Constraint Query Languages

Paris C. Kanellakis[1]
Gabriel M. Kuper[2]
Peter Z. Revesz[3]

Technical Report No. CS-90-31

November 26, 1990

[1] Computer Science Dept., Brown University, Providence, RI 02912
[2] IBM T.J. Watson Research Center, Yorktown Heights, NY
[3] Computer Science Dept., Brown University, Providence, RI 02912

CONSTRAINT QUERY LANGUAGES*

(November 26, 1990)

Paris C. Kanellakis† Gabriel M. Kuper‡ Peter Z. Revesz§

Abstract

We investigate the relationship between constraint programming and database query languages. We show that bottom-up, efficient, declarative database programming can be combined with efficient constraint solving. The key intuition is that the generalization of a ground fact, or tuple, is a conjunction of constraints. We propose a set of basic Constraint Query Language design principles and show how they can be realized with four different classes of constraints: polynomial, rational order, equality, and boolean constraints.

Keywords: database queries, data complexity, quantifier elimination, constraint logic programming, relational calculus, Datalog.

1 Introduction

Q: What's in a tuple?
A: Constraints.

Constraint programming paradigms are inherently "declarative", since they describe computations by specifying how these computations are constrained [5, 39] (for a survey see [29]). A major recent development in logic programming systems is the integration of logic and constraint paradigms, e.g., in CLP [20], in Prolog III [12], and in CHIP [13]. One intuitive reason for this successful integration is as follows. A strength of Prolog is its top-down, depth-first search strategy. The operation of first-order unification at the forefront of this search is a special form of efficient constraint solving. Additional constraint solving increases the depth of the search and, thus, the effectiveness of the approach.

The "declarative" character of database query languages is an important aspect of database systems. Indeed, having such a language for ad-hoc database querying is a requirement today.

*A preliminary version of the results in this paper appeared in [23].

†Brown University, Providence, RI. The work of this author was supported by IBM, by an Alfred P. Sloan Fellowship, and by ONR grant N00014-83-K-0146 ARPA Order No. 4786.

‡IBM T.J. Watson Research Center, Yorktown Heights, NY.

§Brown University, Providence, RI. The work of this author was supported by NSF grant IRI-8617344 and by NSF-INRIA grant INT-8817874.

It is rather surprising that constraint programming has not really influenced database query language design. There has been some research on the power of constraints for the implicit specification of temporal data [10], for extending relational algebra [17], and for magic set evaluation [35], but no overall design principles. The bottom-up and set-at-a-time style of evaluation emphasized in databases, and more recently in knowledge bases, seems to contradict the top-down, depth-first intuition behind Constraint Logic Programming.

The main contribution of this paper is to show that it is possible to bridge the gap between: *bottom-up, efficient, declarative database programming and efficient constraint solving.* A key intuition comes from Constraint Logic Programming: *a conjunction of constraints is the correct generalization of the ground fact.* The technical tools for this integration are: *data complexity* [7, 45] from database theory, and *quantifier elimination* methods from mathematical logic.

First, let us provide some motivation for the integration of database and constraint solving methods. We refer to Section 2.1 for some concrete examples from computational geometry (see also Examples 2.5 and 5.2 for other application areas). Manipulation of geometric data is an important application area (e.g., geographic databases) that requires both relational query language techniques and arithmetic calculations. Indexes for range searching and modeling of complex structures have been used to bridge the gap between declarative accessing of large volumes of spatial data and performing common computational geometry tasks. However, even with these data model extensions arithmetic calculations have not been given first-class citizen status.

What could be sound criteria for achieving integration of data processing and other computations, such as arithmetic calculations? Here are some examples:

(1) Preserving the declarative language style is desirable.
(2) Additional expressive power is desirable, but must come without a serious loss of efficiency.
(3) Bottom-up processing is desirable, since it is a good candidate for many optimizations.

These criteria are satisfied by the *Constraint Query Language* (CQL) design principles outlined below and illustrated in Figure 1.

- *A generalized k-tuple is a conjunction of constraints on k variables.* In the relational database model $R(3,4)$ is a tuple of arity 2. It can be thought of as a single 2-dimensional point and also as $R(x,y)$ with $x = 3$ and $y = 4$. In our framework, $R(x,y)$ with $x = y$ and $x < 2$ is a generalized tuple of arity 2, and so is $R(x,y)$ with $x + y = 2.5$. Hence, a generalized tuple of arity k is a finite representation of a possibly infinite set of tuples of arity k (or equivalently of k-dimensional points).
- *A generalized relation of arity k is a finite set of generalized k-tuples.* It is a disjunction of conjunctions (DNF) of constraints. Each conjunction uses at most k variables, where k is the arity of the relation. In summary:

  > A generalized database is a finite set of generalized relations. Each generalized relation of arity k is a quantifier-free DNF formula of the logical theory of constraints used, that contains at most k distinct variables and that describes a possibly infinite set of k-dimensional points.

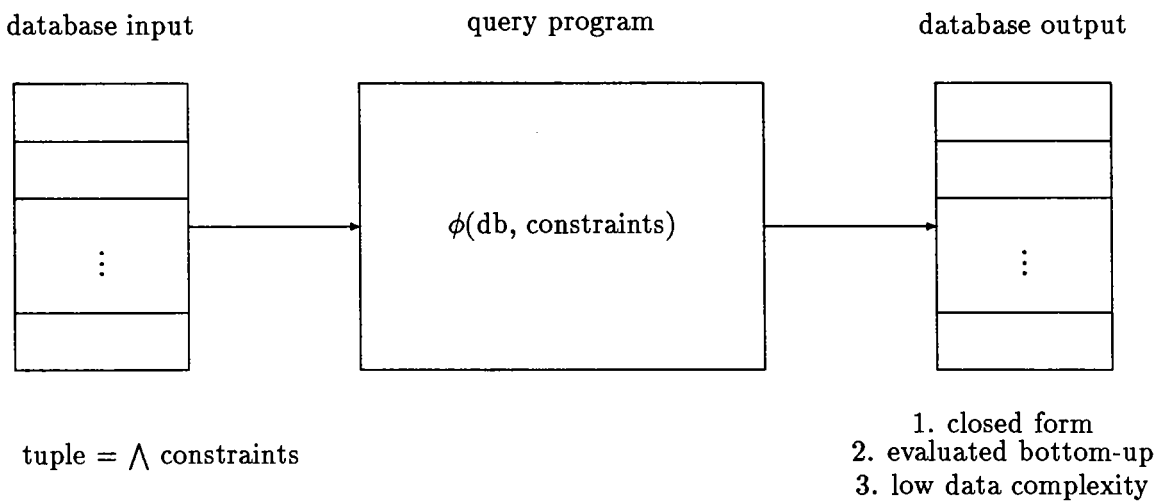

Figure 1: A database framework for CQL

- *The syntax of a CQL is the union of an existing database query language and a decidable logical theory.* For example: Relational calculus [11] + the theory of real closed fields [40] (Section 2); Inflationary Datalog$^\neg$ [1, 26] + the theory of rational order with constants (Section 3); Inflationary Datalog$^\neg$ + the theory of equality of an infinite set with constants (Section 4); and Datalog + boolean equations (Section 5). In each of these cases, we combine in the obvious way the syntax of the database language and the logical theory.

- *The semantics of a CQL is based on that of the decidable logical theory, by interpreting database atoms as shorthands for formulas of the theory.* Let ϕ be a query program using predicate symbols $R_1, \ldots, R_n$ for the inputs and let $r_1, \ldots, r_n$ be the corresponding input generalized relations. We interpret the program in the context of its input. Let $\phi[r_1/R_1, \ldots, r_n/R_n]$ be the formula of the theory that is obtained by replacing in ϕ each database atom $R_i(x_1, \ldots, x_k)$ by the DNF formula for input generalized relation r_i, with its variables appropriately renamed to $x_1, \ldots, x_k$. (Note that, without loss of generality, an occurrence of a database atom in ϕ is of the form $R_i(x_1, \ldots, x_k)$ $1 \leq i \leq n$, where R_i is a predicate symbol of arity k and $x_1, \ldots, x_k$ are distinct variables; this is because in our frameworks we can always use equality constraints among variables in ϕ.) In summary:

  > Query program ϕ applied to input database $r_1, \ldots, r_n$ is a formula of the logical theory of constraints used, i.e., $\phi[r_1/R_1, \ldots, r_n/R_n]$. The output is the possibly infinite set of multi-dimensional points, such that valuating the free variables of this formula to any one of these points makes the formula true.

- *The queries must be evaluable in closed form and bottom-up.* By closed form we mean that the output of any query program applied to any input generalized relations must be

a generalized relation. The analogue for the relational model is that relations are finite structures, and queries are supposed to preserve this finiteness. This is a requirement that creates various "safety" problems in relational databases. In our framework, it is finiteness of representation of constraints that must be preserved. In this framework, evaluation of a query corresponds to an instance of a decision problem and, interestingly, many quantifier elimination procedures realize the goal of closed form and bottom-up evaluation.

- *The queries must be evaluable efficiently.* Database atomic formulas indicate, in the declarative query language itself, the parts that can grow asymptotically versus the parts that are constant-size. By fixing the program size and letting the database grow, we can prove that the evaluation can be performed in PTIME or in NC or in LOGSPACE, depending on the constraints that we consider (for the various complexity classes see [15]).

1.1 Some Basic Definitions

We consider many kinds of constraints.

In Section 2, a *polynomial constraint* is of the form $p(x_1, \ldots, x_n)\theta 0$, where p is a polynomial in $x_1, \ldots, x_n$, and θ is either $=$ or $<$. Special cases of polynomial constraints are linear equations, quadratic equations, linear inequalities, etc. Variables range over the domain of real numbers. For these constraints we interpret $+, *, <$ and constants as the arithmetic operations, the order relation and the real numbers.

In Section 3, a *rational order constraint* is of the form $u = v, u \neq v, u \leq v$ or $u < v$, where u and v are variables or constants. A nontrivial property of these constraints is the density of rational order. Variables range over the domain of rational numbers. Constants and $<, \leq$ are interpreted as rational numbers and their ordering.

In Section 4, an *equality constraint* is of the form $u = v$ or $u \neq v$, where u and v are variables or constants. Here we have an infinite domain: the integers. Each constant denotes an integer, but we have no order, successor, test-for-zero, or arithmetic operations.

In Section 5, a *boolean constraint* is an equation of the form $s =_B t$, where s, t are terms constructed using variables, operators, and generators of a boolean algebra B [6, 27, 32]. (We focus on free boolean algebras with m generators, for which we provide definitions in Section 5.)

In our exposition, we assume some familiarity with the elements of database theory [22, 41].

In particular, we refer readers for detailed, but fairly standard, definitions: to [11, 41] for *relational calculus*, to [1, 22, 26, 41, 42] for *Datalog* and its extensions such as *Inflationary Datalog*$^\neg$, and to [2, 8, 25, 41] for *tableau* query programs. (Tableau are a special case of relational calculus and Datalog. We provide definitions and examples for them in Section 2.2). Let us now describe how to extend relational calculus, Datalog, and Inflationary Datalog$^\neg$ with constraints.

Relational calculus + constraints: The syntax is that of traditional relational calculus where the atomic formulas can also be constraints. Depending on the specific CQL studied these could be polynomial or rational order or equality or boolean constraints (as described above). An example query, in the case of rational order constraints, is the formula:

$$\phi(x_1, x_2) \equiv R_1(x_1, x_2) \vee \exists y R_1(x_1, y) \wedge R_1(y, x_2) \wedge x_1 \leq x_2 \wedge x_2 \leq y.$$

The semantics of relational calculus + constraints is defined using the semantics of relational calculus on *unrestricted (finite or infinite) relational databases.* That is, we use the semantics of relational calculus on relational structures with a finite or infinite set of tuples [31].

We pick our domain of interest, e.g., the reals or the rationals or the integers or a free boolean algebra. We interpret the function symbols (e.g., $+, *$) and the predicate symbols (e.g., $<, =$) in the constraints over our domain of interest. We use D for the domain and δ for the interpretations of the symbols in constraints.

Let ϕ be any relational calculus query with constraints that uses some number of predicate symbols $R_1, \ldots, R_n$ and some number of free variables $x_1, \ldots, x_k$ (in the example above $n = 1$ and $k = 2$). Let $r_1, \ldots, r_n$ be the generalized database relations that are given for the predicate symbols occurring in ϕ. Each one of these relations finitely represents an unrestricted (finite or infinite) relational database, which we call $\rho_1, \ldots, \rho_n$. Using the standard first order meaning of $\models$ we define:

$$\phi(r_1, \ldots, r_n) \equiv \{a_1, \ldots, a_k \in D^k \mid < D, \delta, \rho_1, \ldots, \rho_n > \models \phi(a_1, \ldots, a_k)\}$$

Remark: We have given the semantics of a relational calculus + constraints query on a generalized database as a mapping from unrestricted relational databases to unrestricted relational databases. It is easy to verify that this definition is equivalent to interpreting database atoms as shorthands for formulas of the theory of constraints, as we required in our CQL design principles. For the cases we investigate in this paper, we show that the output of a query on a generalized database is also representable as a generalized database.

Datalog + constraints: The syntax is that of traditional Datalog where the bodies of rules can also contain constraints. Depending on the specific CQL studied these could be rational order or equality or boolean constraints. We only consider polynomial constraints for nonrecursive Datalog or tableau in Section 2.2.

For example P, a *Datalog* program with rational order, is a finite set of rules of form:

$$t_0 :\!- t_1, t_2, \ldots, t_l.$$

The expression t_0 (the rule *head*) must be an atomic formula of the form $R(x_1, \ldots, x_n)$, and the expressions $t_1, \ldots, t_l$ (the rule *body*) must be atomic formulas of the form $x_i = x_j$, $x_i \neq x_j$, $x_i < x_j$, $x_i \leq x_j$, or $R(x_1, \ldots, x_n)$, where R is some predicate symbol.

The semantics of a Datalog program P on a generalized database $r_1, \ldots, r_n$, representing unrestricted relational database $\rho_1, \ldots, \rho_n$, is the least fixpoint of the monotone mapping defined

by a first-order formula ϕ_P (see example right below) and $\rho_1, \ldots, \rho_n$. This is as in the case without constraints, the only difference being the use of unrestricted relational databases [22, 31].

For example, consider the Datalog with rational order query P:

$$R(x,y) :\!-\; R(x,z), R_0(z,y), x \leq y, y \leq z$$

$$R(x,y) :\!-\; R_0(x,y)$$

Let this query be applied to a generalized database r_0 representing the unrestricted relation ρ_0. Then ϕ_P is the following first-order formula, where R_0 is interpreted as ρ_0:

$$\phi_P(x,y) \equiv \psi(x,y;R) \equiv R_0(x,y) \vee \exists z R(x,z) \wedge R_0(z,y) \wedge x \leq y \wedge y \leq z.$$

This formula defines a mapping from unrestricted relations ρ of arity 2 to unrestricted relations of arity 2. Here R is singled out because it is the predicate variable that is initialized to ρ and will receive the output of ϕ_P.

$$\rho \longrightarrow \{a, b \in D^2 | < D, \delta, \rho_0, \rho > \models \phi_P(a,b)\}$$

This mapping is monotone with respect to set inclusion for ρ. By the Tarski fixpoint theorem it has a least fixpoint, which is the output of the query program applied to input r_0.

Inflationary Datalog$^\neg$ + constraints: The syntax is that of Datalog + constraints with one addition. We allow in a rule body expressions of the form $\neg R(x_1, \ldots, x_n)$, where R is some predicate symbol. We give the language inflationary semantics. Namely, to obtain a monotone mapping for Datalog$^\neg$ programs we treat negation in an inflationary fashion [1, 16, 26]. In the inflationary semantics after each iteration the set of facts derived is added to the set of facts that were derived in the previous iterations.

Remark: We have given the semantics of a Datalog$^{(\neg)}$ + constraints query on a generalized database as the least fixpoint of a monotone (inflationary) mapping from unrestricted relational databases to unrestricted relational databases. In the syntax of these languages there are two kinds of predicate symbols: those corresponding to input relations called EDBs and those defined by rules called IDBs. It is easy to verify that our definition is equivalent to interpreting EDB atoms as shorthands for formulas of the theory of constraints, as we required in our CQL design principles. For the cases we investigate in this paper, we show that the IDB unrestricted relations produced by applying a query program on a generalized database are also representable as a generalized database.

Data-Complexity: We show that in a variety of cases, queries of relational calculus + constraints and Datalog$^{(\neg)}$ + constraints can be evaluated in *closed form*. That is, for inputs that are generalized databases the outputs can be represented as generalized databases.

We define *data-complexity* as the complexity of evaluating a representation for the output, of a *fixed* program and a *variable* input generalized database. Thus, when we refer to PTIME or NC or LOGSPACE data-complexity we mean computing closed forms for the output under these resource bounds. (This is slightly more general than the use of these complexity classes for yes/no decision problems).

1.2 Overview of Contributions

Let us now outline our results. From Codd's original work [11] it follows that: *safe relational calculus can be evaluated bottom-up in closed form and LOGSPACE data complexity* (where Codd defines safe formulas via syntactic restrictions on relational calculus). We provide as evidence of the soundness of our design principles: four variations of this theorem, each one with a different set of constraints, and three further specializations of it.

1. Relational calculus with polynomial constraints can be evaluated bottom-up in closed form and NC data complexity. This is a direct consequence of [4, 28, 40] and is contained in Section 2.1. In Sections 2.2 and 2.3, as part of our analysis of the relational calculus and polynomial constraints, we provide a geometric interpretation of the homomorphism theorem for tableau query containment [2, 8, 25] and we propose the use of high-dimension variables as a CQL primitive. We also show that deciding containment between tableau queries with linear equalities is NP-complete, but that with quadratic equalities it is Π_2^p–hard.

2. Relational calculus with rational order constraints can be evaluated bottom-up in closed form and LOGSPACE data complexity. This is shown by adapting the proof of [14] and is contained in Section 3.1. The same technique can be extended to show that:

3. Inflationary Datalog$^\neg$ with rational order constraints can be evaluated bottom-up in closed form and PTIME data complexity. In Section 3.2, for Datalog with rational order constraints, we develop a bottom-up evaluation method that is closer to the classical foundations of logic programming [30] and knowledge bases [41, 42]. This allows us to show that:

4. Piecewise linear Datalog with rational order constraints can be evaluated bottom-up in closed form and NC data complexity. See Section 3.3.

5. Relational calculus with equality constraints can be evaluated bottom-up in closed form and LOGSPACE data complexity. See Section 4. Together with the following result this work extends the approach to safe queries of [3, 18, 24, 35].

6. Inflationary Datalog$^\neg$ with equality constraints can be evaluated bottom-up in closed form and PTIME data complexity. See Section 4.

7. Datalog with rational order and boolean constraints can be evaluated bottom-up in closed form. The data complexity here is higher than in the previous cases and it depends on the use of free boolean algebras with m generators. In Section 5, we partly analyze this data complexity and show it to be Π_2^p–hard if m is unbounded; for m fixed we have PTIME data complexity.

2 Relational Calculus and Polynomial Constraints

In this section we give our first example of a CQL by combining relational calculus with the theory of real closed fields. Note that, further extending this CQL setting with recursion, via Datalog or a fixpoint operator, would give us Turing computability.

In Section 2.1, we argue (using recent developments from the theory of algorithms [4, 28]) that it is possible to bridge the gap between: "declarative and efficient" database programming without arithmetic calculations and most geometric data manipulations with arithmetic calculations. In this argument, the key notion from database theory is *data complexity* [7, 45].

In Section 2.2, we move from data complexity to optimization of tableau queries. These queries form the most common subset of relational calculus queries and are equivalent to nonrecursive Datalog queries. We augment them using special polynomial constraints such as linear equations, quadratic equations, and simple inequalities without arithmetic operations.

In Section 2.3, we introduce a "programming language construct" for relational calculus and polynomial constraints that is outside the traditional constraint logic programming and decision procedure frameworks.

2.1 Data Complexity and Computational Geometry

Consider a generalized database relation to be a propositional formula in DNF that is special in two ways: (1) the clauses are conjunctions of polynomial constraints, and (2) the number of variables in this formula is at most the arity of the relation.

For example, let r be a generalized relation with arity 2 and containing two generalized tuples $(y = 2x \wedge x \neq y)$ and $(x+y \geq 1)$. Thus, it is the DNF formula $(y = 2x \wedge x \neq y) \vee (x+y \geq 1)$. It describes an infinite set of two dimensional points, namely the half plane $x + y \geq 1$ and the line $y = 2x$ without the point $x = y = 0$. We can regard each DNF clause as a generalization of the notion of a tuple in a relational database, because the relational database tuple $(c_1, \ldots, c_k)$ can be expressed by the constraints $(x_1 = c_1) \wedge \cdots \wedge (x_k = c_k)$. We can then think of a generalized database relation as a finite set of these generalized tuples, in the same way that a database relation is a finite set of tuples. Note that both relations and generalized relations are sets of (multi-dimensional) points. However, relations are finite sets of points, whereas generalized relations are possibly infinite sets of points that have some finite representation using constraints.

Consider a query language consisting of all first-order formulas over the database predicates together with polynomial constraints. The syntax is the union of relational calculus with that of the theory of real closed fields [40]. For the semantics we use the theory of real closed fields. As described in Section 1, the database atomic formulas will be used as shorthands for large formulas of the theory of real closed fields.

For example, let $R(x, y)$ be an atomic formula appearing in a query ϕ. If we apply ϕ to the generalized relation $r \equiv (y = 2x \wedge x \neq y) \vee (x + y \geq 1)$, then $R(x, y)$ is a shorthand of this

formula.

The critical observation is that: *database atomic formulas express and highlight, in the declarative query language itself, the parts that can grow asymptotically versus the parts that are constant-size calculations.* That the database size n dominates the query size by many orders of magnitude, is the rationale of data complexity. In the following examples n is the only parameter that grows asymptotically and $R(\ldots)$ are the only "database accesses".

To illustrate this discussion, we show how three basic operations of computational geometry, i.e., line intersection, convex hull, and Voronoi diagram (see [34]) can be described in a "declarative and efficiently bottom-up evaluable" query language.

Example 2.1 *Line intersection*: The database relation r consists of n lines in the plane, described as pairs of points. In other words, a tuple in r is of the form (x_1, y_1, x_2, y_2), and represents the line through (x_1, y_1) and (x_2, y_2). We want to compute all the points on the intersections of pairs of these lines (l, l'), for l, l' distinct. To do this, we first define a predicate $On(x, y, x_1, y_1, x_2, y_2)$ that means that the point (x, y) is on the line through (x_1, y_1) and (x_2, y_2), i.e.,

$$(\exists t)(x = tx_1 + (1-t)x_2 \wedge y = ty_1 + (1-t)y_2)$$

Then the intersections can be computed by the formula ϕ with free variables x, y:

$$\begin{aligned}\phi(x,y) \equiv (\exists x_1, x_2, x_3, x_4, y_1, y_2, y_3, y_4)(&R(x_1, y_1, x_2, y_2) \wedge R(x_3, y_3, x_4, y_4) \\ &\wedge (x_1, y_1, x_2, y_2) \neq (x_3, y_3, x_4, y_4) \wedge On(x, y, x_1, y_1, x_2, y_2) \wedge On(x, y, x_3, y_3, x_4, y_4))\end{aligned}$$

where the inequality is a shorthand for the constraints that exclude two tuples representing one line and where On is as defined above. □

Example 2.2 *Convex hull*: The database consists of a 2-dimensional relation r, that describes n points of the plane. We want to select those points from r that form the convex hull. To do this, observe that a point (x, y) is *not* a convex hull point iff there are 3 other points in r such that (x, y) is inside the triangle that they generate. Using constraints, we can define a predicate $Intriangle(x, y, x_1, y_1, x_2, y_2, x_3, y_3)$ that holds when (x, y) is in the triangle generated by (x_1, y_1), (x_2, y_2) and (x_3, y_3). Point (x, y) in r will be in the convex hull iff there do not exist points in r such that $Intriangle(x, y, x_1, y_1, x_2, y_2, x_3, y_3)$.

The naive algorithm based on this observation, known as Floyd's method, runs in PTIME. Although it cannot compete with various known $n \log n$ algorithms, it is still useful in combination with other convex hull techniques. □

Example 2.3 *Voronoi diagram*: We show how to find the graph called the *dual* of the Voronoi diagram [34]. To do this, note that two points u and v are adjacent in the Voronoi dual iff all the points on the line from u to v are closer to u or to v than to any other point in the database. This condition can easily be expressed in our language. □

This list of examples could be extended to most of [34]. The few exceptions typically involve reachability conditions, such as computing Euclidean Spanning Trees.

Queries in the language of relational calculus and polynomial constraints can be evaluated bottom-up in closed form, i.e., the result of a query on a generalized relation is also a generalized relation. This closure property follows immediately from the decision procedure of Tarski for the theory of real closed fields [40]. One can think of Tarski's procedure as a generalized relational algebra, where all the operations are simple variants of the familiar database ones except for projection. Projection corresponds to quantifier elimination and is the nontrivial operation. Unfortunately, Tarski's decision procedure has extremely high complexity (even in our setting). In general, the decision problem for the theory of real closed fields requires nondeterministic exponential time.

Fortunately, our setting has much more structure than geometric theorem proving. If we focus our attention on data complexity, the problem is tractable. If we have a fixed query on a generalized database, we have a fixed bound on the number of variables and on the quantifier depth. We can then use the results of [4, 28] to show that the data complexity is in NC. Furthermore, the [28] algorithm can be used to output a formula in DNF (of size polynomial in the input) that represents the output generalized database.

It is obvious that the general-purpose bottom-up evaluation based on geometric theorem proving is not as efficient as the various specialized computational geometry algorithms. But it can be thought of as a statement that the potential for optimization is present. The following theorem easily follows from [4, 28].

Theorem 2.4 Relational calculus with polynomial constraints can be evaluated bottom-up in closed form and NC data complexity. □

2.2 Tableau Query Equivalence Revisited

Data complexity is based on the assumption that there are sufficient resources for unlimited processing of a query program. This is only a theoretical approximation, and many sophisticated techniques have been developed for query optimization. A key problem for optimization is testing containment and equivalence of queries.

Each query ϕ computes for any input generalized database d an output generalized relation $\phi(d)$. Recall that generalized relations are possibly infinite sets of (multi-dimensional) points. We say that a query ϕ_1 is contained in query ϕ_2, denoted $\phi_1 \subseteq \phi_2$, iff for each input generalized database d, all the points in $\phi_1(d)$ are also in $\phi_2(d)$.

We now examine *conjunctive queries*, or equivalently *nonrecursive Datalog queries*. We augment these queries using special polynomial constraints such as linear equations, quadratic equations, and inequalities without $+, *$. For instance:

Example 2.5 Using nonrecursive Datalog notation and a single linear equation constraint we express the following "balanced checkbook" query.

$$Balanced(x) :\!\!- Expenses(x, f, r, m), Savings(x, s), Income(x, w, i), f + r + m + s = w + i$$

This is a conjunctive query with *Expenses, Savings* and *Income* input relations, *Balanced* output relation, and a single linear equation constraint: x is name, f is amount spent for food, r for rent, m for miscellaneous, s for transfer to savings, w for wages, and i for interest. The intended output of this query is the list of persons whose checkbooks balance. □

An alternative name for conjunctive queries is *tagged untyped tableau* queries, because they can be represented in tabular form, with the relation names as row-tags, and with the same (untyped) variable appearing in many columns. For example, the checkbook query can be represented by a four row tableau with balanced, expenses, savings, and income row-tags. The linear equation constraint would be extra. The tableau would have four columns, and each relation would be extended to arity four by adding dummy arguments denoted by new distinct variables. The first row corresponding to the lefthand side of the query, in this case the balance relation, is called the *summary row.* For the detailed terminology see [2].

We present an extension of the homomorphism technique of [2, 8], which is widely used in the optimization of tagged untyped tableau queries, to such queries with linear equations.

Theorem 2.6 Containment of queries that are expressed by a tagged untyped tableau with a conjunction of linear equation constraints is NP-complete.

Proof: NP-hardness is immediate, since it is NP-complete to determine containment for such queries without linear equations. The new part is showing membership in NP. We show that for two queries ϕ_1 and ϕ_2 in normal form: ϕ_1 is contained in ϕ_2 iff there is a homomorphism mapping ϕ_2 into ϕ_1.

