD (non tabled) resolution steps between any two SLG (tabled) resolution steps, the bodies of the non tabled clauses can be *unfolded* in the bodies of their callers. This completely eliminates the need for non tabled resolution steps.

3. Program transformation similar to factoring. The previous folding transformation may introduce unnecessary recomputation of conjunctions of goals. This can be avoided by the use of *factoring* technique that *folds* common parts of bodies of the clauses and creates separate clauses. Note, that the recomputation is avoided by making the heads of such clauses tabled.

The last two optimizations are based on program transformation. However, in contrast to Magic Set transformation, these two transformations are completely query-independent.

4 Magic Set Transformation for $\mathcal{C}$ (MST$^{\mathcal{C}}$)

This section describes a simple version of the second approach to goal-oriented query evaluation in constraint deductive databases—Magic Set Transformation. The transformation has to be slightly modified in the context of constraint databases as follows:

Definition 4.1 *Let x be a variable and $C \in \mathcal{C}$. We say that x is* free *with respect to C if for a valuation θ if $\theta \models C$ then $\forall a \in Dom(x).\theta[x/a] \models C$. Otherwise we say that x is* bound *in C. We denote* bound(C) *the set of variables bound in C.*

Note that this definition does not take into account the mutual dependencies between different variables (in the case of ground Datalog this was not an issue: a variable could be either free or bound to a single constant). We take an optimistic approach to MST: we use only two adornments $\{b, f\}$ with the following meaning: if an argument is adorned by f then it is definitely free (at the time of the goal processing), if it is adorned by b then it *may* be bound by some constraint. Note, that the b adornment does not guarantee, that the variable is bound: consider a goal $p(x)$ and a constraint $x < y$. Then x in p is adorned by b but in fact it is not constrained at all. However, to avoid this situation, a much more complicated

[6] The performance is approximately the same as evaluating the uninstantiated query first.

analysis of the constraint query is needed (including the analysis of the data stored in the constraint database which may not be feasible in general). In the case of Datalog (i.e., the constraint class generated from the set $\{x = a : a \in A\}$) MST$^{\mathcal{C}}$ reduces to the standard MST for Datalog [1].

Definition 4.2 (Adornment) *Let G be an atom of arity k. An* adornment *A for this atom is a string over $\{b, f\}$ of length k. We say that the i-th argument of G is adorned by b (f) if the i-th position in A is b (f), respectively.*

We say that A is an adornment of G with respect to $C \in \mathcal{C}$ *if the i-th position of A is b if i-th argument of A is bound in C or the i-th position of A is f if i-th argument of A is free in C.*

Let $G \leftarrow D, B_1, \ldots, B_n \in P$ be a clause and A be an adornment of G. Then an adornment of the clause *with respect to G and A in a set of adornments $\{A_{B_1}, \ldots, A_{B_n}\}$ such that A_{B_i} is an adornment for B_i. The adornment for the atoms in the body of the clause is constructed as follows: the j-th argument of B_i is adorned by b in A_{B_i} if:*

1. *the j-th argument of B_i occurs in the head of the clause G as an argument that is adorned by b in A, or*
2. *the j-th argument of B_i is a variable bound in D, or*
3. *the j-th argument of B_i occurs as a variable in a subgoal preceding B_i.*

Otherwise the j-th argument of B_i is adorned by f.

A magic atom *for an atom G and an adornment A is the atom* magicG^A *that has only those arguments of G that are adorned by b in A (i.e., the arity of* magicG^A *is less or equal to the arity of G depending on how many arguments of G are adorned by b. Arguments adorned by f are removed).*

For simplicity, only left-to-right SIPS[7] is used (rule 3 of the previous definition). This corresponds to the selection rule used in Constraint Memoing. In both cases different selection rules may improve the efficiency of query evaluation [13]. However, in the case on MST, the SIPS is fixed during the *program transformation* phase (and also there are difficulties with combining different SIPS in one program). In the case of SLG$^{\mathcal{C}}$ evaluation, the selection rule can be adjusted during the evaluation process dynamically while preserving correctness of the answers.

