DISCO: A Constraint Database System with Sets

Jo-Hag Byon and Peter Z. Revesz

Department of Computer Science and Engineering
University of Nebraska–Lincoln, Lincoln, NE 68588, USA
email: {byon,revesz}@cse.unl.edu

Abstract. This paper describes the implementation of a constraint database system with integer and set of integers data types. The system called DISCO allows Datalog queries and input databases with both integer gap-order [30] and set order constraints [31]. The DISCO query language can easily express many complex problems involving sets. The paper also presents efficient running times for several sample queries.

1 Introduction

Recently there has been much interest in constraint databases that generalize relational databases by allowing infinite relations that are finitely represented using constraint tuples (ex., [23, 3, 4, 8, 17, 21, 25, 28]).

DISCO (short for *Datalog with Integer and Set order COnstraints*) is a constraint database system being developed at the University of Nebraska. DISCO implements a particular case of constraint query languages for which a general framework was proposed in [23] analogously to the constraint logic programming framework of Jaffar and Lassez [18].

The particular type of constraints implemented in DISCO are integer gap-order and set order constraints. The incorporation of these constraints into databases was described theoretically in [30, 31, 34] but not implemented before.

DISCO combines the advantages of constraint logic programing and database systems. DISCO provides a non-procedural, logic-based query language and allows users to express the database inputs in a compact and often the only natural way, i.e., using constraints. Like many other database systems, DISCO also works by a translation to a procedural, algebraic language and a bottom-up evaluation that is guaranteed to terminate with some constraint database output. These features make DISCO applicable in various problems in computer-aided design, scientific databases and other areas where set-type data are used.

The current version of DISCO also incorporates several optimization methods. The running time of DISCO as measured on some traditional problems like boolean satisfiability is quite reasonable compared with other solutions. Since DISCO has a DEXPTIME-complete data complexity, any implementation will have some bad worst cases. (Data complexity measures the computational complexity of evaluating fixed size query programs as the input database size varies [7, 38].) However, the running times of most user queries may be faster in

the average case. Developing good benchmark problems for constraint database systems is a topic of research in the future.

The paper is organized as follows. Section 2 describes the DISCO query language. Section 3 presents the outline of the implementation of DISCO, including the data structures, the optimization methods used and the algorithm for query evaluation. In Section 4 we present the testing results on some example queries. In Section 5 we mention related work. Finally, in Section 6 we list some open problems to improve the system that we are working on.

2 The DISCO Query Language

The syntax of the query language of DISCO, denoted $Datalog^{<\mathbf{Z},\subseteq\mathbf{P(Z)}}$, is that of traditional Datalog (Horn clauses without function symbols) where the bodies of rules can also contain a conjunction of integer or set order constraints. That is, each program is a finite set of rules of the form: $R_0 :\!-\, R_1, R_2, \ldots, R_l$. The expression R_0 (the rule *head*) must be an atomic formula of the form $p(v_1, \ldots, v_n)$, and the expressions $R_1, \ldots, R_l$ (the rule *body*) must be atomic formulas of one of the following forms:

1. $p(v_1, \ldots, v_n)$ where p is some predicate symbol.
2. $v\theta u$ where v and u are integer variables or constants and θ is a relational operator $=, \neq, <, \leq, >, \geq, <_g$ where g is any natural number. For each g the atomic constraint $v <_g u$ is used as shorthand for the expression $v + g < u$.
3. $V \subseteq U$ or $V = U$ where V and U are set variables or constants.
4. $c \in U$ or $c \notin U$ where c is an integer constant and U is a set variable or constant.

Atomic formulas of the form (2) above are called gap-order constraints and of the form (3-4) are called set order constraints. In this paper we will always use small case letters for integer variables and capital letters for set variables. Set variables always stand for a finite or infinite set of integers.

Remark: In $Datalog^{<\mathbf{Z},\subseteq\mathbf{P(Z)}}$ the left hand side of any $\in, \notin$ constraint must be a constant. Without this restriction the query language is not evaluable [33].

Next we give as an example of $Datalog^{<\mathbf{Z},\subseteq\mathbf{P(Z)}}$ the query for testing the satisfiability of a propositional formula in conjunctive normal form.

Example 2.1 We assume that the input propositional formulas are in conjunctive normal form and contain only the propositional variables $x_1, x_2, \ldots$.

We describe each input propositional formula ϕ using three EDB (extensional database [37]) relations: *No_vars*, *No_clauses*, *Clause*. The unary EDB relations *No_vars* and *No_clauses* describe respectively the number of distinct variables and clauses in ϕ.

Each clause of ϕ is represented by a constraint tuple of the *Clause* EDB relation. In each constraint tuple the first argument gives the clause number and the second argument gives the elements of the clause. The second argument is a

set in which integer i (or $-i$) appears if and only if variable x_i occurs positively (or negated) in the clause.

For example, let ϕ be the propositional formula $(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4) \wedge (\neg x_3 \vee x_4)$. Then ϕ is represented by the following constraint EDB.

$No\_vars(4).$
$No\_clauses(3).$

$Clause(1, \{1, 2, 3\}).$
$Clause(2, \{-1, 2, 4\}).$
$Clause(3, \{-3, 4\}).$

We use the EDB relation *Literal* to express all the possible literals that may occur in ϕ. We will also use the EDB relation *Choice* to express the fact that either variable i or its negation is true for each propositional variable.

