Languages[Query Languages] Logical Design[data models]

This work was supported in part by NSF grants IRI-9625055 and IRI-9632871 and by a Gallup Research Professorship.


Name: Peter Z. Reveny
Address: Department of Computer Science and Engineering
and Center for Communication and Information Science
University of Nebraska-Lincoln, Lincoln, NE 68588 USA
Affiliation: University of Nebraska-Lincoln
Biography: Peter Z. Reveny is an assistant professor of computer science and engineering at the University of Nebraska-Lincoln since August 1992. He obtained his B.S. degree, Summa cum Laude, from Tulane University in 1985, and M.S. and Ph.D. degrees all from computer science from Brown University in 1987 and 1991, respectively. He was a postdoctoral visiting fellow at the University of Toronto during 1991-92.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works, requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM Inc., 1515 Broadway, New York, NY 10036 USA, fax +1 (212) 869-0481, or permissions@acm.org.
Safe Query Languages for Constraint Databases

Peter Z. Revesz
University of Nebraska–Lincoln

In the database framework of [Kanelakis et al. 1990] it was argued that constraint query languages should take as input constraint databases and give as output other constraint databases that use the same type of atomic constraints. This closed-form requirement has been difficult to realize in constraint query languages that contain the negation symbol. This paper describes a general approach to restricting constraint query languages with negation to safe subsets that contain only programs that are evaluable in closed-form on any valid constraint database input.

Categories and Subject Descriptors: H.2.3 [Software]: Database Management; H.2.1 [Software]: Database Management

General Terms: Database, Languages

Additional Key Words and Phrases: Constraint databases, datalog, relational calculus, safety

1. INTRODUCTION

Constraint databases describe extensional database relations as quantifier-free first-order formulas. Constraint databases in the form of nonground facts were used in constraint logic programming [Cohen 1990; Jaffar and Lassez 1987; Jaffar and Maher 1994; Van Hentenryck 1989] for almost ten years. Constraint databases are also increasingly adopted for database use [Kanelakis 1995]. In the database framework of [Kanelakis et al. 1990] it was argued that constraint query languages should take as input constraint databases and give as output other constraint databases that use the same type of constraints. This has been called the closed-form evaluation requirement.

There are several motivations for closed-form evaluation. A closed-form evaluation enables easy composition of queries. That is convenient in the information market where companies buy raw data and sell refined data both in the form of some databases. For example, a chain of companies could produce a refined product like a geographic map. Another motivation for a closed-form evaluation is that it also allows the addition of aggregate operators as is done recently in [Chomicki and Kuper 1995].

The good news is that the closed-form evaluation requirement is met by several constraint query languages. For example, Relational Calculus with equality and inequality constraint databases, Relational calculus with polynomial inequality constraints, Datalog with rational order constraints can be evaluated in closed-form in PTIME data complexity [Kanelakis et al. 1990]. (Data complexity is the measure of the computational complexity of fixed queries as the size of the input database grows [Chandra and Harel 1982; Vardi 1982]. The rationale behind this measure is that in practice the size of the database typically dominates by several orders of magnitude the size of the query.)

Datalog with integer (gap)-order constraints programs and Datalog with \( \subseteq \) constraints on set variables are evaluable in closed-form on constraint databases with
DEXPTIME-complete data complexity [Revesz 1993; Cox and McAloon 1993; Revesz 1995]. Datalog $S$ [Chomicki 19], an extension of Datalog with a successor function applied always to the first argument of relations, can be evaluated in closed-form and has PSPACE-complete data complexity. Datalog with periodicity constraints [Toman et al. 1994], relational calculus with linear repeating points [Kabanza et al. 1990] and temporal constraints [Koubarakis 1994] can be also evaluated in closed-form.

The bad news is that many other interesting languages do not guarantee a closed-form evaluation. For example, the temporal database queries of [Baudinet et al. 1991], and stratified Datalog with integer (gap-)order constraints [Revesz 1993] can express any Turing-computable function, hence in these languages termination of query evaluation cannot be guaranteed.

In this paper we take an inspiration from the area of relational databases. In relational databases queries are required to be functions from finite input to finite output relational databases. This is an analogue of the closed-form evaluation requirement. It is traditional in the relational database literature to define various syntactical “safety restrictions” on languages to ensure that queries in the restricted language always yield finite database outputs on finite database inputs [Abiteboul et al. 1995; Kifer 1988; Ullman 1989]. In this paper we generalize this notion of safety by considering syntactical restrictions on languages to guarantee closed-form evaluation. Safety can be tested independently of database inputs.

The syntactical notion of safety in this paper corresponds to a subset of the semantical notion of evaluability in the following way. Every safe query is evaluatable in closed-form on any valid database input, but some evaluable queries are not safe. [Stolboushkin and M.A. 1997] proved recently that, unfortunately, any definition of safety must leave out some queries that are evaluable in closed-form. Nevertheless, the safe constraint queries defined in this paper already extend greatly the expressive power of safe relational queries.

This paper is organized as follows. Section 2 describes basic definitions and previous related work. Section 3 describes a general approach to safety for both relational calculus and Datalog query languages with and without negation.

Section 4 defines safe relational calculus queries with integer gap-order constraint databases, safe relational calculus queries with integer set order constraint databases and the combination of these two languages. Section 4 also presents terminating evaluation algorithms for queries in these languages and algorithms to test the safety of relational calculus queries.

Section 5 defines safe stratified Datalog queries with integer gap-order constraint databases, safe stratified Datalog queries with integer set order constraint databases and their combination. Section 5 also presents terminating evaluation algorithms for queries in these languages and algorithms to test the safety of Stratified Datalog queries.

Section 6 shows that the computational complexity of testing whether relational calculus with gap-order or set order or stratified Datalog with gap-order or set order programs are safe is in PTIME in their size. Section 6 also considers the data complexity of evaluating yes/no relational calculus and stratified Datalog queries with gap-order or set order constraints. The data complexity results in this paper are summarized in the following table:
<table>
<thead>
<tr>
<th>language</th>
<th>integer gap-order</th>
<th>integer set order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Safe RC</td>
<td>in PTIME</td>
<td>PSPACE-hard, in DEXPTIME</td>
</tr>
<tr>
<td>Datalog</td>
<td>DEXPTIME-complete</td>
<td>DEXPTIME-complete</td>
</tr>
<tr>
<td>Safe Datalog</td>
<td>nonelementary-hard, closed</td>
<td>DEXPTIME-complete</td>
</tr>
</tbody>
</table>

It is also shown that for safe stratified Datalog with integer gap-order constraints, in the data complexity result the level of exponentiation can grow linearly with the number of strata in the programs. Finally Section 7 gives some conclusion and direction for future work.

2. BASIC CONCEPTS
In this section we review the basic concepts of constraint databases and constraint query languages and discuss some related works on closed-form evaluation and query languages.

2.1 Constraint Databases
Constraint databases are a finite set of constraint relations. Each constraint relation is a quantifier-free first-order DNF (disjunctive normal form) formula in some constraint theory. We call each disjunct of the DNF formula a constraint tuple. Each constraint theory is distinguished by the set of atomic formulas in it. The following are some example constraint theories.

**Theory of Integer (Gap)-Order Constraints:** The domain of variables is the integers, and the allowed constraints are \( x = y, x \neq y, x <_g y \) where \( g \) is any nonnegative integer and \( x, y \) are integer variables or constants. The last of these is called a gap-order constraint. The meaning of a gap-order constraint \( x <_g y \) is that \( x \) is less than \( y \) with at least \( g \) integers between them, or equivalently, \( x + g < y \).

**Theory of Integer Set Order Constraints:** The domain of variables is the finite and infinite set of integer tuples, and the allowed constraints are \( \vec{x} \in X, \vec{x} \not\subseteq X, \text{and } X \subseteq Y \) where \( \vec{x} \) is any tuple of integer constants and \( X, Y \) are integer set variables or constants.

We will denote these theories by \( C_{=, \neq, <_U} \) and \( C_{=, \neq, <_U} \), respectively. We will also consider the combination of these two theories, which we denote \( C_{=, \neq, <_{U \cup V}} \). In this combination the domain of variables is either the integers or the finite and infinite sets of integers. To keep our notation simple, we will denote integer variables in lower and set of integer variables by upper case letters. We also denote by \( \mathbb{Z} \) the integers, by \( \mathbb{N} \) the nonnegative integers, and by \( P(\mathbb{Z}^a) \) the powerset of the integer tuples of arity \( a \). Unless we say explicitly otherwise, \( a \) will be one in the examples in the paper.

The following is an example constraint relation over the theory of integers and \( C_{=, \neq, <_U} \):

\[
R_5(x, y) = 20 <_5 x \lor y <_4 7 \lor x = y
\]

This relation has three constraint tuples.

Let \( R \) be any constraint relation. The meaning of \( R \), denoted \( \text{points}(R) \), is the unrestricted (finite or infinite) relation that consists of the set of tuples that satisfy the DNF formula of \( R \).
In the above, \( \text{points}(R_3) \) is an infinite relation which includes for example the tuples \((30, 15), (10, 2) \) and \((10, 10) \).

2.2 Constraint Query Languages

In this section we review the most important constraint query languages, including their syntax and semantics.

2.2.1 Relational Calculus Queries. The syntax of relational calculus queries is the language of relational calculus formulas. Relational Calculus formulas are built from variable symbols \( x, y, z, v, u, \ldots \), constant symbols \( a, b, c, \ldots \), relation symbols, the conjunction connective symbol \( \land \) and the existential quantifier symbol \( \exists \) according to the following rules.

— If \( R \) is an \( n \)-ary relation symbol and \( x_1, \ldots, x_n \) are variables or constants, then \( R(x_1, \ldots, x_n) \) is a relational calculus formula.
— If \( \phi_1 \) and \( \phi_2 \) are relational calculus formulas, then \( \phi_1 \land \phi_2 \) is a relational calculus formula.
— If \( \phi \) is a relational calculus formula, then \( \neg \phi \) is a relational calculus formula.
— If \( \phi \) is a relational calculus formula and \( x \) is a variable, then \( \exists x(\phi) \) is a relational calculus formula.

In the last line of the above definition, we call each occurrence of variable \( x \) within \( \phi \) a bound variable. Variables that are not bound are called free in a formula. Sometimes we abbreviate a formula of the form \( \exists x_1, \ldots, \exists x_n \phi \) by writing \( \exists \phi \). We will also use sometimes the identity \( \forall \phi \equiv \neg \exists \neg \phi \).

Each relational calculus formula is evaluatable as either true or false with respect to the input database and an assignment to the free variables. A relational calculus query is a function from a relational database to an unnamed relation. The unnamed relation will contain those assignments from the domain \( \delta \) to the free variables that make the relational calculus formula true.

More precisely, let \( x_1, \ldots, x_n \) be the set of free variables of a relational calculus formula \( \phi \) in some fixed order and let \( \phi(a_1, \ldots, a_n) \) denote the formula obtained by substituting \( a_i \) for \( x_i \) for each \( 1 \leq i \leq n \). Each input database \( d \) is an assignment of a finite number of tuples over \( \delta^n \) to each \( R_i \) with arity \( n_i \). The output database is a single relation of arity \( n \) defined as \( \{(a_1, \ldots, a_n) : < \delta, d > \models \phi(a_1, \ldots, a_n) \} \).

Here \( < \delta, d > \models \phi(a_1, \ldots, a_k) \) if \( \phi \) is a relational calculus formula and \( (a_1, \ldots, a_k) \) is a tuple in \( \delta^n \), then

\[
< \delta, d > \models (\phi \land \psi) & \quad \text{iff} \quad < \delta, d > \models \phi \quad \text{and} \quad < \delta, d > \models \psi \\
< \delta, d > \models (\neg \phi) & \quad \text{iff} \quad \text{not} \quad < \delta, d > \models \phi \\
< \delta, d > \models (\exists x \phi) & \quad \text{iff} \quad < \delta, d > \models \phi[a_i/a_j] \quad \text{for some} \quad a_j \in \delta \\
\]

where \( [x_i/a_j] \) means the instantiation of the free variable \( x_i \) by \( a_j \).

2.2.2 Relational Algebra Queries. Relational algebra expressions are built from the well-known relational algebra operators of select \( (\sigma) \), project \( (\pi) \), rename \( (\rho) \), join \( (\times) \) and difference \( (\setminus) \). The relational algebra plays an important part in evaluating non-procedural relational database queries [Abiteboul et al. 1989; Ullman 1989]. The algebraic operators can be extended for constraint databases. The
constraint algebra operators on constraint relations \(R_1, R_2, \ldots\) correspond to relational algebra operators on the unrestricted relations \(\text{points}(R_1), \text{points}(R_2), \ldots\). More precisely:

Let \(R, R_1, R_2\) be constraint relations over some domain and constraint theory \(\mathcal{C}\). We say that \(\sigma, \pi, \rho, \forall, \exists\) and \(-\) are select, project, rename, join and difference operators for constraint databases over \(\mathcal{C}\) if they map input constraint relations over \(\mathcal{C}\) to output constraint relations over \(\mathcal{C}\) such that the following hold:

\[
\text{points}(\rho_{\mathcal{C}}(R)) \equiv \rho_{\mathcal{C}}(\text{points}(R))
\]

\[
\text{points}(\sigma_{\mathcal{C}}(R)) \equiv \sigma_{\mathcal{C}}(\text{points}(R))
\]

\[
\text{points}(\pi_{x_1, \ldots, x_n}(R)) \equiv \pi_{x_1, \ldots, x_n}(\text{points}(R))
\]

\[
\text{points}(R_1 \bowtie R_2) \equiv \text{points}(R_1) \bowtie \text{points}(R_2)
\]

\[
\text{points}(R_1 \ominus R_2) \equiv \text{points}(R_1) - \text{points}(R_2)
\]

where \(\mathcal{C}\) is any valid rename or select condition.

We say that an algebraic operator is closed under a constraint theory \(\mathcal{C}\) if it maps constraint relations over \(\mathcal{C}\) to constraint relations over \(\mathcal{C}\).

### 2.2.3 Datalog Queries

Datalog is a rule-based language. That means that syntactically a Datalog program \(\Pi\) is a finite set of rules of the form:

\[
R_0(x_1, \ldots, x_k) :\leftarrow R_1(x_{1,1}, \ldots, x_{1,k_1}), \ldots, R_n(x_{n,1}, \ldots, x_{n,k_n})
\]

where \(R_0, \ldots, R_n\) are not necessary distinct relation symbols and the \(x_i\)'s are either variables or constants. We call \(R_0(x_1, \ldots, x_k)\) the head and \(R_1(x_{1,1}, \ldots, x_{1,k_1}), \ldots, R_n(x_{n,1}, \ldots, x_{n,k_n})\) the body of the rule. We associate with each rule \(r\) of the above form the formula \(\phi_r\).

\[
\phi_r(\mathcal{F}) = \exists \mathcal{Y}(R_1(x_{1,1}, \ldots, x_{1,k_1}) \land \ldots \land R_n(x_{n,1}, \ldots, x_{n,k_n}))
\]

where \(\mathcal{F}\) is the list of variables in the head and \(\mathcal{Y}\) the variables that occur only in the body.