Algorithm 4.3 (MST program transformation)

$$\begin{aligned}
\mathrm{mst}(G, A, P) = \{\ & G^A \leftarrow D, \mathrm{magic}G^A, B_1^{A_{B_1}}, \ldots, B_k^{A_{B_k}} \\
& \mathrm{magic}B_1^{A_{B_1}} \leftarrow D, \mathrm{magic}G^A \\
& \qquad \vdots \\
& \mathrm{magic}B_k^{A_{B_k}} \leftarrow D, B_1^{A_{B_1}}, \ldots, B_{k-1}^{A_{B_{k-1}}}, \mathrm{magic}G^A \\
& \qquad\qquad \text{for } G \leftarrow D, B_1, \ldots, B_k \in P\ \} \\
\cup & \bigcup_{0<i\leq k} \mathrm{mst}(B_i, A_{B_i}, P)
\end{aligned}$$

where $\{A_{B_1}, \ldots, A_{B_k}\}$ is the adornment of $G \leftarrow D, B_1, \ldots, B_k$ with respect to G and A and magicG^A, magic$B_1^{A_{B_1}}, \ldots,$ magic$B_k^{A_{B_k}}$ *are the magic atoms for $G, B_1, \ldots, B_k$, respectively.*

Definition 4.4 (MST for $\mathcal{C}$) *Let (G, C, P) be a query. A* MST$^{\mathcal{C}}$ *of this query is the query (G^A, C, P') where*

$$P' = \mathrm{mst}(G, A, P) \cup \bigcup_{C_i \in \exists_G . C} \{\mathrm{magic}G^A \leftarrow C_i\}$$

[7] Sideways Information Passing Style.

Test cases for (ground) Datalog:

Transitive closure (tc)	chain(N)	cycle(N)	tree(N)
`tc(X,Z):- tc(X,Y),tc(Y,Z).`	`edge(1,2).`	`edge(1,2).`	`edge(1,2).`
`tc(X,Y):- edge(X,Y).`	`...`	`...`	`...`
	`edge(N-1,N).`	`edge(N-1,N).`	`edge(N-1,2N-2).`
		`edge(N,1).`	`edge(N-1,2N-1).`

Queries: `tc(X,Y)`, `tc(1,Y)`, `tc(1,32)` for tc+chain(32), tc+cycle(32), and tc+tree(32).

Test cases for Datalog with gap-order constraints over Z:

Monotonic Path (path)	
`path(A,C,X,Z):- X<Y<Z, path(A,B,X,Y), path(B,C,Y,Z).`	
`path(A,B,X,Y):- edge(A,B,X,Y).`	
chain with unit edges (echain(N))	chain with equidistant nodes (nchain(N))
`edge(1,2,X,Y):- X<Y.`	`edge(1,2,X,Y):- 0<X<2, 1<Y<3.`
`...`	`...`
`edge(N-1,N,X,Y):- X<Y.`	`edge(N-1,N,X,Y):- N-2<X<N, N-1<Y<N+1.`

Queries: `path(A,B,X,Y)` and `path(A,B,X,Y), 5<X,Y<11` for path+echain(16)
`path(A,B,X,Y)` and `path(A,B,X,Y), -1<X,X+14<Y,Y<16` for path+nchain(16)

Figure 2: Test Programs

for A being an adornment for G with respect to C.

The only difference between the classical version of the MST and our version is in the definitions of the *free* variable and the *seed* of the magic computation, the clauses magic$G^A \leftarrow C_i$, derived by the quantifier elimination procedure from the original query.

5 Performance

In this section an analytical complexity bound for the Constraint Memoing evaluation is provided. As the complexity of top-down evaluation depends on the particular class of constraints $\mathcal{C}$ our analysis is based on the relative comparison to the (time- and space-) complexity of the bottom-up evaluation procedure. The analytical results are confirmed by the experimental results that show the performance gain achieved by Constraint Memoing.

5.1 Theory

We show, that the complexity of Constraint Memoing is no worse than complexity of the bottom-up evaluation:

Theorem 5.1 *Let* $\mathrm{TP}^{\omega}_{\mathcal{C}}(\emptyset)$ *be the result of the bottom-up evaluation of the query* (G, C, P)*. Let* $f(n)$ *be a function such that* $|\mathrm{TP}^{\omega}_{\mathcal{C}}(\emptyset)| \in O(f(|P|))$*. Then*

1. *the SLG-forest* $\mathrm{SLG}(G, C)$ *has at most* $O(f^2)$ *nodes.*
2. *there are at most* $O(f^2)$ *applications of the SLG*$^{\mathcal{C}}$ *rules in the evaluation of the query.*

P r o o f: (1) follows from the observation that every SLG-tree in the SLG forest has at most $O(f)$ leaves and thus also at most $O(f)$ nodes (the height of the trees is fixed by the number of subgoals in the clauses of P) and there are at most $O(f)$ different SLG trees in the SLG-forest (the size of $\mathrm{TP}^{\omega}_{\mathcal{C}}(\emptyset)$ limits the number of possible roots of the SLG-trees). (2) follows immediately from (1) as every application of a rule creates at least one new node in the SLG-forest.