$Literal(\{1\}).$
$Literal(\{2\}).$
$Literal(\{3\}).$
$Literal(\{4\}).$
$Literal(\{-1\}).$
$Literal(\{-2\}).$
$Literal(\{-3\}).$
$Literal(\{-4\}).$

$Choice(1, X)$:- $1 \in X, -1 \notin X.$
$Choice(1, X)$:- $1 \notin X, -1 \in X.$
$Choice(2, X)$:- $2 \in X, -2 \notin X.$
$Choice(2, X)$:- $2 \notin X, -2 \in X.$
$Choice(3, X)$:- $3 \in X, -3 \notin X.$
$Choice(3, X)$:- $3 \notin X, -3 \in X.$
$Choice(4, X)$:- $4 \in X, -4 \notin X.$
$Choice(4, X)$:- $4 \notin X, -4 \in X.$

We also assume that the input database contains the *Next* relation with the tuples $Next(0, 1)$, $Next(1, 2)$, $Next(2, 3)$, $Next(3, 4)$. The *Next* relation is used for iterating over the number of clauses and variables. Clearly the clauses are satisfiable if and only if there is a valid and satisfying truth assignment. Hence a satisfying truth assignment will be found by first finding all possible truth assignments by iterating over the variables and making choices over each variable and second testing for each assignment whether it satisfies all the clauses.

We represent truth assignments to a set of propositional variables by a set of integer numbers where the set contains integer i if x_i is true and $-i$ if x_i is false. For example, the truth assignment in which x_1 and x_2 are true and x_3 and x_4 are false is represented by the set $\{1, 2, -3, -4\}$.

$Assgn(0, X)$.
$Assgn(j, X)$:- $Assgn(i, X), Next(i, j), Choice(j, X)$.

$Sat(0, X)$:- $No\_vars(n), Assgn(n, X)$.
$Sat(j, X)$:- $Sat(i, X), Next(i, j), Clause(j, C), Literal(V), V \subseteq C, V \subseteq X$.

$Yes(X)$:- $No\_clauses(m), Sat(m, X)$.

Let's see how the program tests that the assignment $X = \{1, 2, -3, -4\}$ is a satisfying truth assignment. Clearly X is a possible truth assingment, hence $Sat(0, \{1, 2, -3, -4\})$ will be true. Since $Next(0, 1)$ and the first clause contains variable x_1 which is true in X, $Sat(1, \{1, 2, -3, -4\})$ will also be true. Similarly we can find $Sat(2, \{1, 2, -3, -4\})$ and $Sat(3, \{1, 2, -3, -4\})$ to be true. Since we have three clauses, by the last rule $Yes(\{1, 2, -3, -4\})$ will be true. Since *Yes* is a non-empty relation the input sentence is satisfiable. □

It is shown in the earlier papers [30, 31] that Datalog with integer gap-order only $Datalog^{<\mathbf{Z}}$
and Datalog with set order constraints only $Datalog^{\subseteq \mathbf{P(Z)}}$ has a fixpoint model which coincides with the least model. We will show that the same property holds for $Datalog^{<\mathbf{Z}, \subseteq \mathbf{P(Z)}}$ queries.

Definition 2.1 Let $\mathcal{M}$ be the set of all possible ground tuples over the integers and sets of integers. Let P be a $Datalog^{<\mathbf{Z}, \subseteq \mathbf{P(Z)}}$ program and d be a constraint database. Let $\mathcal{D}$ be the set of ground tuples implied by d. The function T_P from and into $\mathcal{M}$ is defined as follows.
$T_P(\mathcal{D}) = \{t \in \mathcal{M}$: there is a rule R_0 :— $R_1, \ldots, R_k$ in P and an instantiation θ such that
$R_0\theta = t$, and $R_i\theta$ holds if R_i is a constraint and $R_i\theta \in \mathcal{D}$ otherwise for each $1 \leq i \leq k$. $\}$
Now we prove that for $Datalog^{<\mathbf{Z}, \subseteq \mathbf{P(Z)}}$ programs the least model and the least fixpoint coincide.

Theorem 2.1 For any $Datalog^{<\mathbf{Z}, \subseteq \mathbf{P(Z)}}$ program P, Least Model of P = Least Fixpoint of T_P. □

Each query evaluation in DISCO yields as output a set of combined graphs for each IDB (intensional database [37]) relation. The termination of DISCO queries follows from the termination proofs in the earlier papers [30, 31]. Therefore we can show the following:

Theorem 2.2 $Datalog^{<\mathbf{Z}, \subseteq \mathbf{P(Z)}}$ queries have DEXPTIME-complete data complexity. □

3 Implementation of the DISCO System

The DISCO database system not only combines the method of dealing with $Datalog^{<\mathbf{Z}}$ described in [30] and the method of dealing with $Datalog^{\subseteq \mathbf{P(Z)}}$ described in [31], but it improves those by the implementation of constraint generalizations

of some well-known Datalog evaluation and optimization techniques. In particular, at present the following techniques are implemented in DISCO.