Let us explain *normal forms*, *symbol mappings*, and *homomorphisms.* We break up each query ϕ into a tagged untyped tableau part T, that consists exclusively of distinct occurrences of variables, and a conjunction of constraints C. This normal form is without loss of generality, since the constraints in C can force any equalities of the distinct symbols in T. A function h is a symbol mapping from the symbols of T_2 to those of T_1 iff it maps the summary row of T_2 into the summary row of T_1, every constant to itself, and the tagged rows of T_2 into similarly tagged rows of T_1. A symbol mapping h extends naturally to rows and to constraints. We shall call such a symbol mapping h an homomorphism from ϕ_2 to ϕ_1 if it also has the property that whenever constraints C_1 are satisfied so are $h(C_2)$, i.e., when constraints C_1 imply constraints $h(C_2)$.

Let a *database state* be a possibly infinite relational structure (in the sense of first order logic). Our analysis applies both to query containment of queries whose inputs are only finite database states, and to queries whose inputs are any database states. We at first prove using standard database theory techniques the following claim:

Claim Let $\phi_1 = (T_1, C_1)$ and $\phi_2 = (T_2, C_2)$ be two normal form tagged untyped tableau queries with polynomial constraints. Let $h_1, \ldots, h_m$ be all the possible symbol mappings from T_2 to T_1. For all database states d, $\phi_1(d) \subseteq \phi_2(d)$ iff C_1 implies $h_1(C_2) \vee \cdots \vee h_m(C_2)$.

Proof of Claim: (If) If θ_1 is any constraint satisfying valuation for ϕ_1 (i.e. θ_1 is an assignment of constant values to variables), then $\theta_1(C_1)$ is true and, by the hypothesis, there is a symbol mapping h_k such that $\theta_1(h_k(C_2))$ is true. Then we can take $\theta_2 = \theta_1 h_k$ as a valuation

for ϕ_2. This implies that for any database state d, $\phi_1(d) \subseteq \phi_2(d)$.

(Only if) Let d be any database state and θ_1 be a valuation for T_1 that satisfies C_1, yielding some summary row output. Then there must be another valuation θ_2 for T_2 that satisfies C_2, yielding the same summary row output. Moreover, we can restrict θ_2 to map the rows of T_2 only to the image of θ_1, i.e. the database tuples accessed to make a valid valuation. This restriction is without loss of generality, because the database could indeed be no larger, and if the query containment holds in this restriction, then it also holds for any larger database that contains the image.

Now take any row t in T_2. θ_2 maps t into a tuple t' in the database. θ_1 also maps at least one row t'' into t' (choose an arbitrary t''). Then we can construct a mapping h from t to t'', by following the arrows in the mapping of θ_2 and reversing the arrows in the mapping of θ_1. For example, if $t = (a, b, c), t' = (5, 8, 5)$ and $t'' = (x, y, z)$, then we can take $h = (a \rightarrow x, b \rightarrow y, c \rightarrow z)$. Moreover, continuing this way h can be expanded into a complete symbol mapping from T_2 to T_1, because the variables are distinct in T_2 so there are no clashes in the symbol mapping. This shows that if $\theta_1(C_1)$ is true then there is a valuation θ_2 and a symbol mapping h such that $\theta_2(C_2)$ is true and $\theta_1 h = \theta_2$ and thus $\theta_1(h(C_2))$ is true.

Therefore we see that for any valid valuation θ_1 of C_1 there is some symbol mapping h depending on θ_1 such that $\theta_1(h(C_2))$ is true. Moreover, the above argument did not use any assumption about C_1. Hence, this shows that for all C_1's, C_1 implies $h_1(C_2) \vee \cdots \vee h_m(C_2)$, where $h_1, \ldots, h_m$ are all the possible symbol mappings from T_2 to T_1. So the claim holds. □

Finally, to complete Theorem 2.6 we use from [37], p. 139, the following simple geometric fact: "an affine space is contained in a union of affine spaces iff it is contained in one member of this union".

For linear equation constraints, each of the conjunction of constraints C_1 and $h_i(C_2)$ describes an affine space. Moreover, C_1 implies $h_1(C_2) \vee \ldots \vee h_m(C_2)$ iff the affine space C_1 is contained in the union of other affine spaces. But this can happen only if one of the affine spaces $h_i(C_2)$ contains the affine space C_1. Therefore one of the symbol mappings must be a homomorphism from ϕ_2 to ϕ_1. Such a homomorphism can be guessed in NP, and containment of an affine space in another can be checked in polynomial time. This suffices to prove Theorem 2.6 and generalizes the homomorphism theorem with linear equations. □

Let $\phi_1 = (T_1, C_1)$ and $\phi_2 = (T_2, C_2)$ be two queries. We say that the queries have the *homomorphism property*, whenever there is a symbol mapping h from ϕ_2 to ϕ_1 such that for all database states d, $\phi_1(d) \subseteq \phi_2(d)$ iff C_1 implies $h(C_2)$. The homomorphism technique could be extended for tagged untyped tableau queries with a conjunction of linear equation constraints, because those have the homomorphism property. In contrast, with quadratic equations we can show that the homomorphism technique does not work. Furthermore:

Theorem 2.7 Containment of queries that are expressed by tagged untyped tableau with a conjunction of quadratic equation constraints is Π_2^p-hard.

Proof: We can give a simple reduction from the $\forall \overline{x} \exists \overline{y} \psi(\overline{x}, \overline{y})$ quantified boolean formula problem, which is known to be Π_2^p-complete (see [15]). Without loss of generality assume that

in ψ negation is only used on the boolean variables (i.e., negation has been pushed to the leaves of the parse tree of ψ).

Let ϕ_2 be:

$$R(\overline{x}) :\!- x_1(1-x_1)=0,\ldots,x_n(1-x_n)=0, y_1(1-y_1)=0,\ldots,y_m(1-y_m)=0, \chi(\overline{x},\overline{y},\overline{s})$$

In ϕ_2 all the constraints except the last one are used to restrict the $\overline{x}=(x_1,\ldots,x_n)$ and the $\overline{y}=(y_1,\ldots,y_m)$ vectors of variables to be either 0's or 1's. The formula $\chi(\overline{x},\overline{y},\overline{s})$ denotes the conjunction of quadratic constraints that is constructed as follows. Let $F_1,\ldots,F_l$ be the subformulas of ψ, with $F_l=\psi$. Let $s_1,\ldots,s_l$ be distinct fresh variables. Then add the conjunct $s_k=s_i+s_j$ whenever $F_k=F_i\wedge F_j$, add $s_k=s_is_j$ whenever $F_k=F_i\vee F_j$, add $s_k=(1-s_i)$ whenever $F_k=\neg F_i$. If $F_k=F_i$ and F_i is a boolean variable x_i or y_i in $\psi(\overline{x},\overline{y})$, add $s_k=(1-x_i)$ or $s_k=(1-y_i)$. Finally add the conjunct $s_l=0$.

By induction, it can be proven that for any truth assignment $\psi(\overline{x},\overline{y})$ is true iff $\exists\overline{s}\chi(\overline{x},\overline{y},\overline{s})$ is true assigning 1 (0) to x_i,y_i if the respective boolean variables get assigned true (false). The basic intuition is that subformula F_i is made true by the assignment iff constraint $s_i=0$ is satisfied. Hence, ϕ_2 will have as ouput all $\overline{x}$ truth assignments for which there is some $\overline{y}$ truth assignment such that $\psi(\overline{x},\overline{y})$ holds. Then let ϕ_1 be:

$$R(\overline{x}) :\!- x_1(1-x_1)=0,\ldots,x_n(1-x_n)=0$$

Note that ϕ_1 will have as output all possible $\overline{x}$ vectors, hence $\phi_1\subseteq\phi_2$ if and only if the quantified boolean formula holds. □

Containment in the presence of inequality constraints without $+$ and $*$ is the problem examined in [25]. For this case proving Π_2^p-hardness is still open, although it is known that the homomorphism technique fails.

For containment of tagged untyped tableau with inequalities and no $+,*$, we can show that the homomorphism technique fails *even for semiinterval queries.* Semiinterval queries are those in which each variable is bounded by a constant from only one side i.e., left or right, and there are no other constraints. [25] shows that the homomorphism technique works for left-semiinterval queries alone or right-semiinterval queries alone and does not work for interval queries (i.e., when variables are bounded from both sides).

Example 2.8 The homomorphism property fails for semiinterval queries.

Proof: We give two example queries ϕ_1 and ϕ_2 such that $\phi_1\subseteq\phi_2$, but there is no homomorphism from ϕ_2 to ϕ_1. The query ϕ_2 is:

$$R''(u) :\!- R'(u), R(x,y), x\leq 4, y\geq 4$$

While the query ϕ_1 is:

$$R''(u) :\!- R'(u), R(a,b), R(b,c), a\leq 4, c\geq 4$$

There are two possible symbol mappings from the symbols of ϕ_2 to the symbols of ϕ_1. The first is $h_1 = (u \to u, x \to a, y \to b)$, and the second is $h_2 = (u \to u, x \to b, y \to c)$.

$\phi_1 \subseteq \phi_2$ is easy to see, because either $b \geq 4$ and $R(a,b), R(b,c), a \leq 4, c \geq 4$ implies $R(a,b), a \leq 4, b \geq 4$, which is the same as in ϕ_2 after renaming, or $b < 4$ and $R(a,b), R(b,c), a \leq 4, c \geq 4$ implies $R(b,c), b \leq 4, c \geq 4$, which is again the same as in ϕ_2 after a different renaming. Therefore, for each database if ϕ_1 gives some output, then ϕ_2 also gives a superset of that output. However, one symbol mapping is not enough to show containment.

Consider now the database $R(1,3), R(3,5), R'(7)$. Then for the valuation $\theta = (a \to 1, b \to 3, c \to 5, u \to 7)$, ϕ_1 yields $R''(7)$. However, $\theta h_1(\phi_2)$ is not a valid valuation for ϕ_2, because $\theta h_1(R(x,y), x \leq 4, y \geq 4) = R(1,3), 1 \leq 4, 3 \geq 4$, i.e., it is unsatisfiable and cannot produce any output tuple.

Similarly, considering the database $R(1,5), R(5,9), R'(7)$ and the valuation $\theta' = (a \to 1, b \to 5, c \to 9, u \to 7)$, ϕ_1 again yields $R''(7)$. However, $\theta' h_2(\phi_2)$ is not a valid valuation, because $\theta' h_2(R(x,y), x \leq 4, y \geq 4) = R(5,9), 9 \geq 4, 5 \leq 4$, i.e., it is also unsatisfiable and cannot produce any output tuple. □

2.3 High-Dimension Variables as a Programming Primitive

By modeling databases as sets of constraints, we in effect have generalized the notion of a tuple. Variables, however, still range over individual points. We could generalize the notion of a variable to let variables range over objects such as lines or planes. There is a certain similarity here with the set *complex object construct* from complex object database systems. We argue (by example) that these high-dimension variables could be very useful programming primitives.

We shall say that a variable $x^{[k]}$ is *of dimension* k if the variable ranges over k-dimensional sets of points. If R is a k-ary relation, the language has atomic formulas of the form $R(x^{[k]})$. Such a formula means that all of the points in the value of $x^{[k]}$ are in the relation R. We also have atomic formulas of the form $x^{[k]} = y^{[k]}$ and $x^{[k]} \subseteq y^{[l]}$, where $k < l$ and the second formula means that all the points in the value of $x^{[k]}$ are also in the value of $y^{[l]}$.

The main advantage in using high-dimension variables is that many queries can then be expressed in a much simpler and more natural way. For example, line intersection can be written as

$$\phi(p) \equiv (\exists l_1, l_2)(R(l_1) \wedge R(l_2) \wedge l_1 \neq l_2 \wedge p \subseteq l_1 \wedge p \subseteq l_2)$$

In this example we use the convention that p is a variable of dimension 0 ranging over points, and l_1 and l_2 are of dimension 1 ranging over lines. Convex hull and Voronoi diagram can also be expressed more naturally using high-dimension variables. It is interesting to note that many geometric problems can be expressed using high-dimension variables and *linear* constraints.

A natural question is whether adding high-dimension variables makes the relational calculus with polynomial constraints more expressive. It turns out that both languages with or without these variables have the same expressive power (this was pointed out to us by Allen Van Gelder).

To see why this is true, we give an example how a formula that uses a 1-dimensional variable l ranging over lines can be converted into a formula with ordinary (i.e., 0-dimensional) variables. The idea is that l can be represented by 4 ordinary variables x_1, y_1, and x_2, y_2, that represent 2 points on the line. Quantification on l is then replaced by quantification over these four variables. The formula $R(l)$ then becomes $(\forall z)R(zx_1 + (1-z)x_2, zy_1 + (1-z)y_2)$, and the formula $(x, y) \subseteq l$ becomes $(\exists z)(zx_1 + (1-z)x_2 = x \wedge zy_1 + (1-z)y_2 = y)$.

We believe that: *high-dimension variables in CQL's are an interesting shorthand that both facilitates writing natural programs and might provide clues, e.g., linear constraints, for simpler evaluation strategies.*

3 Datalog$^\neg$ and Rational Order Constraints

As shown in [19] and [45] the fixpoint queries of [7] together with a finite discrete order express exactly PTIME. It follows from the proofs of this fact, as well as the normal form results of [19, 16, 1] that Inflationary Datalog$^\neg$ with a finite discrete order expresses exactly PTIME. These simulation proofs can be modified, by making all programs use only constants appearing in the database, to show that Inflationary Datalog$^\neg$ with rational order expresses *at least* PTIME.

Given this observation, it is interesting to consider an extended setting with generalized databases and inequality constraints, but no $+$ or $*$. Using data complexity analysis, we show that Inflationary Datalog$^\neg$ with rational order expresses *at most* PTIME.

In Section 3.1, we examine our new framework with constraints of the form $x < y$, $x = y$, $x \neq 2.5$ etc, over the domain of the rational numbers. We show that the relational calculus with rational order constraints can be evaluated bottom-up in closed form and LOGSPACE data complexity. That result does not follow from the analysis of polynomial constraints, the data complexity is tighter than NC.

In Section 3.2, we first prove that Inflationary Datalog$^\neg$ with rational order can be evaluated bottom-up in PTIME in the size of the database, using our evaluation method for relational calculus with rational order constraints. We then provide a second proof for Datalog, that emphasizes logic programming tools (as opposed to decision procedures for logical theories). The motivation for this is to gain more intuition about Herbrand atoms, minimal models, derivation trees, and the other machinery of constraint logic programming [20, 30].

In Section 3.3, we examine the parallel bottom-up evaluation of Datalog programs with rational order constraints, using the logic programming tools developed in 3.2.

3.1 Relational Calculus and Rational Order Constraints

This section contains a detailed proof of the following:

Theorem 3.1 Relational calculus with rational order constraints can be evaluated bottom-up in closed form and LOGSPACE data complexity.

Proof: We extend the proof of [14] to languages with constants and adapt it to data complexity. Our basic technical contribution is the appropriate definition of r-configuration.

In order to use the decision procedure techniques of [14], we transform the query program applied to the input set of constraints into one semantically equivalent formula ϕ of the theory of rational order with constants (see Section 1). For example, let $R(x, y)$ be a binary relation containing the constraints $x < y$, $x < 5$ and $y = 4$. Consider the query $(\exists z)(R(x, z) \wedge R(z, y))$ applied to this generalized database. Then the equivalent formula is

$$\phi(x, y) \equiv (\exists z)((x < z \vee x < 5 \vee z = 4) \wedge (z < y \vee z < 5 \vee y = 4))$$

Furthermore, we can, within the required resource bounds, eliminate all occurrences of $=$ and just use $<$, by replacing $x = y$ by $\neg((x < y) \vee (x > y))$. We similarly eliminate all logical connectives but $\vee, \neg, \exists$. This is for minimizing the case analysis in the proof.

In what follows: *we fix the query program and the input generalized database and consider the equivalent formula* ϕ (as in the above example). The following definition of r-configuration is the key one. Note that it is with respect to the formula ϕ, which we consider fixed. We shall refer throughout to r-configurations, without mentioning the formula ϕ.

Definition 3.2 Let D_ϕ be the set of constants that appear in ϕ. An *r-configuration* $\xi = (\overline{f}, \overline{l}, \overline{u})$ of size n consists of a sequence $\overline{f} = (f_1, \ldots, f_n)$, where $\{f_1, \ldots, f_n\} = \{1, \ldots, j\}$, for some $j \leq n$, and two sequences $\overline{l} = (l_1, \ldots l_n)$ and $\overline{u} = (u_1, \ldots, u_n)$, where the l_i's are in $D_\phi \cup \{-\infty\}$, and u_i's are in $D_\phi \cup \{+\infty\}$, such that:

1. For all i, $l_i \leq u_i$.
2. There is no constant c in D_ϕ with the property that $l_i < c < u_i$.
3. Whenever $f_i < f_j$, then $l_i < u_j$.
4. Whenever $f_i = f_j$, then $l_i = l_j$ and $u_i = u_j$. □

The idea behind r-configurations is as follows. Consider two rational n-dimensional points $\overline{x} = (x_1, \ldots, x_n)$ and $\overline{y} = (y_1, \ldots, y_n)$. We want to know whether they can be distinguished using the order constraints and the available constants. We say that: these points *can be distinguished* if

1. The relative order of the x_i's is different from the relative order of the y_i's, or
2. Some x_i is in a different relation to some constant in D_ϕ than some y_i.

Each r-configuration characterizes a set of non-distinguishable points. The f_i's describe the relative order of the x_i's, i.e., $x_i < x_j$ iff $f_i < f_j$. l_i and u_i, on the other hand, bound x_i from below and above by constants from $D_\phi \cup \{+\infty, -\infty\}$ in the tightest fashion possible.

Example 3.3 Assume that the constants in D_ϕ are $\{0,1,2,3\}$. The sequence of numbers $(0.5, 3.5, 1.5, 1.5, 2)$ can then be represented by the r-configuration consisting of

1. $\overline{f} = (1,4,2,2,3)$ describing the order between the elements of the sequence.
2. $\overline{l} = (0,3,1,1,2)$
3. $\overline{u} = (1,+\infty,2,2,2)$□

We also need some technical definitions: (1) Express an r-configuration as a conjunction of constraints. (2) The points satisfying this conjunction are the indistinguishable points denoted by the r-configuration. (3) An r-configuration can be extended by adding other variables.

Definition 3.4 The formula $F(\xi)$, with n free variables $\{x_1, \ldots, x_n\}$, corresponding to an r-configuration $\xi = (\overline{f}, \overline{l}, \overline{u})$, of size n, is the conjunction of: (1) $x_i < x_j$, whenever $f_i < f_j$. (2) $x_i = x_j$, whenever $f_i = f_j$. (3) $l_i < x_i < u_i$ whenever $l_i < u_i$. (4) $x_i = l_i$ whenever $l_i = u_i$. □

Definition 3.5 $\models F(\xi)(a_1, \ldots, a_n)$ will mean that $F(\xi)$ is satisfied by the assignment of each a_i to the corresponding variable x_i.□

Definition 3.6 Let $\xi = (\overline{f}, \overline{l}, \overline{u})$ be an r-configuration of size n. An r-configuration $\xi' = (\overline{f}', \overline{l}', \overline{u}')$ of size $n+1$ is an *extension* of ξ iff

1. $\overline{f}'$ is an extension of $\overline{f}$. This means that for all i, j, $1 \leq i, j \leq n$, $f'_i < f'_j$ iff $f_i < f_j$,
2. $\overline{l}' = (l_1, \ldots, l_n, l_{n+1})$, and
3. $\overline{u}' = (u_1, \ldots, u_n, u_{n+1})$.□

The idea behind the proof is as follows. We show that the r-configurations partition multi-dimensional space in such a way that to test whether a subformula of the query holds throughout an r-configuration, it suffices to test whether it holds at an arbitrary point in this configuration. This partitioning is made formal using the following five Lemmas 3.7–11.

We use this partitioning to construct an algorithm ALG, which evaluates the query in closed form. The output of ALG consists of a set of r-configurations. Its correctness is described in the next two Lemmas 3.12–13. We then argue that ALG can be implemented within the required resource bounds.

Lemma 3.7 Let ξ be an r-configuration of size n, and let ξ' be an extension of size $n+1$. Then we have that $\models F(\xi)(a_1,\ldots,a_n)$ iff for some $a \models F(\xi')(a_1,\ldots,a_n,a)$.

Proof: Since $F(\xi)$ is a conjunction of some of the conjuncts in $F(\xi')$, $F(\xi')(a_1,\ldots,a_n,a)$ implies $F(\xi)(a_1,\ldots,a_n)$. For the converse,

1. If for some i, $f_{n+1} = f_i$, let $a = a_i$.

2. Otherwise, let i and j be such that f_i and f_j are maximal and minimal, respectively, satisfying $f_i < f_{n+1} < f_j$ (the construction can easily be modified to handle the boundary cases with i or j nonexistent). If $l_{n+1} = u_{n+1}$, let $a = l_{n+1}$. It then follows that $a_i < a < a_j$. Otherwise, pick a arbitrarily satisfying the conditions $a_i < a < a_j$ and $l_{n+1} < a < u_{n+1}$. The reason such an a exists is shown as follows. Such an a would not exist only if $a_j \leq l_{n+1}$ or $u_{n+1} \leq a_i$. Assume $a_j \leq l_{n+1}$. $f_{n+1} < f_j$, together with part 3 of Definition 3.2, implies that $l_{n+1} < u_j$. However, by the definition of $F(\xi)$, $l_j \leq a_j$. Since $l_j \leq a_j \leq l_{n+1} < u_j$, it follows that $l_j < u_j$; and by the definition of $F(\xi)$ once more, $l_j < a_j$. We therefore have

$$l_j < a_j \leq l_{n+1} < u_j$$

   which contradicts part 2 of Definition 3.2. The second case is similar.

In both cases, it follows that $(a_1,\ldots,a_n,a)$ satisfies $F(\xi')$.□

Lemma 3.8 Let ξ be an r-configuration of size n. There exist rational numbers $a_1, \ldots, a_n$, such that $\models F(\xi)(a_1,\ldots,a_n)$.

Proof: Induction on the size of ξ, using Lemma 3.7.□

Lemma 3.9 Let $a_1, \ldots, a_n$ be rational numbers. There exists a unique r-configuration ξ of size n such that $\models F(\xi)(a_1,\ldots,a_n)$.

Proof: Uniqueness follows from the fact that if $\xi^1 \neq \xi^2$, then $F(\xi^1) \wedge F(\xi^2)$ is unsatisfiable. To show this, suppose that $(a_1,\ldots,a_n)$ satisfies $F(\xi^1) \wedge F(\xi^2)$. Let $\xi^1 = (\overline{f}^1, \overline{l}^1, \overline{u}^1)$ and $\xi^2 = (\overline{f}^2, \overline{l}^2, \overline{u}^2)$.

1. Let $f_i^1 \neq f_i^2$, w.l.o.g, $f_i^1 < f_i^2$. Since $\overline{f}^2$ is a sequence that consists of *all* the integers from 1 to some k (possibly with repetitions), it follows that for some j, $f_j^2 = f_i^1$. But then, by Definition 3.4, $a_j < a_i$. This implies, using Definition 3.4 again, that $f_j^1 < f_j^2$. Repeating this, we get an infinite descending sequence of positive integers, a contradiction.

2. Let $l_i^1 \neq l_i^2$, w.l.o.g., $l_i^1 < l_i^2$. Suppose first that $l_i^1 = u_i^1$. Then, by Definition 3.4, $l_i^1 = a_i < l_i^2 \leq a_i$, a contradiction. On the other hand, if $l_i^1 < u_i^1$, then part 2 of Definition 3.2 implies that $u_i^1 \leq l_i^2$. Once more, by Definition 3.4, $a_i < u_i^1 \leq l_i^2 \leq a_i$, a contradiction.

3. The case when $u_i^1 \neq u_i^2$ is similar.

Existence is shown by induction on n. The case $n = 1$ is trivial. Assuming that the result holds for n, let $a_1, \ldots, a_n, a_{n+1}$ be given, and let ξ be such that $(a_1, \ldots, a_n)$ satisfies $F(\xi)$. We show how to extend ξ to ξ' such that $(a_1, \ldots, a_{n+1})$ satisfies $F(\xi')$. There are two cases to consider for $a = a_{n+1}$:

1. For some i, $a = a_i$. ξ' is defined by $f'_j = f_j$ for $i \leq n$, $f'_{n+1} = f_i$, $l'_{n+1} = l_i$ and $u'_{n+1} = u_i$.
2. Let i and j be such that a_i and a_j are maximal and minimal, respectively, satisfying $a_i < a < a_j$ (the construction can easily be modified to handle the boundary cases). l_{n+1} will be the largest constant in D_ϕ such that $l_{n+1} \leq a$, u_{n+1} the smallest such that $a \leq u_{n+1}$. $\overline{f}'$ is defined in such a way that it is compatible with the ordering of the a_i's, i.e.,
   (a) If $f_k \leq f_i$, then $f'_k = f_k$.
   (b) If $f_k \geq f_j$, then $f'_k = f_k + 1$.
   (c) $f'_{n+1} = f_j$.

In both cases, $\models F(\xi')(a_1, \ldots, a_{n+1})$.□

Lemma 3.10 Let ψ be a formula, with at most k free variables, using only constants in D_ϕ. Let ξ be an r-configuration of size k and $\models F(\xi)(a_1, \ldots, a_k)$ and $\models F(\xi)(a'_1, \ldots, a'_k)$, then $\models \psi(a_1, \ldots, a_k) \Leftrightarrow \models \psi(a'_1, \ldots, a'_k)$.

Proof: We show, by induction on the size of ψ, that the result holds for all r-configurations ξ and all a_i's and a'_i's.

1. Atomic formulas. There are two types of atomic formulas $x_i < x_j$ and $x_i < c$. In the first case, $a_i < a_j$ iff $f_i < f_j$, and likewise, $a'_i < a'_j$ iff $f_i < f_j$. Therefore, $a_i < a_j$ iff $a'_i < a'_j$. In the second case, $a_i < c$ iff $u_i < c$ or $l_i < u_i \leq c$, and similarly for a'_i.
2. If ψ is of the form $\neg\psi_1$ or $\psi_1 \vee \psi_2$, the proof is straightforward.
3. If ψ is $(\exists x)\psi'$, then $(a_1, \ldots, a_k)$ satisfies ψ iff for some a, $(a_1, \ldots, a_k, a)$ satisfies ψ'. By Lemma 3.9, there is an r-configuration ξ' such that $(a_1, \ldots, a_k, a)$ satisfies $F(\xi')$. It is easy to see (by the existence argument in Lemma 3.9) that ξ' must be an extension of ξ. Since $(a'_1, \ldots, a'_k)$ satisfies $F(\xi)$, by Lemma 3.7 there is an a' such that $(a'_1, \ldots, a'_k, a')$ satisfies $F(\xi')$. But then, by the induction hypothesis, $(a'_1, \ldots, a'_k, a')$ satisfies ψ', and therefore $(a'_1, \ldots, a'_k)$ satisfies ψ. □

Lemma 3.11 Let ξ be an r-configuration, and ψ a formula using only the constants in D_ϕ. Then $F(\xi) \rightarrow \psi$ is valid (i.e., true for all assignments to the free variables) iff $F(\xi) \wedge \psi$ is satisfiable (i.e., true for some assignment to the free variables).

Proof: If $F(\xi) \rightarrow \psi$ is valid, then Lemma 3.8 implies that $F(\xi) \wedge \psi$ is satisfiable. On the other hand, if $F(\xi) \wedge \psi$ is satisfiable, say by $(a_1, \ldots, a_n)$, and $F(\xi)$ is satisfied by $(a'_1, \ldots, a'_n)$, then Lemma 3.10 implies that ψ is satisfied by $(a'_1, \ldots, a'_n)$. □

Query Evaluation Algorithm ALG

Input of ALG: A formula $\phi \equiv$ (program + input generalized database) with n free variables. Let D_ϕ be the set of constants in ϕ.

For each r-configuration ξ of size n (with constants from D_ϕ), test whether $F(\xi) \rightarrow \phi$ is valid. This test is performed using recursive algorithm BALG. The algorithm BALG takes as input an r-configuration ξ' and a formula ψ that uses only constants in D_ϕ. It returns 1 iff $F(\xi') \rightarrow \psi$ is valid. The various cases of BALG for ψ are:

1. ψ is an atomic formula $x_i < x_j$.
If $f_i < f_j$ (where f_i, f_j are from ξ') then return 1 else return 0.

2. ψ is an atomic formula $x_i < c$.
If $l_i = u_i < c$ or $l_i < u_i \leq c$ (where l_i, u_i are from ξ') then return 1 else return 0.

3. ψ is $\psi_1 \vee \psi_2$.
If result of BALG(ξ',ψ_1) is 1 then return 1, else return result of BALG(ξ',ψ_2).

4. ψ is $\neg\psi'$.
If result of BALG(ξ',ψ') is 1 then return 0 else return 1.

5. ψ is $(\exists x)\psi'$.
For every extension ξ'' of ξ', do BALG(ξ'',ψ').
If the result on one of these r-configurations is 1 then return 1 else return 0.

Output of ALG: The disjunction of the $F(\xi)$'s for which $F(\xi) \rightarrow \phi$ is valid.

We now prove correctness of BALG and ALG.

Lemma 3.12 The algorithm BALG(ξ', ψ) described above returns 1 iff $F(\xi') \rightarrow \psi$ is valid.

Proof: The proof is by induction on the structure of ψ.

1. ψ is an atomic formula $x_i < x_j$. The proof of correctness is trivial.
2. ψ is an atomic formula $x_i < c$. Correctness when $l_i = u_i < c$ is trivial. When $l_i < u_i \leq c$, correctness follows from the fact that $c \in D_\phi$ and thus $c < u_i$ implies $c \leq l_i$.
3. ψ is $\psi_1 \vee \psi_2$. Correctness follows from Lemma 3.11, together with the fact that $F(\xi') \wedge (\psi_1 \vee \psi_2)$ is satisfiable iff $(F(\xi') \wedge \psi_1) \vee (F(\xi') \wedge \psi_2)$ is satisfiable.
4. ψ is $\neg\psi'$. To show correctness we must show that $F(\xi') \rightarrow \neg\psi'$ is valid iff $(F(\xi') \rightarrow \psi')$ is not valid. By Lemma 3.11, $F(\xi') \rightarrow \neg\psi'$ is valid iff $F(\xi') \wedge \neg\psi'$ is satisfiable. But $F(\xi') \wedge \neg\psi'$ is the same as $\neg(F(\xi') \rightarrow \psi')$, which completes the proof.

5. ψ is $(\exists x)\psi'$. To show correctness it suffices, by Lemma 3.11, to show that $F(\xi') \wedge (\exists x)\psi'$ is satisfiable iff $F(\xi'') \wedge \psi'$ is satisfiable for some ξ''.

   For the if, assume that the formula $F(\xi'') \wedge \psi'$ is satisfied by some $(a_1, \ldots, a_n, a)$. Lemma 3.7, then implies that $F(\xi') \wedge (\exists x)\psi'$ is satisfied by $(a_1, \ldots, a_n)$. For the only if, assume that $F(\xi') \wedge (\exists x)\psi'$ is satisfied by $(a_1, \ldots, a_n)$. There must then exist a rational number a, such that $(a_1, \ldots, a_n, a)$ satisfies ψ'. By Lemma 3.9, there is a r-configuration ξ'' such that $(a_1, \ldots, a_n, a)$ satisfies $F(\xi'')$. By restricting ξ'' to the first n variables, and using Lemma 3.7 together with the uniqueness part of Lemma 3.9, it follows that ξ'' is an extension of ξ'.

To show that ALG is correct, we have to show that it outputs a formula in DNF that is equivalent to ϕ. The formula output by ALG is clearly in DNF, and it therefore suffices to show:

Lemma 3.13 The result of ALG is equivalent to the formula ϕ.

Proof: Let $S = \{\xi_1, \ldots, \xi_n\}$ be the set of those configurations for which of $F(\xi_i) \rightarrow \phi$ is valid. Clearly, $\bigvee_{1 \leq i \leq n} F(\xi_i) \rightarrow \phi$ is valid. For the converse, let $(a_1, \ldots, a_k)$ be an assignment of values to the free variables of ϕ such that $\models \phi(a_1, \ldots, a_k)$. By Lemma 3.9, there exists a configuration ξ such that $(a_1, \ldots, a_k)$ satisfies $F(\xi)$. But then $(a_1, \ldots, a_k)$ satisfies $F(\xi) \wedge \phi$, and by Lemma 3.11, it follows that $\xi \in S$, completing the proof. □

We still have to show that the algorithm is in LOGSPACE. First, consider BALG. The first problem we have is that we cannot construct the formula ϕ in LOGSPACE. We therefore have to use the query formula as given, switching to the database whenever the query contains a predicate symbol, rather than copying explicitly the contents of the relation at this point.

To run BALG we need to store the current configuration. Since we have a fixed query formula, we have a bound on the number of quantifiers, and hence on the maximum size of the configurations we have to consider. It then follows that we can store each configuration in LOGSPACE.

For a given configuration ξ, we use a fixed number of pointers to find the subformulas of ϕ and perform the appropriate steps of BALG on them. Whenever we encounter a predicate symbol, we use one pointer to remember where we are in the query, and a second pointer to scan the database, as though it were part of the query formula at this point. The first pointer is to remember where to return to after we reach the end of the database relation.

Most of the subcases of BALG are straightforward. When we are considering a quantifier, however, we have to iterate over all extensions of ξ. We can do this in LOGSPACE by considering each extension to ξ in turn, and first testing whether it is a legal configuration or not. This shows that BALG is a LOGSPACE algorithm.

The general algorithm ALG iterates over all configurations ξ, and performs BALG on each one. As before, we can easily iterate over all configurations in LOGSPACE, and this shows that ALG is also in LOGSPACE. This completes the proof of Theorem 3.1. □

3.2 Datalog Bottom-up Evaluation Revisited

Theorem 3.14 Inflationary Datalog$^\neg$ with rational order constraints can be evaluated bottom-up in closed form and PTIME data complexity.

Proof: If we are required to iterate a relational calculus formula, we can use ALG of the previous section as one iteration of the query. Since the relational calculus formula is *fixed*, there are at most a polynomial number of r-configurations. Since under inflationary semantics we can only add r-configurations at each iteration, we obtain a polynomial time algorithm for Inflationary Datalog$^\neg$ with rational order constraints. □

Let us now consider an alternative proof for the Datalog case of this theorem. The main idea comes from the semantics of Constraint Logic Programming [20]. It involves generalizing the notion of a Herbrand atom. The result is a "natural" bottom-up evaluation for Datalog with rational order constraints.

Definition 3.15 Let P be a *generalized database logic program*, i.e., a Dåtalog + constraints program defining the IDB predicates *and* a generalized database defining the EDB predicates.

1. A *generalized EDB Herbrand atom* is an EDB predicate symbol with distinct variable symbols as arguments and a conjunction of rational order constraints on these variables. (Note that these atoms are generalized tuples).
2. A *generalized IDB Herbrand atom* is an IDB predicate symbol with distinct variable symbols as arguments and an r-configuration ξ on these variables. $F(\xi)$, as in Definition 3.4, is a conjunction of constraints on these variables. (Note that these atoms are generalized tuples of a special form). □

Example 3.16 For example, an r-configuration can also describe a generalized Herbrand atom when it is attached to a predicate symbol. Start from the r-configuration ξ of Example 3.3, assume predicate R has arity 5 and the database has only the constants $\{0,1,2,3\}$ in it. The conjunction of constraints $F(\xi)$ gives us a generalized Herbrand atom denoted as:

$$R(x_1, x_2, x_3, x_4, x_5) :\!- 0 < x_1 < 1 < x_3 = x_4 < 2 = x_5 < 3 < x_2.$$

□

First, let us observe that r-configurations are closed under projection. We say that an r-configuration is *projected* onto a subset of its variables when all the variables outside this subset are eliminated, i.e., their bounds are deleted from $\overline{l}, \overline{u}$ and they are removed from the ordering $\overline{f}$. It is easy to see that the projection of an r-configuration is an r-configuration of smaller size. Now, let us develop the necessary constraint logic programming terminology.

The evaluation of a generalized database logic program P starts by collecting in a set H all the predicate, variable, and rational constant symbols that occur in the program, that is, either in its rules or in its database part. We call H the *generalized Herbrand base* of P, because

all symbols that are ever used during our evaluation must be in H. Since each program is by definition finite, H must be also a finite set.

The *generalized Herbrand universe* of P consists of all generalized EDB Herbrand atoms of P (these remain fixed throughout the evaluation of the program) and *all* generalized IDB Herbrand atoms that can be built out of the generalized Herbrand base. Since there is a finite number of r-configurations the generalized Herbrand universe is finite. Its lattice of subsets is also finite and thus complete.

We let an *interpretation* I of P be a subset of its generalized Herbrand universe, containing all the generalized EDB Herbrand atoms of P.

Definition 3.17 Let P be a generalized database logic program and H be its generalized Herbrand base. Let I be an interpretation of P. All generalized EDB Herbrand atoms of I are *derivable in one rule firing from P and I*. Generalized IDB Herbrand atoms are *derivable in one rule firing from P and I* as follows:

Choose any rule $A_0 :\!-\ A_1, A_2, \ldots, A_k, C$ of P, where $A_0, A_1, A_2, \ldots, A_k$ are relational atoms and C is a conjunction of rational order constraints. Without loss of generality the n occurrences of variables in the relational atoms are distinct and any equalities are part in C.

1. Choose any r-configuration ξ of size n built from H.
2. Check that $F(\xi) \rightarrow C$ is valid, i.e., true for all values of the free variables.
3. If $A_i, 1 \leq i \leq k$ is an EDB relational atom $R(\ldots)$ then project ξ onto its variables to produce r-configuration ξ_i. Check that for some generalized EDB Herbrand atom $R(\ldots) :\!-\ \psi$ in I we have that $F(\xi_i) \rightarrow \psi$ is valid.
4. If $A_i, 1 \leq i \leq k$ is an IDB relational atom $R(\ldots)$ then project ξ onto its variables to produce r-configuration ξ_i. Check that generalized IDB Herbrand atom $R(\ldots) :\!-\ F(\xi_i)$ is in I.
5. If all tests are true and if A_0 is the IDB relational atom $R(\ldots)$ then project ξ onto its variables to produce r-configuration ξ_0. *Fire the rule once to derive* generalized IDB Herbrand atom $R(\ldots) :\!-\ F(\xi_0)$.

We define a function T_P from interpretations to interpretations as follows:

$$T_P(I) = \{A : A \textit{ is derivable in one rule firing from } P \textit{ and } I\}$$

□

In the above definition, generalized IDB Herbrand atoms are effectively r-configurations and are treated purely syntactically. The constraints C and the generalized EDB Herbrand atoms are treated in a slightly different fashion. This is because we avoid transforming them into disjunctions of r-configurations. This could be done but would be rather awkward and unnecessary. Let us comment on the tests involving C and the EDBs.

(1) Checking that $F(\xi) \rightarrow C$ is valid can be done simply by picking one assignment to the variables that satisfies $F(\xi)$ and verifying that it satisfies C. This is by Lemmas 3.10 and 3.11.

(2) Checking that, for some generalized EDB Herbrand atom $R(\ldots) :\!\!-\ \psi$, the implication $F(\xi_i) \rightarrow \psi$ is valid can be done scanning the input and for each tuple checking as in (1) above. The reason for this test is to guarantee that the multi-dimensional points of $F(\xi_i)$ are points of the input generalized EDB relation r that corresponds to R. Recall that r is a DNF formula, in fact, it is the disjunction of all possible ψ's of this test. It is interesting to note that by Lemmas 3.10 and 3.11: $F(\xi_i) \rightarrow r$ is valid iff $F(\xi_i) \rightarrow \bigvee \psi$ is valid iff $F(\xi_i) \wedge \bigvee \psi$ is satisfiable iff for some ψ, $F(\xi_i) \wedge \psi$ is satisfiable iff for some ψ, $F(\xi_i) \rightarrow \psi$ is valid.

We define a *model* M of P to be an interpretation of P such that $T_P(M) \subseteq M$. P may have several models ordered according to set inclusion. We have the following analogues to traditional logic programming:

Theorem 3.18 Let P be a generalized database logic program and L_P the intersection of all models of P. Then we have that,
(1) L_P is the unique least model of P.
(2) L_P is the unique least fixpoint of T_P.
(3) L_P can be produced by a finite number of iterations of the mapping T_P.
(4) Each generalized Herbrand atom in L_P is derivable by a finite number of rule firings from the interpretation with empty IDBs.
(4) L_P can be evaluated bottom-up in PTIME data complexity, by rule firings starting from the interpretation with empty IDBs.

Proof: (1) Identical to traditional logic programming model intersection property for Horn clauses. (2) By Tarski's theorem on the complete lattice of subsets of the generalized Herbrand universe. (3) By the finiteness of the generalized Herbrand universe. (4) and (5) The arguments are the same as for Datalog "naive" bottom-up evaluation. □

Call *generalized naive evaluation* the bottom-up evaluation by rule firings starting from the interpretation with empty IDBs. This evaluation has the well defined finite output L_P. Is this generalized database output the desired result according to the semantics defined in Section 1? Recall that for these semantics we interpreted programs as mappings from unrestricted relational databases to unrestricted relational databases. These semantics are computable via *naive evaluation* of Datalog rules on unrestricted relational databases.

The following theorem expresses the fact that generalized naive bottom-up evaluation of P is semantically sound and complete. Namely, given any generalized database representing an input unrestricted relational database, then its generalized database output L_P finitely represents the output of naive evaluation on the unrestricted relational database.

Theorem 3.19 Let P be a generalized database logic program with generalized EDB Herbrand atoms d_1. Let d_2 be the unrestricted relational database represented by the generalized database d_1, that is, $points(d_1) = d_2$. If $L_P(d_1)$ is the output of the *generalized naive evaluation* of P and $L_P(d_2)$ is the ouput of the *naive evaluation* of $P[d_2/d_1]$ (with input d_2 instead of d_1) then $points(L_P(d_1)) = L_P(d_2)$.

Proof: We need to prove two directions. The soundness direction, that is, any point p in $L_P(d_1)$ is also in $L_P(d_2)$, and the completeness direction, that is, any point p in $L_P(d_2)$ is also in $L_P(d_1)$.

We prove each direction by induction on the number of iterations i of the two evaluations. We shall show that for each i, $\mathit{points}(T_P^i(d_1)) = T_P^i(d_2)$. Since generalized naive evaluation is finite this also proves that a finite number of iterations suffice for naive evaluation as well!

For $i = 0$, we have $\mathit{points}(T_P^0(d_1)) = \mathit{points}(d_1) = d_2 = T_P^0(d_2)$, hence the claim holds. Now we assume that $\mathit{points}(T_P^i(d_1)) = T_P^i(d_2)$ is true and prove the equality for $i+1$.

For the soundness direction, let $A_0 :\!-\ A_1, A_2, \ldots, A_k, C$ be any rule of P where C are rational order constraints. To perform a rule firing of generalized naive evaluation, assume that an r-configuration ξ is chosen whose set of satisfying points also satisfy C. Suppose that each of the projections of ξ onto the A's of the body are r-configurations whose set of satisfying points are also satisfied by some atoms of $T_P^i(d_1)$. Then by our rule application ξ is projected onto the head of the rule and is turned into an atom of $T_P^{i+1}(d_1)$. Now let p be any point within ξ. Then all the projections of p are points. By the induction hypothesis, $\mathit{points}(T_P^i(d_1)) = T_P^i(d_2)$. Therefore, each projection of p onto the A's of the body must be points within $T_P^i(d_2)$. Hence, taking that collection of points from $T_P^i(d_2)$ we can derive via naive evaluation into $T_P^{i+1}(d_2)$ the projection of p on the rule head. Thus, $\mathit{points}(T_P^{i+1}(d_1)) \subseteq T_P^{i+1}(d_2)$.

For the completeness direction, let $A_0 :\!-\ A_1, A_2, \ldots, A_k, C$ be any rule of P where C are any rational order constraints. To perform a rule firing in naive evaluation, assume that some multi-dimensional point p is chosen as the values of all variables in the rule. Suppose that the projections of p are all present among the atoms of $T_P^i(d_2)$, and hence the projection of p on the head is derived into $T_P^{i+1}(d_2)$ (call it p'). By Lemma 3.9 p satisfies some unique r-configuration ξ. Use that ξ and generalized naive evaluation. The point p must also satisfy C and by induction some atoms in $T_P^i(d_1)$. By Lemma 3.10 all points in ξ satisfy exactly the same formulas. Hence, taking ξ a generalized naive rule firing can derive (as a projection of ξ onto the head) an atom of $T_P^{i+1}(d_1)$ that contains p'. We can reason similarly for each point, proving that $T_P^{i+1}(d_2) \subseteq \mathit{points}(T_P^{i+1}(d_1))$. □

3.3 Datalog Derivation Trees and Parallelism

A *generalized derivation tree* for generalized Herbrand atom A using program P is a tree whose nodes are labelled as follows:

1. A labels the root.
2. Every leaf is labelled by a generalized EDB Herbrand atom of P.
3. Every internal node is labelled by a generalized IDB Herbrand atom B. Let its children have labels $B_1, \ldots, B_k$. There is a rule in P and an r-configuration such that: B is derived by one firing of this rule using $B_1, \ldots, B_k$ as atoms and this r-configuration.

Each generalized derivation tree illustrates one possible sequence of rule firings to derive the label of its root. An obvious parallel evaluation method tries all possible ways of firing each rule in every iteration step. Therefore the number of iteration steps necessary for this parallel algorithm to derive an IDB atom is exactly the minimum depth generalized derivation tree for the IDB atom.

This observation motivates the definition of a *generalized polynomial fringe property.* We say that a program P has the generalized polynomial fringe property, if each atom in L_P has a generalized derivation tree with at most a fixed polynomial (in the size of the EDB part of P) number of leaves.

The (generalized) polynomial fringe property is a semantic notion. A natural class of queries can be described purely syntactically by *piecewise linear programs* –see [42] for the exact definition. Following [42] one can show that piecewise linear programs always have the (generalized) polynomial fringe property.

Theorem 3.20

1. Datalog programs with rational order constraints that have the generalized polynomial fringe property can be evaluated bottom-up in closed form and NC data complexity.
2. Piecewise linear Datalog with rational order constraints can be evaluated bottom-up in closed form and NC data complexity.

Proof: Use the analysis of parallelism in Datalog programs by Ullman and van Gelder [42] by substituting "generalized derivation tree" for "derivation tree". □

We close this section with the observation that our development of constraint logic programming machinery can have many applications. For example, various forms of analysis of Datalog$^\neg$ in logic programming (e.g stratified or inflationary semantics) can be directly translated into Datalog$^\neg$ + rational order constraints.

4 Datalog$^\neg$ and Equality Constraints

In this section, we study equality constraints, i.e., constraints of the form $x = y$, $x \neq c$, etc., over the integers. Our analysis also applies to any countably infinite domain. The only properties we use are that (1) there are countably infinite elements in the domain, and (2) different constants of our vocabulary denote different elements. In this sense, we have a special case of Datalog$^\neg$ and rational order constraints. We present a separate analysis because for rational order constraints we expressed $\neq$ using $<$; here we do not.

We are interested in this class of constraints for the following reason. Suppose we have a relational database. Any finite relation in this database can be represented as a set of equality constraints. However, in the relational data model, there are "unsafe" queries for which the

result is not finite. In our language "safety" is not a problem, since we still get a closed, finite description of this infinite set.

For the relational calculus and Datalog$^\neg$ applied to finite relations there are similar results in the literature [3, 24, 18]. Our work extends some of the results of [3, 24, 18], by adding recursion and handling any input generalized relation. In fact, the specific number of constants added in [3] and [18] to the "active domain" (the set of constants that appear in the database and in the query) corresponds precisely to the induction depth used in proving our evaluation method correct. We now show that:

Theorem 4.1

1. Relational calculus with equality constraints can be evaluated bottom-up in closed form and LOGSPACE data complexity.
2. Inflationary Datalog$^\neg$ with equality constraints can be evaluated bottom-up in closed form and PTIME data complexity.

Proof: These complexity results are the same as those we have shown for rational order in the previous section. The algorithm and proofs follow the same outline as those in Section 3.1. We have to use a different notion of configuration.

As in that Section 3.1, let ϕ be the query formula applied to the given database instance, and let D_ϕ be the set of constants that appear in ϕ. We will also have a special symbol o whose meaning will be explained below.

Definition 4.2 An *e-configuration* $\xi = (\overline{e}, \overline{v})$ of size n consists of an equivalence relation $\overline{e}$ on $\{1, \ldots, n\}$ and a sequence $\overline{v} = (v_1, \ldots v_n)$, where each v_i is in $D_\phi \cup \{\mathrm{o}\}$, such that,

1. If $i \,\overline{e}\, j$, then $v_i = v_j$, and
2. If $v_i = v_j \neq \mathrm{o}$, then $i \,\overline{e}\, j$. □

Consider a point $\overline{x} = (x_1, \ldots, x_n)$. The e-configuration that contains $\overline{x}$ is determined as follows. The equivalence relation $\overline{e}$ is defined by $i \,\overline{e}\, j$ iff $x_i = x_j$. If x_i is equal to some constant v in D_ϕ, we set v_i equal to v. Otherwise v_i will be the special symbol o whose meaning is that x_i is not equal to any constant in D_ϕ.

Example 4.3 Let D_ϕ be the set $\{1, 2\}$. The sequence $(1, 1, 2, 4, 2, 4, 3)$ is represented by the e-configuration that consists of the equivalence relation $\overline{e}$ $\{\{1, 2\}, \{3, 5\}, \{4, 6\}, \{7\}\}$ together with the sequence $\overline{e} = (1, 1, 2, \mathrm{o}, 2, \mathrm{o}, \mathrm{o})$. □

We now proceed as in Section 3.1.

Definition 4.4 The formula $F(\xi)$ with n free variables $\{x_1, \ldots, x_n\}$ that corresponds to an e-configuration $\xi = (\overline{e}, \overline{v})$ of size n is the conjunction of: (1) $x_i = x_j$, whenever $i\ \overline{e}\ j$, (2) $x_i \neq x_j$, whenever $\neg(i\ \overline{e}\ j)$, (3) $x_i = v_i$ whenever $v_i \neq 0$, and (4) for all v in D_ϕ, $x_i \neq v$ whenever $v_i = 0$. □

Definition 4.5 $\models F(\xi)(a_1, \ldots, a_n)$ means that $F(\xi)$ is satisfied by the assignment of each a_i to the corresponding variable x_i.□

Definition 4.6 Let $\xi = (\overline{e}, \overline{v})$ be an r-configuration of size n. An r-configuration $\xi' = (\overline{e}', \overline{v}')$ of size $n+1$ is an *extension* of ξ iff

1. For all i and j, $1 \leq i, j \leq n$, $i\ \overline{e}\ j$ iff $i\ \overline{e}'\ j$, and
2. $\overline{v}' = (v_1, \ldots, v_n, v'_{n+1})$.□

Lemma 4.7 Let ξ be an e-configuration of size n, and let ξ' be an extension of size $n+1$. Then $\models F(\xi)(a_1, \ldots, a_n)$ iff for some a, $\models F(\xi')(a_1, \ldots, a_n, a)$.

Proof: Since $F(\xi)$ is a conjunction of some of the conjuncts in $F(\xi')$, $F(\xi')(a_1, \ldots, a_n, a)$ implies $F(\xi)(a_1, \ldots, a_n)$. For the converse,

1. If for some i, $i\ \overline{e}'\ n+1$, let $a = a_i$.
2. Otherwise, there are two cases. If v'_{n+1} is in D_ϕ, let $a = v'_{n+1}$. Otherwise, let a be some element of the domain different from $a_1, \ldots, a_n$, and not in D_ϕ. This is possible because there is an unbounded supply of integers. In both cases, it follows that $(a_1, \ldots, a_n, a)$ satisfies $F(\xi')$.□

Lemma 4.8 Let ξ be an e-configuration of size n. Then there exist domain elements $a_1, \ldots, a_n$, such that $\models F(\xi)(a_1, \ldots, a_n)$.

Proof: Induction on the size of ξ, using Lemma 4.7.□

Lemma 4.9 Let $a_1, \ldots, a_n$ be domain elements. There exists a unique e-configuration ξ such that $\models F(\xi)(a_1, \ldots, a_n)$.

Proof: Uniqueness follows from the fact that if $\xi^1 \neq \xi^2$, then $F(\xi^1) \wedge F(\xi^2)$ is unsatisfiable. To show this, suppose that $(a_1, \ldots, a_n)$ satisfies $F(\xi^1) \wedge F(\xi^2)$. Let $\xi^1 = (\overline{e}^1, \overline{v}^1)$ and $\xi^2 = (\overline{e}^2, \overline{v}^2)$.

1. Suppose that $\overline{e}^1 \neq \overline{e}^2$. Let i and j be such that $i\ \overline{e}^1\ j$ but $\neg(i\ \overline{e}^2\ j)$. Then $F(\xi^1)$ contains the conjunct $x_i = x_j$, while $F(\xi^2)$ contains the conjunct $x_i \neq x_j$, and therefore $F(\xi^1) \wedge F(\xi^2)$ is unsatisfiable.

2. Suppose that $\overline{e}^1 = \overline{e}^2$ but $\overline{v}^1 \neq \overline{v}^2$. Let i be such that $v_i^1 \neq v_i^2$. If neither v_i^1 nor v_2^i is equal to $\circ$, then $F(\xi^1)$ contains the conjunct $x_i = v_i^1$, while $F(\xi^2)$ contains the conjunct $x_i = v_i^2$. If on the other hand, say, $v_i^1 = \circ$, then $F(\xi^1)$ contains the conjunct $x_i \neq v_i^2$, while $F(\xi^2)$ contains the conjunct $x_i = v_i^2$.

Existence is shown by induction on n. The case $n = 1$ is trivial. Assuming that the result holds for n, let $a_1, \ldots, a_n$, a_{n+1} be given, and let ξ be such that $(a_1, \ldots, a_n)$ satisfies $F(\xi)$. We show how to extend ξ to ξ' such that $(a_1, \ldots, a_{n+1})$ satisfies $F(\xi')$. There are two cases to consider for $a = a_{n+1}$:

1. For some i, $a = a_i$. ξ' is defined by $j\ \overline{e}'\ k$ iff $j\ \overline{e}\ k$ whenever $1 \leq j, k \leq n$, and $j\ \overline{e}'\ (n+1)$ whenever $i\ \overline{e}\ j$. Also set $v'_{n+1} = v_i$.

2. If $a \neq a_i$, for all i, we define the extension $\overline{e}'$ of $\overline{e}$ by $\neg(i\ \overline{e}'\ n+1)$ for all i from 1 to n. If $a \in D_\phi$, then $v'_{n+1} = a$, otherwise $v'_{n+1} = \circ$.

In both cases, $\models F(\xi')(a_1, \ldots, a_{n+1})$.□

Lemma 4.10 Let ψ be a formula, with at most k free variables, using only constants in D_ϕ. Let ξ be an e-configuration of size k and $\models F(\xi)(a_1, \ldots, a_k)$ and $\models F(\xi)(a'_1, \ldots, a'_k)$, then $\models \psi(a_1, \ldots, a_k) \Leftrightarrow \models \psi(a'_1, \ldots, a'_k)$.

Proof: We show, by induction on the size of ψ, that the result holds for all e-configurations ξ and all a_i's and a'_i's.

1. ψ is $x_i = c$. Then c must be in D_ϕ, and therefore $F(\xi)$ must contain one of the conjuncts $(x_i = c)$ if $v_i = c$ or $(x_i = c')$ if $v_i = c', c \neq c'$ or $(x_i \neq c)$ if $v_i = \circ$. Since we assume that distinct constant symbols denote distinct elements we have: $a_i = c$ iff $a'_i = c$.

2. If ψ is of the form $\neg\psi_1$ or $\psi_1 \vee \psi_2$, the proof is straightforward.

3. If ψ is $(\exists x)\psi'$, then $(a_1, \ldots, a_k)$ satisfies ψ iff for some a, $(a_1, \ldots, a_k, a)$ satisfies ψ'. By Lemma 4.9, there is an e-configuration ξ' such that $(a_1, \ldots, a_k, a)$ satisfies $F(\xi')$. It is easy to see that ξ' must be an extension of ξ. Since $(a'_1, \ldots, a'_k)$ satisfies $F(\xi)$, by Lemma 4.7 there is an a' such that $(a'_1, \ldots, a'_k, a')$ satisfies $F(\xi')$. But then, by the induction hypothesis, $(a'_1, \ldots, a'_k, a')$ satisfies ψ', and therefore $(a'_1, \ldots, a'_k)$ satisfies ψ. □

Lemma 4.11 Let ξ be an e-configuration, and ψ a formula using only the constants in D_ϕ. Then $F(\xi) \rightarrow \psi$ is valid (i.e., true for all assignments to the free variables) iff $F(\xi) \wedge \psi$ is satisfiable (i.e., true for some assignment to the free variables).

Proof: If formula $F(\xi) \rightarrow \psi$ is valid, then Lemma 4.8 implies that $F(\xi) \wedge \psi$ is satisfiable. On the other hand, if $F(\xi) \wedge \psi$ is satisfiable, say by $(a_1, \ldots, a_n)$, and $F(\xi)$ is satisfied by $(a'_1, \ldots, a'_n)$, then Lemma 4.10 implies that ψ is satisfied by $(a'_1, \ldots, a'_n)$. □

In the same way as for rational order, we can define algorithms ALG and BALG. They are exactly as before but with e-configurations. For example, the algorithm BALG takes as input an e-configuration ξ and a formula ψ that uses only constants in D_ϕ. It returns 1 iff $F(\xi) \rightarrow \psi$ is valid. Only the base cases of BALG(ξ, ψ) are different from the rational order case:

1. ψ is an atomic formula $x_i = x_j$. BALG(ξ, ψ) returns 1 iff $i \,\overline{e}\, j$ in ξ.
2. ψ is an atomic formula $x_i = c$. BALG(ξ, ψ) returns 1 iff $c = v_i$ in ξ.

The proof of correctness and the complexity analysis proceed in an analogous way to the analysis of rational order. This completes the proof of Theorem 4.1. □

Finally, note that for equality constraints we can develop constraint logic programming machinery analogous to that of the previous sections.

5 Datalog and Boolean Constraints

In this section, we describe the addition of boolean constraints to Datalog. The idea is to use boolean operations as shorthands for manipulating various finite domains. From a practical point of view, boolean constraints can be added on top of the Datalog framework with rational order that we already examined in Section 3.2. We can strictly type the arguments of each database predicate. Each argument can range either over the rationals or over a finite domain. All of our results still hold in this combined framework.

We first give some basic definitions about boolean constraints and constraint solving or boolean unification. In Section 5.1, we describe the syntax and the semantics of the language, and present a canonical example. In Section 5.2, we use a lower bound on boolean unification to derive a lower bound on data complexity.

For boolean operators, we use $\wedge$ for *logical and*, $\vee$ for *logical or*, $\neg$ for *logical not*, and $\oplus$ for *exclusive-or*. In each boolean algebra B we always have two constants $0, 1$ in its vocabulary and at least two distinct elements zero and one: constant 0 is interpreted by the zero element and constant 1 by the one element. The special boolean algebra over two elements represented by 0, 1 and with no other constants is called **2**.

Here, we use a *boolean algebra* B, that can have m other constants besides 0 and 1 in its vocabulary. We use boolean algebra terms s, t, i.e., terms built using the operators, the constants 0, 1, the m other constants, and variables. Solving the equation $s =_B t$ means finding an assignment of elements of B to the variables, so that after this assignment s and t denote the same element of the boolean algebra B. For example, solving equations in **2** is propositional satisfiability, and is therefore NP-complete. A useful fact about solving equations in a boolean algebra B is Boole's Lemma (see [32]):

> Equation $t(x_1, x_2, \ldots, x_n) =_B 0$ has a solution in boolean algebra B iff equation $t(1, x_2, \ldots, x_n) \wedge t(0, x_2, \ldots, x_n) =_B 0$ has a solution in B iff $\bigwedge_{\overline{b} \in \{0,1\}^n} t(\overline{b}) =_B 0$.

(The correctness of this test is trivial if B happens to be **2**; otherwise the product may well be 0 even if none of its factors are.)

In *boolean unification* ([32]), we are given two boolean terms $s = s(x_1, \ldots, x_n)$ and $t = t(x_1, \ldots, x_n)$. We assume that each term is built from the operators $\oplus$ *exclusive-or*, $\wedge$ *logical and*, from constants $0, 1$ from at most m other constants, and from variables $x_1, \ldots, x_n$. We want to know if we can substitute for the variables new boolean terms, such that after the substitution s and t can be rewritten into each other using the boolean equational axioms right below.

The restriction to operators $\oplus$ and $\wedge$ is without loss of generality. Using these operators only we work in *the boolean ring that corresponds to the boolean algebra.* We thus use $\neg a$ for $1 \oplus a$ and $a \vee b$ for $a \oplus b \oplus (a \wedge b)$. One possible presentation of the equational axioms is as follows. For arbitrary elements a, b, c:

$$\begin{array}{rclrcl} a \oplus b & = & b \oplus a & a \wedge b & = & b \wedge a \\ (a \oplus b) \oplus c & = & a \oplus (b \oplus c) & (a \wedge b) \wedge c & = & a \wedge (b \wedge c) \\ a \oplus (b \wedge c) & = & (a \oplus b) \wedge (a \oplus c) & a \wedge (b \oplus c) & = & a \wedge b \oplus a \wedge c \\ a \oplus 0 & = & a & a \wedge 1 & = & a \\ a \oplus a & = & 0 & a \wedge a & = & a \end{array}$$

An equivalent definition of *boolean unification* is: s and t are unifiable if there is a substitution that makes them equal in B_m, the free boolean algebra with m generators (these are denoted by the m constants besides $0, 1$). Recall that the free boolean algebra B_m has 2^{2^m} elements and B_0 is **2**. Thus, boolean unification is solving equations in free boolean algebra B_m. For m unbounded its complexity can be high. We shall show that: deciding whether two boolean expressions are unifiable is Π_2^p–complete.

5.1 Bottom-up Evaluation and Boolean Constraints

We only describe the language for the boolean constraint part. This can easily be combined with the language for rational order constraints. For the rest of the paper we assume $B = B_m$.

Syntax: (1) The *facts* have the form, $R_0(\overline{x}) :\!-\ \psi_0(\overline{x}) =_B 0$.
(2) The *rules* have the form: $R_0(\overline{x}) :\!-\ R_1(\overline{x}, \overline{y}), \ldots, R_k(\overline{x}, \overline{y}), \psi_{k+1}(\overline{x}, \overline{y}) =_B 0$.
For simplicity of presentation, we use the convention that the various occurrences of $\overline{x}$ and $\overline{y}$ are possibly different vectors of variables from the set $\{x_1, \ldots, x_k, y_1, \ldots, y_l\}$, and $\psi_0(\overline{x}) =_B 0$ and $\psi_{k+1}(\overline{x}, \overline{y}) =_B 0$ are boolean constraints, i.e., equations over those variables.

Semantics: A set of rules and facts defines, in the standard fashion, a monotone mapping from relations whose elements are from B to relations whose elements are from B. The semantics of this set of facts and rules is the least fixpoint of this mapping.

Remark: We have used only a single constraint in facts and rules. This is because, by rewriting from the equational axioms one can show the equivalences: (1) $a =_B b$ iff $a \oplus b =_B 0$, and

(2) $a =_B 0$ and $b =_B 0$ iff $a \vee b =_B 0$ iff $a \oplus b \oplus (a \wedge b) =_B 0$. Using these properties many constraints can be equivalently replaced by one.

Bottom-up Evaluation: We describe the *firing* of one rule. Assume that we have either given EDB or derived IDB facts $R_i(\overline{x}, \overline{y}) :\!-\ \psi_i(\overline{x}, \overline{y}) =_B 0$ $1 \leq i \leq k$, and that we have the rule $R_0(\overline{x}) :\!-\ R_1(\overline{x}, \overline{y}), \ldots, R_k(\overline{x}, \overline{y}), \psi_{k+1}(\overline{x}, \overline{y}) =_B 0$. We produce a quantified description of the set of tuples from B we will derive.

$$R' = \{R(\overline{x}) : \exists \overline{y}\, \psi_1(\overline{x}, \overline{y}) =_B 0, \ldots, \psi_k(\overline{x}, \overline{y}) =_B 0, \psi_{k+1}(\overline{x}, \overline{y}) =_B 0\}$$

From the above remark, it follows that R' can be represented as a set of facts that satisfy *one* equation, except for the quantified variables on the right side. These extra variables can be eliminated by using repeatedly Boole's Lemma that $\exists y\, \psi(y, \overline{x}) =_B 0$ iff $\psi(0, \overline{x}) \wedge \psi(1, \overline{x}) =_B 0$. This allows us to add R' in unquantified single equation form to the derived facts. (We use Boole's Lemma to eliminate quantifiers, one could also use here some other boolean unification algorithm).

Soundness and Completeness: The bottom-up evaluation consists of repeatedly firing rules starting from the input (EDB) facts and rules. Since the transformation into a single constraint and the quantifier elimination preserve the set of solutions, this bottom-up evaluation does implement the intended semantics.

Termination: As shown in the following theorem, this bottom-up evaluation can be made to terminate after a finite set of firings, by using the property that terms can be rewritten (via the axioms) into equivalent conjunctive normal forms.

Theorem 5.1 Datalog with boolean constraints over B_m can be evaluated bottom-up in closed form. If m is a fixed integer, then this evaluation can be done in PTIME data complexity.

Proof: We can evaluate the query by an iterative method in which we add all new facts that can be derived from the already known facts and rules. From the derived facts added we eliminate quantified variables on the right side. Furthermore we always keep every fact in conjunctive normal form. Since the arity of each relation is fixed (max v), and since the evaluation does not introduce any new constants and the number of constants (m) is also given, every relation R can be represented by at most $2^{2^{v+m}}$ facts (by counting only normal forms). This means that every new fact can be compared syntactically with facts known and added if it is not already there. Firings continue until no more facts are added. This must terminate because there are only a finite number of facts that can be added. □

We present now the canonical "adder circuit" example of Buttner and Simonis [6], but in this case we use bottom-up evaluation.

Example 5.2 An adder circuit can be built from two half-adder circuits. We define the operation of the half-adder by a simple database fact using boolean constraints over **2**.

$$\mathit{Halfadder}(x, y, z, w) :\!-\ x \oplus y =_B z, x \wedge y =_B w$$

where x and y are input variables, z is the sum and w is the carry.

The application of two simple rewrite rules, namely (1) $a =_B b$ iff $a \oplus b =_B 0$, and (2) $a =_B 0$ and $b =_B 0$ iff $a \vee b =_B 0$ yields:

$$\mathit{Halfadder}(x, y, z, w) :\!\!- (x \oplus y \oplus z) \vee ((x \wedge y) \oplus w) =_B 0$$

The adder circuit can be described by the use of two half-adder circuits and an extra constraint, where x and y are input variables, c is carry in, s is sum, and d is carry out.

$$\mathit{Adder}(x, y, c, s, d) :\!\!- \mathit{Halfadder}(x, y, s_1, c_1), \mathit{Halfadder}(s_1, c, s, c_2), d =_B c_1 \vee c_2$$

When using these definitions as a database query, the evaluation proceeds bottom-up, substituting the constraints of the halfadder into the rule for the adder. That yields:

$$\mathit{Adder}(x, y, c, s, d) :\!\!- (c_1 \vee c_2) \oplus d =_B 0, (x \oplus y \oplus s_1) \vee ((x \wedge y) \oplus c_1) =_B 0,$$
$$(s_1 \oplus c \oplus s) \vee ((s_1 \wedge c) \oplus c_2) =_B 0$$

By the rewrite rules (1) and (2) above and using Boole's lemma that "$\exists y\, \psi(y, \overline{x}) =_B 0$ iff $\psi(0, \overline{x}) \wedge \psi(1, \overline{x}) =_B 0$" to eliminate s_1, c_1, c_2 we can transform the above into:

$$\mathit{Adder}(x, y, c, s, d) :\!\!- (x \oplus y \oplus c \oplus s) \vee ((x \wedge y) \oplus (x \wedge c) \oplus (y \wedge c) \oplus d) =_B 0$$

Then by boolean unification methods we get expressions for any subset of the variables in the constraint (say s and d) in terms of the remaining variables (viewed as constants $\mathcal{X}, \mathcal{Y}, \mathcal{C}$). For example, one solution would be $s = \mathcal{X} \oplus \mathcal{Y} \oplus \mathcal{C}$ and $d = (\mathcal{X} \wedge \mathcal{Y}) \oplus (\mathcal{X} \wedge \mathcal{C}) \oplus (\mathcal{Y} \wedge \mathcal{C})$. □

5.2 Data Complexity and Boolean Unification

We assume in this subsection that the number of boolean constants m is unbounded.

Theorem 5.3 Deciding whether two boolean expressions are unifiable is Π_2^p–complete.

Proof: The proof is based on combining Boole's method of variable elimination with Martin and Nipkow's method of constant elimination, as follows.

First, we show that any instance of boolean unification can be put into the normal form $s = \psi(x_1, \ldots, x_n)$ and $t = 0$, by using the property that $a =_B b$ iff $a \oplus b =_B 0$. For any given boolean algebra B, by Boole's method $\psi(\overline{x}) =_B 0$ has a solution iff $\bigwedge_{\overline{b} \in \{0,1\}^n} \psi(\overline{b}) =_B 0$ has a solution. (Note that $\bigwedge_{\overline{b} \in \{0,1\}^n} \psi(\overline{b})$ might contain constants other than 0 and 1).

Martin and Nipkow's method translates the original unification problem in the algebra B_m with m constants, into 2^m independent unification problems in the algebra **2**, by replacing the *constants* by 0's and 1's in all possible ways. The original problem has a solution in the algebra B_m iff each of the 2^m new problems has a solution in the algebra **2**.

This method can be applied using only the constants $\mathcal{Y}$ that appear in $\psi(\overline{x}, \overline{\mathcal{Y}})$. Suppose that there are $k \leq m$ such constants. A unification instance $s = \psi(\overline{x}, \overline{\mathcal{Y}}), t = 0$ with variables $\overline{x}$

and constants $\overline{\mathcal{Y}}$ has a solution iff $\forall_{\overline{a}\in\{0,1\}^k}$ the equation $\psi(\overline{x},\overline{a}) =_{\mathbf{2}} 0$ has a solution by Martin and Nipkow's method. By Boole's method this implies that $\forall_{\overline{a}\in\{0,1\}^k}\bigwedge_{\overline{b}\in\{0,1\}^n}\psi(\overline{b},\overline{a}) =_{\mathbf{2}} 0$ holds. Finally, note that a product of boolean expressions in $\mathbf{2}$ equals 0 iff one of its terms equals 0. Therefore the original problem has a solution iff $\forall_{\overline{a}\in\{0,1\}^k}\exists_{\overline{b}\in\{0,1\}^n}\psi(\overline{b},\overline{a}) =_{\mathbf{2}} 0$ holds. This problem is the quantified boolean formula problem with one alternation of quantifiers, which is Π_2^p–complete [15]. □

Theorem 5.3 assumes that the number of variables in the terms unified can be arbitrarily large. However, in any fixed Datalog program the arity of each predicate symbol is fixed. Therefore, the result of Theorem 5.3 does not imply directly any data complexity lower bound on Datalog with boolean constraints.

Theorem 5.4 There is a yes/no query expressible in Datalog with boolean constraints (over a boolean algebra B_m) whose evaluation is Π_2^p–hard.

Proof: Let $s = \psi(x_1,\ldots,x_n)$ and $t = 0$ be any instance of boolean unification in the boolean algebra B_c with constants $0, 1, \mathcal{A}_1,\ldots,\mathcal{A}_c$. By the previous theorem this is a Π_2^p–hard problem (NP-hard if $c = 0$).

We shall produce a yes/no query expressed in Datalog with boolean constraints, over boolean algebra B_m, such that its answer is $\mathcal{YES}$ iff s, t have a unifier. In this reduction $m = O(|\psi|)$. (If $c = 0$ this is an NP-hardness reduction).

Our construction proceeds in a number of steps:

(1) We build a "tree" circuit for $\psi(\overline{x})$ using some number of gates $\mathcal{G}_1,\ldots,\mathcal{G}_{top}$ with $\mathcal{G}_{top}$ as the output gate of the whole circuit. For each x_i we create a new constant $\mathcal{B}_i$ and substitute the variables by these constants while creating the circuit.

Suppose that formula $\psi(\overline{x})$ has a total of l occurences of constants and variables in it. We enter for each occurrence of $\mathcal{A}_i$ and for each occurrence of x_j the values $\mathcal{A}_i$ and $\mathcal{B}_j$ via the distinct gates $\mathcal{G}_1,\ldots,\mathcal{G}_l$. We create l database facts as follows. If the k^{th} occurrence of a constant (or variable) in the formula is $\mathcal{A}_i$ (or x_j), we create the database fact:

$$Value(\mathcal{G}_k, \mathcal{A}_i)$$

or

$$Value(\mathcal{G}_k, \mathcal{B}_j)$$

Next we look at the parse tree of the formula ψ. We tag each boolean operator $\oplus$ and $\wedge$ in the parse tree by a new distinct gate. We tag the operator at the root by the gate $\mathcal{G}_{top}$. Then for every i, j, k if an $\oplus$ (or $\wedge$) operator is tagged by gate $\mathcal{G}_i$ and has as left child an operator tagged by gate $\mathcal{G}_j$ and has as right child an operator tagged by gate $\mathcal{G}_k$, we create the database fact:

$$Xorgate(\mathcal{G}_i, \mathcal{G}_j, \mathcal{G}_k)$$

or

$$Andgate(\mathcal{G}_i, \mathcal{G}_j, \mathcal{G}_k)$$

The expression corresponding to $\psi(\overline{x})$ is built bottom-up by the evaluation method using the following rules:

$$Value(k,z) :\!\!- Xorgate(k,i,j), Value(i,x), Value(j,y), z =_B x \oplus y$$

$$Value(k,z) :\!\!- Andgate(k,i,j), Value(i,x), Value(j,y), z =_B x \wedge y$$

The bottom-up evaluation derives as the value of the topmost gate $\mathcal{G}_{top}$ some ψ' such that $\psi' =_B \psi(\overline{\mathcal{B}})$. As the rules are evaluated variables i, j, x, y are eliminated but constants $\mathcal{G}, \mathcal{A}, \mathcal{B}$ remain. The result is therefore:

$$Value(\mathcal{G}_{top}, \psi') :\!\!- \psi' =_B \psi(\overline{\mathcal{B}})$$

Remark: Although the $\mathcal{B}$'s within ψ' are constants from the boolean algebra B, the Datalog query program that we write below will treat them as if they were still variables. If a unification is possible between s and t, then there is a particular solution, that is, there is a substitution for the variables, i.e. the $\mathcal{B}$'s, that uses only constant expressions, i.e. expressions over the $\mathcal{A}$'s.

(2) We create two unary database relations $A(x)$ and $B(x)$ to store all $\mathcal{A}$'s and $\mathcal{B}$'s respectively. We also create a unary relation *Aexpr(e)*, which when evaluated bottom-up will contain in it all the expressions of the algebra which has only $\mathcal{A}$'s and the boolean operators $\oplus$ and $\wedge$ in them.

$$Aexpr(e) :\!\!- A(e)$$

$$Aexpr(e) :\!\!- Aexpr(e_1), Aexpr(e_2), e =_B e_1 \wedge e_2$$

$$Aexpr(e) :\!\!- Aexpr(e_1), Aexpr(e_2), e =_B e_1 \oplus e_2$$

We use the terminating form of our bottom-up evaluation, i.e., we keep terms in conjunctive normal form. Note that if $c = 0$ we can just take $0, 1$ as the contents of *Aexpr(e)*.

(3) Our main relation for replacement is $Replace(a, \mathcal{B}, c, d)$, whose intended meaning is that if in expression a all occurrences of constant $\mathcal{B}$ are substituted by an expression c which has only $\mathcal{A}$'s and the boolean operators $\oplus$ and $\wedge$ in it, then we get the expression d. The actual bottom-up replacement can be done recursively as follows:

$$Replace(b,b,c,c) :\!\!- B(b), Aexpr(c)$$

$$Replace(a,b,c,a) :\!\!- A(a), B(b)$$

$$Replace(0,b,c,0) :\!\!- B(b)$$

$$Replace(1,b,c,1) :\!\!- B(b)$$

$$Replace(o,x,y,n) :\!\!- Replace(o_1,x,y,n_1), Replace(o_2,x,y,n_2), o =_B o_1 \wedge o_2, n =_B n_1 \wedge n_2$$

$$Replace(o,x,y,n) :\!\!- Replace(o_1,x,y,n_1), Replace(o_2,x,y,n_2), o =_B o_1 \oplus o_2, n =_B n_1 \oplus n_2$$

(4) To make the substitutions in sequence for each $\mathcal{B}_i$, we implicitly order the $\mathcal{B}$ constants using database facts:

$$Next(0, \mathcal{B}_1)$$

$$Next(\mathcal{B}_i, \mathcal{B}_{i+1})$$

$$Last(\mathcal{B}_n)$$

Now we are ready to put everything together:

$$F(expr, 0) :\!-\ Value(\mathcal{G}_{top}, expr)$$

$$F(new, j) :\!-\ F(old, i), Next(i, j), Replace(old, j, y, new), Aexpr(y)$$

$$Final(expr) :\!-\ Last(k), F(expr, k)$$

At this point, when the last $\mathcal{B}$ constant is replaced, there are only $\mathcal{A}$ constants present within *expr*. Finally, we have the forced unification by:

$$Output(\mathcal{YES}) :\!-\ Final(0).$$

The rule will fire only when $expr =_B 0$. Since all possible substitutions of expressions containing only $\mathcal{A}$'s for the $\mathcal{B}$'s are tried by the bottom-up evaluation, there is any output iff there is a unifier iff s and t are unifiable in algebra B_c.

This completes the reduction. It is clearly a linear-time reduction, which proves the desired data complexity bound. □

6 Conclusions and Open Questions

Constraint logic programming paradigms are currently attracting a great deal of attention in many areas of computer science, e.g., languages for operations research applications [43, 44] or concurrent languages [38]. In this paper, we have presented many examples of "declarative and efficiently evaluable" constraint *database query* languages. Our framework is based on a study of quantifier elimination procedures combined with a use of data complexity.

A number of interesting problems remain open. (1) What is the complexity of tableau containment with inequalities? (2) What is the precise data complexity of Datalog with rational order and boolean constraints? (3) Is it possible to develop the framework for any arbitrary infinite discrete order with constants? Note that progress has been made recently on this question in [36]. Discrete order can be used to model temporal databases. For recent developments of constraint-based approaches to temporal databases we refer to [9, 21]. (4) How do various optimization methods combine with our framework? This would involve extending [35] or the more recent [33].

It would be very interesting to study the implementation of the "declarative and efficiently evaluable" languages outlined in this paper. The results presented here should be properly viewed as positive arguments for the feasibility of such an effort. However, some critical research

questions remain. (1) Although they do not appear as part of the relational model, many physical access structures (using indexes, B-trees, hashing etc) are critical in the implementation of relational databases. Are there analogous simple access structures in this more general setting? What are these access structures for processing r-configurations? (2) The technology of algorithms for logical theories is still rather complex. Are there special cases, for which simple algorithmic techniques can be implemented? These would be analogous to the special treatment of project-select-join query programs in the relational model. (3) CQL's should be designed in an extendible way. For example, this would make it possible to integrate a select set of computational geometry algorithms as primitives in a bottom-up evaluation. (4) CQL's should be designed with helpful features, such as high-order variables and complex database object types.

References

[1] S. Abiteboul, V. Vianu. Procedural and Declarative Database Update Languages. *Proc. 7th ACM PODS*, 240–250, 1988.

[2] A.V. Aho, Y. Sagiv, J.D. Ullman. Equivalences among Relational Expressions. *SIAM J. of Computing*, 8:2:218–246, 1979.

[3] A.K. Aylamazyan, M.M. Gilula, A.P. Stolboushkin, G.F. Schwartz. Reduction of the Relational Model with Infinite Domain to the Case of Finite Domains. *Proc. USSR Acad. of Science (Doklady)*, 286(2):308–311, 1986.

[4] M. Ben-Or, D. Kozen, J. Reif. The Complexity of Elementary Algebra and Geometry. *JCSS*, 32:251–264, 1986.

[5] A.H. Borning. The Programming Language Aspects of ThingLab, A Constraint-Oriented Simulation Laboratory. *ACM TOPLAS* 3:4:353–387, 1981.

[6] W. Buttner, H. Simonis. Embedding Boolean Expressions into Logic Programming. *Journal of Symbolic Computation*, 4:191–205, 1987.

[7] A.K. Chandra, D. Harel. Computable Queries for Relational Data Bases. *JCSS*, 21:156–178, 1980.

[8] A.K. Chandra, P.M. Merlin. Optimal Implementation of Conjunctive Queries in Relational Databases. *Proc. ACM STOC*, 77–90, 1976.

[9] J. Chomicki. Polynomial Time Query Processing in Temporal Deductive Databases. *Proc. 9th ACM PODS*, 379–391, 1990.

[10] J. Chomicki, T. Imielinski. Relational Specifications of Infinite Query Answers. *Proc. ACM SIGMOD*, 174–183, 1989.

[11] E.F. Codd. A Relational Model for Large Shared Data Banks. *CACM*, 13:6:377–387, 1970.

[12] A. Colmerauer. *An Introduction to Prolog III.* Draft, 1987.

[13] M. Dincbas, et al. The Constraint Logic Programming Language CHIP. *Proc. Fifth Generation Computer Systems*, Tokyo Japan, 1988.

[14] J. Ferrante, J.R. Geiser. An Efficient Decision Procedure for the Theory of Rational Order. *Theoretical Computer Science*, 4:227–233, 1977.

[15] M.R. Garey, D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP–completeness.* Freeman, 1979.

[16] Y. Gurevich, S. Shelah. Fixed-Point Extensions of First-Order Logic. *Annals of Pure and Applied Logic*, 32, 265–280, 1986.

[17] M.R. Hansen, B.S. Hansen, P. Lucas, P. van Emde Boas. Integrating Relational Databases and Constraint Languages. *Computer Languages*, 14:2:63–82, 1989.

[18] R. Hull, J. Su. Domain Independence and the Relational Calculus. *Technical Report* 88–64, University of Southern California, submitted to IPL, 1989.

[19] N. Immerman. Relational Queries Computable in Polynomial Time. *Information and Control*, 68:86-104, 1986.

[20] J. Jaffar, J.L. Lassez. Constraint Logic Programming. *Proc. 14th ACM POPL*, 111–119, 1987.

[21] F. Kabanza, J-M. Stevenne, P. Wolper. Handling Infinite Temporal Data. *Proc. 9th ACM PODS*, 392–403, 1990.

[22] P.C. Kanellakis. Elements of Relational Database Theory. *Handbook of Theoretical Computer Science*, Vol. B, chapter 17, (J. van Leeuwen, A.R. Meyer, N. Nivat, M.S. Paterson, D. Perrin editors), North-Holland, 1990.

[23] P. C. Kanellakis, G. M. Kuper, P. Z. Revesz. Constraint Query Languages. *Proc. 9th ACM PODS*, 299–313, 1990.

[24] M. Kifer. On Safety, Domain Independence, and Capturability of Database Queries. *Proc. International Conference on Databases and Knowledge Bases*, Jerusalem Israel, 1988.

[25] A. Klug. On Conjunctive Queries Containing Inequalities. *JACM*, 35:1:146–160, 1988.

[26] P. Kolaitis, C.H. Papadimitriou. Why not Negation by Fixpoint? *Proc. 7th ACM PODS*, 231–239, 1988.

[27] D. Kozen. Complexity of Boolean Algebras. *Theo. Comp. Sci.*, 10, 221-247, 1980.

[28] D. Kozen, C. Yap. Algebraic Cell Decomposition in NC. *Proc. 26th IEEE FOCS*, 515–521, 1985.

[29] W. Leler. *Constraint Programming Languages.* Addison Wesley, 1987.

[30] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1984.

[31] Y.N. Moschovakis. *Elementary Induction on Abstract Structures.* North Holland, 1974.

[32] U. Martin, T. Nipkow. Unification in Boolean Rings. *Journal of Automated Reasoning*, 4:381-396, 1988.

[33] I. N. Mumick, S. J. Finkelstein, H. Pirahesh, R. Ramakrishnan. Magic Conditions. *Proc. 9th ACM PODS*, 314–330, 1990.

[34] F.P. Preparata, M.I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, 1985.

[35] R. Ramakrishnan. Magic Templates: A Spellbinding Approach to Logic Programs. *Proc. 5th International Conference on Logic Programming*, 141–159, 1988.

[36] P.Z. Revesz. A Closed Form for Datalog Queries with Integer Order. *Proc. 3rd International Conference on Database Theory*, 1990.

[37] H.L. Royden. *Real Analysis.* 2^{nd} Ed., 1983.

[38] V.A. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie Mellon University, 1989.

[39] G.L. Steele. The Definition and Implementation of a Computer Programming Language Based on Constraints. Ph.D. thesis, MIT, AI-TR 595, 1980.

[40] A. Tarski. *A Decision Method for Elementary Algebra and Geometry.* University of California Press, Berkeley, California, 1951.

[41] J.D. Ullman. *Principles of Database Systems.* Computer Science Press, 2^{nd} Ed., 1982.

[42] J.D. Ullman, A. Van Gelder. Parallel Complexity of Logical Query Programs. *Algorithmica*, 3:5-42, 1988.

[43] P. Van Hentenryck. A Logic Language for Combinatorial Optimization. *Annals of Operations Research*, 21, 247–274, 1989.

[44] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

[45] M.Y. Vardi. The Complexity of Relational Query Languages. *Proc. 14th ACM STOC*, 137–146, 1982.