A careful implementation needs to store only a single path (of fixed length) in every SLG-tree in the SLG forest. Thus the space requirements can be reduced to $O(f)$. On the other hand, the quadratic number of rule applications can not be avoided in general. However, by analysis of the bottom-up algorithm the number of applications of *clauses* in P is also quadratic with respect to f (the bottom-up algorithm can recompute the same element of the interpretation several times even if it is added only once).

5.2 Experimental Results

Both bottom-up (including the MST optimization) and the Constraint Memoing algorithms have been implemented[8]. Table 1 summarizes the running time (in milliseconds) for the test programs in Figure 2. The examples of (partially) instantiated queries are those, where the optimization achieves the *least* effect. The results show that while the implementations of the various evaluation methods are comparable (the results on uninstantiated queries are approximately the same), the evaluation of instantiated queries is much more efficient using Constraint Memoing. This boost is inherent to the top-down evaluation method as it is not caused by a more sophisticated implementation.

Query	Data	Naive Bottom-up	Semi-naive Bottom-up	Semi-naive with MST	Top-down
`tc(X,Y)`	tc,chain(24)	61934	31766	60134	27634
`tc(1,Y)`	tc,chain(24)	61283	31583	41333	2533
`tc(1,24)`	tc,chain(24)	61166	31567	59517	3766
`tc(X,Y)`	tc,cycle(24)	331134	273768	536466	238567
`tc(1,Y)`	tc,cycle(24)	330034	273233	278850	14100
`tc(1,24)`	tc,cycle(24)	329701	272616	306134	15917
`tc(X,Y)`	tc,tree(24)	12450	7300	15084	5800
`tc(1,Y)`	tc,tree(24)	12384	7267	13083	1450
`tc(1,24)`	tc,tree(24)	12333	7166	19967	2966
`path(A,B,X,Y)`	path,echain(16)	28317	17950	52685	14250
`path, 5<X,Y<11`	path,echain(16)	27633	17450	6832	2450
`path(A,B,X,Y)`	path,nchain(16)	40617	25283	78048	24468
`path, -1<X+14<Y<16`	path,nchain(16)	41050	25565	47347	11116

Table 1: Runing times of test queries for various evaluation procedures (in msec).

6 Conclusion

We have proposed a practical approach to evaluation of queries in generalized databases. Both the analytical and the empirical results show that Constraint Memoing is no worse than comparable bottom-up method (and in many cases the practical performance is much better even when using a very naive implementation). The performance of the Constraint Memoing can be boosted by utilizing the compilation methods developed in [19] and performance similar to ground Datalog can be expected.

Future research in this area will focus on following issues:

Compilation of constraints. To achieve an efficient implementation of Constraint Memoing, data structures for efficient representation of the constraints have to be developed. There are two main differences to be addressed:

- In most cases, the size of the constraint representation is bounded (with respect to the arity of an atom). However, general Logic Programming engines

[8] We use a straightforward implementation of all the algorithms in Sicstus Prolog.

allow unbounded terms to be built. Exploring this property may lead to an efficient *stack*-based implementation (i.e., without heap) of the evaluation procedure.

- On the other hand, classical Logic Programming assumes that every (logical) variable is either free or bound to a single term (and this binding can be changed only by backtracking). This assumption is no longer valid in the presence of constraints as more restrictive conditions may be derived after the variable was originally bound. Also, the constraints specify complex relations *between* individual variables, which is not possible in the standard approach.

Development of such representation enables building of very efficient query evaluation engines based on partial evaluation of the atomic constraints in a given class (similarly to the WAM abstract code [24]).

Analysis of binding patterns. Similarly to the MST transformation, the queries can be analyzed to determine the *flow of information* in clause bodies [13]. This is a considerably more complicated task in the presence of constraints: it is no longer sufficient to focus on single variables; the relationships between groups of variables have to be taken into consideration (as noted in Section 4). Also, the assumption, that all EDB relations are ground (i.e., after resolution of an EDB goal all variables are bound to constants) is no longer valid—the generalized relations store representation of sets of tuples that may be infinite. Such analysis can be used for several purposes: query optimization (MST-like rewriting), optimization of access to the constraint database (indexing), goal reordering, etc.

Interface to an existing RDBMS. As the constraints can be finitely encoded, their representation can be stored as tuples in a standard relational database system. However, query evaluation has to be carried out with respect to the semantics of such encoding (i.e., to perform, e.g., join, of two constraint relations, we can not use the join operation of the underlying RDBMS directly). We propose that the top-down evaluation procedure be used as a front-end built on top of a standard relational DBMS. The techniques proposed above can be directly applied here.

Negation. Adding negation to Datalog$^{\mathcal{C}}$ in such a way that termination is preserved, is a nontrivial task: adding negation often leads immediately to Turing completeness. Essentially, adding negation (and preserving termination) would require the constraint class to be closed under negation (complementation) while preserving constraint-compactness. This condition is easy to satisfy in the case of finite domain constraints. However, for constraints over infinite domains (e.g., integers) this is often not possible (e.g., gap-order constraints over integers [14]).