3.1 Translation to Constraint Algebra

In DISCO the first step of the evaluation of any $Datalog^{<\mathbf{Z},\subseteq\mathbf{P(Z)}}$ query is the translation to a SELECT-PROJECT-JOIN like constraint algebra of the right hand side of each rule in the program. The operators in this algebra work on relations that are put into a combined graph form. That means that each constraint tuple in the input database is converted from a conjunction of integer gap-order and set order constraints to a *combined graph.*

A combined graph contains a gap-graph described in [30] and a set-graph described in [31]. The first step in the conversion is the elimination of $\in, \notin$ constraints by rewriting them into other types of constraints. For example the constraint $5 \in x$ is replaced by the constraint $\{5\} \subseteq x$. The second step is to represent the remaining gap-order constraints by a gap-graph and the remaining set-order constraints by a set-graph as in [30, 31]. The vertices of the gap-graph are integer variables or constants and the vertices of the set-graph are integer set variables. Directed edges with a nonnegative integer *gap-value* label in the gap-graph denote order constraints $(<, <_g)$ and undirected edges equality constraints $(=)$. Directed edges in the set-graph denote subset constraints $(\subseteq)$ and undirected edges equality constraints $(=)$.

Once we have this representation for the input database it is possible to evaluate any conjunctive or Datalog query by replacing the join (or project) operation by a *merge* (or *shortcut*) operation defined on combined graphs. Let t_1 and t_2 be two constraint tuples. Assume that the combined graph representation of t_1 contains the gap-graph g_1 and set-graph s_1 and that of t_2 contains g_2 and s_2. The merge operation in DISCO will apply the merge operation defined in [30] to g_1 and g_2 and the merge operation defined in [31] to s_1 and s_2. The shortcut operation in DISCO will apply either the shortcut operation defined in [30] or [31] to either g_1 or s_1 depending on whether the variables to be eliminated are integer or set type. If R_1 and R_2 are two constraint relations in combined graph form, then the generalized join will apply the merge operation to each possible pair of tuples from R_1 and R_2 and the generalized project will apply the shortcut operation to each tuple in R_1 (or R_2).

3.2 Semi-Naive Evaluation

In the naive evaluation strategy, the bodies of rules are converted into relational algebra expressions and then evaluated. The naive evaluation technique is so termed because it fails to recognize a key aspect of the growth of the relations involved in an iterative evaluation of a DISCO program. Using this strategy, we see that for each iteration, all the tuples of the relation in previous iterations are used for the next iteration in the evaluation of the query. This is extremely inefficient, because only the most recently added tuples can possibly generate any new tuples. Failing to make use of this important fact makes the naive

evaluation strategy inefficient in terms of both space and time, as the size of the intermediate tables are almost always bigger due to the presence of tuples that do not contribute to the growth of the relation corresponding to the output relations of the query. The semi-naive query evaluation technique, on the other hand, uses only the set of tuples added to a relation in the previous iteration. This set of most recently added tuples is called the "delta set". The semi-naive evaluation technique is also implemented in DISCO. This optimization technique alone gave substantial improvements in running time.

3.3 Pushing down Selection and Projection

This is done similarly to the algorithms of pushing down selection and projection in [37] for regular relational algebra expressions. The idea is that selections and projections are made as early as possible to reduce the size of the intermediate relations during the evaluation, thereby improving the overall efficiency of the query evaluation (see [37]). We found that the savings by this method alone were substantial even including the time of pushing down selection and projection. (In Section 4 the time of pushing down selection and projection is included in the total running time.) In the semi-naive evaluation strategy, the selection and projection pushing is applied to the delta relations as opposed to the entire relation.

3.4 Outline of the DISCO System

DISCO is implemented in the C++ programming language. The pseudo-code of the DISCO query processor is given at the end of this section. The main data structures used by the query processor are described below and an overview of them is given in Figure 1.

Argument Class: Each Argument is a token returned by the parser with type indicated to be an integer constant, an integer variable, a set constant, or a set variable.

Constraint Class: This class represents DISCO atomic constraints which are described in Section 2.

Predicate and Rule Classes: In DISCO a Predicate consists of a predicate name and a parenthesized, comma-separated argument list of variables or constants. A Rule consists of predicates and constraints as described in Section 2. This class contains the member function *rectify* that is used to rectify the predicates and rules similar to [37].

GapGraph, SetGraph and CombinedGraph Classes: In DISCO the data structure used to represent each combined graph is a pair of adjacency matrices with the first matrix representing the gap-graph and the second the set-graph part. The rows and columns in the first matrix correspond to variables or constants and in the second matrix only to variables. The i, jth entry in the first matrix is -2 if there is no edge from the ith vertex to the jth vertex, -1 if they are equal and a non-negative integer if there is an edge from i to j with the same gap-value. The i, jth entry in the second matrix is -2 if i and j are not

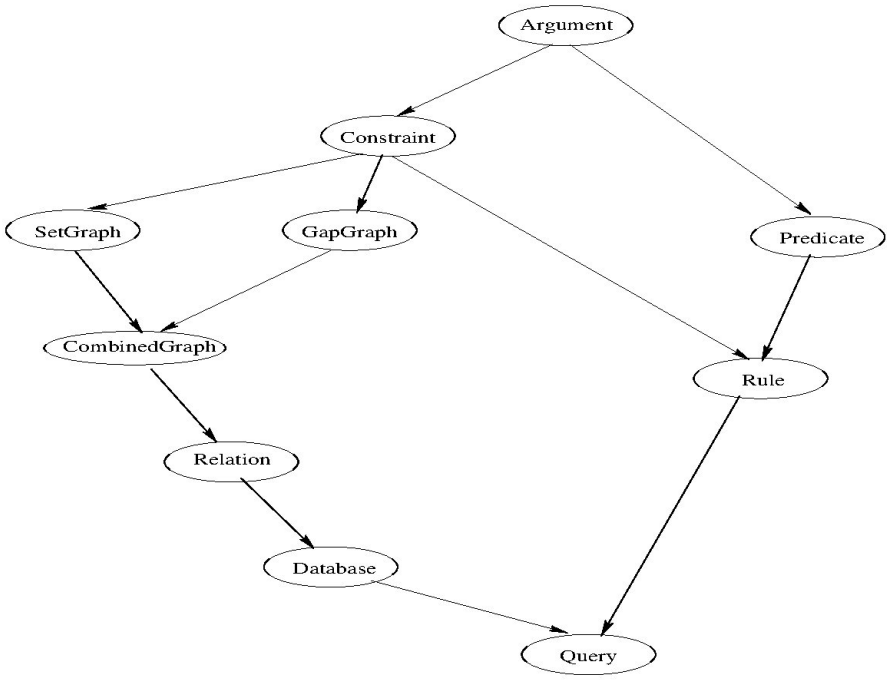


Fig. 1. The class hierarchy used in the implementation

connected, -1 if the ith vertex is $\subseteq$ the jth vertex, 0 if they are equal and 1 if the ith vertex is $\supseteq$ the jth vertex. In addition the last two columns of the second matrix are used to describe the set-constant lower and upper bounds for each variable (with the emptyset and the set of integers Z as the default values). The member functions in each of these classes are: Consistent, Satisfies, Subsume, Shortcut and Merge.

Relation Class: This class is used to represent a generalized relation. It is comprised of a name, an argument list and two tables of generalized tuples represented by a CombinedGraph, one being the table of actual relational tuples in the relation, the other being the "delta table", i.e., a table containing references to those tuples added to the relation in the most recent iteration of the query evaluation routine. This is to facilitate the implementation of the semi-naive query evaluation algorithm. The Relation has member functions that implement the Select, Project and Join operations, as well as a member function AddTuple which ensures that a tuple being added to the relation is consistent, and that it is not subsumed by any other generalized tuple in the relation.

Database Class: This class implements a generalized database. It consists of a set of instances of the Relation class, and member functions that implement insertion and deletion of relations into and from the database. It is in the Database class that the tokens returned by the lexical front-end are stored and manipu-

lated. The Database class defines the concept of a module, which is a list of DISCO facts and rules bracketed by a begin and an end. The ModuleHandler member function of this class evaluates the DISCO query in a semi-naive way.
Query Class: Each query is a combination of a program and a database. This class contains the functions of insert-relation, delete-relation and evaluate-query.

The following algorithm in pseudo-code describes the working of the query evaluation technique used in DISCO. Eventhough there were two evaluation techniques are used for each versions of DISCO, only the algorithm for semi-naive evaluation is represented. The algorithm for naive evaluation technique can be achieved by changing the body of repeat statement to 'call Function *Query.Eval*'. In function *Relation.CreateEDB()* all the facts are converted into the set of combined graphs and stored into the database. The reason that a fact clause is converted into a set of combined graphs is that an integer gap-order constraint of the body of the fact could be converted into two gap-graphs (eq. two gap-graphs of $x < y$ and $x > y$ for the integer gap-order constraint $x \neq y$) and each of two gap-graphs combines with a set graph resulting in two combined graphs. The function *Rule.Rectify()* takes care of the rectification of the rules. The fuction *Query.Eval* produces a set of new tuples satisfying rule clauses having predicates in its body. The algorithm terminates when there are no more tuples that can be added to any IDB relations by the function *Query.Eval.*

Algorithm 3.1 : The Query Evaluation Algorithm in Pseudo-Code

Function *Relation.CreateEDB()*
begin
  for each EDB predicate P_i **do**
  begin
    initialize a relation E_i;
    for each fact F_j with predicate P_i **do**
    begin
      create a list L of combined graphs in the following manner:
       construct a disjunction of gap-graphs corresponding to the integer
       order constraints in the body of F_j;
       construct a set-graph corresponding to the set order constraints
       in the body of F_j;
       construct a set of combined graphs by combining each gap-graph
       to the set-graph;
      for each combined graph c in L **do**
        if (c does not subsume any combined graph in P_i)
          add c to E_i;
    end
    if (E_i is not in the database)
      add E_i to the database;
  end
end /* end of Function Relation.CreateEDB */

```
Function Rule.Rectify()
begin
    for each rule r_i in module M do
    begin
        initialize an empty attribute list A_i;
        for each predicate symbol P_j in the body of r_i do
        begin
            rectify P_j for both integer and set variables by calling Pred.Rectify;
            /* see [37] for the description of how predicates are rectified. */
        end
        for each constraint c_j in the body of r_i do
        begin
            if (any operand of c_j is a integer or set variable)
                add this operand to the attribute set of the relation A_i;
            if (c_i can be applied to any P_l of r_i as a selection constraint)
                add c_j to selection list of P_l of r_i as a selection constraint;
        end
        mark rule r_i as rectified;
    end
end /* end of Function Rule.Rectify */

Function Query.Eval()
begin
    initialize I_j corresponding to IDB predicate P_j to be an empty set.
    repeat
        for each IDB predicate symbol P_j do
        begin
            call EVAL-INCR() analogously to [37].
            It returns ΔI_j, the list of newly added combined graphs;
            if (any combined graph in ΔI_j does not subsume those already in I_j)
                add that new combined graph to I_j;
        end
    until (no new combined graphs are added to any I_j)
end /* end of Function Query.Eval */


/* main program begins here */

call Function Parse();
call Function Relation.CreateEDB();
call Function Rule.Rectify();
call Function Query.Eval();
```

4 Testing Results

We first describe two more examples adapted from [31].

Example 4.1 We can express the successor function for values between 1 and 2^s using only $O(s)$ space. The idea is to encode the binary notation of each number as some subset of $\{s1, s0, \ldots, 21, 20, 11, 10\}$, where $i1$ or $i0$ will be present according to whether in the binary encoding the ith digit from the right is 1 or 0, respectively. For example, let $s = 4$. Then the number 9 can be represented as $\{41, 30, 20, 11\}$.

We first create a relation $Digit(N, i, x)$ which is true if and only if in the binary notation of n the ith digit from the right is x.

$Digit(N, 1, 0)$:— $10 \in N, 11 \notin N$.
$Digit(N, 1, 1)$:— $11 \in N, 10 \notin N$.
...
$Digit(N, s, 0)$:— $s0 \in N, s1 \notin N$.
$Digit(N, s, 1)$:— $s1 \in N, s0 \notin N$.

We also add to the input database the facts $Next(0, 1), \ldots, Next(s-1, s)$ and the fact $No\_digits(s)$ and $Time\_bound(\{s1, \ldots, 11\})$ that describe that we have s binary digits in each number and the largest number is 2^s. Note that the size of the database is $O(s)$. Now we express the successor relation $Succ(N, M)$ which is true if and only if $m = n+1$ for any $n, m \leq 2^s$ where N and M express the integers n and m respectively.

$Succ(N, M)$:— $Succ2(N, M, s), No\_digits(s)$.

$Succ2(N, M, i)$:— $Succ2(N, M, j), Next(j, i), Digit(N, i, x), Digit(M, i, x)$.
$Succ2(N, M, 1)$:— $Digit(N, 1, 0), Digit(M, 1, 1)$.
$Succ2(N, M, i)$:— $Succ3(N, M, j), Next(j, i), Digit(N, i, 0), Digit(M, i, 1)$.

$Succ3(N, M, i)$:— $Succ3(N, M, j), Next(j, i), Digit(N, i, 1), Digit(M, i, 0)$.
$Succ3(N, M, 1)$:— $Digit(N, 1, 1), Digit(M, 1, 0)$.

It is easy to see that in the above $Succ3(N, M, i)$ is true if the first i digits of N are all 1's and of M are all 0's. The second and third rules of $Succ2$ ensure that for some i the first i digits of N represent $2^i - 1$ and those of M represent 2^i. The first rule for $Succ2$ ensures that N and M agree on all subsequent digits, hence $N = M + 1$ will be true for $Succ2(N, M, s)$ for any s greater than i.

Note that in the succesor example the size of the output database is exponential in the size of the input database. We could not have such a case without constraints. The next example concerns inheritance hierarchies.

Example 4.2 Consider an inheritance hierarchy in which *People* is the superclass of the *Employee* and *Customer* classes and *Employee* is the superclass of the *Manager* class. Suppose that each class in the hierarchy has some set constraints (lower and upper bounds) on its set of elements. The lower bounds tell which persons are definitely in the class, while the upper bounds tell which persons may be in the class. Any set in between the two is a possible solution.

Consider the input database shown below. There the given constraints for the Employee class allows two possible solutions, one is $\{al, bob, carl\}$ the other is $\{al, bob, carl, dave\}$. (We use lower case character strings instead of fixed ID numbers to make the example clearer.)

$Root(person)$
$Subclass(1, person, employee)$
$Subclass(2, person, customer)$
$Subclass(1, employee, manager)$

$No\_subclasses(person, 2)$
$No\_subclasses(employee, 1)$
$No\_subclasses(manager, 0)$
$No\_subclasses(customer, 0)$

$Next(0, 1)$
$Next(1, 2)$

$Inbounds(person, P)$
$Inbounds(employee, E)$:— $\{al, bob, carl\} \subseteq E, E \subseteq \{al, bob, carl, dave\}$
$Inbounds(manager, M)$:— $\{al, bob\} \subseteq M$
$Inbounds(customer, C)$:— $\{ed, fred, greg\} \subseteq C, C \subseteq \{ed, fred, greg, li\}$, $C \subseteq \{ed, fred, greg, han, ken\}, joe \notin C$

A natural question is to find the tightest lower and upper bounds for each class implied by the entire inheritance hierarchy. This can be done by:

$Upper\_bound(c_2, S_2)$:— $Subclass(n, c_1, c_2), Upper\_bound(c_1, S_1), S_2 \subseteq S_1$, $Inbounds(c_2, S_2)$
$Upper\_bound(c, S)$:— $Root(c), Inbounds(c, S)$

$Lower\_bound(c, S)$:— $Inbounds(c, S), Lower\_bound2(n, c, S)$, $No\_subclasses(c, n)$.

$Lower\_bound2(n, c, S)$:— $Lower\_bound2(m, c, S), Lower\_bound(c_2, S_2)$, $S_2 \subseteq S, Next(m, n), Subclass(n, c, c_2)$
$Lower\_bound2(0, c, S)$

The intuitive idea is that the upper bound of a superclass is also an upper bound of its subclasses. Similarly, the lower bound of a subclass is a lower bound of its superclass.

For example, the program will find the upper bound for the class Manager to be $M \subseteq \{al, bob, carl, dave\}$, and the lower bound for the class Person to be $\{al, bob, carl, ed, fred, greg\} \subseteq P$. □

Both the naive and the semi-naive versions of the evaluation routines included an algebraic optimization of the right hand sides of the rules. The optimization implemented uses pushing down selections and projections (see Section 3). The CPU times shown in Figures 2 and 5 are measured in seconds on a Silicon Graphics 4.1.3 mainframe machine. The figures show the total CPU times for parsing,

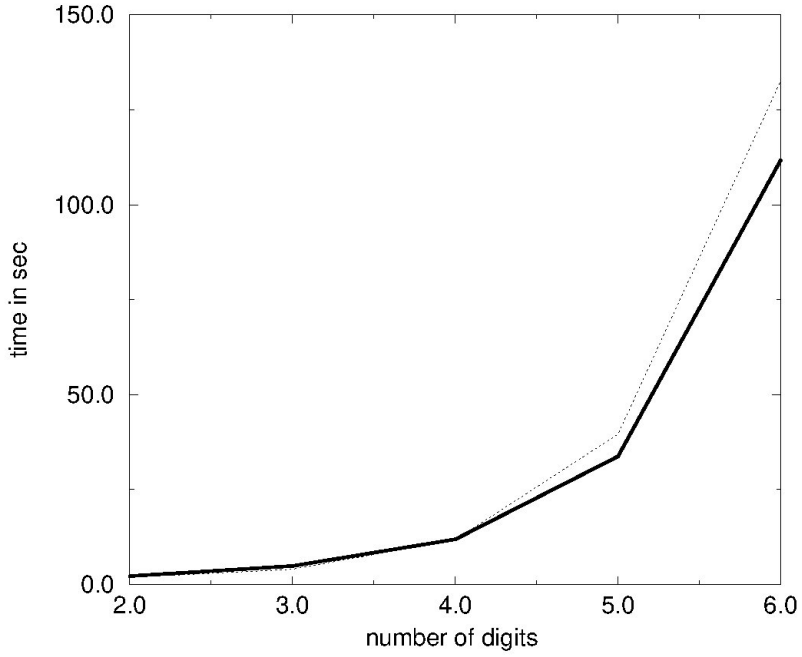


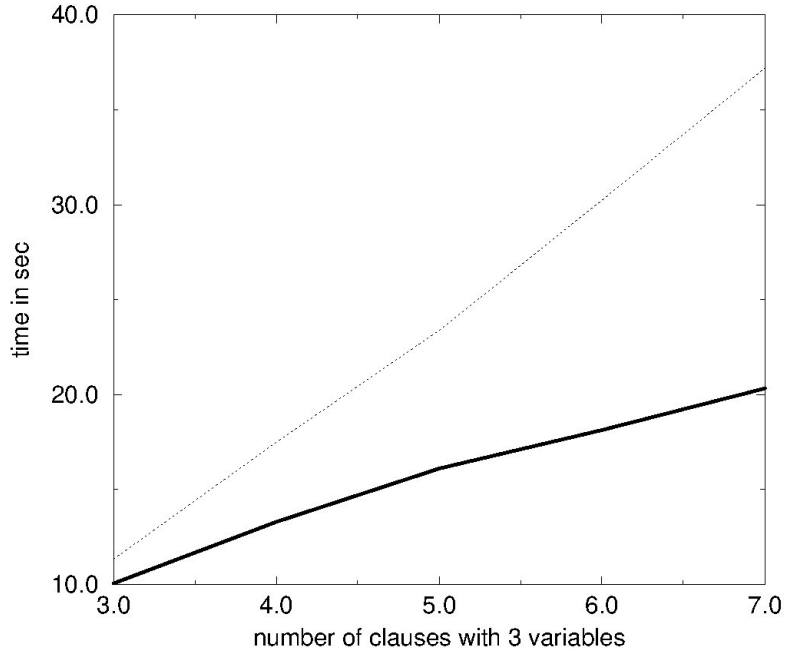
Fig. 2. Plot of running times for successor query

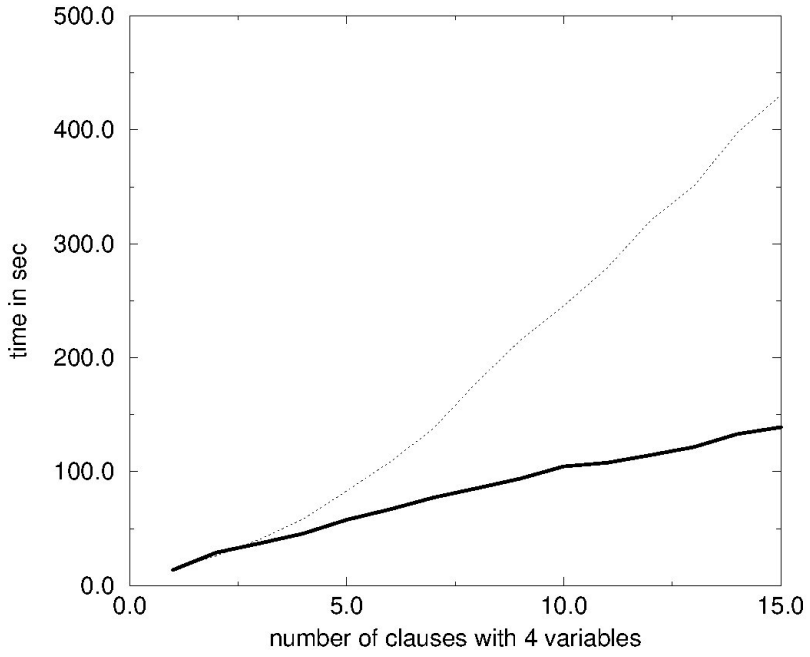
rectifying, algebraic optimizations, as well as the naive (dotted line) or semi-naive (solid line) fixpoint evaluations of the queries. Not shown in the figures are the running times for the inheritance query. We found that the naive evaluation for the inheritance query took 3.92 seconds and the semi-naive evaluation took 2.98 seconds.

5 Related Work

There is a number of constraint logic programming sytems that allow finite sets as a data type and constraints on finite sets, inlcuding ECLIPSE [14], CLPS [27], Conjuncto [15], and {log} [13]. In addition, we should mention CHIP [12] that allows linear arithmetic constraints over both the rationals and bounded subsets of the integers and CLP(Σ^*) [40] that allows finite sets of strings as a data type. DISCO is different from these in its main evaluation method and in allowing infinite sets.

Constraint logic programming systems without set type data also share with


40.0
30.0
20.0
10.0
time in sec
3.0
4.0
5.0
6.0
7.0
number of clauses with 3 variables


500.0
400.0
300.0
200.0
100.0
0.0
time in sec
0.0
5.0
10.0
15.0
number of clauses with 4 variables

DISCO the idea of constraint tuples and constraint-driven evaluation. Among these we can mention Prolog III that allows constraints over the 2-valued Boolean algebra and linear arithmetic constraints over the rationals [10], CLP($\mathcal{R}$) [19] that provides polynomial constraints over the reals, LIFE [1] that allows constraints over feature trees and also provides a notion of objects and Trilogy [39] that allows constraints over integers, real numbers and strings.

In constraint logic programming languages program termination is not guaranteed, but in DISCO the evaluation of queries is guaranteed to terminate, as is usually necessary in database systems. The optimization techniques best suitable for constraint query languages may be different from those for constraint logic programs. Therefore the area of optimization techniques and their implementations needs to be further explored. A recent advance in this area is the compile-time constraint solving method of [16] that is implemented within the DeCoR database system.

Within the area of nested databases that allow abstract data types with nesting of tuple and set constructors several related languages have been proposed, for example LDL [36], CORAL [29] and ELPS [26]. The SEL [20] logic programming language also incorporates the union $\cup$ set constructor as a basic primitive within Prolog. These languages however do not allow the use of set constraints within the input (or output) database; therefore they fit neither the CLP nor the CQL frameworks [18, 23]. Also these systems do not take advantage of quantifier elimination for set order constraints but use set-matching and other techniques instead (see [2] for a survey on such techniques). Some recent language proposals within nested or object-oriented databases [5, 17, 34] allow constraint tuples but they also fall outside the contraint query languages framework.

6 Conclusions and Open Problems

We are currently working on adding to DISCO further optimization methods like magic sets, a better indexing method for constraint tuples (eg. [24, 6]) and extending the types of constraints in the system. In particular we plan to add periodicity constraints [35]. Also, in the next version of DISCO we plan to include string data type. That would enable to express the inheritance hierarchy example with ease. We also would like to explore adding aggregate operations and negation to the DISCO query language in a safe way. Some preliminary theoretical results towards these are described in [9, 32].

References

1. H. Aït-Kaci, A. Podelski. Towards a Meaning of LIFE. *Journal of Logic Programming*, 16, 195–234, 1993.
2. A. Aiken. Set Constraints: Results, Applications and Future Directions. *Proc. 2nd Workshop on Principles and Practice of Constraint Programming*, 171–179, 1994.

3. F. Afrati, S.S. Cosmadakis, S. Grumbach, G.M. Kuper. Linear vs. Polynomial Constraints in Database Query Languages. *Proc. 2nd Workshop on Principles and Practice of Constraint Programming*, 152–160, 1994.
4. A. Brodsky, J. Jaffar, M. J. Maher. Toward Practical Constraint Databases, *Proc. VLDB*, 567–580, 1993.
5. A. Brodsky, Y. Kornatzky. The $L_{yri}C$ Language: Querying Constraint Objects. *Proc. SIGMOD*, 1995.
6. A. Brodsky, C. Lassez, J-L. Lassez, M. J. Maher. Separability of Polyhedra for Optimal Filtering of Spatial and Constraint Data, *Proc. ACM PODS*, 54–65, 1995.
7. A.K. Chandra, D. Harel. Computable Queries for Relational Data Bases. *Journal of Computer and System Sciences*, 21:156–178, 1980.
8. J. Chomicki, T. Imielinski. Finite Representation of Infinite Query Answers. *ACM Transactions of Database Systems*, 181–223, vol. 18, no. 2, 1993.
9. J. Chomicki, G. Kuper. Measuring Infinite Relations, *Proc. 14th ACM PODS*, 78–85, 1995.
10. A. Colmerauer. *An Introduction to Prolog III. CACM*, 28(4):412–418, 1990.
11. J. Cox, K. McAloon. Decision Procedures for Constraint Based Extensions of Datalog. In: *Constraint Logic Programming*, MIT Press, 1993.
12. M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. *Proc. Fifth Generation Computer Systems*, 1988.
13. A. Dovier, G. Rossi. Embedding extensional finite sets in CLP. *International Logic Programming Symposium*, 1993.
14. ECLIPSE. *Eclipse user manual.* Technical report. ECRC, 1994.
15. C. Gervet. Conjunto: Constraint Logic Programming with Finite Set Domains. *Proc. International Logic Programming Symposium*, 339–358, 1994.
16. R. Gross, R Marti. Compile-time Constraint Solving in a Constraint Database System. *Proc. Post-ILPS'94 Workshop on Constraints and Databases*, 13–25, 1994.
17. S. Grumbach, J Su. Finitely Representable Databases. *Proc. 14th ACM PODS*, 66–77, 1995.
18. J. Jaffar, J.L. Lassez. Constraint Logic Programming. *Proc. 14th ACM POPL*, 111–119, 1987.
19. J. Jaffar, S. Michaylov, P.J. Stuckey, R.H. Yap. The CLP(R) Language and System. *ACM Transactions on Programming Languages and Systems*, 14:3, 339-395, 1992.
20. B. Jayaraman, A. Nair. Subset-Logic Programming: Applications and Implementation. Univ. North Carolina Tech. Report TR-88-011, 1988.
21. F. Kabanza, J-M. Stevenne, P. Wolper. Handling Infinite Temporal Data. *Proc. 9th ACM PODS*, 392–403, 1990.
22. P.C. Kanellakis, D.Q. Goldin. Constraint Programming and Database Query Languages. *Proc. 2nd TACS*, 1994.
23. P. C. Kanellakis, G. M. Kuper, P. Z. Revesz. Constraint Query Languages. *Journal of Computer and System Sciences*, vol. 51, 26–52, 1995.
24. P.C. Kanellakis, S. Ramaswamy, D.E. Vengroff, J.S. Vitter. Indexing for Data Models with Constraints and Classes *Proc. 12th ACM PODS*, 1993.
25. M. Koubarakis. Representing and Querying in Temporal Databases: the Power of Temporal Constraints. *Proc. Ninth International Conference on Data Engineering*, 1993.

26. G. M. Kuper. Logic Programming with Sets. *Journal of Computer and System Sciences*, 41, 44-64, 1990.
27. B. Legeard, E. Legros. Short overview of the CLPS System. *Proc. PLILP*, 1991.
28. J. Paradeans, J. Van den Bussche, D. Van Gucht. Towards a Theory of Spatial Database Queries. *Proc. 13th ACM PODS*, 279–288, 1994.
29. R. Ramakrishnan, D. Srivastava, S. Sudarshan. CORAL: Control, Relations and Logic. *Proc. VLDB*, 1992.
30. P. Z. Revesz. A Closed Form Evaluation for Datalog Queries with Integer (Gap)-Order Constraints, *Theoretical Computer Science*, vol. 116, no. 1, 117-149, 1993.
31. P. Z. Revesz. Datalog Queries of Set Constraint Databases, *Fifth International Conference on Database Theory*, Springer-Verlag LNCS 893, pp. 425–438, Prague, Czech Republic, January, 1995.
32. P. Z. Revesz. Safe Stratified Datalog with Integer Order Programs, *First International Conference on Principles and Practice of Constraint Programming*, Springer-Verlag LNCS 976, pp. 154–169, Cassis, September, 1995.
33. P. Z. Revesz. Set Constraint Databases and Query Languages, *Pre-PODS'95 Workshop on Theory of Constraint Databases*, San Jose, California, 1995.
34. D. Srivastava, R. Ramakrishnan, P.Z. Revesz. Constraint Objects. *Proc. 2nd Workshop on Principles and Practice of Constraint Programming*, 274–284, 1994.
35. D. Toman, J. Chomicki, D.S. Rogers. Datalog with Integer Periodicity Constraints. *Proc. ILPS*, 1994.
36. S. Tsur and C. Zaniolo. LDL: A Logic-Based Data-Language. *Proc. VLDB*, pp 33-41, 1986.
37. J.D. Ullman. *Principles of Database and Knowledge-Base Systems*, Vols 1&2, Computer Science Press, 1989.
38. M. Vardi. The Complexity of Relational Query Languages. *Proc. 14th ACM Symposium on the Theory of Computing*, 137–145, 1982.
39. P. Voda. Types of Trilogy. *Proc. 5th International Conference on Logic Programming*, 580–589, 1988.
40. C. Walinsky. CLP(Σ^*): Constraint logic programming with regular sets. *Proc. 6th International Conference on Logic Programming*, 181–190, 1989.

This article was processed using the LaTeX macro package with LLNCS style