Constraint Databases: A Survey*

Peter Z. Revesz
Department of Computer Science and Engineering
University of Nebraska–Lincoln
revesz@cse.unl.edu

Abstract. Constraint databases generalize relational databases by finitely representable infinite relations. This paper surveys the state of the art in constraint databases: known results, remaining open problems and current research directions. The paper also describes a new algebra for databases with integer order constraints and a complexity analysis of evaluating queries in this algebra.

In memory of Paris C. Kanellakis

1 Introduction

There is a growing interest in recent years among database researchers in constraint databases, which are a generalization of relational databases by finitely representable infinite relations. Constraint databases are parametrized by the type of constraint domains and constraint used. The good news is that for many parameters constraint databases leave intact most of the fundamental assumptions of the relational database framework proposed by Codd. In particular,

1. Constraint databases can be queried by constraint query languages that
   (a) have a semantics based on first-order or fixpoint logics with constraints.
   (b) can be evaluated in closed-form.
   (c) can be algebraic and (efficiently) evaluated set-at-a-time.
2. Constraint databases can be efficiently indexed.
3. Integrity constraints can be enforced on constraint databases.
4. Aggregate operators can be applied to constraint databases.
5. Constraint databases can be extended with indefinite information.
6. Constraint databases can be extended with sets, nesting and complex objects.

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

This survey will emphasize points 1(b) and 1(c), and give extensive references for points 2-6. For point 1(a) one can find several good books on first-order logic, for example [53], or good surveys on constraint logic programming and fixpoint semantics [42, 75, 153, 154]. There are also good introductions to logic programming (without constraints) in [10, 50, 101] and to database query languages (also without constraints) in [3, 79, 149]. One can also find an earlier tutorial on constraint databases in [80], and an applications-oriented survey in [61].

Points 1(b) and 1(c) will be illustrated by a new algebra for constraint databases with integer order constraints. It also analyses the computational complexity of evaluating relational calculus queries with integer order constraints.

The survey is organized as follows. Section 2 reviews first-order theories. Section 3 reviews relational databases and query languages. This section contains a comparison between relational databases and logic programming. Section 4 reviews constraint databases and query languages. This section contains a comparison of constraint databases and constraint logic programming. Section 5 summarizes known complexity results. Section 6 lists results on expressive power of languages. Section 7 discusses various semantic extensions of constraint databases. Section 8 mentions some prototype constraint database systems. Finally Section 8 lists some outstanding open problems.

2 First-Order Theories

2.1 Predicate Calculus

First we define the *first-order predicate calculus* language. The *alphabet* of this language consists of the following:

- A countably infinite set of variables. We usually denote variables by $x, y, z, v, u, \ldots$.
- A countably infinite set δ of constants. We usually denote constants by $a, b, c, \ldots$. The domain of each variable is δ.
- A set $\mathcal{R}$ of relation symbols. We usually denote relation symbols by $P, Q, R, \ldots$.
- Connectives $\wedge$ (conjunction), $\vee$ (disjunction), $\neg$ (negation).
- Quantifiers, $\exists$ (there exists) and $\forall$ (for all).

It is usual in the literature to restrict consideration in the definitions and proofs only to the connectives $\neg, \wedge$ and the quantifier $\exists$ because using only these all the others can be expressed. It is also possible to express implication $\rightarrow$ and double implication $\leftrightarrow$ using only those connectives.

Each relation symbol R has a fixed *arity*, that is, number of arguments. We denote the arity of R by $\alpha(R)$. 0-ary relations are permitted.

We define *predicate calculus formulas*, abbreviated PC formulas, inductively as follows:

- If R is an n-ary relation symbol and $x_1, \ldots, x_n$ are variables or constants, then $R(x_1, \ldots, x_n)$ is a PC formula.
- If ϕ is an PC formula, then $\neg\phi$ is an PC formula.
- If ϕ_1 and ϕ_2 are PC formulas, then $\phi_1 \wedge \phi_2$ and $\phi_1 \vee \phi_2$ are PC formulas.
- If ϕ is a PC formula and x is a variable, then $\exists x(\phi)$ and $\forall x(\phi)$ are PC formulas.

In the last line of the above definition, we call each occurrence of variable x within ϕ a *bound* variable. Variables that are not bound are called *free* in a formula.

One of the older problems that was considered by logicians is to decide whether a given predicate calculus formula is satisfiable, that is, whether there are relations that can be assigned to the relation symbols within the formula so that the formula becomes true. Another problem of interest in the area of theorem proving is whether all possible assignments make the formula true. Formulas with this property are called valid.

It was shown by Church that whether an unrestricted (both finite and infinite assignments are permitted) predicate calculus sentence (formula without free variables) is valid is undecidable. More precisely, the set of valid sentences is r.e. (recursively enumerable) but not co-r.e. Even if we restrict all relations to be finite, the problem remains undecidable. Here the set of valid sentences is co-r.e. and not in r.e. [140]. The satisfiability problem is also undecidable because a formula is satisfiable if and only if its negation is not valid. These negative results imply that there is no hope of using first-order predicate calculus as a query language. Fortunately, satisfiability can be solved for many subsets of the predicate calculus described later in the survey.

2.2 First Order Theories with Constraints

In this survey by constraints we mean special named relations. For example, let's suppose that our domain is the set of integers. Then the equality constraint $=$ means the infinite binary relation $\{(i, i) : i \in \mathbf{Z}\}$. (In the paper we use $\mathbf{R}$ for reals, $\mathbf{Q}$ for rationals, $\mathbf{N}$ for natural numbers, and $\mathbf{Z}$ for integers.) Similarly, the addition constraint $+$ means the infinite ternary relation such that the sum of the first two arguments is equal to the third argument. In the following we will keep using the usual infix notation for common constraints.

By first-order theories with constraints we mean various subsets of first-order predicate calculus in which the only relations are constraints. In these

theories there is only one assignment of interest: for each constraint symbol the corresponding constraint relation is assigned.

During the past hundred years many quantifier elimination procedures were developed for various first-order theories with constraints. By quantifier elimination we mean rewriting a formula with quantifiers into an equivalent one without quantifiers. It is out of the scope of this survey to review quantifier elimination procedures. However, we give a chronological outline of the major results in this area.

Some works considered only existentially quantified formulas. In the following table we indicate this as $\exists$QE. (For example, we mark as $\exists$QE Fourier's method even though it can be easily extended to deal with universal quantifiers.) It should be remembered that the following is only a partial list.

1828: $\exists$QE for $Th(\mathbf{R}, <, +)$ by Fourier [58].
1856: $\exists$QE for boolean algebras by Boole.
1927: QE for $Th(\mathbf{Q}, <)$ by Langford [99].
1929: QE for $Th(\mathbf{Z}, <, +)$ by Presburger [113].
1931: no QE for $Th(\mathbf{Z}, +, *)$ by Gödel [62].
1951: QE for $Th(\mathbf{R}, <, +, *)$ by Tarski [141].
1970: no $\exists$QE for $Th(\mathbf{Z}, +, *)$ by Matiyasevich [104].
1976: $\exists$QE for integers with modulus constraints by Williams [157].
1982: no QE for $Th(\mathbf{R}, +, *, exp)$ by van den Dries [150].

The decision problem is the problem of identifying whether a first-order formula without free variables is true or false. It is clear that the elimination of all quantifiers from a formula without free variables should leave as value either true or false. Hence the decision problem reduces to the quantifier elimination problem. Most negative results for quantifier elimination follow via reduction from the undecidability of the decision problem for the corresponding theories.

3 Relational Databases and Query Languages

3.1 Relational Databases and Their Problems

It is well-known that the relational data model enjoys great success in the business world. However, the causes of its success are less well-known. Sometimes it is wrongly assumed that its success is solely due to being based on logic. In reality the relational data model provides solutions to many practical problems that more complex logics often do not solve. Let's see some of these problems.

In the information market there are three important problems that databases need to solve.

1. **Ad-Hoc User's Problem:** Many database users want to enter ad-hoc queries that they improvise on-the-fly. They also want answers to every syntactically correct ad-hoc query.
2. **Application Vendor's Problem:** Application vendors sell libraries of queries to users. Application vendors must guarantee termination of the queries on each valid database input. In theory they may use any programming language for writing queries. However, guaranteeing termination is difficult in many programming languages. Hence they also want a restricted language in which any query they write terminates.
3. **Data Vendor's Problem:** Data vendors buy raw data in the form of databases and produce refined data in the form of databases which they sell to other data vendors. In the information market, there is often a long chain of data vendors from raw data to the users. The data model must serve as a common data communication standard. Each data vendor uses queries to transform raw data to the data product. Often these queries are secret or patented.