Storage and access methods. To achieve an efficient implementation of constraint databases, new storage management techniques have to be developed: access methods suitable for fast retrieval of the stored information, efficient updates of generalized relations, indexing techniques [8], etc.

Benchmarks. The performance of various implementations of Logic Programming languages (e.g., Prolog) is often judged by the performance on a standard benchmarks (e.g., `nrev`). We propose to develop similar benchmarks for query evaluation methods in constraint databases. The benchmarks should be independent of the particular class of constraints. Such test suite would allow to compare performance of various query evaluation methods.

References

[1] Bancilhon F., Maier D., Sagiv Y., Ullman J. D. Magic sets and other strange ways to implement logic programs. In *Proc. ACM Symposium on Principles of Database Systems*, 1986.

[2] Bol, R, Degerstedt, L. The Underlying Search for Magic Templates and Tabulation. Proc. *International Conference on Logic Programming*, 1993.

[3] Clocksin, W. F., Mellish, C.S. Programming in Prolog. Springer Verlag 1987.

[4] Gao, H., Warren, D. S. A Powerful Evaluation Strategy For CLP Programs. Proc. *Intl. Workshop on Principles and Practice of Constraint Programming*, 1993.

[5] Jaffar J., Maher, M. J. Constraint Logic Programming: A Survey. J. Logic Programming 1994, 19. 20:503–581.

[6] Johnson, M. Memoization in Constraint Logic Programming. Proc. *Intl. Workshop on Principles and Practice of Constraint Programming*, 1993.

[7] Kanellakis, P. C., Kuper, G. M., Revesz, P.Z . Constraint Query Languages, Proc. *9th ACM Symposium on Principles of Database Systems,* pp 299–313, 1990.

[8] Kanellakis, P. C., Ramaswamy, S., Vengroff, D. E., Vitter, J. S. Indexing for Data Models with Constraints and Classes. Proc. *ACM Symposium on Principles of Database Systems,* pp 233–243, 1993.

[9] Kemp, D. B., Stuckey, P. J. Analysis based constraint query optimization. Proc. *International Conference on Logic Programming*, 1993.

[10] Lim, P., Stuckey, P.J. A Constraint Logic Programming Shell. PLILP 1990.

[11] Lloyd, J. Foundations of Logic Programming. Springer-Verlag, 1987.

[12] Mumick, I. S. Query Optimization in Deductive and Relational Databases. PhD thesis, Stanford University, 1991.

[13] Ramakrishnan, R. Magic Templates. A spellbinding approach to logic programs. J. Logic Programming 1991, 11:189–216.

[14] Revesz, P. Z. A Closed Form Evaluation for Datalog Queries with Integer (Gap)-Order Constraints. *Theoretical Computer Science*, vol. 116, no. 1, 117–149, 1993.

[15] Srivastava, D., Ramakrishnan, R., Revesz, P. Z. Constraint Objects. Proc. *Intl. Workshop on Principles and Practice of Constraint Programming*, 218–228, 1994.

[16] Srivastava, D., Ramakrishnan, R. Pushing Constraint Selections. J. Logic Programming 1993:16:361–414, 1993.

[17] Stuckey, P. J., Sudarashan, S. Compiling Query Constraints. Proc. *ACM Symposium on Principles of Database Systems,* 1994.

[18] Swift, T., Warren, D. S. Analysis of SLG-WAM Evaluation of Definite Programs. Proc. *1994 International Logic Programming Symposium*, MIT Press, 1994, 219–235.

[19] Swift, T., Warren, D. S. An Abstract Machine for SLG Resolution: Definite Programs. Proc. *1994 International Logic Programming Symposium*, MIT Press, 1994, 633–652.

[20] Swift, T., Warren, D. S., Chen, W. Operational semantics of SLG evaluation. Tech. report, SUNY at Stony Brook, 1994.

[21] Tamaki, H., Sato, T. OLD-Resolution with Tabulation. Proc. *3-rd International Conference on Logic Programming*, 84–98, 1986.

[22] Toman, D., Chomicki, J., Rogers D. S. Datalog with Integer Periodicity Constraints. Proc. *1994 International Logic Programming Symposium*, MIT Press, 1994, 189–203.

[23] Ullman J. D. Principles of Database and Knowledge-base Systems, Vol. 1,2. Computer Science Systems, 1989.

[24] Warren, D. H. D. An Abstract Prolog Instruction Set, *Tech. Report 309*, SRI International, AI Center, Palo Alto, 1983.