CONSTRAINT QUERY LANGUAGES

(extended abstract)

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

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 [B81, S80] (for a survey see [Le87]). A major recent development in logic programming systems is the integration of logic and constraint paradigms, e.g., in CLP [JL87], in Prolog III [C87], and in CHIP [D88]. 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 unification at the forefront of this search is a feasible form of special constraint solving. Additional constraint solving increases the depth of the search.

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. So, 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 data [ChI89] and for magic set evaluation [R88], 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* [CH80, V82] 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 examples and a more detailed discussion. Manipulation of geometric data is an important application area (e.g., geographic databases) that requires both relational query language techniques and arithmetic calculations. Note that indexes for range searching and modeling of complex structures still do not bridge the gap between declarative accessing of large volumes of spatial data and performing common computational geometry tasks. The arithmetic calculations have to be given first-class citizen status.

*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.

What could be sound criteria for achieving integration of data processing and numerical computations? 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 we understand. These criteria are satisfied by the *Constraint Query Language* (CQL) design principles that are outlined below and illustrated in Figure 1.

- *A generalized tuple is a conjunction of constraints.* In the relational database model $R(3,4)$, or $R(x,y)$ with $x=3$ and $y=4$ is a tuple of arity 2. 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 $x+y=2.5$. So a generalized tuple of arity k is a finite representation of a possibly infinite set of k-dimensional points.
- *A generalized relation is a finite set of generalized tuples.* It is a disjunction of conjunctions (DNF) of constraints using at most k variables, where k is the arity of the relation. In the relational model, relations are finite structures, and queries are supposed to preserve this finiteness. This is a requirement that creates various safety problems. In our framework, it is finiteness of representation that must be preserved.
- *The syntax and semantics of a CQL are the union of an existing database query language and a decidable logical theory.* For example: Relational calculus [Co70] + the theory of real closed fields [T51] (Section 2); Inflationary Datalog$^\neg$ [AV88, KP88] + 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).
- *The queries must be evaluable bottom-up and in closed form.* This avoids problems of safety and is good for optimization. The idea is that database atoms are shorthands for formulas in the decidable theory. With some abuse of notation $R(t)$ means that $t=t_1 \vee \cdots \vee t=t_n$ where t_1, $\ldots$, t_n are generalized tuples. Evaluation of a query corresponds to an instance of the decision problem, that is solved bottom-up and using quantifier elimination.
- *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 [GJ79]).

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. In Section 3, a *rational order constraint* is of the form $u=v$ or $u<v$, where u and v are variables or constants. A nontrivial property of these constraints is the density of rational order. 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, but no order. In Section 5, a *boolean constraint* is of the form $s =_B t$, where s,t are terms constructed using variables, constants, and operations of a boolean algebra B ([MN88, BS87]).

Let us now outline our results. In our exposition, we assume some familiarity with the elements of database theory [U82]. From Codd's original work [Co70] it follows that: *safe relational calculus can be evaluated bottom-up in closed form and LOGSPACE data complexity.* 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.

In Section 2 we use [T51, BKR86] to show that: 1. Relational calculus with polynomial constraints can be evaluated bottom-up in closed form with NC data complexity. In Section 3, adapting the proof