By looking at the context of how database systems are used in practice we can see that two important requirements have to be met. First, each query must terminate. Second, each query must give as output a database. We might call these the termination and the constructibility requirements.

Both of the above requirements are satisfied in the relational data model [41, 3, 79, 149], which is illustrated in Figure 1. Each relational database is a finite set of tables. Each table is an abstract data type and is a relation containing a finite set of tuples. A requirement in the relational data model is that queries be evaluable functions from finite databases to finite databases. That requirement is met by three relational query languages that we review in the next three subsection: (1) Relational Calculus, (2) Datalog, and (3) Stratified Datalog.

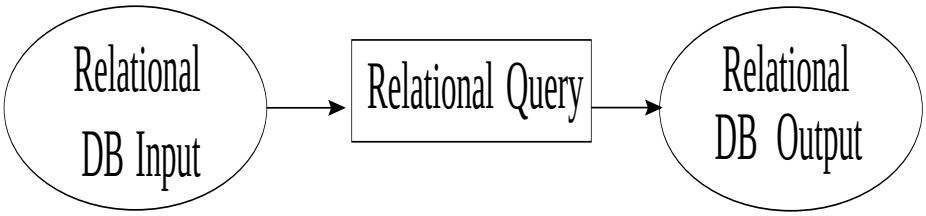

Fig. 1. The Relational Data Model

3.2 Relational Calculus

Syntactically relational calculus and predicate calculus are the same. However, in relational calculus –like in first-order theories with constraints– we are interested in only one assignment to the relations. We assign to each relation symbol an input relation given by the user.

The semantics of any relational calculus query ϕ is a mapping from relational databases to relational databases. Let $x_1, \ldots, x_n$ be the set of free variables of ϕ in some fixed order and let $\phi(a_1, \ldots, a_n)$ denote the formula obtained by substituting a_i for x_i for each $1 \leq i \leq n$.

Each input database d is an assignment of a finite or infinite number of tuples over $\delta^{\alpha(R_i)}$ to each R_i. The output database is a single relation of arity n defined as $\{(a_1, \ldots, a_n) :< \delta, d >\models \phi(a_1, \ldots, a_n)\}$. Here $< \delta, d >\models$ means *satisfaction* with respect to a domain δ and database d and is defined as follows. If r_i is the set of tuples assigned to relation symbol R_i, then

$$< \delta, d >\models R_i(a_1, \ldots, a_k) \text{ iff } (a_1, \ldots, a_k) \in r_i \quad (1)$$

$$< \delta, d >\models (\phi \wedge \psi) \text{ iff } < \delta, d >\models \phi \text{ and } < \delta, d >\models \psi \quad (2)$$

$$< \delta, d >\models (\neg\phi) \text{ iff not } < \delta, d >\models \phi \quad (3)$$

$$< \delta, d >\models (\exists x_i \phi) \text{ iff } < \delta, d >\models \phi[x_i/a_j] \text{ for some } a_j \in \delta \quad (4)$$

where $[x_i/a_j]$ means the instantiation of the free variable x_i by a_j.

Remark: Unfortunately, not all relational calculus queries preserve finiteness. For example, if r is any finite unary relation assigned to relation symbol R and δ is any infinite domain, then the query $\{(a) :< \delta, r >\models \neg R(a)\}$ defines an infinite relation. There are several approaches to guarantee finiteness by syntactically restricting relational calculus queries to various "safe" subsets [13, 72, 86, 149]. Safe relational calculus queries are translatable to relational algebra, a procedural language.

3.3 Datalog

Datalog [3, 11, 50, 101, 149] is a fragment of predicate calculus extending relational calculus with intensionally defined relations. In relational calculus each query defines a single output relation which is not named explicitly and all other relations are inputs. In contrast in Datalog a query may define several output relations which are referred to by name within the query. Hence in Datalog the output relations are defined using references to themselves.

Syntactically a *Datalog* program Π is a finite set of rules of the form:

$$R_0(x_1, \ldots, x_k) :\!\!- R_1(x_{1,1}, \ldots, x_{1,k_1}), \ldots, R_n(x_{n,1}, \ldots, x_{n,k_n}).$$

where the Rs are relation symbols and the xs are either variables or constants.

Semantically each Datalog program is a mapping. To explain this mapping, we first associate with each rule of the above form the formula:

$$\forall v_1 \ldots \forall v_m (R_0(x_1, \ldots, x_k) \vee \neg R_1(x_{1,1}, \ldots, x_{1,k_1}) \vee \ldots \vee \neg R_n(x_{n,1}, \ldots, x_{n,k_n}))$$

where $v_1, \ldots, v_m$ are the variables in the rule. We also associate with Π the conjunction of the formulas associated with each rule in Π. Let F_Π denote this formula.

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

We call an *interpretation* of Π any assignment I of a finite or infinite number of tuples over $\delta^{\alpha(R_i)}$ to each R_i that occurs in Π. A *model* of Π is an interpretation of Π satisfying F_Π. A model I is a *minimum model* if and only if $I \subseteq J$ for all models J.

We call an interpretation an *input database* if it assigns to each R_i that occurs at least once on the left hand side of a rule in Π the empty set of tuples.

Each Datalog program Π is a mapping from input databases to interpretations. Let d be an input database. Then the output of Π on d, denoted $\Pi(d)$, is the minimum model of Π containing d. The following theorem assures us that we really have a mapping.

Proposition 3.1 Let Π be any Datalog program and d be any input database. Then there exists a minimum model of Π containing d. □

Alternative Operational Semantics An important alternative definition of the semantics of Datalog programs is called *fixpoint* semantics.

We call *valuation* any function from (tuples of) variables to (tuples of) constants satisfying the following: for all tuples t_1 and t_2, $\nu(t1, t2) = (\nu(t_1), \nu(t_2))$, and for all constants c, $\nu(c) = c$.

The *immediate consequence operator* of a Datalog program Π, denoted T_Π, is a mapping from interpretations to interpretations as follows. For each interpretation I:

$R_0(a_1, \ldots, a_n) \in T_P(I)$ iff there is a valuation ν and a rule of the form $R_0(x_1, \ldots, x_k) :\!-\ R_1(x_{1,1}, \ldots, x_{1,k_1}), \ldots, R_n(x_{n,1}, \ldots, x_{n,k_n})$ in Π such that $\nu(x_1, \ldots, x_n) = (a_1, \ldots, a_n)$ and $< \delta, d > \models R_i(\nu(x_{i,1}, \ldots, x_{i,k_i}))$ for each $1 \leq i \leq n$.

An interpretation I is called a *fixpoint* of a program Π iff $T_\Pi(I) = I$.

Proposition 3.2 Let Π be any Datalog program and d be any input database. Each fixpoint of Π is a model of F_Π. Moreover, there is a minimum fixpoint of Π containing d. □

The following states that the model-theoretic and the fixpoint semantics coincide.

Proposition 3.3 Let Π be any Datalog program and d be any input database. The minimum fixpoint of Π containing d equals $\Pi(d)$. □

3.4 Stratified Datalog

Stratified Datalog [3, 50, 149] extends Datalog with negation. Intuitively, an expression like $R_0 :\!-\ R_1 \wedge \neg R_2$ means that R_0 is the set difference of the relations R_1 and R_2. The problem is that to say anything definite about R_0 we must know the full value of the negated relation. That motivates the following definitions.

We call *semipositive* those Datalog programs with negation that only allow negation of EDBs. (See [3, 50] for more discussion.)

Semantically, each semipositive Datalog program is a mapping from databases to interpretations. We can define F_Π similarly to the case of Datalog. We have that:

Proposition 3.4 Let Π be any semipositive Datalog program and d be any input database that assigns non-empty relations only to the EDBs. Then there exists a minimum model of Π containing d. □

Another extension of Datalog is the class of *stratified Datalog* programs. Each stratified Datalog program Π is the union of semipositive programs $\Pi_1, \ldots, \Pi_k$ satisfying the following property: no relation symbol R that occurs negated in a Π_i is an IDB in any Π_j with $j \geq i$.

It is usual to associate with each stratified Datalog program Π a function σ from rules to positive integers. The function σ indicates the grouping of the rules into semipositive programs in an order that satisfies the above property. We will assume that a σ is associated with each stratified Datalog program.

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

Proposition 3.5 Let Π be any stratified Datalog program consisting of the union of some semipositive programs $\Pi_1, \ldots, \Pi_k$ with the above property. Let d be any database that assigns non-empty relations only to the EDBs of Π_1. Then there exists a minimum model of Π containing d. □

Remark: An explicit specification of σ is not required. [12] describes an algorithm to find a σ satisfying the above property. Moreover, [12] also shows that all σ's satisfying the above property are semantically equivalent.