The above looks very much like a positive query. The difference is that in Datalog rules some of the relation symbols stood for defined relations. For example, some \(R_i\) may be equal to \(R_0\). That is, the output relations may be defined using references to themselves.

We call extensional database relations, or EDBs, those relations whose symbol occurs only on the right hand side of rules. We call the other relation symbols the intensional database relations, or IDBs. In each Datalog query the EDBs are the input relations assigned by the user and the IDBs are the output relations to which assignments are sought. The EDBs and IDBs are disjoint in each Datalog program.

We call an interpretation of a Datalog program \(\Pi\) any assignment \(I\) of a constraint relation to each \(R_i\) that occurs in \(\Pi\). We call an interpretation an input database if it assigns to each IDB relation the empty relation.

Each Datalog program \(\Pi\) is a function from input databases to interpretations. Let \(d\) be an input database. Then the output of \(\Pi\) on \(d\), denoted \(\Pi(d)\), is the set of tuples \(t\) that are satisfied by the domain and the input database, denoted
$\delta, d \vdash t$, where we use the first condition of satisfaction only for EDB relations and we use the following for IDB relations:

$\delta, d \vdash R_i(\overline{a})$ iff there is a rule $r$ with head $R_i$ and $\delta, d \vdash \phi_r(\overline{a})$

The output of a Datalog query is called the least model.

2.2.4 Stratified Datalog Queries. Each stratified Datalog [Abiteboul et al. 1995; Chandra and Harel 1982; Apt et al. 1988; Doets 1994; Ullman 1989] program is a composition through negation of Datalog programs. We explain that composition in the following way.

We call semipositive those Datalog programs that allow negation of EDBs.

Semantically each semipositive Datalog program $\Pi$ is a mapping from input databases to interpretations. On any input database $d$ the output of the semipositive Datalog program is $\Pi(d)$ where $\Pi$ is the Datalog program in which each negated occurrence of an EDB relation $R$ is replaced with the complement of $R$.

Each stratified Datalog program $\Pi$ is a list of semipositive programs $\Pi_1, \ldots, \Pi_k$ satisfying the following property: no relation symbol $R$ that occurs negated in a $\Pi_i$ is an IDB in any $\Pi_j$ with $j \geq i$. Each $\Pi_i$ is called a stratum of $\Pi$.

Semantically each stratified Datalog program is a mapping from databases to interpretations. In particular, if $\Pi$ is the list of the semipositive programs $\Pi_1, \ldots, \Pi_k$ with the above property, then the composition $\Pi_1(\ldots \Pi_1(\ldots ))$ is the semantics of $\Pi$.

The output of a stratified Datalog query is called the perfect model.

2.3 Related Work

Constraint algebra operators and closed-form query evaluations were proven for relational calculus with linear arithmetic constraints [Brodsky et al. 1993], with linear repeating points [Kabananza et al. 1990] with rational order constraints [Kanellakis and Goldin 1994], with temporal constraints [Koubarakis 1994], with spatial constraints [Paradeans 1994] and with polynomial arithmetic constraints over the reals [Kanellakis et al. 1990], and for Datalog$_{S}$ [Chomicki 1991] and Datalog with periodicity constraints [Toman et al. 1994].

Koubarakis [Koubarakis 1994] considers temporal difference constraint databases which contain atomic constraints of the form $x_i - x_j \theta c$ where $x_i, x_j$ are integer variables, $c$ is an integer constant and $\theta$ is one of $=, \leq, \geq$. It is shown that relational calculus programs are closed under temporal difference constraint databases. This implies that safety restrictions could be avoided in the case of relational calculus with integer gap-order constraints. However, Koubarakis’s results do not extend to the language described in Section 4.2, while this paper is concerned with a general approach to safe queries.

Constraint select, project, rename and join operators that are closed for $C_{=, \neq, <, >}$ were described and the following proposition was proven in [Revesz 1993].

**Proposition 2.1.** Let $\Pi$ be a Datalog program and $d$ an input constraint database over $C_{=, \neq, <, >}$. A constraint database representing $\Pi(points(d))$ can be computed in finite time by algorithm Find-Closed-Form shown below.

---

**Algorithm Find-Closed-Form**

**INPUT:** A Datalog program $P$ and a constraint relation $\phi_i$ for each $p_i$. 

---
For the defined relations $\phi$, is false.

**OUTPUT:** The least model of $P$ in constraint database form.

**REPEAT**

**FOR** each relation $p_m$ **DO**

Let $\psi_m = \phi_m$.

**END-** **FOR**

**FOR** each $r_j$ of form $p_0(x_1, \ldots, x_k) \leftarrow p_1, \ldots, p_n$ **DO**

$T_j = \hat{\rho}_{j,1}(\phi_1) \otimes \ldots \otimes \hat{\rho}_{j,n}(\phi_n)$.

$F_j = \pi_{x_1, \ldots, x_k} T_j$.

Delete all inconsistent constraint tuples from $F_j$.

Add to $\phi$ each constraint tuple in $F_j$ that does not imply another in $\phi$.

**END-** **FOR**

**UNTIL** $\psi_m = \phi_m$ for each $m$

Contraint relational algebra operators closed for $C_{\subseteq, cd, \subseteq}$, i.e., when the arity $a$ is one, were described and closed-form evaluation of Datalog queries was proven in [Revesz 1995]. (Algorithm Find-Closed-Form with the relational algebra operator symbols reinterpretated can still compute $\Pi(d)$.)

Nested databases allow set type data as well as complex data, (e.g. sets of sets etc.), but they do not allow variables and constraints on them. Relational calculus and rule-based query languages for nested or flat databases that allow sets are considered in [Hull and Su 1993; Kuper 1990; Ramakrishnan et al. 1992; Tsar and Zaniolo 1986; Vadaparty 1994] among others. Nested databases and some constraint databases can be combined. For example, relational calculus queries of nested databases with dense and set order constraints were considered in [Grumbach and Su 1995]. Object-oriented databases and constraint databases can be also combined and queried by an SQL-like language [Brodsky and Kornatzky 1995] or a refinement rule-based language [Srivastava et al. 1994].

The constraint logic programming systems {log} [Dovier and Rossi 1993], ECLIPSE [Eclipse 1994], Conjunto [Gervet 1994] and CLPS [Legrand and Legros 1991] allow finite set domains and constraints on them. Since the domain of sets is finite, these systems also allow set constraints like union and intersection, which are not considered in this paper, and {log} and CLPS also allow a finite depth nesting of sets. The main issue in [Dovier and Rossi 1993; Eclipse 1994; Gervet 1994; Legrand and Legros 1991] is the efficiency of the constraint satisfaction techniques used in testing the satisfiability of the set constraint expressions allowed and not the efficiency of the closed-form evaluation of the constraint logic programs.

Set constraints over infinite sets are used in program analysis (see the surveys [Aiken 1994; Heintze and Jaffar 1994]). The main issue in this line of work is decision procedures for systems of set constraints. These set constraints are used to describe information about the behavior of programs concerning for example type-checking or optimization. This line of work is not concerned with using set constraints within constraint databases and query languages.
3. A GENERAL APPROACH TO SAFE QUERIES

3.1 An Approach to Queries without Negation

In this section we give a general definition of the constraint database operators \( \tilde{\rho} \) (rename), \( \tilde{\sigma} \) (select), \( \tilde{\pi} \) (project), and \( \tilde{\bowtie} \) (join). We will only assume that the constraint database is is represented in a constraint theory \( C \) with the following properties: (1) there exists a function \( sat \) from constraint tuples in \( C \) to \{true, false\} that returns true or false depending on whether the constraint tuple is satisfiable in \( C \), and (2) there exists a function \( elim \) from pairs of variables and constraint tuples in \( C \) to constraint tuples in \( C \) such that for each variable \( x \) and constraint tuple \( t \) with \( x \) in it, \( elim(x, t) \) returns a constraint tuple \( t_2 \) such that \( \exists t \) and \( t_2 \) are semantically equivalent.

Let \( \phi = t_1 \lor \ldots \lor t_n \) and \( \psi = s_1 \lor \ldots \lor s_m \) be constraint relations over some constraint theory \( C \) with the above properties.

The rename operation \( \tilde{\rho} \) returns the constraint database with the specified substitutions. That is,

\[
\tilde{\rho}_{x_1/y_1, \ldots, x_k/y_k} \phi = \phi[x_1/y_1, \ldots, x_k/y_k]
\]

where \( x/y \) means the substitution of \( x \) by \( y \).

The selection operation returns the conjunction of the constraint tuples and the selection condition when it is satisfiable. That is,

\[
\tilde{\sigma}_{x_1=a_1, \ldots, x_k=a_k} \phi = \bigvee_{1 \leq i \leq n, \ sat(t_i \land x_1=a_1 \land \ldots \land x_k=a_k)} t_i \land x_1 = a_1 \land \ldots \land x_k = a_k
\]

The projection operation eliminates the necessary variables from each constraint tuple and returns them. Let \( X = \{x_1, \ldots, x_k\} \) be the set of variables in \( \phi \), let \( x_j \in X \), and let \( X' = X \setminus \{x_j\} \).

\[
\tilde{\pi}_{X'} \phi = \bigvee_{1 \leq i \leq n} elim(x_j, t_i)
\]

The join operation pairs each constraint tuple in the first relation with each constraint tuple in the second relation. A pairing is kept as a constraint tuple of the output only if it is satisfiable.

\[
\phi \tilde{\bowtie} \psi = \bigvee_{1 \leq i \leq n, 1 \leq j \leq m, sat(t_i \land s_j)} (t_i \land s_j)
\]

**Lemma 3.1.** The following are true for any \( R, R_1, R_2 \) constraint relations over any \( C \) with the sat and elim functions:

1. \( points(\tilde{\rho}_C(R)) \equiv \rho_C(points(R)) \)
2. \( points(\tilde{\sigma}_C(R)) \equiv \sigma_C(points(R)) \)
3. \( points(\tilde{\pi}_{X'}(R)) \equiv \pi_{X'}(points(R)) \)
4. \( points(R_1 \tilde{\bowtie} R_2) \equiv points(R_1) \bowtie points(R_2) \)

**Proof.** The first and second equivalences follow from the fact that substituting variables by other variables does not change the set of models of a constraint formula.

To show equivalence (3): Suppose that \( a_1, \ldots, a_{j-1}, a_{j+1}, \ldots, a_k \) is a tuple in \( \pi_{X'} points(R) \). Then there must be a tuple \( a_1, \ldots, a_k \) in \( points(R) \). Also, \( a_1, \ldots, a_k \)
must be a model of some constraint tuple \( t_i \) of \( R \). Then by the definition of variable elimination \( a_1, \ldots, a_{j-1}, a_{j+1}, \ldots, a_k \) is a model of \( \text{elim}(x_j, t_i) \). By the definition of \( \pi \) the tuple \( a_1, \ldots, a_{j-1}, a_{j+1}, \ldots, a_k \) must belong to \( \text{points}(\pi_X, R) \).

For the reverse direction, suppose that \( a_1, \ldots, a_{j-1}, a_{j+1}, \ldots, a_k \) is a tuple in \( \text{points}(\pi_X, R) \). Then there must be a constraint tuple \( t_i \) in \( R \) such that \( a_1, \ldots, a_{j-1}, a_{j+1}, \ldots, a_k \) is a model of \( \text{elim}(x_j, t_i) \). Therefore, there is some \( a_j \) value such that \( a_1, \ldots, a_k \) is a model of \( t_i \). Then \( a_1, \ldots, a_k \) must be in \( \text{points}(R) \). Hence \( a_1, \ldots, a_{j-1}, a_{j+1}, \ldots, a_k \) must be in \( \pi_X \cdot \text{points}(R) \).

To show equivalence (4): Suppose that \( a_1, \ldots, a_{k_1} \) is in \( \text{points}(R_1) \) and \( b_1, \ldots, b_{k_2} \) is in \( \text{points}(R_2) \), and \( c_1, \ldots, c_k \) that is the combination of \( a_1, \ldots, a_{k_1} \) and \( b_1, \ldots, b_{k_2} \) such that the same attributes are assigned the same values is in \( \text{points}(R_1) \equiv \text{points}(R_2) \). Then \( a_1, \ldots, a_{k_1} \) is a model of some constraint tuple \( t_i \) in \( R_1 \) and \( b_1, \ldots, b_{k_2} \) is a model of some constraint tuple \( s_j \) in \( R_2 \). By the definition of \( \equiv \) then \( (t_i, s_j) \) is a constraint tuple in \( R_1 \otimes R_2 \) and \( c_1, \ldots, c_k \) is a model of \( R_1 \otimes R_2 \). Hence \( c_1, \ldots, c_k \) must be in \( \text{points}(R_1 \otimes R_2) \).

For the reverse direction, suppose that \( c_1, \ldots, c_k \) is in \( \text{points}(R_1 \otimes R_2) \). Then by the definition of \( \otimes \) there must be a tuple of the form \( (t_i, s_j) \) in \( R_1 \otimes R_2 \) such that \( t_i \) is a constraint tuple in \( R_1 \) and \( s_j \) is a constraint tuple in \( R_2 \). That means that there must be projections of \( c_1, \ldots, c_k \) onto the attributes of \( R_1 \) and \( R_2 \) that yield tuples \( a_1, \ldots, a_{k_1} \) and \( b_1, \ldots, b_{k_2} \) respectively and that \( a_1, \ldots, a_{k_1} \) is a model of \( R_1 \) and \( b_1, \ldots, b_{k_2} \) is a model of \( R_2 \). Hence \( a_1, \ldots, a_{k_1} \) is in \( \text{points}(R_1) \) and \( b_1, \ldots, b_{k_2} \) is in \( \text{points}(R_2) \). Therefore \( c_1, \ldots, c_k \) must be in \( \text{points}(R_1) \equiv \text{points}(R_2) \). \( \square \)

Next we describe example variable elimination algorithms for constraint tuples in \( C=\searrow, \searrow, \leq \) and \( C=\searrow, \neq, \leq \) and in \( C=\searrow, \searrow, \leq \).

**Example 3.1.** Consider the constraint theory \( C=\searrow, \leq \). Here the function \( \text{elim}(x, t) \) returns the conjunction of the constraints in the following set.

\[
\{ \text{those constraints in } t \text{ that do not contain the variable } x \}\cup
\{ y = z : y = x \text{ and } z \text{ occur in } t \}\cup
\{ y <_{\leq} z : y = x \text{ and } x <_{\leq} z \text{ occur in } t \}\cup
\{ y <_{\leq} z : y <_{\leq} x \text{ and } x = z \text{ occur in } t \}\cup
\{ y <_{\leq} x + 1 : y <_{\leq} x \text{ and } x <_{\leq} z \text{ occur in } t \}.
\]

In each of the above \( y \) and \( z \) are constants or variables other than \( x \). The above variable elimination function adds only constraints that logically follow by transitivity from the set of original constraints in \( t \). Hence if the original constraint tuple is satisfiable, then the returned constraint tuple is also satisfiable. Furthermore, if any instantiation satisfies the returned constraint tuple, then it is possible to extend that instantiation with an instantiation for \( x \) such that the combined instantiation satisfies \( t \).

The \( \text{sat}(t) \) function can be defined using the \( \text{elim} \) function. \( \text{sat}(t) \) eliminates each variable one by one from \( t \) until no variables remain. Then it returns “true” if all remaining atomic constraints (that can involve only constants and no variables) are true, else it returns “false”. It is possible to write computationally more efficient functions for eliminating variables and testing satisfiability. We refer to [Revesz 1993] for computationally more efficient algorithms. \( \square \)

Now let’s consider the constraint theory \( C_{\leq, \neq, \leq} \). Let \( A = \{ c_1, \ldots, c_n \} \) be the
set of constants that occur explicitly in a $k$-ary constraint relation $p(X_1, \ldots, X_k)$ in $C_{\leq, \neq, \subseteq}$. Let $\phi(X_1, \ldots, X_k)$ be any constraint tuple in $p$. Then in $\phi$ each $X_i$ could have in it (1) any of the constraints $\tau \in X$ or $\tau \notin X$ for each $\tau \in \mathcal{A}^0$, and (2) any of the constraints $B \subseteq X_i$ or $X_i \subseteq B$ for each $B \subseteq \mathcal{A}^0$. Further, among the $k$ argument variables of $\phi$ we can have the constraints $X_i \subseteq X_i$ where $1 \leq i, l \leq k$.

We define a normal form for constraint tuples as follows. First, replace each constraint $\tau \in X$ or $\tau \notin X$ by the equivalent constraint $\{\tau\} \subseteq X_i$ or $X_i \subseteq Z^0 \setminus \{\tau\}$. Second, replace the conjunction of the lowerbound constraints $B_1 \subseteq X_1, \ldots, B_m \subseteq X_i$ by the equivalent constraint $(\bigcup B_j) \subseteq X_i$. Similarly, replace the conjunction of upperbound constraints $X_i \subseteq B_1, \ldots, X_i \subseteq B_m$ by the equivalent constraint $X_i \subseteq (\bigcap B_j)$. We call the normal form of a constraint tuple the constraint tuple with the above replacements. It is clear that every constraint tuple and its normal form are semantically equivalent.

We say that $p$ is in normal form if every constraint tuple of $p$ is in normal form. Since the ordering of the atomic constraints within a constraint tuple does not change its meaning, we will ignore the ordering of the atomic constraints when talking about normal forms.

**Example 3.2.** Consider the constraint theory $C_{\leq, \neq, \subseteq}$. Here the function $\text{elim}(X, t)$ returns the conjunction of the constraints in the following set. Let $t'$ be the normal form of $t$.

\{$\text{those constraints in } t' \text{ that do not contain the variable } X\} \cup$

\{$Y \subseteq Z : Y \subseteq X \text{ and } X \subseteq Z \text{ occur in } t'\}$

In the above $Y$ and $Z$ are constants or variables other than $X$. The above variable elimination function adds only constraints that logically follow by transitivity from the set of original constraints in $t'$. Hence if the original constraint tuple is satisfiable, then the returned constraint tuple is also satisfiable. Furthermore, if any instantiation satisfies the returned constraint tuple, then it is possible to extend that instantiation with an instantiation for $X$ such that the combined instantiation satisfies $t$ and $t'$.

The $\text{sat}(t)$ function can be defined using the $\text{elim}$ function similarly to the previous example. □

**Example 3.3.** Finally, let’s consider the constraint theory $C_{\neq, <, \subseteq} \cup C_{\leq, \neq, \subseteq}$. Let $t$ be any tuple in this constraint theory. Each atomic constraint in $t$ belongs to either the first or the second theory. Let $t_1$ and $t_2$ be the subsets of atomic constraints in $t$ that belong to $C_{\neq, <, \subseteq}$ and to $C_{\leq, \neq, \subseteq}$ respectively. The variable elimination function in this case can be called with either an integer variable $x$ or a set of integers variable $X$. In the first case it should return the conjunction of $\text{elim}(x, t_1)$ in Example 3.1 and $t_2$, while in the second case it should return the conjunction $t_1$ and $\text{elim}(X, t_2)$ in Example 3.2. The satisfiability testing function should return true if and only if both $\text{sat}(t_1)$ in Example 3.1 and $\text{sat}(t_2)$ in Example 3.2 return true. □

### 3.2 An Approach to Safe Queries with Negation

A closed-form evaluation often cannot be guaranteed once negation is added to a constraint query language. This is because negation is not closed under several types of constraint relations.
For example, \( R(x, y) \equiv x <_5 y \) is a constraint relation over the integers and \( C_{\equiv, \neq, <_a} \), but \( \neg R \) cannot be represented as a constraint relation over the integers and \( C_{\equiv, \neq, <_a} \).

As another example, \( R_2(X) \equiv X \subseteq \{1, 2, 3, 4, 5\} \) is a constraint relation over sets of integers and \( C_{\subseteq, \neq, \subseteq} \), but \( \neg R_2(X) \) cannot be represented similarly.

On the other hand, many examples can be given where negation is unproblematic. For example, \( R_3(x, y) = 20 <_5 x \lor y <_4 7 \lor x = y \) is a constraint relation over the integers and \( C_{\equiv, \neq, <_a} \). Here \( \neg R_3(x, y) \) can be represented as \( x <_0 26, 2 <_0 y, x \neq y \), which is also a constraint relation over the integers and \( C_{\equiv, \neq, <_a} \).

The difference between the problematic and unproblematic cases of negation stem from the occurrence in the former of atomic constraints that are unnegatable within \( C \). For example, gap-order constraints over two variables and non-zero gap-value \( g \) are not negatable in \( C_{\equiv, \neq, <_a} \).

Our general approach to safe queries over constraint databases with some \( \delta \) and \( C \) will be to assign a type to each input constraint relation. The type of the input constraint relation will tell whether it may contain an unnegatable atomic constraint. Then a type checking is performed during which it is tested whether the output relation may be always represented as a constraint relation over \( \delta \) and \( C \). If it is, then the query is called safe.

Safe queries can be always evaluable in closed-form on any valid constraint database input. By a valid constraint database input we mean one in which each relation has the required type.

The evaluation of safe queries reduces to an evaluation of negation-free queries because the negation can be evaluated by a type-restricted complement operator. The type restricted complement operator will be from constraint relations of a specified type to constraint relations of the same type. In this paper, in particular we will be interested in the following:

Let \( C_{\equiv, \neq, <} \) be the subset of \( C_{\equiv, \neq, <} \) where each gap-order constraint has a zero gap-value or has at least one constant on the right or left hand side.

The type-restricted complement operator \( \Gamma \) from \( C_{\equiv, \neq, <} \) to \( C_{\equiv, \neq, <} \) can be defined using De Morgan’s laws.

Similarly, let \( C_{\subseteq, \neq, \subseteq} \) be the subset of \( C_{\subseteq, \neq, \subseteq} \) where in each \( \subseteq \) constraint the left hand side is a set constant.

The type-restricted complement operator \( \Gamma_2 \) from \( C_{\subseteq, \neq, \subseteq} \) to \( C_{\subseteq, \neq, \subseteq} \) can be defined using De Morgan’s laws and noting that \( \neg(\{c_1, \ldots, c_n\} \subseteq X) \equiv c_1 \not\subseteq X \lor \cdots \lor c_n \not\subseteq X \). The type-restricted complement operator \( \Gamma_3 \) from \( C_{\equiv, \neq, <} \cup C_{\subseteq, \neq, \subseteq} \) to \( C_{\equiv, \neq, <} \cup C_{\subseteq, \neq, \subseteq} \) can be similarly defined.

It is well-known that each relational calculus formula is equivalent to a relational algebra expression with the select, project, rename, join, and complement operators, where each negation is translated as a complement operator. Safe relational calculus formulas are equivalent to relational algebra expressions that can be evaluated with a complement operator that is closed. Similarly, each stratified Datalog program can be evaluated stratum by stratum as a Datalog query after application of a type-restricted complement operator that is closed.
4. SAFE RELATIONAL CALCULUS QUERIES

In this section we define safe relational calculus queries over gap-order and safe relational calculus queries over set order constraint databases, denoted $RC^{<z}$ and $RC_{\leq z}$, respectively, and show that they are evaluable in closed-form. We also consider the combination of these two languages, that is, safe relational calculus queries over both gap-order and set order constraint databases, denoted $RC^{<z,\leq z}$.

4.1 Safe Relational Calculus for Gap-Order Constraint Databases

We assign to each input constraint relation of arity $k$ a type that is called an arguments connection graph or congraph. Intuitively, each congraph shows the possible connections via $<_g$ constraints among the arguments of a relation. Each congraph of arity $k$ is a directed graph $C(V,E,\equiv,\neq)$ where $V$ is the set of argument variables, $E \subseteq V \times V$ is the set of edges, $\equiv \subseteq V \times V$ is the set of equalities among the argument variables, and $\neq \subseteq V \times V$ is the set of inequalities among the argument variables.

We say that a constraint relation $p(x_1,\ldots,x_k)$ has congraph type $C(V,E,\equiv,\neq)$ or is valid with respect to $C(V,E,\equiv,\neq)$ if for every constraint $x_i <_g x_j$ in $p$, the edge $(x_i,x_j) \in E$, and for every constraint $x_i = x_j$ (or $x_i \neq x_j$) in $p$, it is true that $(x_i,x_j) \in \equiv$ (or $(x_i,x_j) \notin \equiv$). (We sometimes abbreviate the latter two conditions as $x_i \equiv x_j$ and $x_i \neq x_j$).

Note that in the above we assume that each relation $p$ is rectified, that is, if it is an EDB then it always appears in the input database and if it is an IDB it always appears in the head of rules with the same list of argument variables. (In the body of the rules the relation symbol $p$ may appear with a different list of variables than in its rectified form.) Rectification is a minor restriction since it is easy to put any Datalog program into an equivalent rectified form [Ullman 1989].

Now we define safe relational calculus with gap-order constraints. We assume that each relation symbol $R$ is already assigned a congraph $C_R(V_R,E_R,\equiv_R,\neq_R)$. Each safe relational calculus formula will also have a type that depends on the type of the relation symbols in it.

— If $R$ is an $n$-ary relation symbol and $x_1,\ldots,x_n$ are variables or constants, then $R(x_1,\ldots,x_n)$ is a safe formula.

— The congraph of $R(x_1,\ldots,x_n)$ will be $C(V,E,\equiv,\neq)$, where $V$ is the set of variables among the $x$’s and $E$ is the edges in $E_R$, $\equiv$ the pairs in $\equiv_R$, and $\neq$ the pairs in $\neq_R$ in which both vertices correspond to $x$’s that are variables.

— If $\phi_1$ and $\phi_2$ are safe formulas, then $\phi_1 \land \phi_2$ is a safe formula.

— Let $C_{\phi_1} = (V_1,E_1,\equiv_1,\neq_1)$ and $C_{\phi_2} = (V_2,E_2,\equiv_2,\neq_2)$. The congraph of $\phi_1 \land \phi_2$ is $C_{\phi_1 \land \phi_2} = (V,E,\equiv,\neq)$, where $V = V_1 \cup V_2$ and $E = E_1 \cup E_2$, $\equiv = \equiv_1 \cup \equiv_2$, and $\neq = \neq_1 \cup \neq_2$.

— If $\phi$ is a safe formula and $C_\phi(V,E,\equiv,\neq)$ is its congraph and $E$ is empty, then $\lnot \phi$ is a safe formula. The congraph of $\lnot \phi$ will be $C_{\phi}(V,E,\neq,\equiv)$.

— If $\phi$ is a safe formula and $x$ is a variable, then $\exists x(\phi)$ is a safe formula.

— Let $C_\phi(V_\phi,E_\phi,\equiv_\phi,\neq_\phi)$ be the congraph of $\phi$. Here $C_{\exists x(\phi)} = (V,E,\equiv,\neq)$ where $V = V_\phi \setminus \{x\}$ and $E = \{(x_i,x_j) : x_i \neq x \neq x_j \text{ and } (x_i,x_j) \in E_\phi \text{ or } (x_i,x),(x,x_j) \in E_\phi \text{ or } (x_i,x),(x,x_j) \in E_\phi \text{ or } (x_i,x) \in E_\phi, (x,x_j) \in \equiv_\phi \text{ or } (x_i,x) \in \equiv_\phi, (x,x_j) \in \equiv_\phi\}$. 

Also, \(\equiv\) = \{(x_i,x_j) : x_i \neq x_j \land (x_i,x_j) \in \equiv^\phi \lor (x_i,x),(x,x_j) \in \equiv^\rho\} \land (x_i,x_j) \in \neq^\delta\).

**Example 4.1.** Let \(\phi = \exists y \exists z (R(x,z) \land \lnot Q(x,y))\) and let \(C_R = (V_R,E_R,\equiv_R,\neq_R)\) where \(V_R = \{x,z\}\) and \(E_R = \{(x,z)\}\) and \(\equiv=\neq = \emptyset\). Also let \(C_Q = (V_Q,E_Q,\equiv_Q,\neq_Q)\) where \(V_Q = \{x,y\}\) and \(E_Q = \emptyset\) and \(\equiv=\neq = \emptyset\).

Here \(\phi\) is safe with the given congraph typing. This is because \(R(x,z)\) and \(Q(x,y)\) are safe and have congraphs \(\{(x)\},\emptyset,\emptyset\) and \(\{(x,y)\},\emptyset,\emptyset\) respectively. The subformula \(\lnot Q(x,y)\) is safe and has congraph \(\{(x,y)\},\emptyset,\emptyset,\emptyset\). Further, \(R(x,z) \land \lnot Q(x,y)\) is safe and has congraph \(\{(x,y)\},\emptyset,\emptyset,\emptyset\). Finally, \(\phi\) is safe and has congraph \(\{(x)\},\emptyset,\emptyset\). \(\square\)

**Theorem 4.1.** Safe \(RC^{<\omega}\) programs are functions from valid gap-order constraint databases to gap-order constraint databases.

**Proof.** We can prove by induction on the structure of the formulas the following:

Let \(d\) be any valid input database. Then when evaluated each formula is valid for its congraph.

The condition is true for the first case. Assume that \(R\) has the scheme \((z_1,\ldots,z_n)\).

Then the first case is evaluated by substituting each \(z_i\) by \(x_i\) and conjoining \(z_i = x_i\) if \(x_i\) is a constant. Clearly, only if there was a constraint between two variables \(z_i\) and \(z_j\) and neither \(x_i\) nor \(x_j\) are constants, can there be a constraint between two variables \(x_i\) and \(x_j\).

The condition is true for conjunction, which is evaluated by natural join, because in the natural join each constraint tuple will be the conjunction of a constraint tuple in the two input constraint relations.

The condition is true for the negation which is evaluated by \(\Gamma\) because \(\Gamma\) always takes in constraint relations over \(C^{=,\neq,<}\) and gives output constraint relations over \(C^{=,\neq,<}\). Having an empty edge relation assures that in the relation to be negated there is no constraint of the form \(x <_g y\) where \(x,y\) are variables and \(g\) is any nonnegative integer constant.

The condition is true for existential quantification, which is evaluated using the variable elimination algorithm described in Example 3.1. As is clear from the algorithm \(x_i <_g x_j\) can be a constraint after the variable elimination only if it was a constraint or three other conditions are true, which are repeated in the definition of \(E\) with the only difference that the gap-values are ignored. Similarly, \(x_i = x_j\) can be a constraint after the variable elimination only if it was a constraint or the second conditions in the algorithm is true, which is repeated in the definition of \(\equiv\).

Finally, the only way \(\neq\) can be true is that it was true before. \(\square\)

### 4.2 Safe Relational Calculus for Set Order Constraint Databases

We assign to each input constraint relation a congraph type \(C(V,E,f)\) where \((V,E)\) is a directed graph and \(f\) is a coloring function from \(V\) to \{green,red\}.

We say that a constraint relation \(p(X_1,\ldots,X_k)\) has congraph type \(C(V,E,f)\) or is valid with respect to \(C(V,E,f)\) if for every constraint \(X_i \subseteq X_j\) in \(p\), the edge \((X_i,X_j) \in E\), and for every constraint \(X_i \subseteq C\) the color of \(X_i\) is red.

Now we define safe relational calculus with set order constraints. We assume that each relation symbol \(R\) is already assigned a congraph \(C_R(V_R,E_R,f_R)\). Each
safe relational calculus formula will also have a type that depends on the type of
the relation symbols in it.

—If \( R \) is an \( n \)-ary relation symbol and \( X_1, \ldots, X_n \) are variables or constants, then
\( R(X_1, \ldots, X_n) \) is a safe formula.

The congraph of \( R(X_1, \ldots, X_n) \) will be \( C(V, E, f) \), where \( V \) is the set of variables
among the \( X \)'s and \( E \) is the edges in \( E_R \) in which both vertices correspond to
\( X \)'s that are variables, and \( f \) assigns red to a vertex \( X \) if it is assigned red by
\( f_R \) or if there is a constant argument \( X_i \) and \( (X_i, X_i) \notin E_R \).

—If \( \phi_1 \) and \( \phi_2 \) are safe formulas, then \( \phi_1 \land \phi_2 \) is a safe formula.
Let \( C_{\phi_1} = (V_1, E_1, f_1) \) and \( C_{\phi_2} = (V_2, E_2, f_2) \). The congraph of \( \phi_1 \land \phi_2 \) is
\( C_{\phi_1 \land \phi_2} = (V, E, f) \), where \( V = V_1 \cup V_2 \) and \( E = E_1 \cup E_2 \). Here for each \( X \in \psi \) the
function \( f(X) = \text{red} \) iff \( (X \in V_1 \) and \( f_1(X) = \text{red} \) or \( (X \in V_2 \) and \( f_2(X) = \text{red} \).

—If \( \phi \) is a safe formula and \( C_{\phi}(V, E, f) \) is its congraph and \( E \) is empty and the
color of each vertex is green, then \( \neg \phi \) is a safe formula.
The congraph of \( \neg \phi \) will be the same as \( C_{\phi}(V, E, f) \).

—If \( \phi \) is a safe formula and \( X \) is a variable, then \( \exists X(\phi) \) is a safe formula.
Let \( C_{\phi}(V, E, f) \) be the congraph of \( \phi \). Here \( C_{\exists X(\phi)} = (V, E, f) \) where \( V = V \setminus \{X\} \)
and \( E = \{(X_i, X_j) : X_i \neq X \neq X_j \) and \( (X_i, X_j) \in E_{\phi} \) or \( (X_i, X_j) \in E_{\phi} \). Also, for each \( X_i \in V \) the value \( f(X_i) = \text{red} \) if \( f(X) = \text{red} \) and \( (X_i, X) \in \phi \).

**Example 4.2.** Let \( \phi = \exists Y(\neg P(X, Y) \land S(Y, Z)) \) and let \( C_P = (V_P, E_P, f_P) \)
where \( V_P = \{X, Y\} \) and \( E_P = \emptyset \) and \( f_P \) assigns green to both \( X \) and \( Y \). Also let
\( C_S = (V_S, E_S, f_S) \) where \( V_S = \{Y, Z\} \) and \( E_S = \{\{Y, Z\}\} \) and \( f_S \) also assigns green
to \( Y \) and red to \( Z \). Then \( \phi \) is safe. This is because \( \neg P(X, Y) \) will have same type
as \( P(X, Y) \) has. Also, the conjunction will have type \( (V, E, f) \) with \( V = \{X, Y, Z\} \)
and \( E = \{\{Y, Z\}\} \) and \( f \) assigns green to \( X \) and \( Y \) and red to \( Z \). Hence \( \phi \) will have
\( V = \{X, Z\} \), \( E = \emptyset \) and \( f_P \) will assign green to \( X \) and red to \( Z \).

**Theorem 4.2.** Safe \( R^{C \subseteq \sigma_{2 \times 1}} \) programs are functions from valid set order
constraint databases to set order constraint databases.

**Proof.** Similarly to the proof of Theorem 4.1, we use induction on the structure
of the formulas.

In the first case, the argument is similar to Theorem 4.1 except for the coloring
function. Note that the only way a vertex \( X \) can have a constant upper bound
is either if it had one before or there was an edge \( (X, X) \) in \( E_R \) and now \( X \) is
assigned a constant value. In the first case, \( X \) already is a red vertex according to
\( f_R \) and \( f \) will keep it red, while in the second case \( X \) will be colored red according
to \( f \). Hence the property is preserved that every vertex with a constant upper
bound is colored red.

The condition is true for conjunction, which is evaluated by natural join, because
in the natural join each constraint tuple will be the conjunction of a constraint
tuple in the two input constraint relations.

The condition is true for the negation which is evaluated by \( \Gamma_2 \) because \( \Gamma_2 \) always
takes in constraint relations over \( C_{\phi} \subseteq \phi \) and \( C \subseteq \phi \) and gives output constraint relations over
\( C_{\phi \subseteq \phi} \). Having an empty edge relation assures that in the relation to be negated
there is no constraint of the form $X \subseteq Y$ where $X,Y$ are integer set variables.

Having each vertex green assures that in the relation to be negated there is no
constraint of the form $X \subseteq C$ where $C$ is an integer set constant.

Existential quantification is evaluated using the variable elimination algorithm in
Example 3.2. The correctness of the coloring function follows from the following.
Suppose $X$ is the variable to be eliminated when the existential quantification is
evaluated. If $X$ is a red vertex, then there could be some constant $C$ such that
$X \subseteq C$ (and if $X$ is green then there cannot be such a $C$). Now if $(X_i, X)$ is an
edge in $E_\phi$, then there could also be a constraint $X_i \subseteq X$ in the constraint relation
associated with the subformula $\phi$. Then by transitivity $X_i \subseteq C$ could be true.

Hence $X_i$ could have a constant upper bound in the constraint relation associated
with the subformula $\exists X \phi$. Hence $X_i$ should be colored red by $f$. \hfill \Box

4.3 Safe Relational Calculus for $C_{\equiv, \neq, <, \cup} \subseteq, \leq}$ Constraint Databases

It is possible to use the results of the previous subsections in defining safe relational
calculus programs for $C_{\equiv, \neq, <, \cup} \subseteq, \leq}$ constraint databases. The idea is to
combine the handling of the integer variables in Section 4.1 and the handling of
set of integer variables in Section 4.2. The congraph will be $C(V, E, \equiv, \neq, \leq, f)$ but
the coloring function will assign a color only to set type vertices and only those
are required to be green before negation. If there are no set type vertices, then the
coloring function will be omitted.

Example 4.3. Suppose that a hospital laboratory tests each patient for a set
of symptoms $s_1, . . . , s_n$. In addition to the test results, it is also known which
disease is associated with which set of symptoms. Which patients are free from all
symptoms of which diseases?

We are going to use the EDB relations $\text{patient_symptom}(p, S)$, $\text{disease_symptom}(d, S)$ and $\text{elem}(s, S)$. Here $\text{patient_symptom}(p, S)$ is true if $p$ is the id number of
a patient and $S$ is the set of symptoms that patient has, $\text{disease_symptom}(d, S)$ is
true if $d$ is a disease and $S$ is the set of symptoms that is associated with it, and
$\text{elem}(s, S)$ is true if $s$ is a symptom that is an element of a set of symptoms $S$. Now
suppose that the congraphs of these relations are the following.

\begin{align*}
C_{\text{patient_symptom}} &= \{(p, S), \emptyset, \emptyset, f_{\text{patient_symptom}}\} \text{ where } f_{\text{patient_symptom}}(S) = \text{green}.
C_{\text{disease_symptom}} &= \{(d, S), \emptyset, \emptyset, \emptyset, f_{\text{disease_symptom}}\} \text{ where } f_{\text{disease_symptom}}(S) = \text{green}.
C_{\text{elem}} &= \{(s, S), \emptyset, \emptyset, \emptyset, f_{\text{elem}}\} \text{ where } f_{\text{elem}}(S) = \text{green}.
\end{align*}

The relational calculus formula $\phi(p, d)$ that we need is the following:
$\exists S_1, S_2 \text{ patient_symptom}(p, S_1) \land \text{disease_symptom}(d, S_2) \land \lnot(\exists s \text{ elem}(s, S_1) \land \text{elem}(s, S_2))$. Here $\phi$ expresses that $p$ is free from all symptoms of disease $d$ if there is no
symptom $s$ which is a common element to the set of symptoms found in patient $p$ and
the set of symptoms commonly associated with disease $d$. We claim that $\phi$ is
a safe query.

Here the congraph of $\phi_1 = \text{elem}(s, S_1) \land \text{elem}(s, S_2)$ is:

\begin{align*}
C_{\phi_1} &= \{(s, S_1, S_2), \emptyset, \emptyset, \emptyset, f_{\phi_1}\} \text{ where } f_{\phi_1}(S_1) = f_{\phi_1}(S_2) = \text{green}.
\end{align*}

The congraph of the subexpression $\exists s \phi_1$ is:

\begin{align*}
C &= \{(S_1, S_2), \emptyset, \emptyset, \emptyset, f\} \text{ where } f(S_1) = f(S_2) = \text{green}.
\end{align*}
The congraph of the negation of this is the same. The congraph of \( \text{patient} \_\text{symptom} (p, S) \land \text{disease} \_\text{symptom} (d, S) \land \neg(\exists s \ (\text{elem} (s, S1) \land \text{elem} (s, S2))) \) is:

\[ C = \{(p, d, S1, S2), 0, 0, 0, f \} \) where \( f(S1) = f(S2) = \text{green}. \]

Finally, the congraph of \( \phi \) is:

\[ C_\phi = \{(p, d), 0, 0, 0) \}

Hence \( \phi \) is a safe query. To make the example more concrete let’s evaluate \( \phi \) on the following EDB instance. The \( \text{patient} \_\text{symptom} \) relation is \( p = 101, S = \{1, 2\} \) \lor \( p = 102, S = \{3, 4\} \) \lor \( p = 103, S = \{3\} \), the \( \text{disease} \_\text{symptom} (d, S) \) relation is \( d = 1, S = \{1, 4\} \) \lor \( d = 2, S = \{2, 3\} \) \lor \( d = 3, S = \{3, 4\} \), and the \( \text{elem} (s, S) \) relation is \( s = 1, 1 \in S \lor \ldots \lor s = 4, 4 \in S \). The congraph of each EDB relation is as required. In this instance, \( \text{elem} (s, S1) \land \text{elem} (s, S2) \) evaluates to:

\[ (s = 1, 1 \in S1, 1 \in S2) \lor \ldots \lor (s = 4, 4 \in S1, 4 \in S2). \]

Hence \( \exists s \ (\text{elem} (s, S1) \land \text{elem} (s, S2)) \) is:

\[ (1 \in S1, 1 \in S2) \lor \ldots \lor (4 \in S1, 4 \in S2). \]

The negation of that will consist of 16 constraint tuples:

\[ (1 \notin S1, 2 \notin S1, 3 \notin S1, 4 \notin S1) \lor \ldots \lor (1 \notin S2, 2 \notin S2, 3 \notin S2, 4 \notin S2) \]

The expression \( \text{patient} \_\text{symptom} (p, S1) \land \text{disease} \_\text{symptom} (d, S2) \land \neg(\exists s \ (\text{elem} (s, S1) \land \text{elem} (s, S2))) \) will be:

\[ (p = 101, d = 3, S1 = \{1, 2\}, S2 = \{3, 4\}) \lor (p = 103, d = 1, S1 = \{3\}, S2 = \{1, 4\}) \]

Hence \( \phi(p, d) \) will be \( (p = 101, d = 3) \lor (p = 103, d = 1) \). This means that patient 101 is free from all symptoms of disease 3 and patient 103 is free from all symptoms of disease 1.

The above query can be also expressed in several languages (SQL and others) that do not use constraint databases [Rao 1996]. However, the use of set variables and set order constraints enabled a more compact and higher-level expression. The use of SQL for this and similar queries is more awkward (see Figure 2 in [Rao 1996]).

5. SAFE STRATIFIED DATALOG QUERIES

5.1 Safe Stratified Datalog for Gap-Order Constraint Databases

We start with some basic definitions.

Let \( C = (V, E, \equiv, \neq) \) be a congraph of a relation over \( C_{=, \neq, <_s} \). The transitive closure of \( C \) is \( C^* = (V, E^*, \equiv^*, \neq) \) where \( \equiv^* \) is the congruence closure of the \( \equiv \) relation, and \( (x_i, x_j) \in E^* \) if and only if it is in \( E \) or there are pairs \( (x_i, z_1), \ldots, (z_i, x_j) \) in \( E \cup \equiv \) with at least one pair in \( E \).

Let \( r \) be any rule with variables \( x_1, \ldots, x_n \), and of the form \( A_0 \leftarrow A_1, A_2, \ldots, A_l \). Then the congraph of \( r \) is the transitive closure of the union of the congraphs of \( A_1, \ldots, A_l \) after the necessary renamings. If two argument variables \( x_i \) and \( x_j \) in the rectified form(s) of some relation(s) are renamed within the rule body by the same variable, then \( (x_i, x_j) \) is added to \( \equiv \).

We define the congraph of IDBs of any semipositive Datalog program as the output of algorithm Find-\text{IDB}-Congraphs.

\textbf{Algorithm Find-\text{IDB}-Congraphs}

\textbf{INPUT:} A semipositive Datalog \( \prec \succeq \) program \( \Pi \) and a congraph for each EDB.
OUTPUT: A congraph of each IDB of $\Pi$.

FOR each IDB relation $p_m(x_1, \ldots, x_k)$ DO
assign to $p_m$ a congraph $C_m(V_m, E_m, \equiv_m, \not\equiv_m) = C_m(\{x_1, \ldots, x_k\}, \emptyset, \emptyset, \emptyset)$.
END-FOR

WHILE any changes in IDB congraphs DO
FOR each rule $r$ with head $p_m(x_1, \ldots, x_k)$ DO
Find $C_r(V_r, E_r, \equiv_r, \not\equiv_r)$ the congraph of rule $r$.
Let $E_m = E_m \cup \{(x_i, x_j) : (x_i, x_j) \in E_r\}$.
Let $\equiv_m = \equiv_m \cup \equiv_r$.
Let $\not\equiv_m = \not\equiv_m \cup \not\equiv_r$.
END-FOR
END-WHILE

In this section let $C_{R, \Pi} = (V_{R, \Pi}, E_{R, \Pi}, \equiv_{R, \Pi}, \not\equiv_{R, \Pi})$ denote the congraph of relation $R$ in program $\Pi$.

Let $\Pi_1$ and $\Pi_2$ be two semipositive $Datalog^{\prec\prec}$ programs. We say that $\Pi_1$ is congraph compatible with $\Pi_2$ if and only if for each relation $R$ that is common to both $\Pi_1$ and $\Pi_2$, $E_{R, \Pi_1} \subseteq E_{R, \Pi_2}$, $\equiv_{R, \Pi_1} \subseteq \equiv_{R, \Pi_2}$, and $\not\equiv_{R, \Pi_1} \subseteq \not\equiv_{R, \Pi_2}$.

We say that a semipositive $Datalog^{\prec\prec}$ program is safe if and only if the congraph of any negated EDB has an empty set of edges.

We say that a stratified $Datalog^{\prec\prec}$ program is safe if and only if it consists of $\Pi_1 \cup \ldots \cup \Pi_n$ where each $\Pi_i$ is a safe semipositive $Datalog^{\prec\prec}$ program and each $\Pi_i$ is congraph compatible with each $\Pi_{i+1}, \ldots, \Pi_n$. Note that by repeatedly calling algorithm Find-IDB-Congraphs on each stratum of a stratified $Datalog$ program, we can test whether it is safe or not.

THEOREM 5.1. Safe stratified $Datalog^{\prec\prec}$ programs are functions from valid gap-order constraint databases to gap-order constraint databases.

PROOF. Let $\Pi = \Pi_1 \cup \ldots \cup \Pi_n$ be a safe stratified $Datalog^{\prec\prec}$ program where each $\Pi_i$ is a safe semipositive $Datalog^{\prec\prec}$ program and each $\Pi_i$ is congraph compatible with each $\Pi_{i+1}, \ldots, \Pi_n$.

We prove by induction on the strata of $\Pi$ that when $\Pi$ is evaluated by algorithm Find-Closed-Form each IDB relation of $\Pi$ is valid for its congraph if each EDB relation of $\Pi$ is valid for its congraph. That is, each IDB relation has in it some gap-order constraint $x_i \prec x_j$ (or $x_i = x_j$ or $x_i \not\prec x_j$) only if the edge (or $\equiv$ or $\not\equiv$) relation of its congraph contains $(x_i, x_j)$.

Let’s consider first the evaluation of $\Pi_1$. The semantics of $\Pi_1$ is equivalent to the semantics of $\Pi_1$ which is $\Pi_1$ with each negated EDB relation replaced by its complement. The complement of each negated EDB relation must be a constraint relation over $C_{\cdot, \not\equiv}$ because each EDB of $\Pi_1$ is also an EDB of $\Pi$ and is valid for its congraph, which by definition of safe semipositive programs cannot contain any edge. Hence $\Pi_1$ can be evaluated by algorithm Find-Closed-Form.

Now suppose that during some iteration of the repeat-until loop of algorithm Find-Closed-Form a constraint tuple $t_0$ is added to an IDB relation $p_0$ such that $t_0$ contains a gap-order constraint $x_{i_1} \theta x_{i_2}$ for some argument variables $x_{i_1}$ and $x_{i_2}$.
and nonnegative integer $g$. There are three cases depending on whether $\theta$ is $<_{g}, =,$

or $\neq$. In these cases we have to prove that the edge, the $\equiv$ or the $\neq$ relation in the

congraph of $p_0$ must contain the pair $(x_{11}, x_{12})$. We prove the first case, the other two cases are similar.

Let $p_0(x_1, \ldots, x_k) \models p_1, \ldots, p_n$ be the rule that was used to derive $t_0$ and let

$y_1, \ldots, y_m$ be the variables that occur only in the rule body. To derive $t_0$

there must be in the previous iteration constraint tuples $t_1, \ldots, t_n$ in $p_1, \ldots, p_n$

respectively such that $(t_1, \ldots, t_n)$ is a constraint tuple in the join of $p_1, \ldots, p_n$ and

$
\pi_{x_1, \ldots, x_k}(t_1, \ldots, t_n) = t_0. Then elim(y_1, \ldots elim(y_m, (t_1, \ldots, t_n))) = t_0
$

by the definition of $\pi$. Further, this chain of variable eliminations in $C_{\equiv, \neq, <_{g}}$ can yield the

constraint $x_{11} <_{g} x_{12}$ only if either it was already a constraint in $t_0$ or there exist

constraints $x_{11} \theta_1 z_1, x_{12} \theta_2 z_2$ where at least one $\theta$ is $<_{g}$ and the others are either

$=$ or $<_{g}$ for some nonnegative integer $g$ and each $z$ is one of $x_1, \ldots, x_k, y_1, \ldots, y_m$.

Since there are such constraints in $(t_1, \ldots, t_n)$, each constraint of the form $z_{j1} \theta z_{j2}$

must occur in at least one $t_i$ for $1 \leq i \leq n$. Then if $\theta =$, then the equivalence

relation $\equiv$ of $t_i$ contains $(z_{j1}, z_{j2})$ and if $\theta <_{g}$ then the edge relation $E$ of $t_i$ contains

$(z_{j1}, z_{j2})$. In either way, by the definition of transitive closure, the transitive

closure of the congraph of the unions of the congraphs of $p_1, \ldots, p_n$ will contain the edge

$(x_{11}, x_{12})$. This shows that the constraint IDB relations of $\Pi_1$ will satisfy their

congraphs. By the definition of congraph compatibility, each IDB relation of $\Pi_1$

that is used in a higher stratum as an EDB relation is valid also for its congraph in that stratum.

Therefore, the above argument for $\Pi_1$ can be repeated for each successive stratum. $\square$

Next we give an example of applying algorithm $Find$-$IDB$-Congraphs.

**Example 5.1.** Suppose that we know the distance in miles between any pair of cities with a direct road connection on a map and we need to find the length of the shortest path between any pair of cities. The following $DataLoc^\ast <_{z}$ program with four rules, $r_1, r_2, r_3, r_4$ respectively, performs this query.

$shortest(x, y, s) \models path(x, y, 0, s), \neg not\_shortest(x, y, s)$.

$not\_shortest(x, y, s_2) \models path(x, y, 0, s_1), path(x, y, 0, s_2), s_1 < s_2$.

$\neg shortest(x, y, s_2) \models path(x, y, 0, s_1), path(x, y, 0, s_2), s_1 < s_2$.

$ \models path(x, y, s_1, s_2) , distance(z, y, s_3, s_2)$.

The input relation $\neg shortest$ describes direct distances between cities in miles using constraint database tuples. For example, the constraint tuple $x = 95, y = 77, s_1 <_{59} s_2$ expresses the fact that city 77 is 60 miles from city 95, or to read it more literally, if we can reach city 95 within $s_1$ miles then we can reach city 77 within $s_2$ miles for any $s_1$ and $s_2$ that satisfies $s_1 <_{59} s_2$.

Let’s compute the IDB congraphs in $\Pi_1$. Figure 1 shows the edges in the congraphs of each rule and each relation at the end of each iteration $i$. For $i = 0$ the congraphs of the input database are shown. The input database relation $\neg shortest$ will have in its congraph only the edge $(s_1, s_2)$, and all the other relations will not have any edge in their congraphs. None of the rule congraphs will have any edge in them either.
After the first iteration, the congraph of \( r_2 \) will have only the edge \((s_1, s_2)\) because of the constraint \( s_1 < s_2 \) occurring in the rule, the congraph of rule \( r_3 \) will have \((s_3, s_2)\) in it because of the renaming of the congraph of the distance relation, while the congraph of \( r_4 \) will have the edge \((s_1, s_2)\) added to it, because on the right hand side the distance relation also contains this edge. Because of the changes in \( r_4 \), the congraph of path will also have the edge \((s_1, s_2)\) added to it.

After the second iteration, the congraphs of rules \( r_2 \) and \( r_4 \) will remain unchanged, while the congraph of \( r_3 \) will have the edge \((s_1, s_3)\) added to it because of the renaming of the congraph of the path relation, and also the edge \((s_1, s_2)\) added to it because it is the shortcut of the edges \((s_1, s_3)\) and \((s_3, s_2)\) already in the congraph. This change in \( r_3 \) however will not cause any change in the congraph of the path relation. Therefore, none of the IDB relation congraphs will change from the end of iteration 1 to the end of iteration 2. Hence the algorithm will terminate and return the congraphs of the IDB relations within the last row.

Further, let’s find the IDB congraphs of \( \Pi_2 \). In there the relations \( path(x, y, 0, s) \) and \( not\_shortest(x, y, s) \) have no edges in their congraphs. Hence the congraphs of rule \( r_1 \) and the shortest relation will also have no edges. Note that \( \Pi_1 \) is congraph compatible with \( \Pi_2 \) and that \( \Pi_1 \cup \Pi_2 \) is a safe program because the congraph of the only relation which is negated contains no edges.

Next we prove that for safe programs the query evaluation algorithm returns in finite time the perfect model as expected.

**Theorem 5.2.** There is an algorithm that for any safe stratified Datalog\(^{-\leq \ast}\) program \( \Pi \) and valid input database \( d \) returns the perfect model of \( \Pi \) in constraint database form.

**Proof.** Let \( \Pi = \Pi_1 \cup \ldots \cup \Pi_n \) be any safe stratified Datalog\(^{-\leq \ast}\) program where each \( \Pi_i \) is a safe semipositive Datalog\(^{-\leq \ast}\) program and each \( \Pi_i \) is congraph compatible with each \( \Pi_{i+1}, \ldots, \Pi_n \).

The perfect model of \( \Pi \) is equivalent to \( \Pi_{\Pi}(\ldots \Pi_1(d)\ldots) \) where each \( \Pi_i \) is \( \Pi_i \) with each negated EDB relation replaced by its complement. The complement of each negated EDB can be found using the type-restricted complement operator \( \Gamma \), which returns a relation over \( C_\pi \). Then the semantics of \( \Pi_i \) can be evaluated using algorithm Find-Closed-Form. This can be repeated for each \( \Pi_i \) for \( 1 \leq i \leq n \) because of the safety restriction and Theorem 5.1. Since each of the successive computation of algorithm Find-Closed-Form terminates as proven in [Revesz 1993], the computation of the perfect model also terminates. The proof that the computation returns the perfect model follows from the general fixpoint semantics theory [van Emde Boas and Kowalski 1976] and Lemma 3.1. \( \Box \)

The next example illustrates that the query evaluation algorithm always returns relations with a type that conforms to our expectations.
EXAMPLE 5.2. Let’s return to Example 5.1. We have seen that it was identified to be a safe query.

Let \( \phi^i_p \) and \( F^i_r \) denote the constraint relations assigned respectively to relation \( R \) and to rule \( r^i \) at the end of iteration \( i \) of the repeat-until loop.

Suppose we have \( \phi \equiv (x = 1, y = 2, s_1 < s_2) \land (x = 1, y = 3, s_1 < s_2) \lor (x = 2, y = 4, s_1 < s_2) \lor (x = 3, y = 4, s_1 < s_2) \). We also have \( \phi^0_p = \phi^0_{ns} = \phi^0_s = \text{false} \).

Let’s see now what happens when algorithm \textit{Find-Closed-Form} is evaluated on stratum 1 which contains rules \( r_2, r_3 \) and \( r_4 \). For each iteration \( i \) of the repeat-until loop, the algorithm finds:

\[
\begin{align*}
F^i_2 &= (\rho_{s_2/s_1} \circ \phi^{i-1}_p) \land (\sigma_{s_2} = 0 \phi^{i-1}_p) \land \phi < \\
F^i_3 &= \pi_{x,y,s_1,s_2} ((\rho_{y/z} \circ \phi^{i-1}_p) \land (\rho_{x/z} \circ \phi_d)) \\
F^i_d &= \phi_d
\end{align*}
\]

In the first iteration of the repeat-until loop, we have \( \phi^0_p = \emptyset \). Therefore, both \( F^2_2 \) and \( F^2_3 \) will be false. As we noted, \( F^2_1 = \phi_d \). This has the net effect of copying each constraint tuple in the distance to the path relation. Hence by the end of the first iteration, we have \( \phi^1_p = \phi^1_{ns} = \phi_s = \text{false} \). Note that the \( \psi \) variables are used only to detect whether any \( \phi \) changed. Since \( \phi_p \) changed in value, we enter the loop again.

In the second iteration of the repeat-until loop, by substituting into the second of the above equations, we find that \( \rho_{y/z} \circ \phi^{i-1}_p \) is:

\[
(x = 1, z = 2, s_1 < s_3) \land (x = 1, z = 3, s_1 < s_3) \land (x = 3, z = 4, s_1 < s_3)
\]

and after projection we get: \( (x = 1, y = 3, s_3 < s_2) \lor (x = 1, y = 4, s_3 < s_2) \lor (z = 1, y = 3, s_3 < s_2) \lor (z = 1, y = 3, s_2 < s_2) \lor (z = 3, y = 4, s_3 < s_2) \lor (z = 3, y = 4, s_3 < s_2) \).

The join of the above two will be:

\[
(x = 1, z = 2, y = 4, s_1 < s_2) \land (x = 1, z = 3, y = 4, s_1 < s_3) \land (x = 3, y = 4, 15 < s_2)
\]

We find that \( \phi^2_p = \phi_d \lor F^2_2 \) and \( \phi^2_{ns} = F^2_2 \). Since there are changes in the IDB relations, we again enter the repeat-until loop.

In the third iteration of the repeat-until loop, similarly to the above, we find that \( F^3_2 =\)

\[
\begin{align*}
F^3_2 &= F^2_2 \lor (x = 1, y = 4, 50 < s_2) \lor (x = 1, y = 4, 60 < s_2) \lor (x = 1, y = 4, 60 < s_2) \lor (x = 1, y = 4, 60 < s_2) \lor (x = 1, y = 4, 60 < s_2) \lor (x = 1, y = 4, 60 < s_2)
\end{align*}
\]

We also find that \( \phi^3_{ns} = \phi^2_{ns} \lor (x = 1, y = 4, 50 < s_2) \lor (x = 1, y = 4, 60 < s_2) \lor (x = 1, y = 4, 60 < s_2) \lor (x = 1, y = 4, 60 < s_2) \lor (x = 1, y = 4, 60 < s_2) \lor (x = 1, y = 4, 60 < s_2) \).

In the fourth iteration of the repeat-until loop, none of the \( F \)s and \( \phi \)s will change.

In stratum 2 the only IDB relation is \textit{shortest}. To find the value of this relation, we have to enter again the repeat-until loop. Here \( \text{path}(x, y, 0, s) \) is the relation:

\[
\begin{align*}
(x = 1, y = 2, 19 < s) \lor (x = 1, y = 3, 44 < s) \lor (x = 2, y = 4, 29 < s) \lor (x = 3, y = 4, 45 < s) \lor (x = 1, y = 4, 49 < s) \lor (x = 1, y = 4, 59 < s) \lor (x = 2, y = 4, 30 < s)
\end{align*}
\]

while \( \text{not shortest}(x, y, s) \) is:

\[
(x = 1, y = 2, 20 < s) \lor (x = 1, y = 3, 45 < s) \lor (x = 2, y = 4, 30 < s)
\]
(x = 3, y = 4, 15 < s) ∨ (x = 1, y = 4, 50 < s) ∨ (x = 1, y = 4, 60 < s)

We find the negation of not _shortest_ using De Morgan's laws and simplifying:

(s < 16) ∨ (x ≠ 3, s < 21) ∨ (y ≠ 4, s < 21) ∨ (x ≠ 1, x ≠ 3, s < 31) ∨
(x ≠ 3, y ≠ 2, s < 31) ∨ (x ≠ 2, x ≠ 3, y ≠ 2, s < 46) ∨ (y ≠ 2, y ≠ 4, s < 46) ∨
(x ≠ 2, x ≠ 3, y ≠ 2, y ≠ 3, s < 51) ∨ (x ≠ 1, x ≠ 2, x ≠ 3) ∨ (x ≠ 1, y ≠ 4) ∨
(y ≠ 2, y ≠ 3, y ≠ 4)

Finally the join of path and the negation of not _shortest_ will be:

(x = 1, y = 2, s = 20) ∨ (x = 1, y = 3, s = 45) ∨ (x = 2, y = 4, s = 30) ∨
(x = 3, y = 4, s = 15) ∨ (x = 1, y = 4, s = 50)

Note that we get a unique s for each pair of x and y. The s is the length of the shortest path between x and y as we expected.

It should be noted that the shortest path length query cannot be expressed using stratified Datalog with only relational databases (see page 934 in [Ullman 1989]).

Hence the use of constraint databases was important in the above example.

### 5.2 Safe Stratified Datalog for Set Order Constraint Databases

For Datalog$^\neg \leq r^{(x^u)}$ programs we define the congraph of a rule as follows.

**Definition 5.1.** Let r be any rule with variables $X_1, \ldots, X_n$ and of the form $A_0 \leftarrow A_1, A_2, \ldots, A_l$. Then the congraph of r is $C_r = (V_r, E_r, f_r)$ where $V_r$ is the union of the vertices and $E_r$ is the transitive closure of the edges in the congraphs of $A_1, \ldots, A_l$ after the necessary renamings. Also, $f_r$ is red for any vertex if and only if it is red in any of the $A_i$.

We define the congraph of IDBs of any semipositive Datalog$^\neg \leq r^{(x^u)}$ program as the output of algorithm *Find-DB-Congraphs*.

---

**Algorithm Find-DB-Congraphs**

**INPUT:** A semipositive Datalog$^\neg \leq r^{(x^u)}$ program $\Pi$ and a congraph for each EDB.

**OUTPUT:** A congraph for each IDB of $\Pi$.

**FOR** each IDB relation $p_m(X_1, \ldots, X_k)$ **DO**

assign to $p_m$ a congraph $C_m(V_m, E_m, f_m)$ with $V_m = \{X_1, \ldots, X_k\}$, $E_m = \emptyset$, and $f_m$ coloring each vertex in $V_m$ green.

**END-FOR**

**WHILE** any changes in IDB congraphs **DO**

**FOR** each rule r with head $p_m(X_1, \ldots, X_k)$ **DO**

Find $C_r(V_r, E_r, f_r)$ the congraph of rule r.

Let $E_m = E_m \cup \{(X_i, X_j) : (X_i, X_j) \in E_r\}$.

Let $f_m(V) = \text{red}$ iff $f_r(V) = \text{red}$ or exist $V_1, \ldots, V_h$ such that $(V, V_1), \ldots, (V_{h-1}, V_h) \in E_r$ and $f_r(V_h) = \text{red}$.

**END-FOR**

**END-WHILE**

---

Let $\Pi_1$ and $\Pi_2$ be two semipositive Datalog$^\neg \leq r^{(x^u)}$ programs. We say that $\Pi_1$ is congraph compatible with $\Pi_2$ if and only if for each relation $R$ that is common to
both $\Pi_1$ and $\Pi_2$, $E_{R,\Pi_1} \subseteq E_{R,\Pi_2}$, and for each $V$ vertex, if $f_{R,\Pi_1}(V) = \text{red}$ then $f_{R,\Pi_2}(V) = \text{red}$.

We say that a semipositive $Datalog^{-\leq v(x^*)}$ program is safe if and only if the congraph of any negated EDB has an empty set of edges and only green vertices.

We say that a stratified $Datalog^{-\leq v(x^*)}$ program is safe if and only if it consists of $\Pi_1 \cup \ldots \cup \Pi_n$ where each $\Pi_i$ is a safe semipositive $Datalog^{-\leq v(x^*)}$ program and each $\Pi_i$ is congraph compatible with each $\Pi_{i+1}, \ldots, \Pi_n$.

Similarly to Theorems 5.1 and 5.2 we can show the following.

**Theorem 5.3.** Safe stratified $Datalog^{-\leq v(x^*)}$ programs are functions from valid set order constraint databases to set order constraint databases.

**Proof.** The proof of this is similar to that of Theorem 5.1. □

**Theorem 5.4.** There is an algorithm that for any safe stratified $Datalog^{-\leq v(x^*)}$ program $\Pi$ and valid input database $d$ returns the perfect model of $\Pi$ in constraint database form.

**Proof.** The proof of this is similar to that of Theorem 5.2. □

### 5.3 Safe Stratified Datalog for $C_{=,\neq,\leq,\prec} \cup C_{\in,\notin,\subseteq}$ Constraint Databases

Safe stratified Datalog programs for $C_{=,\neq,\leq,\prec} \cup C_{\in,\notin,\subseteq}$ constraint databases, denoted $Datalog^{-\leq v(x^*)}$, can be defined and handled by a combination of the techniques in the previous two subsections. We will illustrate the combination in the following example.

**Example 5.3.** Let's consider the following semipositive $Datalog^{-\leq v(x^*)}$ program $\Pi = \{r_1, r_2, r_3\}$.

- $\text{out}(S) := \text{select}(k, S), \text{last}(k)$.
- $\text{select}(j, S) := \text{select}(i, S), \text{next}(i, j), \neg \text{cond}(j, S)$.
- $\text{select}(0, S) := S \subseteq C$.

where $C$ is some set constant. In $\Pi$ the EDB relations are $\text{next}(i, j)$, $\text{last}(k)$, and $\text{cond}(j, S)$, while the IDB relations are $\text{select}(j, S)$ and $\text{out}(S)$. Program $\Pi$ can be used to make a selection from a group of items $C$ by taking care that a set of conditions is avoided. We illustrate this further in Example 5.4, but here we only show that $\Pi$ is a safe program assuming that the EDBs have the following congraphs.

- $C_{\text{next}} = \{(i, j), \emptyset, \emptyset, \emptyset\}$.
- $C_{\text{last}} = \{(k), \emptyset, \emptyset, \emptyset\}$.
- $C_{\text{cond}} = \{(j, S), \emptyset, \emptyset, \emptyset, f_{\text{cond}}\}$ where $f_{\text{cond}}(S) = \text{green}$.

Note that the $\neg \text{cond}$ is safe and its congraph will be the same as that of the $\text{cond}$ relation. Now let's look at how the IDBs change. Initially there are no edges in the IDB congraphs and all set type vertices are green. In the first iteration we find that

- $C_{r_1} = \{(k, S), \emptyset, \emptyset, \emptyset, f_{r_1}\}$ where $f_{r_1}(S) = \text{green}$.
- $C_{r_2} = \{(i, j, S), \emptyset, \emptyset, \emptyset, f_{r_2}\}$ where $f_{r_2}(S) = \text{green}$.
- $C_{r_3} = \{(S), \emptyset, \emptyset, \emptyset, f_{r_3}\}$ where $f_{r_3}(S) = \text{red}$.

In the above $f_{r_3}(S)$ is red because of the upper bound constant $C$. From the rule congraphs we calculate that:

- $C_{\text{out}} = \{(S), \emptyset, \emptyset, \emptyset, f_{\text{out}}\}$ where $f_{\text{out}}(S) = \text{green}$.
\[ C_{\text{select}} = (\{j, S\}, \emptyset, \emptyset, f_{\text{select}}) \text{ where } f_{\text{select}}(S) = \text{red.} \]

In the second iteration we find no changes. Hence we exit the while loop and conclude that \( \Pi \) is a safe program. \( \square \)

We continue the previous example by considering the following problem:

**Example 5.4.** A department needs to select a team of students to participate in a programming contest. The students eligible to participate are Cathy, David, Pat, Mark, Tom, Lilly, and Bob. The selection must avoid the following conditions. (1) Bob is selected and David is not selected. (2) David, Pat, and Mark are all selected. (3) Tom, Cathy, and Bob are all selected. (4) Pat is selected and neither Tom nor Lilly is selected. (5) Neither Cathy nor Lilly is selected. (6) Both Cathy and Lilly are selected. Find all possible teams that may be sent to the programming contest.

Let \( c, d, p, m, t, l, b \) be the integer constants denoting the id numbers of the candidate team members. We use the program in Example 5.3 with \( C = \{c, d, p, m, t, l, b\} \), the \( \text{last} \) relation equal to \( k = 6 \), the \( \text{next} \) relation equal to \( (i = 0, j = 1) \lor \ldots \lor (i = 5, j = 6) \) and the \( \text{cond} \) relation, expressed in Prolog style, equal to:

\[
\text{cond}(1, S) := b \in S, d \notin S.
\]

\[
\text{cond}(2, S) := \{d, p, m\} \subseteq S.
\]

\[
\text{cond}(3, S) := \{t, c, b\} \subseteq S.
\]

\[
\text{cond}(4, S) := p \in S, t \notin S, l \notin S.
\]

\[
\text{cond}(5, S) := c \notin S, l \notin S.
\]

\[
\text{cond}(6, S) := \{c, l\} \subseteq S.
\]

Clearly the congraph of each of the EDB relations is as required in Example 5.3. \( \square \)

**6. The Computational Complexity of Safe Queries**

In this section we analyze the time and space required for testing whether a program is safe and for evaluating safe queries (safe programs + valid constraint database inputs).

When analyzing the evaluation of queries, we are interested in *data complexity*. Data complexity is the measure of the computational complexity of fixed queries as the size of the input database grows [Chandra and Harel 1982; Vardi 1982]. The rationale behind this commonly used measure is that in relational database practice the size of the database typically dominates by several orders of magnitude the size of the query. We assume that data complexity will be also a realistic measure for constraint databases. However, this assumption may or may not be actually true in future constraint database systems.

**6.1 The Complexity of Testing the Safety of Programs**

At first we show that it is relatively easy to test whether a given program is safe or not.
Theorem 6.1. Whether a \( RC^{<z} \), \( RC^{<z} \), stratified \( Datalog^{<z} \), or stratified \( Datalog^{<z} \) program is safe can be tested in PTIME in its size.

Proof. The proof in the case of \( RC^{<z} \) and \( RC^{<z} \) follows from induction on the structure of the formulas. For each expression of the form \( \phi \wedge \psi \), or \( \neg \phi \), or \( \exists x \phi \) the congrap can be found in PTIME in the size of the congraphs of \( \phi \) and \( \psi \). The proof in the case of \( Datalog^{<z} \) and \( Datalog^{<z} \) follows from the fact that each EDB and IDB relation has a unique congrap and that the computation of the congraphs by algorithm Find-IDB-Congraps is monotone. That is, if \( C \) is the congrap of any IDB relation of the form \( p_0(x_1, \ldots, x_h) \), then its set of vertices is fixed \( V = \{ x_1, \ldots, x_h \} \), while its edge relation is a subset of \( V \times V \) and is monotone increasing. Further, in the case of \( Datalog^{<z} \) the \( \equiv \) and \( \neq \) relations are also a subset of \( V \times V \) and are monotone increasing. In the case of \( Datalog^{<z} \) the coloring function is also monotone as it can only change a green color to a red, but never a red to green. Hence for each stratum the computation of algorithm Find-IDB-Congraps must terminate in PTIME in the size of \( V \), which is bounded linearly by the size of the program. Finally, the number of IDB relations and the number of strata are also bounded linearly by the size of the program.

6.2 The Complexity of Safe Relational Calculus Queries

Since relational calculus queries can be translated into relational algebra queries, we will study first the computational complexity of the algebraic operators for constraint databases.

Let's consider constraint databases over \( C_{\neq, <, =} \). We define a normal form for constraint tuples as follows. The normal form contains at most one constraint between any two distinct variables, and at most one upper bound and one lower bound constraint for each variable. A constraint relation is in normal form if all tuples in it are in normal form. Putting a constraint relation into normal form requires only a polynomial time in the number of atomic constraints in it [Revesz 1993].

Theorem 6.2. Let \( p_1(x_1, \ldots, x_k) \) and \( p_2(y_1, \ldots, y_k) \) be any two fixed relation schemes where some of the \( x \)s may equal some of the \( y \)s. Then for any constraint relation instances of \( p_1 \) and \( p_2 \) in normal form with \( n_1 \) and \( n_2 \) tuples respectively, the projection, selection, rename, and complement operators on \( p_1 \) can be done in time polynomial in \( n_1 \) and the join of \( p_1 \) and \( p_2 \) can be done in time polynomial in \( n_1 + n_2 \).

Proof. Note that here we take \( k \) and \( k_2 \) to be fixed constants while \( n_1 \) and \( n_2 \) are variables. Also, \( n_1 \) and \( n_2 \) are proportional to the size of the constraint relation instances for \( p_1 \) and \( p_2 \) because in normal form each constraint tuple has at most a constant \( 2k + k(k - 1) \) number of atomic constraints. Hence for each tuple the \( \text{elim} \) and \( \text{sat} \) functions can be done in constant time. Hence it is easy to see that for the projection operation the time complexity is proportional to the number of tuples and for the join operation the time complexity is proportional to the product of the number of tuples in \( p_1 \) and \( p_2 \). It is also straightforward that for the select and rename operations the time complexity is linear in \( n_1 \).

For the type-restricted complement operator we can assume that the constraint database instance of \( p_1 \) is over \( C_{\neq, <, =} \). Let \( p_1 \) be \( t_1 \vee \ldots \vee t_m \) where each \( t_i = \)
(a_1 \land \ldots \land a_{i,t_i}) for t_i \leq 2k + k(k - 1). The complement of p_i can be found using
De Morgan’s law as follows:
\[ \neg(t_1 \lor \ldots \lor t_{n_1}) = \neg t_1 \land \ldots \land \neg t_{n_1} = \]
\[ (-a_{1,1} \lor \ldots \lor -a_{1,t_1}) \land \ldots \land (-a_{n_1,1} \lor \ldots \lor -a_{n_1,t_1}) \]

We need to put the above formula into disjunctive normal form. A naive way of
doing that would be the following. Let relation temp_1 contain the negation of the
atomic formulas in t_1. Let for each 2 \leq i \leq n_1 the relation temp_i contain the join of
relation temp_{i-1} with the negation of the atomic formulas in t_i. The complement of
p_1 is equal to temp_{n_1}. This process in the worst case may take O((2k + k(k - 1))^{n_1})
time and yield that many constraint tuples. However, we can do better than that.
Let S = \{c + g + 1 : \exists x such that (c < g x) occurs in p_1 \} \cup \{c - g - 1 : \exists x
such that (x < g c) occurs in p_1 \}. Note that \neg(c < g x) \equiv x <_0 (c + g + 1) and
\neg(x < g c) \equiv (c - g - 1) <_0 x. Hence it is easy to see that the complement relation
of p_1 can be written such that it contains only atomic constraints =, \neq, <, < s
where s \in S. Let m be the size of S. Obviously, m is at most linear in the size of n_1
because there are only a constant number of atomic constraints in each normalized
constraint tuple.
Within normal form tuples between each pair of vertices there is one of the
following: an equality constraint, an inequality constraint, a less-than constraint
with zero gap-value, a greater-than constraint with zero gap-value, or no constraint.
Also, each vertex has no lower (upper) bound or one of the elements of S as a lower
(upper) bound. Hence there are at most 5^{k(k-1)} \times (m + 1)^k \times (m + 1)^k = O(n_1^2)
number of possible tuples in normal form in the complement of p_i. Hence if we
modify the naive evaluation suggested above by putting the temp relation into
normal form after each join and eliminating duplicate tuples, we obtain an algorithm
that runs in polynomial time in n_1 as required.

Now let’s consider constraint relations over C_{\leq, \leq, \neq, \leq}, for which we already
described a normal form.

Theorem 6.3. Let p_1(X_1, \ldots, X_k) and p_2(Y_1, \ldots, Y_k) be any two fixed relation
schemes where some of the Xs may equal some of the Ys. Then for any constraint
relation instances of p_1 and p_2 in normal form with n_1 and n_2 tuples respectively,
the projection, selection, and rename operators on p_1 can be done in time polynomial
in n_1 and the complement of p_i can be done in DEXPTIME in n_1. For any positive integer
n_1 there is an input relation p_i of size n_1 such that the complement of p_i
requires 2^{n_1} number of tuples. The join of p_1 and p_2 can be done in time polynomial
in n_1 + n_2.

Proof. The proof is similar to Theorem 6.2 for the operators of project, select,
rename and join. For the complement operator a modification similar to that in the
proof of Theorem 6.2 yields DEXPTIME complexity because the possible number of
constraint tuples in normal form is exponential in n_1 (Counting the set of possible
constraint tuples can be done similarly as in Lemma 6.2.)

For the lower bound of the complement operator consider the constraint relation:
\[ r(X, Y) = (c_1 \notin X \land c_1 \notin Y) \lor \ldots \lor (c_{n_1} \notin X \land c_{n_1} \notin Y). \]
where \( c_i \) for \( 1 \leq i \leq n_1 \) are distinct constants. Let \( S = \{ c_1, \ldots, c_{n_1} \} \). The complement of \( r \) is:

\[
\text{co}_r(X,Y) = \bigvee_{(S_1 \cup S_2 = S) \land (S_1 \cap S_2 = \emptyset)} S_1 \subseteq X \land S_2 \subseteq Y.
\]

This representation cannot be reduced to fewer tuples because no tuple entails another. Clearly, the size of \( r \) is \( n_1 \), while the size of \( \text{co}_r \) is \( 2^{n_1} \) number of constraint tuples. \( \square \)

We will now consider the computational complexity of yes/no relational calculus programs. These programs have zero arity output relations which either contain the empty tuple (true) or no tuples (false). Consideration of yes/no relational calculus programs is convenient for analyzing data complexity, because many complexity classes are defined based on yes/no decision problems.

**Theorem 6.4.** For each fixed program \( \Pi \) in safe \( RC^{<z} \) deciding whether \( \Pi(d) \) is yes for variable database \( d \) is in PTIME. For each fixed program \( \Pi \) in safe \( RC^{\leq p(z^*)} \) deciding whether \( \Pi(d) \) is yes for variable database \( d \) is \( \Sigma^p_k \)-hard for some constant \( k \) and is in DEXPTIME.

**Proof.** For any fixed safe \( RC^{<z} \) or safe \( RC^{\leq p(z^*)} \) program we always need to perform a fixed number of constraint select, project, rename, join, or type-restricted complement operations. Since by Theorems 6.2 and 6.3 the computational complexity of these operators is in PTIME (respectively DEXPTIME) in the size of the constraint relations that are the arguments to these operators, any fixed safe \( RC^{<z} \) (respectively safe \( RC^{\leq p(z^*)} \)) program can be evaluated in PTIME (respectively DEXPTIME) in the size of the input relations.

For the lower bound we will show that there is a fixed safe \( RC^{\leq p(z^*)} \) program with variable input database that expresses the class of Quantified Boolean Formulas of the form \( \exists x_1 \forall x_2 \exists x_3 \ldots \forall x_k \phi \) where without loss of generality \( \phi \) is a boolean formula in disjunctive normal form. This subclass of quantified boolean formulas is referred to as \( \Sigma^p_k \) and forms a complexity class. Therefore, we will show that there is a safe \( RC^{\leq p(z^*)} \) program with \( \Sigma^p_k \)-hard data complexity for each \( k \). Since \( \text{PSPACE} = \bigcup_{k} \Sigma^p_k \), it follows that the class of safe \( RC^{\leq p(z^*)} \) programs has \( \text{PSPACE} \)-hard data complexity.

In our translation set \( S_i \) will represent the variables \( \bar{x}_i \). A variable in \( \bar{x}_i \) being true or false will correspond to belonging or not belonging to set \( S_i \). The safe \( RC^{\leq p(z^*)} \) program expressing the quantified boolean formula problem above will be the following:

\[
\exists S_1 \forall S_2 \exists S_3 \ldots \forall S_k p(S_1, \ldots, S_k)
\]

where \( p(S_1, \ldots, S_k) \) is the translation of the formula \( \phi \). The above \( RC^{\leq p(z^*)} \) program has a fixed size even though the exact number of variables bounded by the quantifiers as well as \( \phi \) may vary. Any change only effects the input database relation \( p \). For example the class of Quantified Boolean formulas with \( k = 2 \) can be expressed by the following \( RC^{\leq p(z^*)} \) query where only \( p \) varies:

\[
\exists S_1 \forall S_2 p(S_1, S_2)
\]
We express any instance of $\Sigma^p_2$ by using an input relation $p$ such that in any satisfying assignment for $\phi$ the variable $x_i$ is true if and only if there is a satisfying assignment for $p$ such that $i \in S_i$.

For example, let $\exists x_1, x_2, x_3 \forall x_4, x_5 \phi$ be the quantified boolean formula where $\phi$ is $(\neg x_1 \land x_2 \land \neg x_4) \lor (\neg x_2 \land \neg x_3 \land x_5) \lor (x_1 \land x_3 \land \neg x_5)$. Then we use the above $RC^{\exists \forall \exists \forall}$ program where the input relation $p$ is $(1 \notin S_1, 2 \in S_1, 4 \notin S_2) \lor (2 \notin S_1, 3 \notin S_1, 5 \in S_2) \lor (1 \in S_1, 3 \in S_1, 5 \notin S_2)$. Clearly, the translation is a safe $RC^{\exists \forall \exists \forall}$ query because only $c \in$ and $c \notin$ constraints are used. □

6.3 The Complexity of Safe Stratified Datalog$^{\neg \exists \leq z}$ Queries

Although safe stratified Datalog$^{\neg \exists \leq z}$ queries can be evaluated in finite time, in this section we show that their evaluation may require a large data complexity. We start with a definition of families of functions $F_i$ of type $N \to N$. Let $F_0$ be the set of polynomial functions, and let $F_i = \{2^f : f \in F_{i-1}\}$ for $i > 0$. If $F$ is a family of functions, let $F$-TIME denote the class of languages that can be accepted within some time $f \in F$. Now we will show using a Turing machine reduction that evaluation of safe stratified Datalog$^{\neg \exists \leq z}$ queries is $F$-TIME-hard.

Let $d$ be a database instance in normal form and let $|d|$ denote its size in number of tuples. Then the size of $d$ in number of bits representation on a tape is $O(|d|)$. Let $D$ denote the set of possible database instances. We define a function $f$ of type $N \times D \to N$ as follows. Let $f(0, d) = |d|$ and $f(i, d) = 2^f(i-1, d)$. (Here $f(i, d) \in F_i - \text{TIME}$.)

We start with a lemma that shows that the successor function on integers from 0 to $f(i, d)$ can be defined using a safe stratified Datalog$^{\neg \exists \leq z}$ program with $i$ strata.

**Lemma 6.1.** There is a safe stratified Datalog$^{\neg \exists \leq z}$ program with a single negation that given as inputs a relation containing the number $s$ and a relation that enables counting from 0 to $s$, defines both (1) a relation containing the number $2^s$ and (2) a relation that enables counting from 0 to $2^s$.

**Proof.** Let us assume that the input relations are $\text{no digits}(s)$ and $\text{next}(0, 1), \ldots, \text{next}(s-1, s)$. Using a safe stratified Datalog$^{\neg \exists \leq z}$ program we define two output relations, (1) a relation $\text{two_to_s}(2^s)$ and (2) the successor relation $\text{succeed}(0, 1), \ldots, \text{succeed}(2^s-1, 2^s)$.

**To show (1):** We write a rule for exponentiation as follows.

\[
\text{exp}(j, x_1, x_2) \leftarrow \text{next}(i, j), \text{exp}(i, x_1, x_3), \text{exp}(i, x_3, x_2).
\]

\[
\text{exp}(1, x_1, x_2) \leftarrow x_1 < x_2.
\]

This will define the constraint tuples $\text{exp}(i, x_1, x_2) : x_1 < 2^{i-1} x_2$ for each $1 \leq i \leq s$. In particular, $\text{exp}(s, x_1, x_2) : x_1 < 2^{s-1} x_2$ is one of the constraint tuples defined. Therefore, we can find the value $2^s$ as follows.

\[
\text{two_to_s}(x) \leftarrow \text{geq firepower}_s(x), \neg \text{gtfirepower}_s(x).
\]

\[
\text{gtfirepower}_s(x) \leftarrow \text{geq firepower}_s(y), y < x.
\]

\[
\text{geq firepower}_s(x) \leftarrow \text{no digits}(s), \text{exp}(s, 0, x).
\]

Notice that we use only one stratified negation in these rules. This is the only place where we need negation.

**To show (2):** In this step it helps to think of each number being written in binary notation. Since the number $2^s$ has $s$ binary digits, what we really need is given a counter on the digits and the value $2^s$ define a counter from 0 to $2^s$. 

We start by representing the value of each digit using a constraint interval, where
the gap-value is one less than the actual value. That is, for each $1 \leq i \leq s$, we
want to represent the value of the $i$th digit from the right as: $digit(i, x_1, x_2) := x_1 < 2^{\ell(i-1)} x_2$. The following rules define the desired constraint tuples.

\[
\begin{align*}
\text{digit}(j, x_1, x_2) &:= \text{next}(i, j), \text{digit}(i, x_1, x_3), \text{digit}(i, x_3, x_2). \\
\text{digit}(1, x_1, x_2) &:= x_1 < x_2. \\
\end{align*}
\]

Note that we can represent each number $i$ by a pair of constraints: $-1 < i x$
(which is equivalent to $(i - 1) < x$) and $x < 2^{s-(i+1)} 2^s$ (which is equivalent to
$x < i + 1$). Since each number can be expressed as the sum of a subset of the values
of the $n$ digits, if we start out from the constraint $-1 < x$ and $x < 2^s$ and choose
to increment for each $1 \leq i \leq s$ either the first or the second gap-value by the value
of the $i$th digit, then we will get a single integer between $0$ and $2^s - 1$ as output.
This gives an idea about how to define any number that we need. For example, the
following rules define the number $2^s - 1$.

\[
\begin{align*}
two_to_s\text{minus_one}(x) &:= \no_digits(s), \add_digit(x, x, s), \\
\add_digit(x_1, x_2, j) &:= \text{next}(i, j), \add_digit(x_3, x_2, i), \text{digit}(j, x_3, x_1). \\
\add_digit(x_1, x_2, 0) &:= -1 < x_1, x_2 < n, \two_to_s(n). \\
\end{align*}
\]

The above rules recursively define $x_1$ to be bounded by higher and higher con-
stants from below while $x_2$ is always bounded by $2^s$ from above. That is, for each
$0 \leq j \leq s$ the value of $\add_digit(x, j)$ will be equivalent to $-1 < 2^{j-1} x_1, x_2 < 2^s$.
Hence in the top rule when $j = s$ we have $x = 2^s - 1$. We used a separate $x_1$ and
$x_2$ in all the rules except the top rule to make it easy to tighten the lower bound
constraint while preserving the upper bound constraint.

In the rules we always added the value of a binary digit to the lower bound. Note
that in general we can define any integer between $0$ and $2^s - 1$ if for each binary
digit value we add it to the lower bound if the corresponding binary digit is $1$ in the
number or subtract it from the upper bound as if the corresponding binary digit is
$0$ in the number we want to define.

To express the successor function, we define pairs of integers. Let $x_1$ and $x_2$
represent the first and $y_1$ and $y_2$ represent the second integer. The following rules
make sure that when we add a digit to the $x$s we also add the same digit to the $y$s
the right way.

\[
\begin{align*}
succ(x, y) &:= two_to_s\text{minus_one}(x), two_to_s(y). \\
succ(x, y) &:= succ2(x, x, y, y, s), no_digits(s). \\
succ2(x_3, x_2, y_3, y_2, j) &:= succ2(x_1, x_2, y_1, y_2, i), \text{next}(i, j), \text{digit}(j, x_1, x_3), \\
&\text{digit}(j, y_1, y_3). \\
succ2(x_1, x_3, y_1, y_3, j) &:= succ2(x_1, x_2, y_1, y_2, i), \text{next}(i, j), \text{digit}(j, x_3, x_2), \\
&\text{digit}(j, y_1, y_2). \\
succ2(x_1, x_3, y_3, y_2, j) &:= succ3(x_1, x_2, y_1, y_2, i), \text{next}(i, j), \text{digit}(j, x_3, x_2), \\
&\text{digit}(j, y_1, y_3). \\
succ3(x_3, x_2, y_1, y_3, j) &:= succ3(x_1, x_2, y_1, y_2, i), \text{next}(i, j), \text{digit}(j, x_1, x_3), \\
&\text{digit}(j, y_1, y_2). \\
succ3(x_1, x_2, y_1, y_2, 0) &:= -1 < x_1, x_2 < n, -1 < y_1, y_2 < n, two_to_s(n). \\
\end{align*}
\]

In this program, in each recursive step, $x_1$ will be bounded by higher and higher con-
hstants from below and $x_2$ will be bounded by lower and lower constants from
above. In the second rule the possible values of \( x_1 \) and \( x_2 \) will overlap exactly on one integer. A similar note applies to \( y_1 \) and \( y_2 \).

As an example, let \( s = 3 \). Then we can prove that \( \text{succ}(4, 5) \) is true. It helps to think that the numbers 4 and 5 are written in binary notation as 100 and 101. The sequence of derived facts leading to the conclusion is the following:

\[
\text{succ3}(x_1, x_2, y_1, y_2, 0) \iff \neg x_1, x_2 < 8, \neg y_1, y_2 < 8 \quad \text{by the last rule.}
\]

\[
\text{succ2}(x_1, x_2, y_1, y_2, 1) \iff \neg x_1, x_2 < 8, \neg (y_1, y_2 < 8) \quad \text{by the fifth rule.}
\]

\[
\text{succ2}(x_1, x_2, y_1, y_2, 2) \iff \neg x_1, x_2 < 3, \neg (y_1, y_2 < 8) \quad \text{by the fourth rule.}
\]

\[
\text{succ2}(x_1, x_2, y_1, y_2, 3) \iff \neg x_1, x_2 < 3, \neg (y_1, y_2 < 8) \quad \text{by the third rule.}
\]

Note that the right hand side of the above is equivalent to \( 4 \leq x_1, x_2 \leq 4 \) and \( 5 \leq y_1, y_2 \leq 5 \). Hence we get \( \text{succ}(4, 5) \) by the second rule. \( \square \)

We call Datalog programs with a selected IDB relation with zero arity a yes/no program. This is because this output relation either contains the empty tuple (true) or no tuples (false). Consideration of yes/no Datalog programs is convenient for analysing data complexity, because many complexity classes are defined based on yes/no decision problems.

**Theorem 6.5.** There is a fixed yes/no program \( \Pi \) in safe stratified Datalog\(^{<\leq z}\) with \( i \) negations such that deciding whether \( \Pi(d) \) is yes for variable database \( d \) is deterministic \( \mathcal{F}_i - \text{TIME} \)-hard.

**Proof.** To prove the theorem we show that we can simulate an \( f(i,d) \)-time bounded deterministic Turing machine using a safe stratified Datalog\(^{<\leq z}\) program with \( i \) negations.

Lemma 6.1 implies that we can find the value of \( f(i,d) \) using a safe stratified Datalog\(^{<\leq z}\) program \( P \). All we have to do is to use \( i \) copies of the program fragment within Lemma 6.1 and rename them such that the output of one copy will be the input to the next copy. We can copy the value \( f(i,d) \) into the \text{time bound} relation:

\[
\text{time bound}(t) \iff \text{two to } \ldots \text{two to } (t).
\]

By Lemma 6.1 the program \( P \) also defines the successor relation on integers from 0 to \( f(i,d) \). Now assume that we want to simulate a deterministic \( f(i,d) \)-time bounded Turing machine running on a tape input of size \( n \), where \( n \) is any integer less than \( f(i,d) \). We record the value of \( n \) into the \text{tape size} relation:

\[
\text{tape size}(n).
\]

Let the deterministic \( f(i,d) \)-time bounded Turing machine be \( T = (K, \sigma, \delta, s_0, h) \), where \( K \) is the set of states of the machine, \( \sigma \) is the alphabet, \( \delta \) is the transition function, \( s_0 \) is the initial state, and \( h \) is the halting state.

First we use a relation \( T \) to describe the initial content of the tape. We create \( n \) facts \( T(i,c_i), \) one for each \( 1 \leq i \leq n \). If \( i > n \), then the content of the \( i \)th tape cell will be a special tape symbol \# denoting that it is blank. (Here \# can be any integer not already denoting a tape symbol.) We express this by:

\[
T(m, \#) \iff \text{tape size}(n), n < m.
\]

Second we use relations \text{Left}, \text{Right} and \text{Write} to describe the transition function \( \delta \) of \( T \). We create for each possible machine input state \( s_1 \), output state \( s_2 \), tape symbols \( c \) and \( \omega \), a fact \( \text{Left}(s_1, c, s_2) \), \( \text{Right}(s_1, c, s_2) \) or \( \text{Write}(s_1, c, s_2, \omega) \) if according to \( \delta \) when the machine is in state \( s_1 \) and pointing to \( c \), then the machine
must go to state $s_2$ and move one tape cell to the left, or to the right, or stay and write $w$ on the tape, respectively.

Third we use a relation $C$ to describe the configuration of the machine. The relation $C(t,i,s)$ describes that at time step $t$ the machine is pointing to tape position $i$ and is in state $s$. We can assume that the Turing machine is pointing at time zero to the first tape cell. Therefore we create a fact $C(0,1,s_0)$.

Fourth we express the sequence of transitions of the machine by a relation $R(t,j,c)$ which is true if and only if at time $t$ the $j^{th}$ tape cell contains the tape symbol $c$. To initialize $R$ we write the rule: $R(0,j,c) \leftarrow T(j,c)$.

We express the requirements for a valid deterministic computation of the machine as follows.

\[
\begin{align*}
C(t_2,o,s_2) & \leftarrow succ(t_2,t_2), C(t,i,s_1), R(t,i,c), Left(s_1,c,s_2), succ(o,i). \\
C(t_2,o,s_2) & \leftarrow succ(t_2,t_2), C(t,i,s_1), R(t,i,c), Right(s_1,c,s_2), succ(i,o). \\
C(t_2,i,s_2) & \leftarrow succ(t_2,t_2), C(t,i,s_1), R(t,i,c), Write(s_1,c,s_2,w). \\
R(t_2,i,c) & \leftarrow succ(t_2,t_2), C(t,i,s_1), R(t,i,c), Left(s_1,c,s_2). \\
R(t_2,i,c) & \leftarrow succ(t_2,t_2), C(t,i,s_1), R(t,i,c), Right(s_1,c,s_2). \\
R(t_2,p,c) & \leftarrow succ(t_2,t_2), C(t,i,s_1), R(t,p,c), i < p. \\
R(t_2,p,c) & \leftarrow succ(t_2,t_2), C(t,i,s_1), R(t,p,c), i > p. \\
\text{yes} & \Leftarrow C(t,i,h), time\_bound(t_2), t < t_2.
\end{align*}
\]

The last rule expresses that by time $f(i,d)$ the machine is in the halting state $h$. \qed

It is easy to see that Theorem 6.5 is true even if the size of each integer constant occurring in the input database $d$ is logarithmic in the size of $d$. [Revesz 1993] proved that yes/no Datalog$^{\leq z}$ programs can be decided in PTIME data complexity if we restrict and in DEXPTIME data complexity if we do not restrict the size of the integer constants in $d$. For the lower bound in the latter case we have the following.

**THEOREM 6.6.** There is a fixed yes/no program $\Pi$ in Datalog$^{\leq z}$ such that deciding whether $\Pi(d)$ is yes for variable database $d$ is DEXPTIME-hard.

**Proof.** If we do not restrict the size of the integer constants in the input database, then $d$ may contain the relation $\text{two\_to\_s}(2^s)$, as well as $\text{no\_digits}(s)$ and the next relation from $\text{next}(0,1)$ to $\text{next}(s-1,s)$. Then we can use part (2) of Lemma 6.1 which does not need any negation and can skip part (1) that is only used to define the relation $\text{two\_to\_s}(2^s)$ which we are given in $d$. That means that we can define the successor function from 0 to $2^s$. Then we can use the rules in the proof of Theorem 6.5 which also do not contain any negation. This shows that we can simulate any DEXPTIME bounded Turing machine with a fixed Datalog$^{\leq z}$ program and variable database $d$. \qed

6.4 The Complexity of Safe Stratified Datalog$^{\leq \ell \in \mathbb{N}, z^*}$ Queries

First we consider the upper bound of the problem of tuple recognition.

**Lemma 6.2.** For any fixed semipositive Datalog$^{\leq \ell \in \mathbb{N}, z^*}$ program $\Pi$ with output relation $r$, variable input database $d$, and set constant tuple $(C_1, \ldots, C_k)$, we can test whether $r(C_1, \ldots, C_k) \in \text{points}(\Pi(d))$ in DEXPTIME in the size of $d$. 

PROOF. Let $A = \{c_1, \ldots, c_n\}$ be the set of integer constants that occur in the program or in the input database. Let $p(X_1, \ldots, X_k)$ be any $k$-ary relation in normal form. In each constraint tuple of $p$ there are $(2^{2^n})^k = 2^{ak^n}$ possible different lower bounds and $2^{k(n+1)}$ possible different upper bounds for each of the $k$ argument variables. (Any subset $B$ of $A^n$ can be a lower bound or an upper bound and $Z^n \setminus B$ can be also an upper bound.) Further, there are $k(k - 1)$ ordered pairs with two distinct argument variables. Between any ordered pair we either have or not have a $\subseteq$ constraint. Hence there are $2^{ak^n} \times 2^{k(n+1)} \times 2^{k(k-1)}$ different normal form tuples for $p$. Since we are interested in data complexity and take the program to be fixed, $k$ is also a fixed constant. Hence there are $O(2^{ak^n})$ different normal form tuples in $p$.

Let $k_{\text{max}}$ be the maximum arity of a relation in $\Pi$. From the above follows that each relation in $\Pi$ will have at most $O(2^{2^{ak_{\text{max}}}n})$ normal form tuples. Further, the complement of any relation will also have at most that many normal form tuples because the type-restricted complement operator $\Gamma_2$ does not introduce any new constants.

Since there is a fixed constant number of IDB relations, $O(2^{2^{ak_{\text{max}}}n})$ is a bound on the number of iterations required in evaluating the model of $\Pi$. Since the program has a fixed size, each iteration will take $O(2^{2^{ak_{\text{max}}}n})$ time where $c$ is the number of relation symbols in the body of any rule (implying a cartesian product operation in the worst case). Clearly $c$ is a fixed constant for each program. □

**Theorem 6.7.** For any fixed stratified $Datalog^{\leq n \cdot 2^n}$ program $\Pi$ with output relation $r$, variable input database $d$, and set constant tuple $(C_1, \ldots, C_k)$, we can test whether $r(C_1, \ldots, C_k) \in \text{points}(\Pi(d))$ in DEXPTIME in the size of $d$.

**Proof.** It follows from Lemma 6.2 that evaluating any stratum of a stratified $Datalog^{\leq n \cdot 2^n}$ program takes DEXPTIME in the size of $A$. Going from one stratum to the next stratum cannot increase $A$. Hence each stratum will be evaluated within DEXPTIME in the size of the original input database $d$. □

Now let's consider the lower bound data complexity of the query evaluation. In this lower bound we will not even use any negation symbol.

**Theorem 6.8.** There is a fixed yes/no program $\Pi$ in $Datalog^{\leq n \cdot 2^n}$ such that deciding whether $\Pi(d)$ is yes for variable database $d$ is DEXPTIME-complete.

**Proof.** The upper bound follows by Theorem 6.7. The lower bound is by simulation of deterministic exponential time bounded Turing machines. At first we show that we can express the successor function for values between 0 and $2^n - 1$ using only $O(s)$ space. The idea is to encode the binary notation of each number as some subset of $\{s_1, s_0, \ldots, 21, 20, 11, 10\}$, where $s_1$ or $s_0$ will be present according to whether in the binary encoding the $i$th 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 $\text{digit}(N, I, D)$ which is true if and only if $N$ represents the integer $n$ as described above and in the binary notation of $n$ the $i$th digit from the right is $d$, and $I = \{i\}$ and $D = \{d\}$.
\[ \text{digit}(N, \{1\}, \{0\}) ::= 10 \in N, 11 \notin N. \]
\[ \text{digit}(N, \{1\}, \{1\}) ::= 11 \in N, 10 \notin N. \]
\[ \ldots \]
\[ \text{digit}(N, \{s\}, \{0\}) ::= s0 \in N, s1 \notin N. \]
\[ \text{digit}(N, \{s\}, \{1\}) ::= s1 \in N, s0 \notin N. \]

We also add to the input database the facts \( \text{nextt}(\{0\}, \{1\}), \ldots, \text{nextt}(\{s-1\}, \{s\}) \) and the fact \( \text{no_digits}(\{s\}) \) and \( \text{time_bound}(\{s1, \ldots, 11\}) \) that describe that we have \( s \) binary digits in each number and the largest number is \( 2^s - 1 \). Note that the size of the input database is \( O(s) \). Now we express the successor relation \( \text{succ}(N, M) \) which is true if and only if \( M, N \) represent the numbers \( m, n \) respectively and \( m = n + 1 \) for any \( 0 \leq n, m < 2^s \).

\[ \text{succ}(N, M) ::= \text{succ}(N, M, S), \text{no_digits}(S). \]

\[ \text{succ2}(N, M, J) ::= \text{succ2}(N, M, J, \text{next}(J, I), \text{digit}(N, I, D), \text{digit}(M, I, D)). \]
\[ \text{succ2}(N, M, \{1\}) ::= \text{digit}(N, \{1\}, \{0\}), \text{digit}(M, \{1\}, \{1\}). \]
\[ \text{succ2}(N, M, I) ::= \text{succ3}(N, M, J, \text{next}(J, I), \text{digit}(N, I, \{0\}), \text{digit}(M, I, \{1\}). \]
\[ \text{succ3}(N, M, J) ::= \text{succ3}(N, M, J, \text{next}(J, I), \text{digit}(N, I, \{1\}), \text{digit}(M, I, \{0\}). \]
\[ \text{succ3}(N, M, \{1\}) ::= \text{digit}(N, \{1\}, \{1\}), \text{digit}(M, \{1\}, \{0\}). \]

During the rest of the simulation the successor relation will be used for counting the current position on the tape and the running time similarly to Theorem 6.5. The only important change is to replace the integer variables by integer set variables, integer constants by integer set constants that contain a single element and instead of the \( < \) relation use the following:

\[ \text{greater}(I, J) ::= \text{succ}(I, K), \text{greater}(K, J). \]
\[ \text{greater}(I, J) ::= \text{succ}(I, J). \]

The \text{greater} relation can be used to initialize and update the first \( 2^s - 1 \) tape cells. That is enough for the simulation because the Turing machine never needs to move beyond the \( 2^s - 1 \)st tape cell due to the time limit. \( \square \)

7. CONCLUSIONS AND FUTURE WORK

The relative expressive power of various constraint query languages is an interesting issue. [Benedikt et al. 1996] proved recently that even simple recursive queries like transitive closure cannot be expressed in relational calculus with polynomial arithmetic constraints over the real numbers. What is the relative expressive power of the various safe stratified Datalog queries of constraint databases?

There are also many practical questions about the implementation of constraint query languages. The issues here include efficient indexing of constraint tuples, integrity constraints, built-in aggregate operators, user interfaces, concurrent access to data, security etc. Many of these problems have to be rethought in the context of constraint databases (see the surveys [Cohen 1990; Jaffar and Maher 1994; Kanellakis 1995; Kanellakis and Gaidin 1994]). The constraint database system DISCO [Byon and Revesz 1995] under development at the University of Nebraska implements the constraint query language \( \text{Datalog}^{\leq x, \lessgtr x, \leq x, \lessgtr x} \). We plan to implement safe stratified \( \text{Datalog}^{\leq x, \lessgtr x, \leq x, \lessgtr x} \) presented in this paper in a future version of that system.
REFERENCES


Safe Query Languages for Constraint Databases