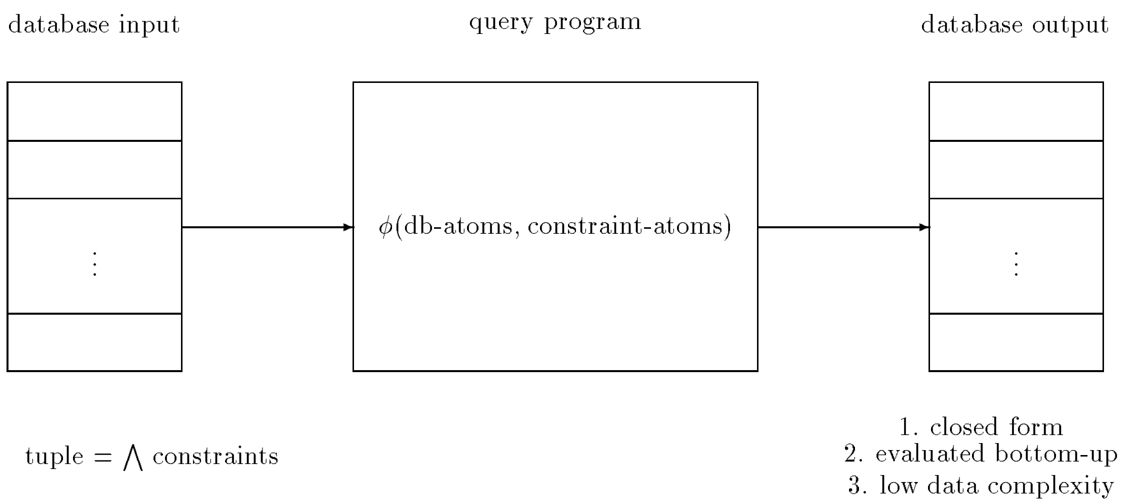

Figure 1: A database framework for CQL

of [FG77], we show that: 2. Relational calculus with order constraints can be evaluated bottom-up in closed form and LOGSPACE data complexity, and 3. Inflationary Datalog$^{\neg}$ with order constraints can be evaluated bottom-up in closed form and PTIME data complexity. For Datalog with rational order constraints, we develop a bottom-up evaluation method that is closer to the classical foundations of logic programming [L84] and knowledge bases [UvG88]. 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. In Section 4, we show that: 5. Relational calculus with equality constraints can be evaluated bottom-up in closed form and LOGSPACE data complexity, and 6. Inflationary Datalog$^{\neg}$ with equality constraints can be evaluated bottom-up in closed form and PTIME data complexity. These results extend some of the work on safety in [AGSS86, Ki88, HS89, R88]. In Section 5, we show that: 7. Datalog with order and boolean constraints can be evaluated bottom-up in closed form. The data complexity here depends on the use of constants in the boolean constraints. In order to analyze this data complexity we show that: deciding whether two boolean expressions are unifiable is Π_2^p–complete.

Finally in Sections 2.2 and 2.3, as part of our analysis of the relational calculus and arithmetic constraints we investigate the use of high-dimension variables and the homomorphism theorem for tableau query containment [CM76, ASU79, Kl88]. We show that the containment of tableau queries with linear equalities is NP-complete, but that with quadratic equalities it is Π_2^p–complete.

2 Relational Calculus and Polynomial Constraints

2.1 Data Complexity and Computational Geometry

In this subsection, we argue (using recent developments from the theory of algorithms) 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* [CH80, V82].

Consider an instance of a generalized database relation as 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. We can regard each 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 (multidimensional) points. However, relations are finite sets of points, whereas generalized relations are possibly infinite sets of points that have some finite representation.

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 [T51]. From an expressibility point of view the database atomic formulas will be used as shorthands for large formulas of the theory of real closed fields. 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 dominates the query size by many orders of magnitude, is the rationale of data complexity.

To illustrate this discussion, we show how three basic operations of computational geometry, i.e., line intersection, convex hull, and Voronoi diagram (see [PS85]) can be described in a "declarative and efficiently bottom-up evaluable" query language. In the following examples n is the only parameter that grows asymptotically and $R(\ldots)$ are the only "database accesses".

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 $\mathrm{On}(x, y, x_1, y_1, x_2, y_2)$ 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:

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

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 $\mathrm{Intriangle}(x, y, x_1, y_1, x_2, y_2, x_3, y_3)$ 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 $\mathrm{Intriangle}(x, y, x_1, y_1, x_2, y_2, x_3, y_3)$.

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 [PS85]. 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 [PS85]. 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 [T51]. 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 much more 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 [BKR86] to show that the data complexity is in NC. Furthermore, the [BKR86] 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 [BKR86] 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 [BKR86].

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

2.2 Query Program 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 optimization. A key problem for optimization is testing containment and equivalence of queries.

In this subsection, we move from data complexity to optimization of conjunctive queries. Conjunctive queries form the most common subset of relational calculus queries. We augment them using special polynomial constraints such as linear equations, quadratic equations, and inequalities without $+, *$. We present an extension of the homomorphism technique of [CM76, ASU79]. For the terminology see [ASU79].

Theorem 2.5 Containment of tagged untyped tableau queries with a conjunction of linear equation constraints is NP-complete. □

NP-hardness is immediate, since it is NP-complete to determine containment for such queries with equalities but without $+$ or $*$. The new part is showing membership in NP. We show that for two queries Q_1 and Q_2 in normal form: Q_1 is contained in Q_2 iff there is a homomorphism mapping Q_2 into Q_1.

Let us explain *normal forms*, *symbol mappings*, and *homomorphisms*. We break up each query Q 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 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, by mapping every constant to itself). We shall call such an h a homomorphism from Q_2 to Q_1 if it also has the property that whenever constraints C_1 are satisfied so are $h(C_2)$.

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 aren any database states. With standard database theory techniques one can prove that:

Lemma 2.6 Let $Q_1 = (T_1, C_1)$ and $Q_2 = (T_2, C_2)$ be two 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, $Q_1(d) \subseteq Q_2(d)$ iff C_1 implies $h_1(C_2) \vee \cdots \vee h_m(C_2)$. $\square$

From [R83], p. 139, we have 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, the above general lemma and geometric fact suffice to show the homomorphism theorem. 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 $C_1(x)$. Therefore one of the symbol mappings must be a homomorphism from Q_2 to Q_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 2.5 and generalizes the homomorphism theorem to linear equational calculations.

In contrast, for quadratic equational calculations we can show that the homomorphism technique does not work. Furthermore:

Theorem 2.7 Testing containment of tagged untyped tableau queries with a conjunction of quadratic equation constraints is Π_2^p-complete. $\square$

Containment in the presence of inequality constraints without $+$ and $*$ is the problem examined in [Kl88]. 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. As a special case, [Kl88] shows that the homomorphism technique works for left-semiinterval queries alone or right-semiinterval queries alone.

2.3 High-Dimension Variables

We have already discussed data complexity and optimization. In this subsection, we introduce a "programming language type construct" that is outside the traditional constraint logic programming and decision procedure frameworks.

By modelling 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

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

In this example we use the convention that p_i is a variable of dimension 0 ranging over points, and l_i is 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. In turns out that both languages with or without these variables have the same expressive power[1]. 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 $(\forall z)R(x_1 + zx_2, y_1 + zy_2)$, and the formula $(x, y) \subseteq l$ becomes $(\exists z)(x_1 + zx_2 = x \wedge y_1 + zy_2 = y)$.

Thus, we believe that: *high-dimension variables are an interesting shorthand that both facilitates writing natural programs and provides clues, e.g., linear constraints, for simpler evaluation strategies.*

3 Datalog$^{\neg}$ and Rational Order Constraints

As shown in [I86] and [V82] the fixpoint queries with a finite discrete order express exactly PTIME. It follows from the proofs of this fact that inflationary Datalog$^{\neg}$ ([AV88, KP88]) with a finite discrete order expresses exactly PTIME. It is easy to modify these simulation proofs (by making all programs use only constants appearing in the database) to show that inflationary Datalog$^{\neg}$ with rational order expresses PTIME. Given this observation, it seems interesting to consider an extended setting with generalized databases and inequality constraints, but no $+$ or $*$.

3.1 Data Complexity and PTIME

In this subsection, 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:

[1]This was pointed out to us by Allen Van Gelder.

Theorem 3.1 (1) Relational calculus with rational order constraints can be evaluated bottom-up in closed form and LOGSPACE data complexity. (2) Inflationary Datalog$^\neg$ with rational order constraints can be evaluated bottom-up in closed form and PTIME data complexity. □

Neither part of the above theorem follows from the analysis of polynomial constraints. The data complexity in (1) is tighter than what we already have using polynomial constraints. Adding recursion as in (2) to polynomial constraints (rather than to rational order) easily gives us a Turing machine capability. Note that closure of evaluation implies that safety is no longer a critical issue.

We show (1) by extending the proof of [FG77] to languages with constants and adapting it to data complexity. The basic technical contribution is the appropriate definition of r-configuration. Part (2) follows from (1) and the fact that there can only be a polynomial number of r-configurations. In the rest of this subsection we outline the proof.

In order to use [FG77], 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. (Recall that with some abuse of notation $R(t)$ means that $t = t_1 \vee \cdots \vee t = t_n$, where $t_1, \ldots, t_n$ are the generalized database relation tuples). In formula ϕ we can, within the required resource bounds, eliminate all occurrences of $=$ and just use $<$. This is for minimizing the case analysis in the proof. The basic idea now is to perform quantifier elimination on ϕ using r-configurations.

Definition 3.2 Let ϕ be the formula we get from applying a query of relational calculus with rational order constraints to a generalized database. Let C_ϕ be the set of constants that appear in ϕ. An *r-configuration* $C = (\mathbf{f}, \mathbf{c}, \mathbf{d})$ of size k consists of a sequence $\mathbf{f} = (f_1, \ldots, f_k)$, where $\{f_1, \ldots, f_k\} = \{1, \ldots, l\}$, for some $l \leq k$, and two sequences $\mathbf{c} = (c_1, \ldots c_k)$ and $\mathbf{d} = (d_1, \ldots, d_k)$, where the c_i's and d_i's are either in C_ϕ or are $\pm\infty$, such that:
1. For all i, $c_i \leq d_i$.
2. There is no constant c in C_ϕ with the property that $c_i < c < d_i$.
3. Whenever $f_i < f_j$, then $c_i < d_j$.
4. Whenever $f_i = f_j$, then $c_i = c_j$ and $d_i = d_j$.

Intuitively, the r-configurations partition the domain into the finest possible segments that are bounded by the available constants. We show that to test whether a formula holds in a particular segment, it suffices to test whether it holds at a single, arbitrary, point in the segment.

Definition 3.3 The formula $F(C)$ corresponding to an r-configuration $C = (\mathbf{f}, \mathbf{c}, \mathbf{d})$ 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) $c_i < x_i < d_i$ whenever $c_i < d_i$. (4) $x_i = c_i$ whenever $c_i = d_i$.

We now describe the quantifier elimination in ϕ. First assume that ϕ is produced from a boolean query, i.e., it has no free variables. We present an algorithm BALG for this special case. We then generalize it to ALG for ϕ from an arbitrary query.

The algorithm BALG takes as input an r-configuration C and a formula ψ that uses only constants in C_ϕ. It returns 1 iff $F(C) \rightarrow \psi$. We initialize it by applying BALG to boolean query ϕ with the trivial r-configuration of size 0. The various cases of BALG for ψ are:

1. If ψ is an atomic formula $x_i < x_j$ then return 1 iff $(f_i < f_j)$.
2. If ψ is an atomic formula $x_i < c$ then return 1 iff $(c_i = d_i < c$ or $c_i < d_i \leq c)$.
3. If ψ is $\psi_1 \vee \psi_2$ then run the algorithm on ψ_1. If the result is 1 then return 1 else run it on ψ_2 and return the result.

4. If ψ is $\neg\psi$ then run the algorithm on ψ and invert the result.
5. If ψ is $(\exists x)\psi'$ then run the algorithm on ψ' for all extensions C' (defined in the obvious way) of the r-configuration C. Return 1 if the output on one of these r-configurations is 1; otherwise return 0.

We show that BALG is correct and that it can be implemented on a LOGSPACE Turing machine. It is an algorithm for answering boolean queries. However, it is possible to modify it into algorithm ALG that handles arbitrary queries as follows: "to answer an arbitrary query ϕ, output those r-configurations C for which $F(C) \to \phi$ and the result of the query is the conjunction of these $F(C)$'s."

Since there are only a polynomial number of r-configurations, we can use ALG as one iteration to obtain a polynomial time algorithm for inflationary Datalog$^\neg$ with order constraints. This suffices for the proof of part (2) of the theorem.

3.2 Datalog Bottom-up Evaluation Revisited

Algorithm ALG of the previous subsection does not give us enough intuition about minimal models, derivation trees and the other machinery of logic programming [L84]. This section gives a more natural bottom-up evaluation for Datalog with rational order constraints.

For simplicity of presentation, we restrict the syntax of our rules. There is only one difference from inflationary Datalog$^\neg$ with rational order constraints. No negated database predicates are allowed in Datalog with rational order constraints, but negation is allowed in constraints.

The idea is to change the notion of a Herbrand atom. This idea is the basis of Constraint Logic Programming [JL87], but in Datalog with rational order we end up with a finite Herbrand universe and a bottom-up evaluator. We show that with this new notion of atom each program still has a least model. This least model has a representation that is evaluable bottom-up in PTIME.

An *h-configuration* is a total ordering of variables and constants (e.g., $2 < x < y < 3 < 5 < z = w < 10$) with at most two adjacent constants. We take a *generalized Herbrand atom* to be a predicate symbol with distinct variable symbols as arguments and an h-configuration on exactly these variables (e.g., $R(x, y, z, w)$ together with $2 < x < y < 3 < 5 < z = w < 10$). Intuitively, a generalized Herbrand atom describes a set of points. A generalized Herbrand atom that describes a singleton set we call a *point*. Generalized Herbrand atom A_1 "covers" another atom A_2 if $points(A_1) \supseteq points(A_2)$. We note that our choice of definitions makes it easy to test for "cover".

We require database *facts* to be generalized Herbrand atoms. (We can show that facts can be transformed into this normal form with no loss of expressibility or efficiency). We require that in each *rule* all arguments be distinct variables and that each rule have a single h-configuration constraint in which all the arguments appear. (We can show that rules can be transformed into this normal form with no loss of expressibility or efficiency).

A *rule application* is the following:
(1) select any h-configuration that is "covered" by the rule h-configuration,
(2) take the projection of the selected h-configuration onto each subgoal,
(3) test that there are database facts that "cover" these projections,
(4) if all tests are true then project the selected h-configuration onto the head and add it as a new fact.

For database facts and program rules viewed together as one *constraint logic program* P, we follow the development of [L84]. We define an *interpretation* I of a constraint logic program P to be a subset of all possible generalized Herbrand atoms formable with predicate, variable and constant symbols appearing in the program. We define a function T_P from interpretations to interpretations as follows. $T_P(I) = \{A : A$ is a derivable atom using rules of P and $I\}$. We define a *model* M of a logic program P to be an interpretation of P such that any atom derivable using rules of P and database M is already in M. Then we can show the following.

Theorem 3.4 Let P be a constraint logic program. Then P has a least generalized Herbrand model L_P and $L_P = lfp(T_P) = T_P \uparrow \omega$. $\Box$

This theorem expresses the fact that our bottom-up evaluation is syntactic, and that by the finiteness of the generalized Herbrand universe it will stop in a finite number of steps (in PTIME using data complexity). It also says that the computation, as a syntactic process, has a unique well defined outcome. However, it says nothing about the semantics.

The following theorem expresses the fact that the bottom-up evaluation method that uses h-configurations is semantically sound and complete.

Theorem 3.5 Let P_1 and P_2 be two constraint logic programs with the same rules. Let P_2 have point facts only, but possibly an infinite set of them. If *points*(facts of P_1) = *points*(facts of P_2) holds, then we also have *points*(Model(P_1)) = *points*(Model(P_2)). $\Box$

We can use this bottom-up evaluation method to take advantage of the finer analysis possible in Datalog. For example, the analysis of parallelism in Datalog programs by Ullman and van Gelder [UvG88] can be used directly by substituting the word "cover" for the word "match". In this way it is possible to define *generalized derivation trees* and prove a *generalized polynomial fringe property.* Therefore:

Theorem 3.6 Piecewise linear Datalog with rational order constraints can be evaluated bottom-up in closed form and NC data complexity.$\Box$

4 Relational Calculus and Equality Constraints

We now look at equality constraints, i.e., constraints of the form $x = y$, $x \neq c$, etc., over an arbitrary infinite domain. We cannot use the previous analysis, because much of it depended on the density of the order. We can still show similar results:

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. $\Box$

Suppose we have a finite relation represented as a set of constraints. Given a relational calculus query on this finite relation, for which the result is infinite, in our language we will still get a closed, finite, description of this infinite set. Thus, because of closure, safety is no longer a critical issue.

For the relational calculus and Datalog$^\neg$ applied to finite relations there are similar results in the literature [AGSS86, Ki88, HS89]. The above theorem extends some of the results from [AGSS86, Ki88, HS89], since the input is is a generalized database.

Remark: The specific number of constants added in [AGSS86] and [HS89] to the set of constants that appear in the database and the query (i.e., to the active domain) corresponds to the induction depth used in proving our evaluation method correct.

For the proof all one has to do is modify the argument of Section 3.1 using a different notion of configuration. Let ϕ and C_ϕ be the analogs of what was defined in Section 3.1. (So ϕ is a formula with constants of the theory of equality of any infinite set).

Definition 4.2 An *e-configuration* $C = (\mathbf{e}, \mathbf{c})$ of size k consists of an equivalence relation on $\{1, \ldots, k\}$ and a sequence $\mathbf{c} = (c_1, \ldots c_k)$ where each c_i is either in C_ϕ or the special element $\circ$, such that,
1. If iej, then $c_i = c_j$.
2. If $c_i = c_j \neq \circ$, then iej.

Definition 4.3 The formula $F(C)$ that corresponds to an e-configuration $C = (\mathbf{e}, \mathbf{c})$ is the conjunction of: (1) $x_i = x_j$, whenever iej. (2) $x_i \neq x_j$, whenever $\neg iej$. (3) $x_i = c_i$ whenever $c_i \neq \circ$. (4) $x_i \neq c$ whenever $c_i \neq c$, for all c in C_ϕ.

As before, the algorithm BALG for boolean queries takes as input a e-configuration C and a formula ψ, and returns 1 iff $F(C) \rightarrow \psi$. BALG can be easily modified into ALG for arbitrary queries. In BALG, the other cases are the same as in Section 3.1, except for the following:

1. If ψ is an atomic formula $x_i = x_j$ then return 1 iff iej.
2. If ψ is an atomic formula $x_i = c$ then return 1 iff $c = c_i$.

5 Datalog and Boolean Constraints

In this section, we describe the addition of boolean constraints to Datalog with rational order. We strictly type the arguments of each database predicate. Each argument can range either over the rationals or over a finite domain. The idea is to use boolean operations as shorthands for manipulating various finite domains. This is on top of the framework with rational order that we examined in the previous section.

In the context of boolean operations, we use $+$ for *exclusive-or*, $\wedge$ for *logical and*, $\vee$ for *logical or*. We use a boolean algebra B, that can have m other constants besides 0 and 1. We use boolean algebra terms s, t (sometimes with constants). The notation $s =_B t$ means that the terms s and t are equivalent in the boolean algebra B.

5.1 The Adder Circuit Example

We first present the canonical example of Buttner and Simonis [BS87], but using bottom-up evaluation.

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 as follows:

$$\mathit{halfadder}(X, Y, Z, W) \text{ :- } X + Y =_B Z, X \wedge Y =_B W$$

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 $a + b =_B 0$, and (2) $a =_B 0$ and $b =_B 0$ iff $a \vee b =_B 0$ yields:

$$\mathit{halfadder}(X, Y, Z, W) \text{ :- } (X + Y + Z) \vee ((X \wedge Y) + W) =_B 0$$

The adder circuit can be described by the use of two half-adder circuits and an extra constraint:

$$\mathit{adder}(X, Y, C, S, D) \text{ :- } \mathit{halfadder}(X, Y, S1, C1), \mathit{halfadder}(S1, C, S, C2), D =_B C1 \vee C2$$

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:

$$\begin{aligned} \mathit{adder}(X, Y, C, S, D) \text{ :- } \quad & (X + Y + S1) \vee ((X \wedge Y) + C1) =_B 0, \\ & (S1 + C + S) \vee ((S1 \wedge C) + C2) =_B 0, \\ & (C1 \vee C2) + D =_B 0 \end{aligned}$$

By the rewrite rules (1) and (2) above and repeatedly using Boole's lemma that "$\exists y f(y, \underline{x}) =_B 0$ iff $f(0, \underline{x}) \wedge f(1, \underline{x}) =_B 0$" we transform the above into:

$$\mathit{adder}(X, Y, C, S, D) \text{ :- } (X + Y + C + S) \vee (X \wedge Y) + (X \wedge C) + (Y \wedge C) + D =_B 0$$

This last constraint can be solved by boolean unification methods to get expressions for any subset of the variables in the constraint (the dependent variables) in terms of the remaining variables (the independent variables). For example, one solution would be $S = X + Y + C$ and $D = (X \wedge Y) + (X \wedge C) + (Y \wedge C)$.

This method is quite powerful when one considers more complicated examples in which the boolean algebra B can have an arbitrary number of constants. For example, if D is the constant d, then the solution has to be expressed in terms of d, rather than as above.

5.2 Boolean Unification

In the problem of boolean unification ([MN88, BS87]), 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 $+$ *exclusive-or*, $\wedge$ *logical and*, from boolean constants, and from boolean 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 are provably equal (using the boolean algebra axioms in [MN88]).

In B we always have at least two constants, i.e., the zero element 0, and the identity element 1. We actually have an infinite set of boolean algebras, depending on the number of additional constants in the algebra. The algebra with no extra constants will be called **2**. Boolean unification in **2** is equivalent to satisfiability, and is therefore NP-complete. However, once we allow other constants, the complexity is much higher. We can show that:

Theorem 5.1 Deciding whether two boolean expressions are unifiable is Π_2^p–complete. □

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 in the normal form $s = f(x_1, \ldots, x_n)$ and $t = 0$, by using the property that $a =_B b$ iff $a + b =_B 0$. By Boole's method this problem has a solution iff $\bigwedge_{\underline{b} \in 2^n} f(\underline{b}) =_B 0$ has a solution. (Note that $\bigwedge_{\underline{b} \in 2^n} f(\underline{b})$ might contain constants other than 0 and 1).

Martin and Nipkow's method translates the original unification problem in the algebra B 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 iff each of the 2^m new problems has a solution.

This method can be applied using only the constants that appear in $f(\underline{x})$. Suppose that there are $k < m$ such constants. A unification instance $s = f(\underline{x}, \underline{y}), t = 0$ with variables $\underline{x}$ and constants $\underline{y}$ has a solution iff the equation $\bigvee_{\underline{a} \in 2^k} f(\underline{x}, \underline{a}) =_2 0$ has a solution (since $a =_B 0$ and $b =_B 0$ iff $a \vee b =_B 0$). Therefore the original problem has a solution iff $\bigvee_{\underline{a} \in 2^k} \bigwedge_{\underline{b} \in 2^n} f(\underline{b}, \underline{a}) =_2 0$ has a solution. This problem is the quantified boolean formula problem with one alternation of quantifiers, which is Π_2^p–complete [GJ79].

5.3 Bottom-up Evaluation and Boolean Unification

In this subsection, we give a brief description of a design for adding boolean constraints to Datalog with rational order. We describe the syntax for the part with boolean constraints. This syntax can be combined with the one for Datalog with rational order. We assume that when order constraints are present, they are always restricted to particular attributes. These attributes are typed and do not interact in any way with the attributes that ntake boolean values.

The syntax of the boolean constraints part is as follows. The database facts have the normal form: (1) $F(\underline{x})$:- $f(\underline{x}) =_B 0$., and (2) The rules have the normal form: $R(\underline{x})$:- $R_1(\underline{x}, \underline{y}), \ldots, R_k(\underline{x}, \underline{y}), r(\underline{x}, \underline{y}) =_B 0$., where for simplicity $\underline{x}$ and $\underline{y}$ will be any vector of variables from the set $\{x_1, \ldots, x_k, y_1, \ldots, y_l\}$.

The bottom-up evaluation proceeds as follows. We describe the "firing" of one rule. Assume that we are either given or have derived the facts $R_i(\underline{x}, \underline{y})$:- $r_i(\underline{x}, \underline{y}) =_B 0$, and that we have the rule $R(\underline{x})$:- $R_1(\underline{x}, \underline{y}), \ldots, R_k(\underline{x}, \underline{y}), r(\underline{x}, \underline{y}) =_B 0$. We produce the set

$$R' = \{R(\underline{x}) : r_1(\underline{x}, \underline{y}) =_B 0, \ldots, r_k(\underline{x}, \underline{y}) =_B 0, r(\underline{x}, \underline{y}) =_B 0\}$$

Since (1) $a =_B b$ iff $a + b =_B 0$, and (2) $a =_B 0$ and $b =_B 0$ iff $a \vee b =_B 0$, it follows that each R' can be represented as a set of facts that are in normal form except for some new variables on the right side. These extra variables can be eliminated by using repeatedly Boole's lemma that $\exists y f(y, \underline{x}) =_B 0$ iff $f(0, \underline{x}) \wedge f(1, \underline{x}) =_B 0$. This allows us to add R' to the derived facts and "fire" another rule.

Boolean unification can also be used to eliminate the quantifiers at each rule "firing". Boolean unification is used to ensure that existing facts are not derived again, and that the "firing" process is finite. We can therefore show:

Theorem 5.2 (1) Datalog with rational order and boolean constraints, over a boolean algebra with m constants, can be evaluated bottom-up in closed form. (2) If m is a fixed, then this evaluation has PTIME data complexity. □

However, m constants can provide much more succinct shorthands. Using an encoding of boolean unification and the analysis of the previous subsection, we can show that:

Theorem 5.3 There is a yes/no query expressible in Datalog with rational order and boolean constraints (over a boolean algebra with m constants) whose evaluation is Π_2^p–hard.□

6 Conclusions and Open Questions

We have presented many examples of "declarative and efficiently evaluable" constraint 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? (4) How do various optimization methods combine with our frameworks (i.e., how to extend [R88])? (5) Is it possible to integrate a select set of computational geometry algorithms as primitives in a bottom-up evaluation?

References

[AV88] S. Abiteboul, V. Vianu. Procedural and Declarative Database Update Languages. *Proc. 7th ACM PODS*, 240–250, 1988.

[ASU79] A.V. Aho, Y. Sagiv, J.D. Ullman. Equivalences among Relational Expressions. *SIAM J. of Computing*, 8:2:218–246, 1979.

[AGSS86] 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.

[B81] A.H. Borning. The Programming Language Aspects of ThingLab, A Constraint-Oriented Simulation Laboratory. *ACM TOPLAS* 3:4:353–387, 1981.

[BKR86] M. Ben-Or, D. Kozen, J. Reif. The Complexity of Elementary Algebra and Geometry. *JCSS*, 32:251–264, 1986.

[BS87] W. Buttner, H. Simonis. Embedding Boolean Expressions into Logic Programming. *Journal of Symbolic Computation*, 4:191–205, 1987.

[C87] A. Colmerauer. *An Introduction to Prolog III.* Draft, 1987.

[CH80] A.K. Chandra, D. Harel. Computable Queries for Relational Data Bases. *JCSS*, 21:156–178, 1980.

[CM76] A.K. Chandra, P.M. Merlin. Optimal Implementation of Conjunctive Queries in Relational Databases. *Proc. ACM STOC*, 77–90, 1976.

[ChI89] J. Chomicki, T. Imielinski. Relational Specifications of Infinite Query Answers. *Proc. ACM SIGMOD*, 174–183, 1989.

[Co70] E.F. Codd. A Relational Model for Large Shared Data Banks. *CACM*, 13:6:377–387, 1970.

[D88] M. Dincbas, etal. The Constraint Logic Programming Language CHIP. *Proc. Fifth Generation Computer Systems*, Tokyo Japan, 1988.

[FG77] J. Ferrante, J.R. Geiser. An Efficient Decision Procedure for the Theory of Rational Order. *Theoretical Computer Science*, 4:227–233, 1977.

[GJ79] M.R. Garey, D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP–completeness.* Freeman, 1979.

[HS89] R. Hull, J. Su. Domain Independence and the Relational Calculus. *Technical Report* 88–64, University of Southern California, submitted to IPL, 1989.

[I86] N. Immerman. Relational Queries Computable in Polynomial Time. *Information and Control*, 68:86-104, 1986.

[JL87] J. Jaffar, J-L. Lassez. Constraint Logic Programming. *Proc. 14th ACM POPL*, 111–119, 1987.

[Ki88] M. Kifer. On Safety, Domain Independence, and Capturability of Database Queries. *Proc. International Conference on Databases and Knowledge Bases*, Jerusalem Israel, 1988.

[Kl88] A. Klug. On Conjunctive Queries Containing Inequalities. *JACM*, 35:1:146–160, 1988.

[KP88] P. Kolaitis, C.H. Papadimitriou. Why not Negation by Fixpoint? *Proc. 7th ACM PODS*, 231–239, 1988.

[Le87] W. Leler. *Constraint Programming Languages.* Addison Wesley, 1987.

[L84] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1984.

[MN88] U. Martin, T. Nipkow. Unification in Boolean Rings. *Journal of Automated Reasoning*, 4:381-396, 1988.

[PS85] F.P. Preparata, M.I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, 1985.

[R83] H.L. Royden. *Real Analysis.* 2^{nd} Ed., 1983.

[R88] R. Ramakrishnan. Magic Templates: A Spellbinding Approach to Logic Programs. *Proc. 5th International Conference on Logic Programming*, 141–159, 1988.

[S80] G.L. Steele. The Definition and Implementation of a Computer Programming Language based on Constraints. Ph.D. thesis, MIT, AI-TR 595, 1980.

[T51] A. Tarski. *A Decision Method for Elementary Algebra and Geometry.* University of California Press, Berkeley, California, 1951.

[U82] J.D. Ullman. *Principles of Database Systems.* Computer Science Press, 2^{nd} Ed., 1982.

[UvG88] J.D. Ullman, A. Van Gelder. Parallel Complexity of Logical Query Programs. *Algorithmica*, 3:5-42, 1988.

[V82] M.Y. Vardi. The Complexity of Relational Query Languages. *Proc. 14th ACM STOC*, 137–146, 1982.