Remark 2: Datalog programs with negation on the right hand side of the rules can be assigned an *inflationary* semantics [4, 64, 89]. This semantics is not equivalent to the one given here and seems less frequently used.

3.5 Relational Databases vs. Logic Programming

It is not our goal here to review logic programming [52, 11], which has known similarities with relational database queries. We would like only to point out some essential differences between these two concepts.

Each logic program is a mapping from a finite set of facts to a least model. Logic programs do not always terminate and least models are not always a finite set of facts. Therefore, logic programs do not satisfy the termination and constructibility requirements, which as we saw are crucial problems in databases.

Every relational query can be expressed as a logic program, but the reverse is not true. Unfortunately, from the database point of view the higher expressive power of logic programs is at the expense of some practicality. The following is a brief historical summary of the developments in the two areas, based on [10]:

1970: Relational Algebra proposed by Codd [41].
1972: Relational Calculus shown equivalent to Relational Algebra by Codd.
1973-74: Prolog by Colmerauer, Kanoui, and Van Caneghem [44] and Kowalski.
1976: Fixpoint Semantics for Prolog by Van Emden and Kowalski.
1979-80: Datalog by Aho and Ullman [7] and Chandra and Harel [31].
1985-88: Stratified Datalog by Chandra and Harel [33] and Prolog by Apt, Blair and Walker [12].

4 Constraint Databases and Query Languages

Constraint programming, or programming with constraints as primitives in the programming language, was first investigated in the early 1960's by Sutherland in the SKETCHPAD system [139]. Since then constraint programming was applied to various problems in artificial intelligence [60, 102, 105, 135], graphical-interfaces [22, 131], and in logic programming and databases. We review only the developments in the latter two areas, and even in logic programming we mention only on those developments that have a relationship to databases. The reader may find detailed surveys on constraint logic programming in [42, 75, 96, 153, 154].

4.1 Constraint Databases and Their Problems

Many database applications have to deal with infinite concepts like time and space. However, in practice only those databases can be used which can be finitely represented. Fortunately, many infinite data can be finitely represented using constraint databases. A general framework for using constraint databases is presented in [82]. The following three definitions are from that paper.

A *generalized k-tuple* is a quantifier-free conjunction of constraints on k variables ranging over a domain δ. Each generalized k tuple represents in a finite way an infinite set of regular k-tuples. For example, suppose that relation R contains the set of points on the line that passes through the origin and has slope two. While R has in it an infinite number of tuples, it can be finitely represented by the generalized 2-tuple $R(x, y) :\!\!- y = 2 * x$.

A *generalized relation of arity k* is a finite set of generalized k-tuples, with each k-tuple over the same variables.

A *generalized database* is a finite set of generalized relations.

If we use generalized databases, then all the definitions of query language semantics still apply except that we have to also generalize the meaning of $\models$. That can be done as follows.

Let r_i be the generalized relation assigned to R_i. We associate with each r_i a formula F_{r_i} that is the disjunction of the formulas on the right hand side of each generalized k-tuple of r_i. Let ϕ be any relational calculus formula. Satisfaction with respect to a domain δ and database d, denoted $< \delta, d >\models$, is defined recursively as follows:

$$< \delta, d > \models R_i(a_1, \ldots, a_k) \text{ iff } F_{r_i}(a_1, \ldots, a_k) \text{ is true} \tag{5}$$

$$< \delta, d > \models (\phi \wedge \psi) \text{ iff } < \delta, d > \models \phi \text{ and } < \delta, d > \models \psi \quad (6)$$

$$< \delta, d > \models (\neg \phi) \text{ iff not } < \delta, d > \models \phi \quad (7)$$

$$< \delta, d > \models (\exists x_i \phi) \text{ iff } < \delta, d > \models \phi[x_i/a_j] \text{ for some } a_j \in \delta \quad (8)$$

The following alternative semantics that is equivalent to the above is discussed in [82]:

Let $\phi = \phi(x_1, \ldots, x_m)$ be a relational calculus program with free variables $x_1, \ldots, x_m$. Let relation symbols $R_1, \ldots, R_n$ in ϕ be assigned the generalized relations $r_1, \ldots, r_n$ respectively. Let $\phi[R_1/F_{r_1}, \ldots, R_n/F_{r_n}]$ be the formula that is obtained by replacing in ϕ each database atom $R_i(z_1, \ldots, z_k)$ by the formula $F_{r_i}[x_1/z_1, \ldots, x_k/z_k]$ where $F_{r_i}(x_1, \ldots, x_k)$ is the formula associated with r_i. The output database of ϕ on input database $r_1, \ldots, r_n$ is the relation $\{(a_1, \ldots, a_m) :< \delta, d > \models \phi_1(a_1, \ldots, a_m)$ where $\phi_1 = \phi[R_1/F_{r_1}, \ldots, R_n/F_{r_n}]\}$.

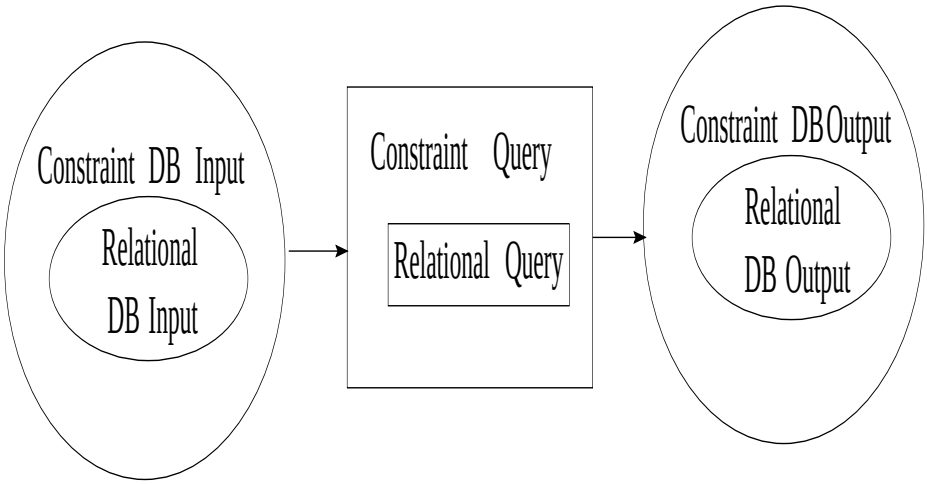

Fig. 2. The Constraint Data Model

In the generalized database model of [82] queries are functions from generalized databases to generalized databases using the same type of constraints. This *closed-form* requirement is the analogue of the termination and constructibility requirements in relational databases and stems from the same practical considerations that were discussed in Section 3.1. Constraint query languages are generalizations of relational query languages with constraints, i.e., reference to built-in relations are allowed in them. These ideas of the constraint data model are summarized in Figure 2. We present an example constraint query language and closed-form evaluation in Section 4.2.

As a consequence of the alternative semantics above we have:

Proposition 4.1 Let $\mathcal{T}$ be any first-order theory with constraint relations. If $\mathcal{T}$ admits quantifier elimination, then the output of each relational calculus query on generalized relations containing only constraints in $\mathcal{T}$ can be evaluated in finite time. Moreover, the output can be represented as a generalized relation containing only constraints in $\mathcal{T}$. □

Proposition 4.1 implies that all of the quantifier elimination results in Section 2.2 help realize the goal of closed-from evaluation. Hence they are very much relevant for querying generalized databases. Because of them relational calculus on many types of constraint databases can be used without any restriction, i.e. not even the "safety" problem is a concern. For Datalog and stratified Datalog queries however closed-form evaluation is more difficult and sometimes impossible (see Section 5). Recently [145] presented broadly applicable sufficient conditions for termination and closed-form evaluation of Datalog queries with constraints.

4.2 An Example Closed-Form Evaluation

Quantifier elimination algorithms provide closed-form evaluations for queries. Unfortunately, many quantifier elimination algorithms are computationally inefficient. Efficiency can be improved by algebrizing the quantifier elimination procedure. Algebraic operators are desirable because they provide efficient set-at-a-time computations with generalized relations. The idea of using algebras instead of simple quantifier elimination goes back to Tarski and Thompson [142]. Codd's *relational algebra* [41] is another example of algebrization of query evaluation. Relational algebra is a procedural language that is equivalent in expressive power to safe Relational Calculus. Relational Calculus queries are translated to relational algebra for quicker evaluation [3, 149].

More recently some algebraic operators are considered in the case of linear constraint databases in [23], in the case of dense order constraints in [81], and in the case of discrete order constraints in [120], which shows a partial algebra with only select, project, and join operators. Next we extend those to the full set of the relational algebra operators and give that as an illustration of algebras for generalized databases. Our definitions will follow mainly [120].

Definition 4.1 Let x and y be any two integer variables or constants. Given some assignment to the variables, a *minimum gap-order* constraint $x <_g y$ for some gap-value $g \in \mathbf{N}$ holds if and only if $g < y - x$ holds in the given assignment. A *minimum gap-order* constraint $x = y$ holds if and only if x and y are equal in the given assignment. A *maximum gap-order* constraint $x <^h y$ for some gap-value $h \in \mathbf{N}$ holds if and only if $0 < y - x < h$ holds in the given assignment. □

Definition 4.2 Let $x_1, \ldots, x_n$ be integer variables and $c_1, \ldots, c_m$ be integer

constants. Any graph with vertices labeled $x_1, \ldots, x_n, c_1, \ldots, c_m$ and at most one undirected edge labeled by $=$ or at most one directed edge labeled by $<_g^h$ for some $g \in \mathbf{N}$ and $h \in \mathbf{N} \cup +\infty$ between any pair of distinct vertices is called a *mm-gapgraph.* □

Remark: Any $g = 0$ and $h = +\infty$ may be considered as default values and not written out explicitly within the mm-graphs.

Definition 4.3 Any mm-gapgraph with vertices labeled with variables $x_1, \ldots, x_n$ and constant 0 is in *normal form.* Furthermore, any set of mm-gapgraphs each with vertices labeled with $x_1, \ldots, x_n, 0$ is in *normal form.* □

It is easy to see that any mm-gapgraph of size n can be put into normal form in $O(n)$ time. We have change each constraint of the form $c <_g x$ (respectively $c <^h x$) where $c > 0$ into $0 <_{c+g} x$ (respectively $0 <_c^{c+h} x$). The cases when $c < 0$ or is on the right hand side of the minimum (respectively maximum) gap-order constraint are similar.

For the rest of this section, assume that all gap-graphs are in normal form. Furthermore, assume that each order constraint has both a minimum and a maximum bound. If that is not true in some constraint than add the default values of 0 for minimum and $+\infty$ for maximum gap-values.

Definition 4.4 Let G be a mm-gapgraph with vertices $y, v_1, \ldots, v_n, 0$. Then a *shortcut* operation over vertex y transforms G into an output mm-gapgraph with vertices $v_1, \ldots, v_n, v_{n+1}$ where $v_{n+1} = 0$ as follows. First, for each $0 < i, j, \leq n+1$ do the following.

If $v_i = y$ and $y = v_j$ are edges in G, then add $v_i = v_j$ as an undirected edge to G.
If $v_i = y$ and $y <_g^h v_j$ are edges in G, then add $v_i <_g^h v_j$ as a directed edge to G.
If $v_i <_g^h y$ and $y = v_j$ are edges in G, then add $v_i <_g^h v_j$ as a directed edge to G.
If $v_i <_{g_1}^{h_1} y$ and $y <_{g_2}^{h_2} v_j$ are edges in G, then add $v_i <_{g_1+g_2+1}^{h_1+h_2-1} v_j$ as a directed edge to G.

Second, if there are two or more edges of the form $<_{g_1}^{h_1}, \ldots, <_{g_k}^{h_k}$ between vertices v_i and v_j then delete these edges and replace these with the edge $v_i <_{\max(g_1,\ldots,g_k)}^{\min(h_1,\ldots,h_k)} v_j$. If more than one edge remains between any two vertices, then the shortcut operation fails, returns an error message, and it does not produce a shortcut mm-gapgraph as output. Otherwise, y and its incident edges are deleted and the resultant mm-gapgraph is returned. □

In the above, the first part of the shortcut operation adds only constraints that follow by transitivity from the set of original constraints represented by the input mm-gapgraph. In the second part the simplification is needed to ensure

that the graph remains in mm-gapgraph form and has at most one edge between any pair of vertices. Note that if after the simplification more than one edge remains between any pair of vertices, then they must be differently oriented or one directed and another undirected, i.e. equality, constraint. This clearly implies that the mm-gapgraph is not satisfiable. Note that apart from this obvious case the shortcut operation does not check the mm-gapgraph for satisfiability.

Definition 4.5 Let G_1 and G_2 be two mm-gapgraphs over some (maybe different) subsets of the variables $v_1, \ldots, v_n$ and the constant 0. Then a *merge* operation on G_1 and G_2 creates a mm-gapgraph G with vertices $v_1, \ldots, v_n, v_{n+1} = 0$ as follows. For each $0 < i, j \leq n+1$ do the following.

If there is no edge between v_i and v_j in G_1 and G_2, then do nothing.
If there is an edge between v_i and v_j in only one of G_1 or G_2, then add that edge to G.
If $v_i = v_j$ is an edge in both G_1 and G_2, then add $v_i = v_j$ as an edge to G.
If $v_i <_{g_1}^{h_1} v_j$ in G_1 and $v_i <_{g_2}^{h_2} v_j$ in G_2 are edges, then add $v_i <_{max(g_1,g_2)}^{min(h_1,h_2)} v_j$ as an edge to G.
If $v_j <_{g_1}^{h_1} v_i$ in G_1 and $v_j <_{g_2}^{h_2} v_i$ in G_2 are edges, then add $v_j <_{max(g_1,g_2)}^{min(h_1,h_2)} v_i$ as an edge to G.
In any other case the merge operation fails, and it does not return any graph. □

In the merge operation we want any assignment that satisfies the output mm-gapgraph to satisfy both of the input mm-gapgraphs. The operation guarantees this by checking that the corresponding edges in the two input mm-gapgraphs are compatible and by combining the stricter minimum and the stricter maximum gap-order constraints in the corresponding edges and adding the combination to the output mm-gapgraph. The last condition "in any other case" includes the cases when G_1 and G_2 are not compatible, for example, when for some variables v and w, one specifies that v is less than w while the other says that v is greater than w. In these cases there is clearly no assignment that can satisfy both graphs, hence the merge operation will fail. Next we prove the semantic correctness of the operations defined, that is, we show that shortcut is a valid existential quantifier elimination procedure and join is consistency preserving.

Lemma 4.1 Let G be a mm-gapgraph over variables $y, v_1, \ldots, v_n$ and constant 0. Let G' be the mm-gapgraph obtained by shortcutting over y in G (if exists). Let $a_0, a_1, \ldots, a_n$ be any sequence of integer numbers. Then $G(a_0, a_1, \ldots, a_n)$ is true if and only if G' exists and $G'(a_0, a_1, \ldots, a_n)$ is true. □

For the merge operation we show that the *and* of the input mm-gapgraphs is consistent if and only if the output mm-gapgraph is consistent.

Lemma 4.2 Let G_1 and G_2 be two mm-gapgraphs over some (maybe different) subsets of the variables $v_1, \ldots, v_n$ and over 0. Let G be the mm-gapgraph obtained by merging G_1 and G_2 (if exists). Let $a_1, \ldots, a_n$ be any sequence of integer numbers. Then $G_1(a_0, a_1, \ldots, a_n)$ and $G_2(a_0, a_1, \ldots, a_n)$ are true if and only if G exists and $G(a_0, a_1, \ldots, a_n)$ is true. □

Based on the above operators, we define the generalized project $\dot{\pi}$ and the generalized join $\dot{\bowtie}$ operators. Let R_1 and R_2 be two relations that are both in normal form. Then the generalized join of R_1 and R_2, denoted $R_1 \dot{\bowtie} R_2$ is the set $\{merge(G_1, G_2) : G_1 \in R_1, G_2 \in R_2\}$. Assume that R_1 is a k-ary relation with argument symbols $S = \{x_1, \ldots, x_k\}$. Then the generalized project of R_1 onto a subset $S' = S \setminus \{x_i\}$ of the arguments is the set $\{shortcut(x_i, G) : G \in R_1\}$.

The generalized selection operation on R, denoted $\dot{\sigma}_C R$, where C is any selection condition that is the conjunction of minimum and maximum gap-order constraints, is the set $\{merge(G, graph(C)) : G \in R\}$. Here *graph* is a function that transforms each conjunction of gap-order constraints into an mm-gapgraph.

The generalized rename operation $\dot{\rho}$ just renames the argument vertices in each mm-gapgraph in the given generalized relation.

Now let R_1 and R_2 be two k-ary generalized relations with the same scheme. Then the generalized union operation $\dot{\cup}$ is defined as the union of the mm-gapgraphs in R_1 and R_2.

Also, the generalized difference of R_1 and R_2, denoted $R_1 \dot{-} R_2$, is defined to be the set $\{R_1 \dot{\bowtie} R_3 : R_3 = \neg R_2\}$ where $\neg R_2$ is the *complement* of R_2 in the standard sense. That is, if R_2 represents any set of regular tuples $\mathcal{B} \subseteq \delta^k$, then $\neg R_2$ is the set of regular tuples $\delta^k \setminus \mathcal{B}$. The following can be proven using De Morgan's laws.

Lemma 4.3 Let R be a k-ary relation represented by a set of mm-gapgraphs of size n. Then a generalized relation representing the complement of R can be found in $O(n^{k^2})$ time. □

We have defined the generalized versions of all the fundamental relational algebra operators. (Note that cross product is the case of join when there are no overlaps in the arguments of the two input relations. Our definition of join allows that possibility.) Next we give an example of the use of the generalized algebraic operators.

Example 4.1 Suppose that two persons want to schedule a meeting during a given month. The first person is free from the 3rd to the 6th and from the 16th to the 26th. The second person is free from the 6th to the 18th and from the 25th to the 30th. Also suppose that they don't want to meet on the 6th and the 13th. Which days could they meet?

The relation FREE telling which person is free on which days and the relation BAD_DAYS can be represented in normal form as follows.

FREE	Person-ID	Day	
	p	t	$0 <^{2} p \wedge 0 <_{2}^{7} t$
	p	t	$0 <^{2} p \wedge 0 <_{15}^{27} t$
	p	t	$0 <_{1}^{3} p \wedge 0 <_{5}^{19} t$
	p	t	$0 <_{1}^{3} p \wedge 0 <_{24}^{31} t$

BAD_DAYS	Day	
	t	$0 <_{5}^{7} t$
	t	$0 <_{12}^{14} t$

The query that finds the good days for meeting is the following:

$$((\dot{\pi}_{Day}\, \dot{\sigma}_{0<^{2}p}\, FREE) \bowtie (\dot{\pi}_{Day}\, \dot{\sigma}_{0<_{1}^{3}p}\, FREE)) \dot{-} BAD\_DAYS$$

Here $\dot{\sigma}_{0<^{2}p} FREE$ will be the first two tuples of FREE. (Note that the selection condition when added to the other tuples will result in an inconsistency.) That projected into *Day* will give a temporary relation $R_1(t) = \{G_1, G_2\}$ where G_1 represents $0 <_{2}^{7} t$ and G_2 represents $0 <_{15}^{27} t$.

Similarly, $\dot{\pi}_{Day}\dot{\sigma}_{0<_{1}^{3}p} FREE$ will yield $R_2(t) = \{G_3, G_4\}$ where G_3 represents $0 <_{5}^{19} t$ and G_4 represents $0 <_{24}^{31} t$. The generalized join of R_1 and R_2 will be $\{merge(G_1, G_3)$,
$merge(G_1, G_4), merge(G_2, G_3), merge(G_2, G_4)\}$. The temporary output will be the following generalized relation:

R_3	Day	
	t	$0 <_{5}^{7} t$
	t	$0 <_{15}^{19} t$
	t	$0 <_{24}^{27} t$

In the above we do not show the merge of G_1 and G_4 because it is inconsistent. Next we take the complement of BAD_DAYS. This will yield:

$$\begin{aligned}
& \neg((0 <_{5}^{7} t) \vee (0 <_{12}^{14} t)) \\
\equiv\ & \neg((0 <_{5} t \wedge 0 <^{7} t) \vee (0 <_{12} t \wedge 0 <^{14} t)) \\
\equiv\ & (\neg(0 <_{5} t) \vee \neg(0 <^{7} t)) \wedge (\neg(0 <_{12} t) \vee \neg(0 <^{14} t)) \\
\equiv\ & ((t < 0 \vee t = 0 \vee 0 <^{6} t) \vee (t < 0 \vee t = 0 \vee 0 <_{6} t)) \wedge \\
& ((t < 0 \vee t = 0 \vee 0 <^{13} t) \vee (t < 0 \vee t = 0 \vee 0 <_{13} t)) \\
\equiv\ & (t < 0 \vee t = 0 \vee 0 <^{6} t \vee 0 <_{6} t) \wedge (t < 0 \vee t = 0 \vee 0 <^{13} t \vee 0 <_{13} t) \\
\equiv\ & (t < 0 \vee t = 0 \vee 0 <^{6} t \vee 0 <_{6}^{13} t \vee 0 <_{13} t)
\end{aligned}$$

Hence the complement of BAD_DAYS is a generalized relation with five tuples. Finally, we take the generalized join of R_3 and the complement of BAD_DAYS. We obtain:

R_5	Day	
	t	$0 <_{15}^{19} t$
	t	$0 <_{24}^{27} t$

The above generalized relation expresses that the two persons could meet any day from the 16th to the 18th and from the 25th to the 26th inclusive.

4.3 Constraint Databases vs. Constraint Logic Programming

Let us briefly compare constraint databases and constraint logic programming [42, 75, 153, 154].

Each constraint logic program is a mapping from a finite set of constraint facts to a least model. The most general framework for constraint logic programming was given by Jaffar and Lassez [74] who show that under very mild assumptions about the constraint predicates, the semantics of Prolog-like languages with constraints changes only slightly, in particular queries can still be given a least model semantics.

Constraint query languages are a subset of constraint logic programs. However, Jaffar and Lassez do not give any criteria for effectively evaluating the least model or even whether the least model is finitely representable. For constraint query languages the least models are effectively computable and finitely representable. Therefore, constraint query languages solve the user's constructibility requirement and the data vendor's problem, which are NOT solved by constraint logic programs in general. The main contribution of constraint database research is the set of algorithmic solutions to these problems.

Some important constraint programming languages are the following: Prolog III which allows constraints over the 2-valued Boolean algebra and linear arithmetic constraints over the rationals [43]. CHIP [49] which allows linear arithmetic constraints over both the rationals and bounded subsets of the integers. CLP($\mathcal{R}$) [76] which provides polynomial constraints over the reals. LIFE [8] which allows constraints over feature trees and also provides a notion of objects. Trilogy [156] which allows constraints over strings, integers, and real numbers. The following is a brief historical outline of the developments in constraint logic programming and constraint databases:

1982: Prolog II, first instance of CLP.
1987: CLP semantics by Jaffar and Lassez [74].
1990: CQL database framework by Kanellakis, Kuper and Revesz [82].

5 Complexity Issues

In this section, we first review the known complexity results for first-order theories with constraints in Section 5.1. Then we review the known data complexity results for both relational and constraint query languages in Section 5.2. We discuss how data complexity is related to closed-form evaluation in Section 5.3. Finally Section 5.4 discusses optimization techniques to speed up query evaluation.

5.1 The Decision Problem for First-Order Theories with Constraints

The computational complexity of the decision problem for first-order theories was investigated in depth during the past twenty years. We list only some of the important results in this area.

The computational complexity of $Th(\mathbf{R}, <, +, *)$ is investigated in [19, 45, 92, 119]. The complexity of $Th(\mathbf{R}, <, +)$, a subset of the previous theory, is in DSPACE(2^{cn}) where n is the size of the formula [55] and also in alternating Turing machine class $TA(2^{cn}, n)$ [29]. (See [77] for definitions of various complexity classes.) Another interesting subset of the above theory allows only difference constraints, i.e. only constraints of the form $x_i - x_j > c$ for x_i, x_j variables and c constant. The complexity of this language is considered in [90, 91] and is shown to be PSPACE-complete in general and to be Σ_k^p-complete for k alternation of $\exists$ and $\forall$ quantifiers in the prenex form. The complexity of the theory of rational order is considered in [54] and is shown to be in DSPACE($n \log n$). The complexity of Presburger arithmetic is considered in [21, 57, 118].

In the above complexity analyses the Ehrenfeucht-Fraïssé game technique [51, 59] plays a major role. In this section, we illustrate a simple case of this important technique by applying it to the first-order theory of integer order.

In general, Ehrenfeucht-Fraïssé games are used to show that quantifiers ranging over all elements of a domain of a theory may be restricted to small finite subsets of the domain. This makes the validity of formulas of a theory decidable by exhaustive search. Upper bounds on the size of the finite subsets yield upper bounds on the time or space required to evaluate the formulas. The theory of integer order provides a simple case of the power of this general technique. Ferrante and Rackoff [55] used Ehrenfeucht-Fraïssé games in a compact proof to show an $O(n^2)$ space upper bound for deciding formulas of size n that have no constants other than possibly 0. Their analysis is more complex than necessary because they are concerned with several theories at once. In the following we simplify their proof for the case of the theory of integer order only and also add a means to deal with constants other than 0.

First, the infinite set of possible integer tuples that may satisfy a first-order formula is divided into a finite number of equivalence classes. The goal is to show that any member of an equivalence class satisfies a formula if and only if every member of the equivalence class satisfies that formula. This limits the number of possible cases that needs to be considered during evaluation, because it is enough to pick just one member from each equivalence class to test whether it is a solution. At first we look at the case when the formulas have no constants in them.

Definition 5.1 Let $\overline{a}_k = (a_1, \ldots, a_k)$ and $\overline{b}_k = (b_1, \ldots, b_k)$ be tuples of integers. For each $k, d \in N$ we write that $\overline{a}_k \sim_{k,d} \overline{b}_k$ if and only if for each $1 \leq i, j \leq k$, if $\mid a_i - a_j \mid \leq 2^{d-k}$, then $a_i - a_j = b_i - b_j$, and if $a_i - a_j > 2^{d-k}$, then $b_i - b_j > 2^{d-k}$. □

In the next lemma, we use $\models$ to mean satisfaction in the standard sense.

Lemma 5.1 Let $F(x_1, \ldots, x_k)$ be any formula of the theory of integer order, and let d be the quantifier depth in F. Then for any tuples $\overline{a}_k, \overline{b}_k$, if $\overline{a}_k \sim_{k,d} \overline{b}_k$, then $\overline{a}_k \models F(x_1, \ldots, x_k)$ if and only if $\overline{b}_k \models F(x_1, \ldots, x_k)$.

Proof: We prove the lemma by induction on the number of $\vee, \neg, \exists$ operators in F. Assume that $\overline{a}_k \sim_{k,d} \overline{b}_k$. For the base case, we have a single order constraint of the form $x_i < x_j$ or $x_i = x_j$. Clearly, each of these is either true or false in both $\overline{a}_k$ and $\overline{b}_k$. For more complex formulas we have the following cases:

(1) $F(\overline{x}) \equiv F'(\overline{x}) \vee F''(\overline{x})$. Suppose $\overline{a}_k \models F'(\overline{x}) \vee F''(\overline{x})$. Then, without loss of generality $a_k \models F'(\overline{x})$. By induction, $\overline{b}_k \models F'(\overline{x})$. Hence, $\overline{b}_k \models F'(\overline{x}) \vee F''(\overline{x})$.

(2) $F(\overline{x}) \equiv \neg F'(\overline{x})$. Suppose $\overline{a}_k \models \neg F'(\overline{x})$. Then, $\overline{a}_k \not\models F'(\overline{x})$. By induction, $\overline{b}_k \not\models F'(\overline{x})$. Hence, $\overline{b}_k \models \neg F'(\overline{x})$.

(3) $F(\overline{x}) \equiv \exists z F'(\overline{x}, z)$. Suppose $a_k \models \exists z F'(\overline{x}, z)$. Then for some a_{k+1}, $\overline{a}_k, a_{k+1} \models F'(\overline{x}, z)$. If $a_{k+1} = a_j$ for some $1 \leq j \leq k$, then we choose $b_{k+1} = b_j$. Otherwise, let $a_{j_1} < a_{k+1} < a_{j_2}$ be the pair of integers within $\overline{a}_k$ between which a_{k+1} lies. If $a_{j_2} - a_{j_1} \leq 2^{d-k}$, then choose $b_{k+1} = b_{j_1} + a_{k+1} - a_{j_1}$. If $a_{j_2} - a_{j_1} > 2^{d-k}$, then choose $b_{k+1} = b_{j_1} + 2^{d-k}$. Since $\overline{a}_k \sim_{k,d} \overline{b}_k$, it is easy to see that in both cases the choices make $\overline{a}_{k+1} \sim_{k+1,d} \overline{b}_{k+1}$. By induction, $\overline{b}_{k+1,d} \models F'(\overline{x}, z)$. Hence, $\overline{b}_k \models F(\overline{x})$.

These cases prove the "if" part of the lemma. By arguing similarly for $\overline{b}_k$ we prove the "only if" part of the lemma. □

The above lemma can be generalized to the case when there are constants in the formula. Moreover it can be turned into an effective quantifier elimination procedure. The quantifier-free formula that is returned however, will have to use *gap-order* constraints, that is atomic formulas of the form $x_i - x_j > k$ where k is a nonnegative integer number.

Theorem 5.1 Let $F(x_1, \ldots, x_k)$ be a formula in the first-order theory of integer order. Then we can find a quantifier-free formula $F'(x_1, \ldots, x_k)$ in the theory of integer gap-order such that $F \leftrightarrow F'$ is true.
Furthermore, the size of F' is at most $O((k!(k-1)^{2^d+2}((2^{d+1}+3)m)^k)n)$ where d is the quantifier depth of F, m is the number of distinct integer constants in F and n is the total size of F.

Proof: We want to find one member from each equivalence class – in the sense of Definition 5.1– whose members satisfy F. By Lemma 5.1, if F had no constants in it, then we would have to try only $O(k!(k-1)^{2^d+2})$ sequences of integers, i.e., $k!$ ordering of the variables in $\overline{a}_k$ and between each adjacent pair trying out only equalities and gaps of size 1 to 2^d+1. However, if F contains constants, then the relative ordering and gap-sizes among the constants and the variables have to be also known to be able to evaluate atomic formulas that contain constants, e.g., $x_i = c$, $x_i < c$, or $x_i \geq c$.

We can extend Definition 5.1 as follows. We write that $\overline{a}_k \sim_{k,d} \overline{b}_k$ if and only if for each constant or variable u in F the following condition holds: for each $1 \leq i \leq k$, if $\mid a_i - u \mid \leq 2^{d-k}$, then $a_i - u = b_i - u$, if $a_i - u > 2^{d-k}$, then $b_i - u > 2^{d-k}$, and if $u - a_i > 2^{d-k}$, then $u - b_i > 2^{d-k}$.

Hence we would need for each of the possible ordering-gaping of the variables only a gap assignment with respect to the constants that occur in the formula. With respect to each constant c, a variable x_i can be placed into $2^{d+1}+3$ distinguishable positions, that is, x_i less than c by $1, \ldots, 2^d+1$, x_i equal to c, or x_i greater than c by $1, \ldots, 2^d+1$. With m constants in F, this leaves $(2^{d+1}+3)m$ choices for each variable. Hence we have the upper bound of $O(k!(k-1)^{2^d+2}((2^{d+1}+3)m)^k)$ many different equivalence classes with constants. (Actually, we may have much less, as only a part of the gap choices will be consistent with the variable orderings, and only part of the gap assignments with respect to the constants will be consistent with the variable orderings and gap choices.)

Similarly to Lemma 5.1 it can be shown that if any member of an equivalence class satisfies the formula, then each member of the equivalence class satisfies the formula. Moreover, we can pick a representative from each equivalence class and test whether it satisfies the formula. It is clear that each equivalence class can be written as a quantifier-free formula over the free variables that is a conjunction of gap-order constraints. The formula F' will be the disjunction of the conjunctive formulas representing the equivalence classes that satisfy F. Clearly the size of each conjunctive formula is less than the size of F. This proves the theorem. □

Lemma 5.1 shows that the formulas with integer order cannot distinguish between gaps than are more than exponential in their size. We show in the following example as a lower bound a formula which does distinguish between gaps that are up to an exponential in the size of the formula.

Many possible formulas can be found with that property. Our example is quite simple and is in disjunctive normal form. That is an improvement over previous examples of hard formulas. That is, Ferrante and Rackoff's method of writing short formulas that define complicated properties exactly exploit the fact that the formula is not in normal form. If their example were put into normal form its size would increase exponentially.

Example 5.1

$$Gap_{2^n}(s,t) \equiv \exists x \forall y \; Gap_{2^n-1}(x,y) \vee (s < y < t) \vee (t < y < s)$$

$$Gap_1(s,t) \equiv s \neq t$$

The formula $Gap_{2^n}(s,t)$ is satisfied if and only if $\mid t - s \mid \geq 2^n$. This is easy to prove by induction as follows. For $n = 0$, $Gap_{2^0}(s,t) \equiv Gap_1(s,t)$ is satisfied if and only if $\mid t - s \mid \geq 1$, as claimed. For Gap_{2^n} without loss of generality let $s < t$. We reason as follows.

(if) Suppose that $t - s \geq 2^n$. Then let $x = s + 2^{n-1}$. Then for all $y \leq s$ the condition $x - y \geq 2^{n-1}$ holds. Also, for all $y \geq t$ the condition $y - x \geq 2^{n-1}$ holds. For all $s < y < t$ the second condition holds. Hence $Gap_{2^n}(s,t)$ holds.

(only if) Suppose that $t - s < 2^n$. For any x if $s < x < t$, then either $x - s < 2^{n-1}$ and $Gap_{2^n-1}(x,t)$ is false or $t - x < 2^{n-1}$ and $Gap_{2^n-1}(x,s)$ is false, and since $s < s < t$ and $s < t < t$ are both false, the formula must be false when either $y = s$ or $y = t$. If $x \leq s$ or $x \geq t$, then the formula is false because each of the three disjuncts fails when $y = x$. □

5.2 Query Evaluation Formulated as a Decision Problem

When analyzing the complexity of query evaluation, it is common to reformulate the query evaluation problem as a decision problem.

Data Complexity: For each fixed program Π we define a language $L_\Pi = \{(t,d) : t \in \Pi(d)\}$. The language L_Π consists of the set of strings which are pairs of a regular relational database tuple t and an input database d (written as a string of generalized relational database tuples) such that t is in the semantically defined output of Π on d (that is, independent of any particular representation). Following Chandra and Harel [31] and Vardi [155] we call *data complexity* the computational complexity of deciding whether a pair (t,d) is in the language L_Π. Data complexity is a commonly used measure in databases because it expresses the intuition that usually the size of the database dominates by several orders of magnitude the size of the query program.

δ	**Constraints**	**Relational Calculus**	**Datalog**	**Stratified Datalog**
D		AC_0 (s) [81] LOGSPACE (s) [31]	PTIME-comp (s) [7, 31]	PTIME-comp (s) [12, 33]
Q	$\neq, \leq, <$	AC_0 [81] LOGSPACE [82, 54]	PTIME-comp [82]	PTIME-comp [82]
N,Z	$\equiv_k$ for fixed set of k's	in PTIME [78, 157]	PTIME-comp [147]	PTIME-comp [145]
N,Z	$1S$	in NC [120]	PSPACE-comp (s) [38]	PSPACE-comp (s) [36]
$P(D)$	$\subseteq, k \in, k \notin$	in PTIME ($\exists$ only) [134]	DEXPTIME-comp [122]	DEXPTIME-comp (s) [124]
B_m	$=_{B_m}$	in PTIME ($\exists$ only) [82]	2^{2^n}-comp [82, 125]	in $2^{2^{2^n}}$ (s) [125]
Z	$\neq, \leq, <, <_k$	in NC [120]	DEXPTIME-comp [121, 122]	non-elem-comp (s) [123]
R	$<, +$	in NC [55]	undec.	undec.
N,Z	$<, +, \equiv_k$	in $O(2^{n^{c^n}})$ [118, 21]	undec.	undec.
R	$+, *$	NC [82, 19, 119]	undec.	undec.
Z	$+, *$	undec. [62]	undec.	undec.
Q	$+, *$	undec. [128]	undec.	undec.

Fig. 3. The Data Complexity of Constraint Query Languages

Figure 3 summarizes some of the known data complexity results for relational and constraint query languages. The yet unmentioned domain symbols in the table are: D for any discrete domain (for example the integers), B_m for free Boolean algebra with m elements, and $\mathbf{P(Z)}$ for sets of integers. In the figure ($\exists only$) means that only relational calculus without $\neg$ and $\forall$ is considered and (s) means other syntactical restrictions. The $=$ is not listed as a constraint in any row of the table, but it is assumed to be present in each row.

Most of the complexity results in Figure 3 assume Turing machines as the computational model. The exceptions are the NC-ness results which assume the PRAM model of computation and the AC_0 results which assume random-access alternating Turing machines [14, 30].

The known data complexity results for relational databases are shown in the first row. For relational databases (safe) relational calculus has LOGSPACE [31] and AC_0 [81] data complexity in the PRAM and Alternating Turing machine models, respectively. Also, for relational databases both Datalog [7, 31] and Stratified Datalog [12, 33] have PTIME-complete data complexity.

In the Figure 3 the constraint $<_k$ where k is a nonnegative integer is called a gap-order constraint. For variables x_i and x_j, the constraint $x_i <_k x_j$ is true if and only if $x_i + k < x_j$ is true.

[120] considered relational calculus with $\neq, \leq, <$ constraints on the integers in the input database and showed it to be decidable within PTIME data complexity. Theorem 5.2 in this paper is an improvement of that result to NC and with the remarks after it an extension with gap-order constraints within the input database. (Note that without the extension the closed-form requirement is not satisfied.)

Koubarakis [91] considered the first-order language of difference constraints, i.e. only constraints of the form $x_i - x_j > c$ for x_i, x_j integer variables and c integer constant. Koubarakis shows that the expression complexity of this language is PSPACE-complete and provides a quantifier elimination method. When the reals are considered as the domain, the expression complexity remains the same.

It turns out that the expressive power of the languages in [120] and [91] in the case of integers is the same. That is because $x_i - x_j > c$ is equivalent to $x_j <_c x_i$ if $c \geq 0$ and is equivalent to $\neg(x_i <_{(-c-1)} x_j)$ if $c < 0$. Furthermore, any gap-order constraint $x_i <_c x_j$ can be expressed by the subformula $\exists z_1 \ldots \exists z_c (x_i < z_1 \wedge z_1 < \ldots < z_c \wedge z_c < x_j)$.

The constraint $\equiv_k$ is the usual modulus constraint on integers, that is, $x_i \equiv_k x_j$ is true if and only if the remainder of x_i divided by k equals the remainder of x_j divided by k where k is a positive integer. The set of solutions $\{c + kn : n \in \mathbf{Z}\}$ of a modulus constraint of the form $x \equiv_k c$ is called a linear repeating point.

Datalog with a successor function, which is denoted $1S$ in the table, has the safety requirement that addition is applied only to the first argument of each relation. This distinguished argument of relations is called a temporal argument. The languages of [78, 38, 39] can express many interesting temporally recurring events. For example, they can express that employees in a company get paid every week. In the language of [39] this would be:

$$paid(t, p) \text{ :— } employee(p, c), paid(t_2, p), t = t_2 + 1 + 1 + 1 + 1 + 1 + 1 + 1$$

However, the languages are different in expressive power. [78] cannot express transitive closure queries, while [39] cannot express the difference of relations. The expressive power of these languages is compared in [16] which also present an undecidable language. The expressive power of point and interval-based query languages is compared in [146].

[147] also combines modulus constraints with integer gap-order constraints. Datalog with gap-order constraints is shown to have a closed form and a DEXPTIME-complete data complexity in [121, 122]. The expression complexity of the same

language is shown to be in DEXPTIME in [47]. Recent work adds stratified negation to Datalog with gap-order constraints with some syntactical safety restrictions. The resultant language has a non-elementary data complexity [123]. The relationship between syntactical safety as given in [123] and semantical safety is discussed in [136] and it is shown that syntactical safety cannot be extended to include all semantically safe queries.

Datalog with set order constraints of the form $U \subseteq V$, $k \in U$, and $k \notin U$, where k is an integer (or element of some other infinite domain D) and U, V are set variables or sets of constants, is considered in [134, 122, 124]. The first reference shows that conjunctive queries have PTIME, and the other two show that Datalog and safe stratified Datalog have a DEXPTIME-complete data complexity.

The data complexity of relational calculus and Datalog queries with Boolean equality constraints over a free Boolean algebra with m generators is analyzed in [82, 125].

Some related results that do not appear in the table are works on deciding set constraints. The set constraints considered in [5] are much more general than the ones in the table. Quantifier elimination is not possible in the more general case considered there. There sets appear as leaves of functional terms and unification and other methods are considered for the decision problem only.

When we have addition and multiplication, then we can express the order relation in both the **R** and the **Z** case. When the domain is **Z**, then we can also express exponentiation. The undecidability results in the table are based in part on these observations.

5.3 From the Decision Problem to Closed-Form Evaluation

Next we give an example of how one can translate the complexity results for the decision problem of a first-order theory with constraints to a closed-form evaluation result for relational calculus queries.

Theorem 5.2 Let Π be any fixed Relational Calculus program. Then Π can be evaluated in NC in the size of any generalized database with integer order constraints such that the output will be a generalized relation with integer gap-order constraints.

Proof: First using Proposition 4.1 translate the query evaluation problem into a quantifier elimination problem in the theory of integer order. Also transform the formula into a prenex normal form, i.e. where all the quantifiers occur only at the front. A formula can be put into prenex-form in NC.

The theorem follows from Theorem 5.1 taking advantage of the fact that the

quantifier depth and the number of free variables are fixed constants. When we test a representative of each equivalence class whether it satisfies the formula for each new existential variable a_{k+i} we need to test only $O((2(k-i)+1)2^d(2^{d+1}+3)m)$ many choices depending where a_{k+i} is inserted into the current order of the variables $a_1, \ldots, a_{k+i-1}$ and the m constants. Since d and k are fixed constants, employing an exhaustive search on these possibilities the total number of cases is still only $O(m^d)$. Since $m < n$ the total number of cases is only a polynomial in the size of the generalized database input. Each case should be tested on a quantifier-free formula which can be done individually in NC and all of them in parallel in NC. Once the value of each case is known, the quantifiers can be evaluated in NC too. □

Remark 1: It is possible to improve the above result by allowing the generalized database input to contain gap-order constraints. If the maximum gap-value in the input database is some constant c then similarly to Example 5.1 we can express it by a formula which adds only $\log c$ quantifier-depth and size. That is, the problem with gap-order constraints in the input generalized database can be still reduced to evaluating in the theory of integer order a formula that has $O(k + \log c)$ quantifier depth. Hence this problem is also in NC.

Remark 2: The theory of integers with a successor relation, i.e. $+1$, can be also reduced to case of the theory of integer order. That is because any constraint of the form $x+1 = y$ in the former can be expressed by the subformula $(x < y) \wedge \neg\exists z(x < z \wedge z < y)$ in the latter. This implies that with successor constraints also query evaluation is in NC.

The following is the analogue of Theorem 5.2 in the case of algebraic queries.

Theorem 5.3 Let Π be any fixed generalized relational algebra query. Then Π can be evaluated in NC in the size of any generalized database input that is in normal form.

Proof: The proof is by induction on the parse tree of the relational algebra expression showing that the temporary relation T_i associated with each internal node i has a size that is polynomial in the size of the input relations (at the leaves) below it and that T_i can be evaluated in NC. To show polynomial size is easy because generalized select, project, and rename can only decrease the number of mm-gapgraphs in the temporary relation and join can only return at most $n * m$ mm-gapgraphs where n and m are the number of mm-gapgraphs in the two input relations. By Lemma 4.3 the number of mm-gapgraphs can also grow only polynomially by each negation (and therefore difference). Since the relational algebra expression is fixed the output relation at the root node must have a number of mm-graphs that is polynomial in the size of the input generalized database. Finally, we note that the size of each mm-gapgraph is bounded by a constant because the number of arguments in each relation is fixed.

To show NC-ness of the evaluation it is enough to show that each generalized algebra operation can be evaluated in NC in the size of the input relation(s). For select, project, and rename and join that is obvious. Negation can be done in NC similarly to Theorem 5.2, that is, we precompute and test in parallel all $O(n^{k^2})$ mm-gapgraphs whether it satisfies the negated formula that is the disjunction of all the mm-gapgraphs in the input relation. □

Unfortunately, there are no general techniques to use data complexity results for Datalog and Stratified Datalog to bound the time required by a closed-form evaluation algorithm. Even worse, for constraint query languages a finite data complexity does not imply that there is a closed-form evaluation.

5.4 Optimization Problems

Many optimization methods are based on the idea of transforming programs into semantically equivalent ones that however can be evaluated faster. Transformation from a relational calculus query to an algebra discussed above is one example. Other examples includes the propagation of selection and projection operators and the ordering of join operators in both relational calculus and similarly by magic set techniques in Datalog. Recent extensions of these techniques for constraint queries can be found in [26, 27, 71, 97, 98, 106, 133, 144, 152].

The testing for equivalence during program transformations often is accomplished by testing for query containment first. Recall that each relational calculus program ϕ defines a mapping from any input database d (which may be represented finitely) to an output database $\phi(d)$ (which may also be represented finitely). We say that a program ϕ_1 is contained in program ϕ_2, denoted $\phi_1 \subseteq \phi_2$, if and only if for each input database d, all the tuples in $\phi_1(d)$ are also in $\phi_2(d)$. The *containment* problem is: Given two programs ϕ_1, ϕ_2 decide whether $\phi_1 \subseteq \phi_2$.

The containment problem is particularly important for a subclass of relational calculus queries that are formed without the connectives $\neg$ and $\vee$ and the quantifier $\forall$. These queries are called *Conjunctive Queries*.

The containment problem is NP-complete for conjunctive queries without constraints [34]. It remains NP-complete with only linear equality constraints [82]. The problem becomes Π_2^p-complete with dense linear order constraints [151]. The containment problem for conjunctive queries with quadratic equation constraints over the reals is Π_2^p-hard [82].

Besides query transformations, good indexing techniques on the generalized tuples in the input database relations can also improve the efficiency of query evaluation. A typical problem in query evaluation is 1-dimensional range searching, which asks to return all tuples that have an x attribute with values between two constants.

Range searching on regular relations can be implemented by B-trees and and B^+-trees [17, 46] which are good in minimizing the number of accesses to secondary storage. Range searching requires $O(\log_B N + K/B)$ secondary memory accesses in the worst case, where B is the number of tuples in a block, N is the total number of tuples in the relation searched, and K is the number of tuples returned.

The problem of range searching on generalized relations can be implemented using grid-files, quad-trees, and R-trees (see the books [112, 129, 130] for a review of these and other spatial data structures). The idea is to index each generalized tuple by the interval that is the projection of the tuple onto the x axis. The data structures mentioned above can easily accommodate insertions and deletions as well as range searching. Unfortunately, they work well only with main-memory storage but do not give worst-case guarantees on the number of accesses to secondary storage. [84] gives a data structure and an algorithm with optimal worst-case performance with regard to secondary storage accesses. The data structure is static in the sense that it allows insertions but no deletions. [117, 116] also considers multi-dimensional searching, i.e. range searching when several attributes are involved.

6 Expressive Power

Relational calculus queries without constraints (see Section 3.2) can be evaluated using a quantifier elimination procedure over the first-order theory of equality if in the input database each relation is a finite set of regular tuples. However, a quantifier elimination procedure may be slow compared to the usual evaluation of "safe" relational calculus queries (see [3, 149] for definition of safe). The efficiency of "safe" queries is due primarily to the fact that in them the quantifiers can be restricted to range over the set of constants occurring explicitly in the query, called the *active domain*, denoted δ_A, without a change in the output relation.

Using a version of Ehrenfeucht-Fraïssé games, Aylamazyan, Gilula, Stolboushkin and Schwartz showed that in any relational calculus query (safe or unsafe) it suffices to let quantifiers range over the active domain plus q number of additional constants from the full domain δ where q is the number of quantified variables [13]. This means that we can evaluate "unsafe" relational calculus queries with almost the same efficiently as safe relational queries.

Let's call active-domain semantics the mapping defined as in Section 3.2 but δ replaced by δ_A. Hull and Su considered whether relational calculus queries under the unrestricted and the active-domain semantics have the same expressive power. They found that when the input database consists of relations with a finite set of regular tuples, then the answer is yes, that is, for each unrestricted query it is possible to find an active-domain query that gives always the same output [72]. Paredaens, Van den Bussche and Van Gucht show the equivalence

when the input database consists of generalized relations with linear inequality constraints over the reals [111].

Another interesting question is the relative expressibility of relational calculus queries over different types of generalized databases. The problem of finding a *convex hull* and a *voronoi diagram* is expressible in relational calculus with polynomial inequality constraints over the reals. These queries cannot be expressed in relational calculus without generalized databases and constraints. That is because the latter queries are always *generic* or isomorphism preserving, that is, if two input databases are isomorphic, then their outputs are also isomorphic [31]. Clearly, this property of genericity is lost when constraints are added. Necessary modifications of the concept of genericity are investigated in [95, 109, 111].

Negative results are also interesting. The strongest negative result is that relational calculus with polynomial constraints over the real numbers cannot express even simple recursive queries like connectivity, transitive closure and parity of a relation [18]. This results extends earlier work on linear inequality constraints [6, 68, 94], and on rational numbers and order constraints [65, 137].

Finally, another negative result from [65] is that valid sentences of the first-order predicate calculus when relations range over finitely representable databases is undecidable. This case is similar to the finite relations case, that is, validity here is also co-r.e. and not in r.e.

7 Further Extensions

There are a number of recent extensions of the constraint data model. Among these are the addition of (1) integrity constraints, (2) aggregation operators, (3) indefinite information (4) complex objects, and (5) spatial and topological databases.

Integrity constraints play an important part in regular relational database design and have been investigated in great detail (see [143]). Dependency theory among integrity constraints for generalized databases is however a fairly open area, which was investigated only recently [15, 103]. [15] defines *constraint generating dependencies* and studies the computational complexity of their implication and consistency problems. Let $emp(name, salary, boss)$ be a relation storing information about employees in a company. An example of a simple case of constraint generating dependency is

$$\forall n_1 \forall s_1 \forall b_1 \forall n_2 \forall s_2 \forall b_2 ((emp(n_1, s_1, b_1) \wedge emp(n_2, s_2, b_2) \wedge b_1 = n_2) \rightarrow (s_1 < s_2))$$

meaning that bosses always earn more than their employees. