CONSTRAINT QUERY LANGUAGES*

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

Abstract

We investigate the relationship between programming with constraints and database query languages. We show that 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 over a small number of variables. We describe the basic Constraint Query Language design principles and illustrate them with four classes of constraints: real polynomial inequalities, dense linear order inequalities, equalities over an infinite domain, and boolean equalities. For the analysis, we use quantifier elimination techniques from logic and the concept of data complexity from database theory. This framework is applicable to managing spatial data and can be combined with existing multi-dimensional searching algorithms and data structures.

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

1 Introduction

1.1 Motivation and Framework

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 [7, 36, 52]. A major recent development in logic programming systems is the integration of logic and constraint paradigms, e.g., in CLP [27], in Prolog III [17], and in CHIP [19], for a recent survey see [16]. 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 term unification, at the forefront of this

*A preliminary version of the results in this paper appeared in [28].

†Brown University, Providence, RI. Research was supported by IBM, by an Alfred P. Sloan Fellowship, and by ONR grants N00014-83-K-0146 ARPA Order No. 4786 and N00014-91-J-4052 ARPA Order No. 8225.

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

§Brown University, Providence, RI. Research was supported by NSF grant IRI-8617344 and by NSF-INRIA grant INT-8817874.

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 style 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. It is rather surprising that constraint programming has not really influenced database query language design. Perhaps 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. There has been some previous database research on the power of constraints for the implicit specification of temporal data [14], for extending relational algebra [23], and for magic set evaluation [44] but no overall design principles. Also, manipulation of constraints has been used to study termination properties of logic programs. For an example related to the frameworks of this paper see [8].

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* [11, 60] from database theory, and *quantifier elimination* methods from mathematical logic.

Let us provide some motivation for the integration of database and constraint solving methods. Manipulation of spatial data is an important application area (e.g., spatial or 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 extensions arithmetic calculations have not been given first-class citizen status in the various query languages used, and the integration of language and application has been "loose". For an example of "tight" integration of application, language paradigm, and implementation, let us review the relational data model.

In the relational data model, [15], an important application area (data processing) is described in a declarative style (relational calculus) so that it can be automatically and efficiently translated into procedural style (relational algebra). Program evaluation is bottom-up and set-at-a-time as opposed to top-down and tuple-at-a-time, because the applications involve massive amounts of structured data. This evaluation may be optimized, e.g., via algebraic transformations, selection propagation etc. It may be performed in-core in PTIME, because of the low complexity of the calculations expressed. Most importantly, it may be implemented efficiently with large amounts of data in secondary storage via indexing and hashing.

Our claim in this paper is that by generalizing relational formalisms to constraint formalisms it is, in principle, possible to generalize all the key features of the relational data model. (1) The language framework that we propose preserves the declarative style and the efficiency of relational database languages. (2) The possible applications of constraint databases include both data processing and numerical processing of spatial data. (3) The implementation technology of spatial access methods (see [48, 49]) naturally matches the new formalism. We will now explain our new framework and give arguments in support of the above (1-3).

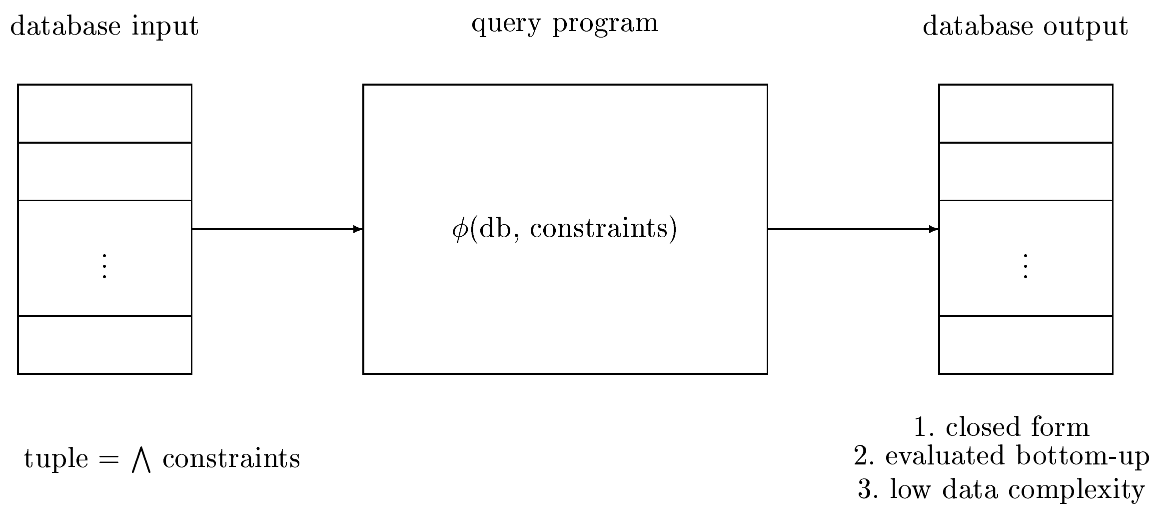

Figure 1: The CQL database framework

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

(a) Preserving the declarative language style is desirable.
(b) Additional expressive power is desirable, but must come without a serious loss of efficiency.
(c) 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). The formal definitions are in Section 1.2.

- *A generalized k-tuple is a quantifier-free conjunction of constraints on k variables, which range over a domain D.* In the relational database model $R(3,4)$ is a tuple of arity 2. It can be thought of as a single point in 2-dimensional space and also as $R(x,y)$ with $x = 3$ and $y = 4$, where x, y range over some finite domain. In our framework, $R(x,y)$ with $(x = y \wedge x < 2)$ is a generalized tuple of arity 2 and so is $R(x,y)$ with $x + y = 2.5$, where x, y range over the rational or the real numbers. Hence, a generalized tuple of arity k is a finite representation of a possibly infinite set of tuples of arity k.

- *A generalized relation of arity k is a finite set of generalized k-tuples, with each k-tuple over the same variables.* It is a disjunction of conjunctions (i.e., in disjunctive normal form DNF) of constraints, which uses at most k variables ranging over domain D.

  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. It contains at most k distinct variables and describes a possibly infinite set of arity k tuples (or points in k-dimensional space D^k).

- *The syntax of a CQL is the union of an existing database query language and a decidable logical theory.* For example: Relational calculus [15] + the theory of real closed fields [53] (Section 2); Inflationary Datalog$^\neg$ [1, 22, 33] + the theory of dense linear order with constants (Section 3); Inflationary Datalog$^\neg$ + the theory of equality on an infinite domain 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 $\phi = \phi(x_1, \ldots, x_m)$ be a query program using free variables $x_1, \ldots, x_m$. Let predicate symbols $R_1, \ldots, R_n$ in ϕ name the input generalized relations and let $r_1, \ldots, r_n$ be corresponding input generalized relations. We interpret the program in the context of such an 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 $R_i(z_1, \ldots, z_k)$ by the DNF formula for input generalized relation r_i, with its variables appropriately renamed to $z_1, \ldots, z_k$. (Note that, without loss of generality, an occurrence of a database atom in ϕ is of the form $R_i(z_1, \ldots, z_k)$ $1 \leq i \leq n$, where R_i is a predicate symbol of arity k and $z_1, \ldots, z_k$ are distinct variables; this is because in our framework we can always use equality constraints among variables in ϕ.) Let D be the constraint domain:

  > Query program $\phi = \phi(x_1, \ldots, x_m)$ 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 points in m-dimensional space D^m, such that instantiating the free variables $x_1, \ldots, x_m$ of this formula to any one of these points makes the formula true.

- *For each input, 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 [15, 54]. The precise analogue in relational databases is the notion of weak safety of [3]. In our framework, it is finiteness of representation of constraints that must be preserved. Evaluation of a query corresponds to an instance of a decision problem. Interestingly, many quantifier elimination procedures realize the goal of closed form. Also, they use induction on the structure of formulas, which leads to bottom-up evaluation.

- *For each input, the queries must be evaluable efficiently in the input size, i.e., with low data complexity.* 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 [21]).

(2) Let us motivate these design principles by a very common task from computational geometry and spatial databases; the problem of computing all rectangle intersections [43, 49]. Note that the theory of constraints used in this simple, but very common, example is the theory of dense linear order with constants (see Section 3).

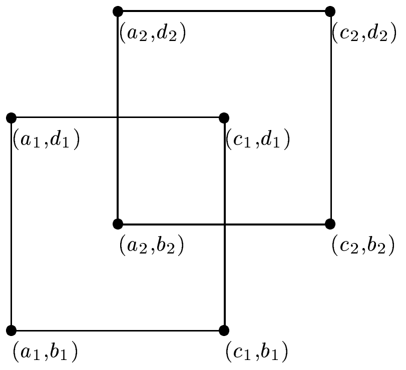

Figure 2: Rectangle intersection

Example 1.1 The database consists of a set of rectangles in the plane, and we want to compute all pairs of distinct intersecting rectangles.

This query is expressible in a relational data model that has a $\leq$ interpreted predicate. One possibility is to store the data in a 5-ary relation named R. This relation will contain tuples of the form (n, a, b, c, d), and such a tuple will mean that n is the name of the rectangle with corners at (a, b), (a, d), (c, b) and (c, d). We can express the intersection query as

$$\{(n_1, n_2) | n_1 \neq n_2 \wedge (\exists a_1, a_2, b_1, b_2, c_1, c_2, d_1, d_2)(R(n_1, a_1, b_1, c_1, d_1) \wedge R(n_2, a_2, b_2, c_2, d_2)$$
$$\wedge (\exists x, y \in \{a_1, a_2, b_1, b_2, c_1, c_2, d_1, d_2\})(a_1 \leq x \leq c_1 \wedge b_1 \leq y \leq d_1 \wedge a_2 \leq x \leq c_2 \wedge b_2 \leq y \leq d_2))\}$$

To see that this query expresses rectangle intersection note the following: the two rectangles n_1 and n_2 share a point if and only if they share a point whose coordinates belong to the set $\{a_1, a_2, b_1, b_2, c_1, c_2, d_1, d_2\}$. This can be shown by exhaustively examining all possible intersecting configurations. Thus, one could eliminate the $(\exists x, y)$ quantification altogether and replace it by a boolean combination of $\leq$ atomic formulas, involving the various cases of intersecting rectangles.

The above query program is particular to rectangles and does not work for triangles or for interiors of rectangles. Recall that, in the relational data model quantification is over constants that appear in the database. By contrast, if we use generalized relations the query can be expressed very simply (without case analysis) and applies to more general shapes.

Let $R(z, x, y)$ be a ternary relation. We interpret $R(z, x, y)$ to mean that (x, y) is a point in the rectangle with name z. The rectangle that was stored above by (n, a, b, c, d), would now be stored as the generalized tuple $(z = n) \wedge (a \leq x \leq c) \wedge (b \leq y \leq d)$. The set of all intersecting rectangles can now be expressed as

$$\{(n_1, n_2) | n_1 \neq n_2 \wedge (\exists x, y)(R(n_1, x, y) \wedge R(n_2, x, y)\}$$

The simplicity of this program is due to the ability in CQL to describe and name point-sets using constraints. The same program can be used for intersecting triangles.

In this paper we shall argue that this simplicity of expression can be combined with efficient evaluation techniques, even if quantification is over an infinite domain.□

We refer to Section 2.1 for more concrete examples from computational geometry. Other examples are presented in Section 2.2 (the balanced checkbook example) and Section 5.2 (the adder circuit example).

Remark A: The constraint theories that we investigate here are applicable to spatial databases. Temporal databases require the development of analogous frameworks for the theory of discrete linear order with constants, e.g., see [28]. For recent developments of constraint-based approaches to temporal databases we refer to [4, 13, 46]. Our results on linear order only apply to dense linear order. The case of discrete, or integer, linear order is analyzed in [46]. □

Remark B: The key concept in CQL, illustrated by Example 1.1, is that constraints describe *point-sets*, such that *all* their points are in the database. With the appropriate constraint theory these point-sets are accurate (and perhaps the most intuitive) representations of spatial objects. Our framework is thus one of *complete information.* Constraint logic programming paradigms are currently attracting a great deal of attention in languages for operations research applications [58, 59] and have also impacted the field of concurrent programming language design [50]. The use of constraints for operations research and for concurrency is sometimes semantically different from their use in our framework. For example: Constraints can be used to represent the many possible states (of which one is true) of a set of concurrent processes. Each individual concurrent process maintains and manipulates constraints that describe the *partial information* it has about the state of all processes. □

(3) The language framework of the relational data model does have low data complexity, but does not account for searches that are logarithmic or faster in the sizes of input relations. Without the ability to perform such searches relational databases would have been impractical. Very efficient use of secondary storage is an additional requirement, beyond low data complexity, whose satisfaction greatly contributes to relational technology.

B-trees and their variants B^+-trees, [5, 18], are examples of important data structures for implementing relational databases. In particular, let each secondary memory access transmit B units of data, let r be a relation with N tuples, and let us have a B^+-tree on the attribute x of r. The space used in this case is $O(N)$. The following operations define the problem of *1-dimensional searching on relational database attribute* x, with the corresponding performance bounds using a B^+-tree on x: (i) Find all tuples such that for their x attribute $(a_1 \leq x \leq a_2)$. If the output size is K tuples, then this *range searching* is in worst-case $O(\log_B N + K/B)$ secondary memory accesses. If $a_1 = a_2$ and x is a key, then this is *key-based searching.* (ii) Insert or delete a given tuple. These are in worst-case $O(\log_B N)$ secondary memory accesses.

The problem of *k-dimensional searching on relational database attributes* $x_1, \ldots, x_k$ generalizes 1-dimensional searching to k attributes, with range searching on k-dimensional intervals. It is a central problem in spatial databases for which there are many solutions with good secondary memory access performance, e.g., grid-files, quad-trees, R-trees (see the surveys [48, 49]).

For generalized databases we can define an analogous problem of *1-dimensional searching on generalized database attribute* x using the operations: (i) Find a generalized database that represents all tuples of the input generalized database such that their x attribute satisfies $(a_1 \leq x \leq a_2)$. (ii) Insert or delete a given generalized tuple.

If $(a_1 \leq x \leq a_2)$ is a constraint of our CQL then there is a trivial, but inefficient, solution to the problem of 1-dimensional searching on generalized database attribute x. One can add constraint $(a_1 \leq x \leq a_2)$ to every generalized tuple (i.e., conjunction of constraints) and naively insert or delete generalized tuples in a table. This would involve a linear scan of the generalized relation and introduces a lot of redundancy in the representation. In many cases, the projection of any generalized tuple on x is one interval $(a \leq x \leq a')$. This is true for Example 1.1, for our CQL's with dense linear order, for relational calculus with linear inequalities over the reals, and in general when a generalized tuple represents a convex set. Under such natural assumptions, there is a better solution for 1-dimensional searching on generalized database attribute x.

- A *generalized 1-dimensional index* is a set of intervals, where each interval is associated with a generalized tuple. Each interval $(a \leq x \leq a')$ in the index is the projection on x of its associated generalized tuple. The two endpoint a, a' representation of an interval is a fixed length *generalized key*.
- Finding a generalized database, that represents all tuples of the input generalized database such that their x attribute satisfies $(a_1 \leq x \leq a_2)$, can be performed by adding constraint $(a_1 \leq x \leq a_2)$ to only those generalized tuples whose generalized keys have a non-empty intersection with it.
- Inserting or deleting a given generalized tuple are performed by computing its projection and inserting or deleting intervals from a set of intervals.

The use of generalized 1-dimensional indexes reduces redundancy of representation and transforms 1-dimensional searching on generalized database attribute x into the problem of on-line intersections in a dynamic set of intervals. This is a well-known problem with many elegant solutions from computational geometry [43]. It is a special case of 2-dimensional searching in relational databases, called 1.5-dimensional searching in [41]. For example, the priority search trees of [41] are a linear space data structure with logarithmic-time update and search algorithms for in-core processing. Grid-files, R-trees, and quad-trees have all been used for solving this problem with good secondary memory access performance.

In summary, *current spatial database access methods are applicable to indexing in our CQL framework* because: If $(a_1 \leq x \leq a_2)$ is a constraint of our CQL and the projection of any generalized tuple on x is an interval $(a \leq x \leq a')$, then the problem of 1-dimensional searching on generalized database attribute x is a special case of 2-dimensional searching in relational databases.

We will now concentrate on the technical development of the CQL framework and on the existence of natural constraint query languages with closed form, bottom-up evaluation of low data complexity.

1.2 Basic Definitions

The framework, of generalized relations with corresponding query languages, can be applied to many different classes of constraints.

Definition 1.2 The classes we consider in Sections 2-4 are as follows.

1. *Real polynomial inequality constraints* are all formulas (and their negations) of the form $p(x_1, \ldots, x_j)\ \theta\ 0$, where p is a polynomial with real coefficients, variables $x_1, \ldots, x_j$, and θ is one of $=, \leq, <$ (or its negation $\neq, >, \geq$). The domain D is the set of real numbers and function symbols $+, *$, predicate symbols θ, and constants are intepreted in the standard way over D.

2. *Dense linear order inequality constraints* are all formulas (and their negations) of the form $x\theta y$ and $x\theta c$, where x, y are variables, c is a constant, and θ is one of $=, \leq, <$ (or its negation $\neq, >, \geq$). We assume D is a countably infinite set (e.g., the rational numbers) with a binary relation which is a dense linear order. Constants, $=, \leq$, and $<$ are interpreted as elements, equality, the dense linear order, and the irreflexive dense linear order of D.

3. *Equality constraints over an infinite domain* are all formulas (and their negations) of the form $x\theta y$ and $x\theta c$, where x, y are variables, c is a constant, and θ is $=$ (or $\neq$). We assume D is a countably infinite set (e.g., the integer numbers) but without order. Constants and $=$ are interpreted as elements and equality of D.

In Section 5, we present the definitions and analysis for *boolean equality constraints.*□

Remark C: There are of course other classes of constraints that could illustrate the CQL framework, e.g., linear inequalities over the reals or discrete linear order constraints. However, the examples we have chosen illustrate all of our analytical techniques. (a) Real polynomial inequality constraints are quite general. They show the possible applicability of the framework to problems of computational geometry and the limits of data complexity analysis. It is possible to combine them with relational calculus, but not with recursive formalisms. (b) Dense linear order constraints are also very general, since one may use them to simulate any PTIME computation (as in [26] and [60]). We devote a large part of our analysis to this case, because it best illustrates the desired integration with relational calculus and various recursive formalisms. Discrete linear order is much harder to combine with recursion [46]. (c) Equality constraints over an infinite domain were chosen as the simplest generalization of the relational data model. The analysis here is very close to that of dense linear order constraints. (d) Finally, boolean equality constraints capture important operations research applications, although their CQL is not as "efficient" as in the other cases. □

Definition 1.3 Let Φ be a class of constraints.

1. A *generalized k-tuple* (over variables $x_1, \ldots, x_k$) is a finite conjunction $\varphi_1 \wedge \cdots \wedge \varphi_N$, where each $\varphi_i, 1 \leq i \leq N$, is a constraint in Φ. Furthermore, the variables in each φ_i are all free and among $x_1, \ldots, x_k$.

2. A *generalized relation of arity k* is a finite set $r = \{\psi_1, \ldots, \psi_M\}$, where each $\psi_i, 1 \leq i \leq M$ is a generalized k-tuple over the same variables $x_1, \ldots, x_k$.

3. The *formula corresponding to a generalized relation r* is the disjunction $\psi_1 \vee \cdots \vee \psi_M$. We use ϕ_r to denote the quantifier-free formula corresponding to relation r.

4. A *generalized database* is a finite set of generalized relations.$\Box$

In database theory, a k-ary relation r is a finite set of k-tuples (or points in a k-dimensional space) and a database is a finite set of relations. However, the relational calculus and algebra can be developed without the finiteness assumption for relations. We will use the term *unrestricted relation* for finite or infinite sets of points in a k-dimensional space. It is possible to develop query languages using such unrestricted relations (e.g., see [39]). In order to be able to do something useful with such unrestricted relations, we need a finite representation that we can manipulate. This is exactly what the generalized tuples provide.

Definition 1.4 Let Φ be a class of constraints interpreted over domain D, r a generalized relation of arity k with constraints in Φ, and $\phi_r = \phi_r(x_1, \ldots, x_k)$ the formula corresponding to r with free variables $x_1, \ldots, x_k$. The generalized relation r represents the unrestricted k-ary relation which consists of all $(a_1, \ldots, a_k)$ in D^k such that $\phi_r(a_1, \ldots, a_k)$ is true. A generalized database represents the finite set of unrestricted relations that are represented by its generalized relations. $\Box$

Example 1.5 This is a generalization of the relational data model. Let relation r consist of the tuples $(1,2)$ and $(3,4)$. These tuples are equivalent to the generalized 2-tuples, $x = 1 \wedge y = 2$ and $x = 3 \wedge y = 4$. Therefore, the r corresponds to the set $\{x = 1 \wedge y = 2, x = 3 \wedge y = 4\}$ and the formula $\phi_r \equiv (x = 1 \wedge y = 2) \vee (x = 3 \wedge y = 4)$. It should be clear that a point (x, y) is in the generalized relation iff it satisfies the corresponding formula.

Let us illustrate our framework using real polynomial inequality constraints. Let generalized relation r consist of two generalized tuples $(y = 2 * x \wedge x \neq y)$ and $(x + y \geq 1)$. Corresponding to this r is the DNF formula $\phi_r = (y = 2 * x \wedge x \neq y) \vee (x + y \geq 1)$. ϕ_r describes an infinite set of points in 2-dimensional space namely the half plane $x + y \geq 1$ and the line $y = 2 * x$ without the point $x = y = 0$. $\Box$

Note that the representation of an unrestricted relation by a finite set of generalized tuples need not be uniquely defined.

Relational calculus + constraints: We present a short but self-contained description of the relational calculus with a given a class of constraints. For more details on the relational calculus in database theory see [15, 29, 54].

Definition 1.6 Let Φ be a class of constraints. Let $R_1, \ldots, R_i, \ldots$ be predicate symbols, each with a fixed arity. A relational calculus + Φ query program is a formula of the first-order predicate calculus with equality, such that its atomic formulas are (1) of the form $R_i(x_1, \ldots, x_j)$, where j is the arity of predicate symbol R_i, or (2) formulas from the class Φ of constraints.□

Example 1.7 Let Φ be the class of dense linear order constraints. If R_1 is a predicate symbol of arity 2, then the following is a query:

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

In order to formally define its meaning, we need interpretations for the predicate symbols. These will come from input generalized relations. We also need interpretations of the symbols in the constraints. These will come from the particular theory of constraints used.□

Definition 1.8 Let D be the domain of constraint class Φ and δ the interpretation of the symbols in these constraints. Let ϕ be a relational calculus + Φ query program with predicate symbols $R_1, \ldots, R_n$ and with free variables $x_1, \ldots, x_m$. Let $r_1, \ldots, r_n$ be generalized relations of the same arities as $R_1, \ldots, R_n$. These generalized relations represent unrestricted relations $\rho_1, \ldots, \rho_n$ (where ρ_i is the set of points that satisfy ϕ_{r_i}). Using the standard first order meaning of $\models$ we define:

$$\rho \equiv \phi\,[\rho_1/R_1, \ldots, \rho_n/R_n] \equiv \{a_1, \ldots, a_m \in D^m \mid \langle D, \delta, \rho_1, \ldots, \rho_n\rangle \models \phi(a_1, \ldots, a_m)\}$$

The query expressed by program ϕ is defined as a mapping: from unrestricted relations $\rho_1, \ldots, \rho_n$ (represented by the input generalized relations $r_1, \ldots, r_n$) to an arity m unrestricted relation ρ. *We also require that ρ be representable by some generalized relation r of arity m.* □

Although unrestricted relation $\rho \equiv \phi\,[\rho_1/R_1, \ldots, \rho_n/R_n]$ is always well defined, the reader should note that our definition requires an additional *closure condition.* Both input and output should be representable by generalized relations.

Remark D: 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. In other words, if we let $\psi \equiv \phi\,[r_1/R_1, \ldots, r_n/R_n]$ be the result of replacing each occurrence of R_i in ϕ by the formula ϕ_{r_i}, then $\phi\,[\rho_1/R_1, \ldots, \rho_n/R_n]$ is precisely the set of points that satisfy ψ. This formula ψ, however, might contain quantifiers and even not correspond to any generalized database. So closure is a non-trivial condition. Quantifier-elimination in the theory of constraints will allow us to satisfy this condition. □

Example 1.9 For a simple example where closure does not hold consider *real polynomial equalities.* These are constraints of the form $p(x_1, \ldots, x_n)\ \theta\ 0$, where θ is $=$ or $\neq$. Let $R(x, y)$ be a binary predicate symbol for the input generalized relation $\{y = x^2\}$. The result (interpreting the generalized relation as an infinite set of points) of $\exists x.R(x, y)$ is the set $\{y | y \geq 0\}$, which cannot be represented by polynomial equality constraints. □

Datalog + constraints: We now consider Datalog with constraints. The syntax is that of Datalog (e.g., see [1, 29, 33, 54, 55]) but we allow the bodies of rules to contain constraints.

Definition 1.10 Let Φ be a class of constraints. Let $R_1, \ldots, R_i, \ldots$ be predicate symbols, each with a fixed arity. A Datalog + Φ query program π is a finite set of rules of the form:

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

t_0, the rule *head*, must be an atomic formula of the form $R(x_1, \ldots, x_k)$, where R is some predicate symbol of arity k. The expressions $t_1, \ldots, t_l$, the rule *body*, are either of the form $R'(x_1, \ldots, x_{k'})$, where R' is some predicate symbol of arity k', or are constraints from Φ. The predicate symbols that appear in heads of rules are called intentional database predicates (IDBs) and the rest are called extensional database predicates (EDBs). $\square$

The meaning of a Datalog + Φ query program π on generalized relations $r_1, \ldots, r_n$, that represent the unrestricted relations $\rho_1, \ldots, \rho_n$, is the least fixpoint of the monotone mapping defined by a first-order formula ϕ_π and $\rho_1, \ldots, \rho_n$. The definition is the same as in the case without constraints, the only difference being the use of unrestricted relational databases [29, 39, 54]. We present this definition by example.

Example 1.11 Consider the Datalog query program π with dense linear order constraints.

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

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

Apply this query program to the generalized database r_0 that represents the unrestricted relation ρ_0. Then ϕ_π is the following first-order formula,

$$\phi_\pi(x,y) \equiv \psi(x,y;R) \equiv R_0(x,y) \vee \exists z(R(x,z) \wedge R_0(z,y) \wedge x \le y \wedge y \le z).$$

ϕ_π defines a mapping from arity 2 unrestricted relations ρ to arity 2 unrestricted relations. Note that, in this formula R_0 is always interpreted as ρ_0. Predicate symbol R is singled out because its interpretation as any value ρ defines the mapping:

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

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. $\square$

The mere existence of the fixpoint, as guaranteed by the Tarski fixpoint theorem, is not enough for our purposes. As in the case of the relational calculus we require that the result of a Datalog query be finitely representable as a generalized database. We shall show that this *closure condition* is satisfied by Datalog, when we consider constraints from the language of dense linear order or equality over an infinite domain. Unfortunately, as the next example shows, this rules out the use of Datalog with real polynomial inequalities.

Example 1.12 Let π be the query program that consists of the rules $S(x,y) :\!\!-\; R(x,y)$ and $S(x,y) :\!\!-\; R(x,z), S(z,y)$ (i.e., S is the transitive closure of R). If the input r for R consists of the generalized relation $y = 2 * x$, then the result of the query is the set of all points (x,y) that satisfy $y = 2^i * x$ for some $i > 0$. This set is not finitely representable in the language of polynomial inequality constraints. $\Box$

Inflationary Datalog$^\neg$ + constraints: The syntax is that of Datalog with constraints with one addition. We allow in a rule body expressions of the form $\neg R'(x_1, \ldots, x_{k'})$, where R' is some predicate symbol of arity k'. We give the language inflationary semantics [1, 22, 33]. 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.

We shall show that the closure results mentioned above, for Datalog with dense order or with equality constraints, hold with inflationary negation as well.

Remark E: We have given the semantics of a Datalog + Φ (Datalog$^\neg$ + Φ) query program on a generalized database as the least fixpoint of a monotone (inflationary) mapping from unrestricted relations to unrestricted relations. 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. $\Box$

Various fragments of relational calculus and Datalog have been found to be particularly useful in databases and have been examined in depth. *Tableaux query programs* form such a fragment. We provide definitions and examples for them in Section 2.2, and refer to [2, 12, 32, 54] for a more detailed treatment.

Complexity: We assume familiarity with the definitions of basic complexity classes such as LOGSPACE, PTIME, NC, and Π_2^p (see [21]).

The prototypical logspace-complete problem in Π_2^p is the *AE-quantified boolean formula problem*: *Input*, a formula $\forall \overline{x} \exists \overline{y} \psi(\overline{x}, \overline{y})$, where $\overline{x}, \overline{y}$ are sets of boolean variables and $\psi(\overline{x}, \overline{y})$ a propositional formula over these variables. *Question*, is the input formula true?

We now define *data complexity.* Our definition involves the complexity of evaluating some representation for the output of a *fixed* query Q, given a *variable* input generalized database. This is more general than the definition of data-complexity for yes/no decision problems.

Definition 1.13 Our sequential machine model is a Turing Machine (TM) with a read-only input tape, a write-only output tape, and a fixed number of work tapes. Our parallel machine model is a Parallel Random Access Machine (PRAM). Our input generalized relations are encoded using some fixed binary encoding.

A query Q has *data complexity* in PTIME (resp. LOGSPACE, NC) if there is a TM (resp. TM, PRAM) which given input generalized relations d produces some generalized relation representing the output of $Q(d)$ and uses polynomial time (resp. logarithmic space on the work tape, polynomial number of processors running in polylogarithmic parallel time). $\Box$

1.3 Overview of Contributions

¿From Codd's original work [15] it follows that: *safe relational calculus can be evaluated bottom-up in closed form and LOGSPACE data complexity.* Codd defines safe formulas via syntactic restrictions on relational calculus. The LOGSPACE data complexity analysis is from [11]. We provide as evidence of the soundness of our design principles many variations of this observation in the context of constraints. The following table summarizes the main data complexity results:

	Polynomial	**Dense Order**	**Equality**
Relational Calculus	NC	LOGSPACE	LOGSPACE
Datalog$^\neg$	Not closed	PTIME	PTIME

In more detail:

1. Relational calculus with real polynomial inequality constraints can be evaluated bottom-up in closed form and NC data complexity. This is a direct consequence of [6, 35, 53] and illustrates the potential applicability of the framework to spatial databases (Section 2.1).

2. As part of our analysis of the relational calculus and real polynomial inequality constraints, we provide a new interpretation of the homomorphism theorem for tableau query containment from [2, 12, 32]. Our interpretation is based on the simple geometric fact that, "an affine space is contained in a finite union of affine spaces iff it is contained in one member of this union" [47], p. 139. We show that deciding containment between tableaux queries with linear equalities is NP-complete, but that with quadratic equalities it is Π_2^p-hard (Section 2.2).

3. Relational calculus (Inflationary Datalog$^\neg$) with dense linear order constraints can be evaluated bottom-up in closed form and LOGSPACE (PTIME) data complexity. This is shown by adapting the proof of [20]. Also, by a slight modification of [26, 60] Inflationary Datalog$^\neg$ with dense linear order expresses *exactly* PTIME (Section 3.1).

4. For Datalog with dense linear order constraints, we develop a bottom-up evaluation method that is closer to the classical foundations of logic programming [38] and knowledge bases [54, 55] (Section 3.2). This allows us to show that piecewise linear Datalog with dense linear order constraints can be evaluated bottom-up in closed form and NC data complexity (Section 3.3).

5. Relational calculus (Inflationary Datalog$^\neg$) with equality constraints over an infinite domain can be evaluated bottom-up in closed form and LOGSPACE (PTIME) data complexity. This extends the approach to safe queries of [3, 25, 31, 44] (Section 4).

6. Finally, Datalog with boolean equality constraints can be evaluated bottom-up and in closed form. For the definitions we refer to Section 5 and [10, 34, 40]. The data complexity here is higher than in the previous cases and it depends on the use of free boolean algebras with m generators. We partly analyze this data complexity and show it to be Π_2^p-hard (Section 5).

2 Real Polynomial Inequality Constraints

Throughout Section 2, we assume that the constraint domain D is the set of real numbers, but our analysis applies to any real closed field.

In Section 2.1, we give our first example of a CQL by combining relational calculus with real polynomial inequalities.

In Section 2.2, we investigate tableaux queries with constraints. We present several results on the optimization of such queries, in the presence of linear equations, quadratic equations, and simple inequalities without arithmetic operations.

2.1 Relational Calculus with Constraints and Computational Geometry

Consider a query language consisting of all first-order formulas over the database predicates together with real polynomial inequality constraints. The syntax is the union of relational calculus with that of the theory of real closed fields [53]. For the semantics, the database atomic formulas will be used as shorthands for large formulas of the theory of real closed fields, as described in Section 1.

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.

In Example 1.1, we already illustrated this language using the problem of object intersection. It is interesting to note that most other basic operations of computational geometry (e.g., Convex Hull and Voronoi diagram – see [43]) can be described in this declarative query language, which also happens to be efficiently bottom-up evaluable.

Example 2.1 *Convex hull*: The database consists of an arity 2 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, takes $O(N^4)$ time, because it involves four database atomic formulas. Although it cannot compete with various known $O(N \log N)$ algorithms, it is still useful in combination with other convex hull techniques. □

Example 2.2 *Voronoi diagram*: We can show how to find the graph called the *dual* of the Voronoi diagram [43]. 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. □

Queries in the language of relational calculus and real polynomial inequality 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 [53] and is one of its basic properties. 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 the general problem of geometric theorem proving. The reason for this is that if we focus our attention on data complexity then 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 [6, 35] to show that the data complexity is in NC.

Theorem 2.3 Relational calculus with real polynomial inequality constraints can be evaluated bottom-up in closed form and NC data complexity.

Proof: This is a direct application of [6, 35]. To see this use the fixed dimension case of the theorem p. 263 in [6]. The cell decomposition method in sections 6-7 of [35] can be used to output a formula in DNF (of size polynomial in the input) that represents the output generalized database. □

It is true 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.

Of course, given the NC data complexity bounds, there are computations that are not expressible in relational calculus with real polynomial inequality constraints. It would be interesting to determine which natural computational problems are or are not expressible. For example, we conjecture that computing Euclidean Spanning Trees is not expressible because it involves reachability computations.

As we pointed out in Section 1, if we consider Datalog with polynomial constraints, the resulting language is not closed. Furthermore, such a language combining arithmetic with recursion has full Turing computability power. It would be interesting to design a CQL with low data complexity which allows limited use of recursion and real polynomial inequalities.

2.2 Tableaux Query Programs and their Containment Problem

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

x				Balanced
x	f	r	m	Expenses
x	s	-	-	Savings
x	w	i	-	Income
				$f+r+m+s=w+i$

Figure 3: The tableau with constraints "balanced checkbook" query program.

techniques have been developed for query optimization. A key problem for optimization is testing containment of query programs.

Each query program ϕ computes for any input generalized database d an output generalized relation $\phi[d]$. Recall that generalized relations represent possibly infinite sets of points. We say that a query program ϕ_1 is contained in query program 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 $\phi_2[d]$. The *containment* problem is: Given two query programs ϕ_1, ϕ_2 decide if $\phi_1 \subseteq \phi_2$.

We now examine *(tagged untyped) tableaux query programs.* These query programs were the subject of many investigations in relational database theory and can be presented as nonrecursive Datalog rules. The terminology (tagged untyped) tableau is used, because each program can be described as a table, with variables or constants appearing in each entry, with the predicate symbols as row-tags, and possibly with some untyped variable appearing in many columns. We augment these queries using special real polynomial inequalities such as linear equations, quadratic equations, and inequalities without $+, *$. For instance:

Example 2.4 In nonrecursive Datalog notation and using 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 query program with *Expenses, Savings* and *Income* input relations, *Balanced* output relation, and a single linear equation constraint: x is user-id, 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 user-ids whose checkbooks balance.

In tableau notation, the checkbook query can be represented by a four row tableau with Balanced, Expenses, Savings, and Income row-tags. The first row corresponds to the head of the rule and is called the *summary row.* The other three rows correspond to predicate symbol occurrences in the body of the rule. Each of these rows has width four, because we add dummy arguments up to the maximum arity, i.e., new distinct variables denoted $-$ (see Figure 3). The linear equation constraint is extra.

For the detailed terminology see [2].□

Let us now explain normal forms, symbol mappings, and homomorphisms. We break up each ϕ, tableau query program with constraints, into a tableau part T, that consists exclusively of *distinct* occurrences of variables, and a conjunction of constraints C. This *normal form* (T, C) is without loss of generality, since the constraints in C can force any equalities of the distinct symbols in T.

Let $\phi_1 = (T_1, C_1)$ and $\phi_2 = (T_2, C_2)$ be two normal form tableaux query programs with real polynomial inequality constraints. A function h is a *symbol mapping* from the symbols of ϕ_2 to those of ϕ_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 a *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 $h(C_2)$.

Lemma 2.5 Let $\phi_1 = (T_1, C_1)$ and $\phi_2 = (T_2, C_2)$ be two normal form tableaux query programs with real polynomial inequality constraints. Let $h_1, \ldots, h_m$ be all the possible symbol mappings from T_2 to T_1. $(\forall d,\ \phi_1[d] \subseteq \phi_2[d])$ iff $(C_1$ implies $h_1(C_2) \vee \cdots \vee h_m(C_2))$.

Proof: (If) If θ_1 is any constraint satisfying valuation for ϕ_1 (i.e., θ_1 is an assignment of values to variables of T_1 satisfying C_1), 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 satisfying valuation for ϕ_2. This implies that for any generalized database d, $\phi_1[d] \subseteq \phi_2[d]$.

(Only if) Let d be any generalized database 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., to 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 $h = (a \to x, b \to y, c \to z)$. 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. $\square$

Let $\phi_1 = (T_1, C_1)$ and $\phi_2 = (T_2, C_2)$ be two tableau query programs with constraints, in normal form. We say that they have the *homomorphism property*, whenever there is a symbol mapping h from ϕ_2 to ϕ_1 such that (for all generalized databases d, $\phi_1[d] \subseteq \phi_2[d]$) iff ($C_1$ implies $h(C_2)$). We will now show that the homomorphism property holds when we have linear equations and is the key to proving containment in NP. This extends the basic technique of [2, 12].

Theorem 2.6 Given two query programs, each a tableau with a conjunction of linear equation constraints, deciding containment is NP-complete.

Proof: NP-hardness is immediate, since it is NP-complete to determine containment for such queries just with equations of the form $x = y$ [2, 12]. The new part is showing membership in NP, given more general linear equations. 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.

We use the previous lemma and from [47], p. 139, the simple geometric fact: "an affine space is contained in a finite 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. □

In contrast, with quadratic equations we can show:

Theorem 2.7 Given two query programs, each a tableau with a conjunction of quadratic equation constraints, deciding containment 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. 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 $s_k = s_i s_j$ 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 output 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. $\Box$

Containment in the presence of inequality constraints without $+$ and $*$ is the problem examined in[32]. For containment of tableaux with inequalities and no $+,*$, we can show that the homomorphism property fails even for semiinterval query programs.

Semiinterval query programs 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. In [32] it is shown that the homomorphism property holds for left-semiinterval queries or right-semiinterval queries alone and does not work for interval queries, i.e., variables are bounded from both sides.

Theorem 2.8 The homomorphism property fails for semiinterval query programs.

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:

$$R''(u) :\!\!- R'(u), R(v,w), v \le 4, w \ge 4$$

While the query ϕ_1 is:

$$R''(u) :\!\!- R'(u), R(x,y), R(y,z), x \le 4, z \ge 4$$

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

$\phi_1 \subseteq \phi_2$ is easy to see, because either $y \ge 4$ and $R(x,y), R(y,z)$, $x \le 4, z \ge 4$ implies $R(x,y)$, $x \le 4, y \ge 4$, which is the same as in ϕ_2 after renaming, or $y < 4$ and $R(x,y), R(y,z)$, $x \le 4, z \ge 4$ implies $R(y,z), y \le 4, z \ge 4$, 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 $\theta = (x \to 1, y \to 3, z \to 5, u \to 7)$, ϕ_1 yields $R''(7)$. However, $\theta h_1(\phi_2)$ is not a valid valuation for ϕ_2, because $\theta h_1(R(v,w), v \le 4, w \ge 4) = R(1,3)$, $1 \le 4$, $3 \ge 4$, 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 $\theta' = (x \to 1, y \to 5, z \to 9, u \to 7)$, ϕ_1 again yields $R''(7)$. However, $\theta' h_2(\phi_2)$ is not a valid valuation, because $\theta' h_2(R(v,w), v \le 4, w \ge 4) = R(5,9), 9 \ge 4, 5 \le 4$, i.e., it is also unsatisfiable and cannot produce any output tuple. $\Box$

3 Dense Linear Order Inequality Constraints

Throughout Section 3, we assume that the constraint domain D is the set of rational numbers, but our analysis applies to any set with a dense linear order.

In Section 3.1, we show that the relational calculus (Datalog$^{\neg}$) with dense linear order inequality constraints can be evaluated bottom-up in closed form and LOGSPACE (PTIME) data complexity. These are tighter bounds than the NC bounds that we get from the analysis of relational calculus with real polynomial inequality constraints.

In Section 3.2, we provide an alternative bottom-up evaluation for Datalog, which 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 [27, 38].

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

3.1 Data Complexity of Relational Calculus and Datalog¬ with Dense Order

We will first show that the relational calculus with dense linear order inequality constraints has LOGSPACE data complexity. The proof of this result is based on the proof of [20], which we extend to languages with constants and adapt for data complexity analysis. Our basic technical contribution is the appropriate definition of r-configuration (for rational-configuration).

In order to use the decision procedure techniques of [20], we transform the query program applied to the input set of constraints into one semantically equivalent formula ϕ of the theory of dense linear order with constants (see Section 1).

For example, let $R(x, y)$ be a binary generalized relation containing the three generalized tuples $x < y$, $x < 5$ and $y = 4$. Consider the query program $(\exists z)(R(x, z) \wedge R(z, y))$ 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 $=, \leq$ and just use atoms of the form $x_i < x_j, x_i < c, c < x_i$. For this replace $x \leq y$ by $(x = y) \vee (x < y)$ and $x = y$ by $\neg((x < y) \vee (y < x))$. 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. Throughout this section, *we use D_ϕ be the set of constants that appear in ϕ.*

Definition 3.1 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 points $\overline{x} = (x_1, \ldots, x_n)$ and $\overline{y} = (y_1, \ldots, y_n)$ in D^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 3.2 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: (3.3) Express an r-configuration as a conjunction of constraints. (3.4) The points satisfying this conjunction are the indistinguishable points denoted by the r-configuration. (3.5) An r-configuration can be extended by adding other variables.

Definition 3.3 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 3.4 $\models F(\xi)(a_1, \ldots, a_n)$ will mean that $F(\xi)$ is satisfied by the assignment of each a_i to the corresponding variable x_i.□

Definition 3.5 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 if

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. $\overline{u}' = (u_1, \ldots, u_n, u_{n+1})$.$\square$

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 five Lemmas 3.6–10. We use this partitioning to construct an algorithm EVAL$_\phi$, which evaluates the query in closed form. The output of EVAL$_\phi$ consists of a set of r-configurations. Its correctness is described in the two Lemmas 3.11–12. We then argue that EVAL$_\phi$ can be implemented in LOGSPACE.

Query Evaluation Algorithm EVAL$_\phi$

Input of EVAL$_\phi$**:** A generalized database. We will assume from now on that ϕ is the result of substituting the definition of the generalized database for each occurrence of a predicate symbol in the query (we comment later on why this does not affect the LOGSPACE complexity).

For each r-configuration ξ of size n, with constants from D_ϕ, test whether $F(\xi) \rightarrow \phi$ is valid (i.e., true for all assignments to its free variables). This test is performed using recursive procedure Boolean-EVAL$_\phi$.

Procedure Boolean-EVAL$_\psi$ takes as input an r-configuration ξ'. It is only called on subformulas of ϕ, thus guaranteeing that all the constants in ψ are in D_ϕ. Boolean-EVAL$_\psi$ returns 1 iff $F(\xi') \rightarrow \psi$ is valid. Its various cases 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 $x_i < c$ or $c < x_i$.
For $x_i < c$, 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.
For $c < x_i$, if $c < l_i = u_i$ or $c \leq l_i < u_i$ (where l_i, u_i are from ξ') then return 1 else return 0.

3. ψ is $\psi_1 \vee \psi_2$.
If the result of Boolean-EVAL$_{\psi_1}(\xi')$ is 1 then return 1 else return result of Boolean-EVAL$_{\psi_2}(\xi')$.

4. ψ is $\neg\psi'$.
If result of Boolean-EVAL$_{\psi'}(\xi')$ is 1 then return 0 else return 1.

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

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

Lemma 3.6 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. Given the density of order, 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 3.1, 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 3.1. The second case is similar.

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

Lemma 3.7 Let ξ be an r-configuration of size n. There exist elements $a_1, \ldots, a_n$ of D, such that $\models F(\xi)(a_1, \ldots, a_n)$.

Proof: Induction on the size of ξ, using Lemma 3.6.□

Lemma 3.8 Let $a_1, \ldots, a_n$ be elements of D. 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 3.3, $a_j < a_i$. This implies, using Definition 3.3 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 3.3, $l_i^1 = a_i < l_i^2 \leq a_i$, a contradiction. On the other hand, if $l_i^1 < u_i^1$, then part 2 of Definition 3.1 implies that $u_i^1 \leq l_i^2$. Once more, by Definition 3.3, $a_i < u_i^1 \leq l_i^2 \leq a_i$, 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 3.9 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 $x_i < c$ ($c < x_i$ is treated similarly). 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 3.8, 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 3.8) that ξ' must be an extension of ξ. Since $(a'_1, \ldots, a'_k)$ satisfies $F(\xi)$, by Lemma 3.6 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 3.10 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 its free variables) iff $F(\xi) \wedge \psi$ is satisfiable (i.e., true for some assignment to its free variables).

Proof: If $F(\xi) \to \psi$ is valid, then Lemma 3.7 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 3.9 implies that ψ is satisfied by $(a'_1, \ldots, a'_n)$. □

We now prove correctness of Boolean-EVAL$_\psi$ and EVAL$_\phi$.

Lemma 3.11 The algorithm Boolean-EVAL$_\psi(\xi')$ described above returns 1 iff $F(\xi') \to \psi$ 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 an atomic formula $c < x_i$. The proof is similar with 2.
4. ψ is $\psi_1 \vee \psi_2$. Correctness follows from Lemma 3.10, 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.
5. ψ is $\neg\psi'$. To show correctness we must show that $F(\xi') \to \neg\psi'$ is valid iff $(F(\xi') \to \psi')$ is not valid. By Lemma 3.10, $F(\xi') \to \neg\psi'$ is valid iff $F(\xi') \wedge \neg\psi'$ is satisfiable. But $F(\xi') \wedge \neg\psi'$ is the same as $\neg(F(\xi') \to \psi')$, which completes the proof.
6. ψ is $(\exists x)\psi'$. To show correctness it suffices, by Lemma 3.10, 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 3.6, 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 an element $a \in D$, such that $(a_1, \ldots, a_n, a)$ satisfies ψ'. By Lemma 3.8, 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 3.6 together with the uniqueness part of Lemma 3.8, it follows that ξ'' is an extension of ξ'.

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

Lemma 3.12 The result of EVAL$_\phi$ is equivalent to the formula ϕ.

Proof: Let $S = \{\xi_1, \ldots, \xi_n\}$ be the set of those configurations for which of $F(\xi_i) \to \phi$ is valid. Clearly, $\bigvee_{1 \leq i \leq n} F(\xi_i) \to \phi$ 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 3.8, 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 3.10, it follows that $\xi \in S$, completing the proof. □

We still have to show that:

Lemma 3.13 EVAL_ϕ can be implemented in LOGSPACE.

Proof: First, consider Boolean-EVAL_ϕ. The first problem we have to address is that the algorithm assumes that we have constructed the formula ϕ, which cannot be done in LOGSPACE. The solution is 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. This can be easily done with only a constant extra memory cost.

To run Boolean-EVAL_ϕ 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 Boolean-EVAL_ψ 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 Boolean-EVAL_ψ 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 Boolean-EVAL_ψ is a LOGSPACE algorithm.

The algorithm EVAL_ϕ iterates over all configurations ξ, and performs Boolean-EVAL_ϕ on each one. As before, we can easily iterate over all configurations in LOGSPACE, and this shows that EVAL_ϕ is also in LOGSPACE. $\Box$

Now we can show that:

Theorem 3.14

1. The relational calculus with dense linear order inequality constraints can be evaluated bottom-up in closed form with LOGSPACE data complexity.
2. Inflationary Datalog$^\neg$ with dense linear order inequality constraints can be evaluated bottom-up in closed form and PTIME data complexity.

Proof: The first part of the theorem follows from Lemmas 3.12–3.13. Note that, EVAL_ϕ proceeds by structural induction on ϕ and all calls to its outermost for can proceed in parallel.

For the semantics of a query program π, of Inflationary Datalog$^\neg$ with dense linear order constraints, we have to iterate a relational calculus formula ϕ_π. We can use EVAL_ϕ of the first part of the theorem 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 dense linear order constraints. These iterations proceed in a bottom-up fashion. $\Box$

A final observation involves the expressive power of Inflationary Datalog$^{\neg}$ with dense linear order inequality constraints.

Theorem 3.15 Inflationary Datalog$^{\neg}$ with dense linear order inequality constraints can express any relational database query computable in PTIME (for a formal definition of these queries see [11]).

Proof: As shown in [26] and [60] the fixpoint queries of [11] together with a finite discrete linear order express exactly PTIME. It follows from the proofs of this fact, as well as the normal form results of [26, 22, 1] that Inflationary Datalog$^{\neg}$ with a finite discrete order expresses exactly PTIME. These simulation proofs can be easily modified, by making all programs use only constants appearing in the database. □

3.2 Datalog Bottom-up Evaluation Revisited

Let us now consider an alternative proof for the Datalog case of Theorem 3.14. The main idea comes from the semantics of Constraint Logic Programming [27]. It involves generalizing the notion of a Herbrand atom. The result is a "natural" bottom-up evaluation for Datalog with dense linear order constraints.

Definition 3.16 Let P be a *generalized database logic program*, that is, a Datalog + 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 dense linear 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 3.17 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 3.2, 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.

The evaluation of a generalized database logic program P starts by collecting in a set H all the predicate, variable, and dense linear 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 3.18 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 dense linear order constraints.

Without loss of generality the n occurrences of variables in the relational atoms are distinct and any equalities are part of C.

1. Choose any r-configuration ξ of size n built from H.
2. Check that $F(\xi) \to C$ is valid, i.e., true for all values of the free variables.
3. If A_i, $1 \le i \le k$ 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 $R(\ldots)$:— ψ in I we have that $F(\xi_i) \to \psi$ is valid.
4. If $A_i, 1 \le i \le k$ 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 3.9 and 3.10.

(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 ϕ_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 3.9 and 3.10: $F(\xi_i) \rightarrow \phi_r$ 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 3.19 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.
5. 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 the semantics in Section 1 we interpreted programs as mappings from unrestricted relations to unrestricted relations. These semantics are computable via *naive evaluation* of Datalog rules on unrestricted relations.

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 input unrestricted relations then its generalized database output L_P finitely represents the output of naive evaluation on the input unrestricted relations.

Theorem 3.20 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 output 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 $points(L_P(d_1))$ is also in $L_P(d_2)$, and the completeness direction, that is, any point p in $L_P(d_2)$ is also in $points(L_P(d_1))$.

We prove each direction by induction on the number of iterations i of the two evaluations. We shall show that for each i, $points(T_P^i(d_1)) = T_P^i(d_2)$. 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 $points(T_P^0(d_1)) = points(d_1) = d_2 = T_P^0(d_2)$, hence the claim holds. Now we assume that $points(T_P^i(d_1)) = T_P^i(d_2)$ 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 dense linear 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, $points(T_P^i(d_1)) = T_P^i(d_2)$. 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, $points(T_P^{i+1}(d_1)) \subseteq T_P^{i+1}(d_2)$.

For the completeness direction, let A_0 :— $A_1, A_2, \ldots, A_k, C$ be any rule of P where C are any dense linear order constraints. To perform a rule firing in naive evaluation, assume that some 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 3.8 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 3.9 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 $T_P^{i+1}(d_2) \subseteq points(T_P^{i+1}(d_1))$. $\square$

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 labeled as follows:

1. A labels the root.
2. Every leaf is labeled by a generalized EDB Herbrand atom of P.
3. Every internal node is labeled 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 [55] for the exact definition. Following [55] one can show that piecewise linear programs always have the (generalized) polynomial fringe property.

Theorem 3.21

1. Datalog programs with dense linear 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 dense linear 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 [55] 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$^\neg$ + dense linear order constraints.

4 Equality Constraints over an Infinite Domain

Throughout Section 4, we assume that the constraint domain D is the set of integer numbers, but our analysis applies to any countably infinite set. In a sense, we have a special case of Datalog$^{\neg}$ and dense linear order constraints. We present a separate analysis because for dense linear order constraints we expressed $\neq$ using $<$, but here we cannot.

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. This is a significant restriction in the relational model, since there is no way to deal with these queries. However, in our generalized setting, the problem goes away. as long as the result has a finite representation of the appropriate kind.

For the relational calculus and Datalog$^{\neg}$ applied to finite relations there are similar results in the literature [3, 31, 25]. Our work extends some of the results of [3, 31, 25], by adding recursion and handling any input generalized relation. In fact, the specific number of constants added in [3] and [25] 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 show similar results to the dense linear order case, namely that the relational calculus with equality constraints can be evaluated bottom-up in closed form and LOGSPACE data complexity, and that inflationary Datalog$^{\neg}$ with equality constraints can be evaluated bottom-up in closed form and PTIME data complexity.

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 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 $\circ$ whose meaning will be explained below.

Definition 4.1 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 $D_\phi \cup \{\circ\}$, such that,

1. If $i \,\overline{e}\, j$, then $v_i = v_j$, and
2. If $v_i = v_j \neq \circ$, then $i \,\overline{e}\, j$. $\square$

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 $\circ$ whose meaning is that x_i is not equal to any constant in D_ϕ.

Example 4.2 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 $\overline{v} = (1, 1, 2, \circ, 2, \circ, \circ)$. $\square$

We now proceed as in Section 3.1.

Definition 4.3 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 $v_i \neq \circ$, and (4) for all v in D_ϕ, $x_i \neq v$ whenever $v_i = \circ$. $\square$

Definition 4.4 $\models F(\xi)(a_1, \ldots, a_n)$ means that $F(\xi)$ is satisfied by the assignment of each a_i to the corresponding variable x_i.$\square$

Definition 4.5 Let $\xi = (\overline{e}, \overline{v})$ be an e-configuration of size n. An e-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. $\overline{v}' = (v_1, \ldots, v_n, v'_{n+1})$.$\square$

Lemma 4.6 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 $F(\xi')$.$\square$

Lemma 4.7 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 4.6.$\square$

Lemma 4.8 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 4.9 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 $x_i = x_j$ or $\neg\psi_1$ 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 4.8, 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 4.6 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 4.10 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 4.7 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 4.9 implies that ψ is satisfied by $(a'_1, \ldots, a'_n)$. □

In the same way as we did for dense linear order, we can define algorithms EVAL_ϕ and Boolean-EVAL_ψ. These algorithms are similar to those for dense order, but use e-configurations. For example, the algorithm Boolean-EVAL_ψ 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 Boolean-$\text{EVAL}_\psi(\xi)$ are different from the dense linear order case:

1. ψ is an atomic formula $x_i = x_j$. Boolean-$\text{EVAL}_\psi(\xi)$ returns 1 iff $i \, \overline{e} \, j$ in ξ.

2. ψ is an atomic formula $x_i = c$. Boolean-$\text{EVAL}_\psi(\xi)$ 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 dense linear order. Also, for equality constraints we can develop constraint logic programming machinery analogous to that of the previous sections. Therefore:

Theorem 4.11

1. Relational calculus with equality constraints over an infinite domain can be evaluated bottom-up in closed form and LOGSPACE data complexity.

2. Inflationary $\text{Datalog}^\neg$ with equality constraints over an infinite domain can be evaluated bottom-up in closed form and PTIME data complexity. □

5 Boolean Equality Constraints

In this section, we describe the addition of boolean equality constraints to Datalog. The idea is to use boolean operations as shorthands for manipulating various boolean domains, finite or infinite.

In order to make the material self-contained, we first give some definitions and basic facts about boolean algebras (Section 5.1). In Section 5.2, we describe the syntax and semantics of a simple language, which we motivate using some examples. Note that, in this section, we "parameterize" the concepts of database and general database. This allows more generality and more flexibility in the language. Finally, in Section 5.3, we describe some lower bounds related to bottom-up evaluation of fixed size programs, i.e., related to data complexity.

5.1 Boolean Algebras

A *boolean algebra* B is a sextuple $< D, \wedge, \vee, ', 0, 1 >$, where D is a set, $\wedge$, $\vee$ are binary functions, $'$ is a unary function and 0, 1 are two specific elements of D (or zeroary functions) such that for any elements x, y, and z in D the following axioms hold:

$$\begin{array}{rclrcl}
x \vee y & = & y \vee x & x \wedge y & = & y \wedge x \\
x \vee (y \wedge z) & = & (x \vee y) \wedge (x \vee z) & x \wedge (y \vee z) & = & (x \wedge y) \vee (x \wedge z) \\
x \vee x' & = & 1 & x \wedge x' & = & 0 \\
x \vee 0 & = & x & x \wedge 1 & = & x \\
0 & \neq & 1 & & &
\end{array}$$

For boolean algebras there is a representation theorem, known as *Stone's theorem*: "Every boolean algebra is isomorphic to a field of sets and every finite boolean algebra is isomorphic to the power set of a finite set". Thus, there is a unique (up to isomorphism) finite boolean algebra for every cardinality 2^m. The boolean algebra of cardinality 2^{2^m} is *the one freely generated by m generators* and is denoted by B_m. For $m = 0$, we have $B_0 = \langle\{0,1\}, \wedge, \vee, ', 0, 1\rangle$.

Let $\langle D, \wedge, \vee, ', 0, 1\rangle$ be any boolean algebra. Then the structure $\langle D, \wedge, \oplus, 0, 1\rangle$, is called a *boolean ring with unity* if we define for any elements x and y the binary function $x \oplus y$ (exclusive-or) as $(x \wedge y') \vee (x' \wedge y)$. Because of one-to-one correspondence between boolean algebras and boolean rings with unity the theory here can be developed in either setting [40]. In what follows we use algebras and, sometimes, the exclusive-or as an abbreviation.

Boolean Terms: We use $T(F, V \cup C)$, for the set of *terms* built in the usual way, from F the set of function symbols $\{\wedge, \vee, ', 0, 1\}$, V a set of variable symbols, and C a of constant symbols distinct from $0, 1$. *Ground terms* are those terms which do not have any variable symbols appearing in them. A (B, σ)-*interpretation* is a pair, where B is a boolean algebra and σ is a mapping of the constant symbols C to the elements of B. For each t in $T(F, V \cup C)$, given a (B, σ)-interpretation and an element of B for each variable symbol appearing in t, we can evaluate t in the usual way and have it denote one element of B.

Boolean equations: An *equation* between terms t_1 and t_2 in $T(F, V \cup C)$ is a statement $t_1 = t_2$. We say that: (1) $t_1 = t_2$ *is true in* B, σ if after applying σ to the constant symbols in t_1, t_2 then for every substitution of the variable symbols by elements of B, t_1 and t_2 denote the same element of B. (2) $t_1 = t_2$ *is true in* B if it is true in B, σ for every σ. (3) $t_1 = t_2$ *is true* if it is true in every B. A number of useful properties hold in all boolean algebras. For example, the following equations are true: $x \vee (y \vee z) = (x \vee y) \vee z$, $x \wedge (y \wedge z) = (x \wedge y) \wedge z$, $x \oplus (y \oplus z) = (x \oplus y) \oplus z$. For another useful example:

Lemma 5.1 Let $t(z_1, z_2, \ldots, z_n)$ be a term, where the z's are the distinct variable or constant symbols occuring in it. Then the following equation is true:

$$t(z_1, z_2, \ldots, z_n) = (t(0, z_2, \ldots, z_n) \wedge z_1') \vee (t(1, z_2, \ldots, z_n) \wedge z_1) \Box$$

Disjunctive Normal Form: We use the convention that z^0 means z' and z^1 means z. Also, that the z's are ordered lexicograpically from z_1 to z_n. Then, we also may write the equation in the previous lemma as $t(z_1, z_2, \ldots, z_n) = \bigvee_{a_1 = \{0,1\}} (t(a_1, z_2, \ldots, z_n) \wedge z_1^{a_1})$. By repeatedly using the above lemma and the nine boolean algebra axioms, it is possible to transform each term into the following *disjunctive normal form*:

$$t(z_1,\ldots,z_n) = \bigvee_{\overline{a}=\{0,1\}^n} (t(a_1,\ldots,a_n) \wedge z_1^{a_1} \wedge z_2^{a_2} \wedge \ldots \wedge z_n^{a_n})$$

where $\bigvee_{\overline{a}=\{0,1\}^n}$ denotes the disjunction of all 0, 1 substitutions for $a_1,\ldots,a_n$. The function determined by $t(z_1,\ldots,z_n)$ depends only on the values of the 2^n expressions $t(a_1,\ldots,a_n)$, where each a_i is either 0 or 1. One can see that each of these 2^n expressions has value either 0 or 1. Hence, it is possible to see that there are 2^{2^n} disjunctive normal forms with n variable and constant symbols.

Constructing B_m: We give a simple example of how to construct the free boolean algebra B_m out of a set of m constant symbols $C = \{c_1,\ldots,c_m\}$. First we build all possible ground terms. Next we find all the equivalence classes of ground terms under the boolean algebra axioms. Each equivalence class is an element of B_m and corresponds to a disjunctive normal form. There are 2^{2^m} distinct equivalence classes or elements of B_m. We call the constant symbols the generators. (Note that, naively we would have used 2^m bits to represent every element of B_m, but this way we can use $\log m$ bits for each generator).

Definition 5.2 A *boolean equality constraint* is a statement of the form $t(\overline{x},\overline{c}) =_{B,\sigma} 0$, for a (B,σ)-interpretation and a term t built using a set of variables $\overline{x}$ and a set of constants $\overline{c}$. (We sometimes omit subscript σ if it is obvious from the context).

Constraint $t(\overline{x},\overline{c}) =_{B,\sigma} 0$ has a solution if the formula $\exists\overline{x}(t(\overline{x},\overline{c}) = 0)$ is true in B,σ (i.e., if after applying σ to the constant symbols in t, there exists a substitution of the variable symbols by elements of B that makes t denote the 0 element of B). A *solution* of a constraint is a substitution that makes t denote 0. □

It is always enough to use boolean constraints of the form above, because the general constraint $t_1 =_{B,\sigma} t_2$ has the same solutions as the constraint $t_1 \oplus t_2 =_{B,\sigma} 0$, just by using the boolean algebra axioms and the definition of $\oplus$.

A useful fact about solving equations in boolean algebras is Boole's Lemma, which can be proven using the nine boolean algebra axioms and the disjunctive normal form described above. Boole's Lemma gives a simple way to eliminate existential quantifiers.

Lemma 5.3 The boolean equality constraint $t(x_1,x_2,\ldots,x_n,\overline{c}) =_{B,\sigma} 0$ has a solution iff the constraint $t(1,x_2,\ldots,x_n,\overline{c}) \wedge t(0,x_2,\ldots,x_n,\overline{c}) =_{B,\sigma} 0$ has a solution iff $\bigwedge_{\overline{b}\in\{0,1\}^n} t(\overline{b},\overline{c}) = 0$ is true in B,σ. Furthermore, the set of substitutions for $x_2,\ldots,x_n$ that make the formula $\exists x_1(t(x_1,x_2,\ldots,x_n,\overline{c}) = 0)$ true in B,σ is the same as the set of solutions of constraint $t(1,x_2,\ldots,x_n,\overline{c}) \wedge t(0,x_2,\ldots,x_n,\overline{c}) =_{B,\sigma} 0$. □

Remark F: In Lemma 5.3 the expression $\bigwedge_{\overline{b}\in\{0,1\}^n} t(\overline{b},\overline{c})$ denotes the conjunction of all 0, 1 substitutions into the variable symbols of $t(x_1,x_2,\ldots,x_n,\overline{c})$ *and is the same expression for all* B,σ. Note that the correctness of Boole's Lemma is trivial if B happens to be B_0, otherwise the conjunction may well be 0 even if none of its conjuncts are. □

5.2 Bottom-Up Evaluation of Datalog with Boolean Equality Constraints

The following evaluation method applies to any B, σ-interpretation. We only describe a language for the boolean equality constraint part. This language can easily be combined with the one for dense linear order constraints, as we illustrate in the last example of this section.

Syntax: Here, 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 $\{x_1, \ldots, x_j, y_1, \ldots, y_l\}$, and $\psi_0(\overline{x}) =_{B,\sigma} 0$ and $\psi_{k+1}(\overline{x}, \overline{y}) =_{B,\sigma} 0$ are boolean equality constraints.
(1) The *facts* (or generalized tuples) have the form, $R_0(\overline{x}) :\!\!-\; \psi_0(\overline{x}) =_{B,\sigma} 0$.
(2) The *rules* have the form: $R_0(\overline{x}) :\!\!-\; R_1(\overline{x}, \overline{y}), \ldots, R_k(\overline{x}, \overline{y}), \psi_{k+1}(\overline{x}, \overline{y}) =_{B,\sigma} 0$.
In these facts and rules all the $\overline{x}$ variables that appear in the body appear in the head. The $\overline{y}$ variables appear only in the body.

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.

One constraint suffices per fact/rule: Because, from the axioms one can show the equivalences: (1) $a =_{B,\sigma} b$ has the same set of siolutions as $a \oplus b =_{B,\sigma} 0$, and (2) $a =_{B,\sigma} 0$ and $b =_{B,\sigma} 0$ have the same set of solutions as $a \vee b =_{B,\sigma} 0$, which has the same set of solutions as $a \oplus b \oplus (a \wedge b) =_{B,\sigma} 0$. 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 $R_i(\overline{x}, \overline{y}) :\!\!-\; \psi_i(\overline{x}, \overline{y}) =_{B,\sigma} 0 \; 1 \leq i \leq k$, and that we have the rule $R_0(\overline{x}) :\!\!-\; R_1(\overline{x}, \overline{y}), \ldots, R_k(\overline{x}, \overline{y}), \psi_{k+1}(\overline{x}, \overline{y}) =_{B,\sigma} 0$. We produce a quantified description of the set of tuples from B that such a firing will derive.

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

¿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 formula $\exists y\, \psi(y, \overline{x}) = 0$ is satisfiable in B, σ iff $\psi(0, \overline{x}) \wedge \psi(1, \overline{x}) =_{B,\sigma} 0$ has a solution. This allows us to add R' in unquantified single equation form to the derived facts. (Note that, here we use Boole's Lemma to eliminate quantifiers instead of 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 disjunctive normal forms.

We present now the simple "adder circuit" example of Buttner and Simonis [10], but in this case we use bottom-up evaluation to illustrate the previous method.

Example 5.4 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 finite boolean equality constraints over B_0. Note that here σ is empty and we omit it.

$$\mathit{Halfadder}(x, y, z, w) :\!\!— x \oplus y =_{B_0} z, x \wedge y =_{B_0} w$$

where x and y are input variables, z is the sum and w is the carry. Turning the two constraints in the body into a single equivalent one we have:

$$\mathit{Halfadder}(x, y, z, w) :\!\!— (x \oplus y \oplus z) \vee ((x \wedge y) \oplus w) =_{B_0} 0$$

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.

$$\mathit{Adder}(x, y, c, s, d) :\!\!— \mathit{Halfadder}(x, y, s_1, c_1), \mathit{Halfadder}(s_1, c, s, c_2), d =_{B_0} c_1 \vee c_2$$

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:

$$\begin{aligned}\mathit{Adder}(x, y, c, s, d) :\!\!— (c_1 \vee c_2) \oplus d =_{B_0} 0, (x \oplus y \oplus s_1) \vee ((x \wedge y) \oplus c_1) =_{B_0} 0,\\ (s_1 \oplus c \oplus s) \vee ((s_1 \wedge c) \oplus c_2) =_{B_0} 0\end{aligned}$$

By transforming the three constraints in the body into one and using Boole's lemma to eliminate s_1, c_1, c_2 we can transform the above into:

$$\mathit{Adder}(x, y, c, s, d) :\!\!— (x \oplus y \oplus c \oplus s) \vee ((x \wedge y) \oplus (x \wedge c) \oplus (y \wedge c) \oplus d) =_{B_0} 0$$

□

Example 5.5 Suppose we replace the variable symbols x, y and c in Example 5.4 by the constant symbols $\mathcal{X}, \mathcal{Y}, \mathcal{C}$. Then we get expressions for a subset of the variables in the constraint, that is, for s and d, in terms of the constant symbols, that is, the parameters, $\mathcal{X}, \mathcal{Y}, \mathcal{C}$. For example, one solution would be $s =_{B_0} \mathcal{X} \oplus \mathcal{Y} \oplus \mathcal{C}$ and $d =_{B_0} (\mathcal{X} \wedge \mathcal{Y}) \oplus (\mathcal{X} \wedge \mathcal{C}) \oplus (\mathcal{Y} \wedge \mathcal{C})$. This says that given any σ that interprets $\mathcal{X}, \mathcal{Y}, \mathcal{C}$ as elements of B_0, then s and d will be computed according to the last two constraints. (See also Remark G below). □

Next we show that this evaluation method works in general by the following theorem.

Theorem 5.6 Let Q be any query program of Datalog with boolean equality constraints over some B, σ. Then Q can be evaluated bottom-up in closed form.

Proof: Let v be the maximum arity of a relation in the given query program. Let m be the number of constant symbols in the program and the input database. Our evaluation method will consist of a number of iterative steps. In each step we add all new facts that can be derived from the already known facts and rules. By the proper substitutions of database facts into the rules we get formulas on the right-hand side of the rules. From these formulas we eliminate existentially quantified variables using Lemma 5.3.

We also have to show that the procedure terminates. To do that, we always keep every fact in disjunctive normal form. Note that Lemma 5.3 does not introduce any new constant symbols, hence the number of constant symbols m does not change during evaluation. The quantifier elimination yields constraints that have up to m constant symbols and v variables. That means that the constraints for each relation R can be represented by at most $2^{2^{v+m}}$ facts by counting only disjunctive normal forms. Hence, after each iteration step, every newly derived constraint can be compared easily with facts already present in the database. If all newly derived constraints are already present, then the iteration can stop, otherwise we add the new constraints. This procedure clearly must terminate because there are only a finite number of facts that can be added. □

A *parametric fact (or parametric generalized tuple)* is a fact (or generalized tuple) for which we have not specified either B or σ or both. A *parametric* generalized database is a set of parametric generalized tuples. (Note the difference between a parametric tuple and a generalized tuple. A generalized tuple describes the set of all tuples that one gets by substituting values for its variables. A parametric tuple describes a single tuple, given a substitution for its parameters).

Example 5.7 As a simple example consider the problem of finding the parity of n bits. By considering these n bits as parameters, a simple solution would be the following:

$$Paritybit(x) :\!\!- \; x =_{B_0,\sigma} \mathcal{Y}_1 \oplus \mathcal{Y}_2 \oplus \ldots \oplus \mathcal{Y}_n$$

Each different σ would be a different assigment of values from B_0 to these n parameters. So if we do not specify σ this would be a parametric fact. □

Remark G: For the procedure described in Theorem 5.6, the particular B, σ in which the constants are interpreted are not important. Syntactically, the same constraints are derived by the evaluation algorithm for each B, σ. This is really a consequence of Lemma 5.3 and Remark F. Therefore, we have that:

The evaluation of Theorem 5.6 can be applied to an input parametric generalized database d to produce an output parametric generalized database $Q(d)$, such that for each $I = (B, \sigma)$ we have $I(Q(d)) = Q(I(d))$. □

¿From a practical point of view, boolean equality constraints can be added on top of the Datalog framework with dense linear order that we already examined in Section 3.2. We can strictly sort the arguments of each database predicate, e.g., each argument can range either over the rationals or over a finite boolean domain. All of our results still hold in this combined framework.

Example 5.8 If the number of bits n of the previous example can vary, then we do not want to write a new program each time n changes. We would like to have a fixed program and change only the input (generalized) database or the input parametric (generalized) database. This we could do with the following program:

$$Paritybit(x) :\!\!- Parity(k,x), Last(k)$$

$$Parity(i,x) :\!\!- Parity(j,y), Next(j,i), Input(i,z), x =_{B_0,\sigma} y \oplus z$$

$$Parity(1,x) :\!\!- Input(1,z)$$

where we use $Next(1,2), \ldots, Next(n-1,n)$, $Last(n)$, and $Input(1, \mathcal{Y}_1), \ldots, Input(n, \mathcal{Y}_n)$ as the parametric database inputs to the Datalog with boolean and dense linear order constraints program. We use in this example n explicitly given elements of the nonboolean domain to order the parity computation. Then the program recursively finds the parity bit for the first i bits for $1 \leq i \leq n$. □

5.3 Data Complexity and Finite Boolean Equality Constraints

For each fixed finite boolean algebra B, fixed Datalog query Q with constraints over B, σ, and variable input generalized database d with constraints over B, σ, query evaluation is a constant size problem.

This is because: Given an input generalized database d (even with an asymptotically growing number of constant symbols) and a fixed finite boolean algebra B and a σ, it is possible to make the number of constant symbols fixed by substituting for them elements of B using σ. B has a constant number of elements and Q has a constant number of variables, because they are fixed. So the constraints can be eliminated by substituting the variables with elements of B, and checking in which substitutions the constraints hold. Each time the constraint holds, record the database tuple implied. That yields a database d' that is equivalent to generalized database d. The size of d' is at most a constant. Hence the query evaluation is in this case a trivial problem. For example, the adder circuit can be described using width 5 constant size relations over 0,1 that define bit addition.

Recall, however, that the procedure described in Theorem 5.6 is "parametric", i.e., syntactically the same constraints are derived by the evaluation algorithm for each B, σ. This might be wasteful for the case of fixed B and Q, but it is a general method applicable when B is part of the problem input and when we do not know σ a priori (so we have to manipulate the constants symbolically).

We now take an algebra that is closer to the idea of our "parametric" evaluation than a fixed algebra. Namely, we assume that the algebra is part of the input. In particular we assume that the algebra is B_m where m is the number of constant symbols $c_1, \ldots, c_m$ in the input database and the program. In the rest of the exposition, we assume that σ maps $c_1, \ldots, c_m$ to the generators of B_m, so we omit the subscript σ in constraints.

Lemma 5.9 Let $\psi(x_1, \ldots, x_n, y_1, \ldots, y_m)$ be a term over variables $x_1, \ldots, x_n$, $y_1, \ldots, y_m$ and the function symbols $\wedge, \vee, '$. Let B_m be the free boolean algebra generated by the constant symbols $c_1, \ldots, c_m$. Then the formula $\forall_{\overline{y}} \exists_{\overline{x}} \psi(x_1, \ldots, x_n, y_1, \ldots, y_m) = 0$ is true in B_0 if and only if the constraint $\psi(x_1, \ldots, x_n, c_1, \ldots, c_m) =_{B_m} 0$ has a solution.

Proof: Note that in this proof we have to reduce a decision problem of size l to another decision problem of size l, where l is the size of the formula ψ. We do the proof by rewriting the first problem to the second one, using identities, i.e. we will use iff steps in the reduction instead of proving both directions separately.

First, we rewrite $\forall_{\overline{y}} \exists_{\overline{x}} \psi(x_1, \ldots, x_n, y_1, \ldots, y_m) = 0$ is true in B_0 into:

$\forall_{\overline{y} \in \{0,1\}^m} (\exists_{\overline{x} \in \{0,1\}^n} \psi(x_1, \ldots, x_n, y_1, \ldots, y_m) = 0)$ is true in B_0 by recognizing that each variable must be either 0 or 1 and by adding parenthesis for clarity.

Note that the subformula within the parenthesis has a solution for the y's if and only if the conjunction of all $0, 1$ substitutions into the x variables has a solution for the y's , that is, if and only if, $\forall_{\overline{y} \in \{0,1\}^m} (\bigwedge_{\overline{x} \in \{0,1\}^n} \psi(x_1, \ldots, x_n, y_1, \ldots, y_m) = 0)$ is true in B_0.

The last formula is true if and only if each of the 2^m subproblems that result from $0, 1$ substitutions to the y variables is true. By using the identity that $a =_B 0$ and $b =_B 0$ iff $a \vee b =_B 0$, the last formula can be rewritten as $\bigvee_{\overline{y} \in \{0,1\}^m} (\bigwedge_{\overline{x} \in \{0,1\}^n} \psi(x_1, \ldots, x_n, y_1, \ldots, y_m)) = 0$ is true in B_0.

Claim: The last formula is equivalent to $\bigwedge_{\overline{x} \in \{0,1\}^n} \psi(x_1, \ldots, x_n, c_1, \ldots, c_m) = 0$ is true in B_m. (This claim is the basis of Martin and Nipkow's method [40], but we can prove it in a simple way as follows).

To prove the claim rewrite the term in the claim into disjunctive normal form. We get $\bigvee_{\overline{y} \in \{0,1\}^m} (\bigwedge_{\overline{x} \in \{0,1\}^n} \psi(x_1, \ldots, x_n, y_1, \ldots, y_m)) \wedge c_1^{y_1} \wedge \ldots \wedge c_m^{y_m}$. Note that in B_m the constant symbols are distinct, and their complements are distinct from each other and from 0 and 1. (It is a well-known fact that each element of a boolean algebra has only one complement.) Hence none of the last m conjuncts is either 0 or a complement of another one among the last m conjuncts. (These are the only ways we can prove a conjunction to be equivalent to 0.) Therefore the conjunction of the last m conjuncts is never 0. Hence, the first conjunct, which could be only 0 or 1, must be always 0 if the whole formula is 0 in B_m. Hence, the formula can be 0 in B_m if and only if $\bigvee_{\overline{y} \in \{0,1\}^m} (\bigwedge_{\overline{x} \in \{0,1\}^n} \psi(x_1, \ldots, x_n, y_1, \ldots, y_m)) = 0$ is true in B_m. Since we don't have any constant symbols in the formula, instead of B_m we may use B_0, which proves the claim.

So far we have rewritten the formula to $\bigwedge_{\overline{x} \in \{0,1\}^n} \psi(x_1, \ldots, x_n, c_1, \ldots, c_m) = 0$ is true in B_m. Finally, we use Boole's Lemma to get the formula $\exists_{\overline{x} \in \{B_m\}^n} \psi(x_1, \ldots, x_n, c_1, \ldots, c_m) = 0$ is true in B_m, which is equivalent to saying that $\psi(x_1, \ldots, x_n, c_1, \ldots, c_m) =_{B_m} 0$ has a solution. $\square$

Note that, we cannot guess and verify a solution of $\psi(x_1, \ldots, x_n, c_1, \ldots, c_m) =_{B_m} 0$ that is small. The size of each element of B_m can be very large. In fact:

Corollary 5.10 Given as inputs any t and m, where t is any boolean formula with constant symbols $c_1, \ldots, c_m$, deciding whether $t =_{B_m} 0$ has a solution is Π_2^p-complete (where B_m is the free boolean algebra generated by the constant symbols $c_1, \ldots, c_m$). $\square$

Theorem 5.11 There is a fixed yes/no query program Q in Datalog with finite boolean equality constraints such that: If for each input database d, with constant symbols $c_1, \ldots, c_m$, we take the σ that maps these constant symbols to themselves and we interpret B as B_m then deciding whether $Q(d)$ is yes is Π_2^p-hard.

Proof: Our reduction will use the AE-quantified boolean formula problem (see Section 1.2). We will produce a yes/no query expressed in Datalog with boolean equality constraints over the finite boolean algebra B_m, which is the free boolean algebra generated by m constant symbols, such that the query answer is $\mathcal{YES}$ iff the quantified boolean formula $\forall_{\overline{y}} \exists_{\overline{x}} \psi(\overline{x}, \overline{y}) = 0$ is true in B_0. In this reduction $m = O(|\psi|)$. Assume $\overline{x} = \{x_1, \ldots, x_n\}$ and $\overline{y} = \{y_1, \ldots, y_p\}$.

When the terms substituted for the variables $x_1, \ldots, x_n$ are terms that contain y variables, the constant symbols 0 and 1, and the boolean operators $\wedge, \vee, '$ in them, then the substitution is called a *parametric* solution. To check whether the quantified boolean formula holds, it is enough to check whether it has a parametric solution. (To see this, suppose that the formula is true. Then there is a solution of x's for any assignment of y's. Take any x_i. For each assignment A of y, x_i is either 0 or 1. We could build a simple formula that says "if A then x_i is 0" or "if A then x_i is 1" as appropriate. We could therefore build a parametric solution for each x_i.)

Our reduction will proceed in four steps. In this reduction we will use script letters for constant symbols.

(1) We build a "tree" circuit for $\psi(\overline{x}, \overline{y})$ using some number of gates. The gates will be referred to by the constants $\mathcal{G}_1, \ldots, \mathcal{G}_{top}$ with $\mathcal{G}_{top}$ as the output gate of the whole circuit. For each x_i we create a new constant symbol $\mathcal{B}_i$ and substitute the x variables by these constant symbols while creating the circuit. Similarly, for each y_i we create a new constant symbol $\mathcal{A}_i$ and substitute the y variables by these constant symbols while creating the circuit. We also have a constant $\mathcal{YES}$. So we have as constants the $\mathcal{A}$'s, $\mathcal{B}$'s, $\mathcal{G}$'s, and $\mathcal{YES}$.

In the reduction we will present a query Q that uses a number of input database relations, called EDB relations, and a number of initially empty output database relations, called IDB relations (following standard database terminology). We have *Input*, *Top*, *Andgate*, *Orgate*, *Notgate*, A, B, *Next* and *Last* as EDB relations. We have as IDB relations, *Value*, *Aexpr*, *Replace*, *Parametric* and *Output*. The yes/no output is carried in *Output*, i.e., yes if *Output*($\mathcal{YES}$) and no otherwise.

Suppose that formula $\psi(\overline{x}, \overline{y})$ has a total of l occurrences of variables in it. We enter for each occurrence of y_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 variable in the formula is y_i (or x_j), we create, using the EDB database predicate *Input*, the database fact:

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

or

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

We also declare $\mathcal{G}_{top}$ as the output gate of the circuit, using the EDB database predicate *Top*:

$$Top(\mathcal{G}_{top})$$

Next we look at the parse tree of the formula ψ. We tag each boolean operator $\wedge, \vee, '$ 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 $\wedge$ (or an $\vee$ or an $'$) 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 (for a $\wedge$ and an $\vee$ gate only) an operator tagged by gate $\mathcal{G}_k$, we create, using one of the EDB database predicates *Andgate*, *Orgate*, or *Notgate*, the database fact:

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

or

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

or

$$Notgate(\mathcal{G}_i, \mathcal{G}_j)$$

The term corresponding to $\psi(\overline{x}, \overline{y})$ is built bottom-up by the evaluation method, using the IDB predicate *Value* and the following rules:

$$Value(k, z) :\!\!-\; Input(k, z)$$

$$Value(k, z) :\!\!-\; Andgate(k, i, j),\; Value(i, x),\; Value(j, y),\; z =_{B_m} x \wedge y$$

$$Value(k, z) :\!\!-\; Orgate(k, i, j),\; Value(i, x),\; Value(j, y),\; z =_{B_m} x \vee y$$

$$Value(k, z) :\!\!-\; Notgate(k, i),\; Value(i, x),\; z =_{B_m} x'$$

The bottom-up evaluation derives as the value of the topmost gate $\mathcal{G}_{top}$ some z_{top} such that $z_{top} =_{B_m} \psi(\overline{\mathcal{B}}, \overline{\mathcal{A}})$. As the rules are evaluated variables i, j, x, y are eliminated but constant symbols $\mathcal{G}_{top}, \mathcal{A}, \mathcal{B}$ remain.

Remark i: We know that we have created *Value* so that $Value(\mathcal{G}_{top}, \psi(\overline{\mathcal{B}}, \overline{\mathcal{A}})$ is in the fixpoint. We will now proceed to construct in the fixpoint of *Axpr* all ground terms e that contain only $0, 1, \mathcal{A}$'s and the boolean operators $\wedge, \vee, '$. We will then replace the $\mathcal{B}$'s in the fixpoint, by all possible such e's in *Replace* and in *Parametric*.

Remark ii: We also know that the constraint $\psi(\overline{e}, \overline{\mathcal{A}}) =_{B_m} 0$ has a solution for the e's in B_m if and only if $\exists_{\overline{x} \in \{B_m\}^n} \psi(\overline{x}, \overline{\mathcal{A}}) = 0$ is true in B_m. By Lemma 5.9 that last formula is true if and only if $\forall_{\overline{y}} \exists_{\overline{x}} \psi(x_1, \ldots, x_n, y_1, \ldots, y_p) = 0$ is true in B_0, which is our original quantified boolean formula problem.

Therefore finding the e's corresponds to finding a parametric solution. We show in the next three steps how a Datalog query can try all possible substitutions that may yield a parametric solution. The proof will follow from Remarks i-ii above.

(2) We create two unary EDB database relations $A(x)$ and $B(x)$ to store all $\mathcal{A}$'s and $\mathcal{B}$'s respectively.

We also create a unary IDB relation *Aexpr(e)*, which when evaluated bottom-up will contain in it all the terms of the algebra which can be written with only $0, 1, \mathcal{A}$'s and the boolean operators $\wedge, \vee, '$ in them.

$$Aexpr(e) :\!\!-\; e =_{B_m} 0$$

$$Aexpr(e) :\!\!- e =_{B_m} 1$$

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

$$Aexpr(e) :\!\!- Aexpr(e_1), Aexpr(e_2), e =_{B_m} e_1 \wedge e_2$$

$$Aexpr(e) :\!\!- Aexpr(e_1), Aexpr(e_2), e =_{B_m} e_1 \vee e_2$$

$$Aexpr(e) :\!\!- Aexpr(e_1), e =_{B_m} e_1'$$

The relation *Aexpr* has many elements. In fact it will consist of all elements that can be written using the boolean operators and the boolean constant symbols $\mathcal{A}_1, \ldots, \mathcal{A}_p$. There are 2^{2^p} distinct elements because there are that many disjunctive normal forms of boolean formulas over those constant symbols. This is large, but still a finite number.

(3) Our main relation for replacement is the IDB relation $Replace(a, \mathcal{B}, c, d)$, whose intended meaning is that if in term a all occurrences of constant $\mathcal{B}$ are substituted by a term c which has only $0, 1$, $\mathcal{A}$'s and the boolean operators $\wedge, \vee, '$ in it, then we get the term d.

The actual bottom-up replacement can be done recursively as follows:

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

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

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

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

$$Replace(o, x, y, n) :\!\!- Replace(o_1, x, y, n_1), Replace(o_2, x, y, n_2), o =_{B_m} o_1 \wedge o_2, n =_{B_m} n_1 \wedge n_2$$

$$Replace(o, x, y, n) :\!\!- Replace(o_1, x, y, n_1), Replace(o_2, x, y, n_2), o =_{B_m} o_1 \vee o_2, n =_{B_m} n_1 \vee n_2$$

$$Replace(o, x, y, n) :\!\!- Replace(o_1, x, y, n_1), o =_{B_m} o_1', n =_{B_m} n_1'$$

(4) To make the substitutions in sequence for each $\mathcal{B}_i$, we order the $\mathcal{B}$ constants, using EDB predicates *Next* and *Last*, and the database facts:

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

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

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

The next two rules using IDB predicate *Parametric* show how in sequence each $\mathcal{B}_i$ can be substituted by terms containing only $0, 1$, $\mathcal{A}$'s and boolean operators $\wedge, \vee, '$.

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

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

When the last $\mathcal{B}$ constant is replaced, there are only $\mathcal{A}$ constants present within *expr*. Hence, we may have one last rule:

$$Output(\mathcal{YES}) :\!\!- Last(k), Parametric(expr, k), expr =_{B_m} 0$$

As shown in Lemma 5.9 deciding whether $expr =_{B_m} 0$ is equivalent to checking whether for each $0, 1$ substitution into the $\mathcal{A}$ constants, the constraint $expr =_{B_0} 0$ has a solution.

Since all possible substitutions of terms containing only $0, 1$, $\mathcal{A}$'s and the boolean operators $\wedge, \vee, '$ for the $\mathcal{B}$'s are tried by the bottom-up evaluation, there is any output iff there is a parametric solution iff $\forall_{\overline{y}} \exists_{\overline{x}} \psi(\overline{x}, \overline{y}) = 0$ is true in B_0. This completes the reduction. □

6 Discussion and Open Questions

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 technical questions remain open. For example: What is the precise data complexity of Datalog with finite boolean equality constraints? Also, is it possible to develop a similar framework for discrete linear order with constants? Note that, progress has been made recently on this question in [46]. Discrete order can be used to model temporal databases. For recent developments of constraint-based approaches to temporal databases we refer to [4, 13, 28]. In Section 2.2, we left open the complexity of tableau containment with dense linear order inequalities. This has been recently shown Π_2^p-complete in [56].

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 data model, many physical access structures, e.g., B-tree indexes, extendible hashing etc, are critical in the implementation of relational databases. In Section 1.1, we argued that there are analogous simple access structures in the more general CQL setting (provided intervals are constraints of the CQL and the projection of any generalized tuple on x is an interval). Is it possible to perform 1-dimensional searching on generalized database attribute x in the same secondary memory access bounds that one uses for 1-dimensional searching on relational database attribute x? Note that, when this problem reduces to on-line interval maintenace, the in-core performance of priority search trees is linear space and logarithmic time. In general, how can grid-files, R-trees, quad-trees and other such structures be used to speed-up CQL evaluation strategies?

(2) The technology of algorithms for logical theories is still rather complex, but much progress has been accomplished in recent years. For example, see [45] for the state-of-the-art in real closed fields. Are there interesting special cases, for which simple algorithmic techniques can be used? These would be analogous to the special treatment of project-select-join query programs in the relational model. In particular, linear inequality constraints should be investigated in a CQL framework.

(3) How do various optimization methods combine with our framework? This would involve extending [44]. For some recent research in this direction we refer to [24, 37, 42, 51]. Constraint

manipulation for database logic program analysis is also related to optimization, e.g., [8, 9, 57].

(4) Constraint query languages 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.

(5) Finally, constraint query languages should be designed with features, such as database types and complex objects. Using such features it might be possible to pose queries about the representation itself and not only about the unrestricted relations represented.

Acknowledgements: We would like to thank Jean-Louis Lassez for his constant encouragement. We would also like to thank Moshe Vardi and Allen Van Gelder for their many constructive comments on the material presented here, as well as Raghu Ramakrishnan and Terry Smith for numerous helpful discussions.

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. Baudinet, M. Niezette, P. Wolper. On the Representation of Infinite Temporal Data and Queries. *Proc. 10th ACM PODS*, 280–290, 1991.

[5] R. Bayer, E. McCreight. Organization of Large Ordered Indexes. *Acta Informatica*, 1:173–189, 1972.

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

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

[8] A. Brodsky, Y. Sagiv. Inference of Monotonicity Constraints in Datalog Programs. *Proc. 8th ACM PODS*, 190–200, 1989.

[9] A. Brodsky, Y. Sagiv. Inference of Inequality Constraints in Logic Programs. *Proc. 10th ACM PODS*, 227–241, 1991.

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

[11] A.K. Chandra, D. Harel. Structure and Complexity of Relational Queries. *JCSS*, 25:1:99–128, 1982.

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

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

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

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

[16] J. Cohen. Constraint Logic Programming Languages. *CACM*, 33:7:52–68, 1990.

[17] A. Colmerauer. An Introduction to Prolog III. *CACM*, 33:7:69–90, 1990.

[18] D. Comer. The Ubiquitous B-Tree. *Computing Surveys*, 11:2:121–137, 1979.

[19] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, F. Berthier. The Constraint Logic Programming Language CHIP. *Proc. Fifth Generation Computer Systems*, Tokyo Japan, 1988.

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

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

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

[23] 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.

[24] R. Helm, K. Marriott, M. Odersky. Constraint-based Query Optimization for Spatial Databases. *Proc. 10th ACM PODS*, 181–191, 1991.

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

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

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

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

[29] P.C. Kanellakis. Elements of Relational Database Theory. *Handbook of Theoretical Computer Science*, Vol. B, chapter 17, (J. van Leeuwen editor), North-Holland, 1990.

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

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

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

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

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

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

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

[37] A. Levy, Y. Sagiv. Constraints and Redundancy in Datalog. *Proc. 11th ACM PODS*, 67–81, 1992.

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

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

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

[41] E. McCreight. Priority Search Trees. *SIAM J. Computing*, 14:257–276, 1985.

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

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

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

[45] J. Renegar. On the Computational Complexity and Geometry of the First-order Theory of the Reals: Parts I–III. *Journal of Symbolic Computation*, 13:255–352, 1992.

[46] P.Z. Revesz. A Closed Form for Datalog Queries with Integer Order. *Proc. 3rd International Conference on Database Theory*, 1990, (to appear in TCS).

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

[48] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS.* Addison-Wesley, Reading MA, 1990.

[49] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, Reading MA, 1990.

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

[51] D. Srivastava, R. Ramakrishnan. Pushing Constraint Selections. *Proc. 11th ACM PODS*, 301–316, 1992.

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

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

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

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

[56] R. van der Meyden. The Complexity of Querying Indefinite Data about Linearly Ordered Domains. *Proc. 11th ACM PODS*, 331–346, 1992.

[57] A. Van Gelder. Deriving Constraints among Argument Sizes in Logic Programs. *Proc. 9th ACM PODS*, 47–60, 1990.

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

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

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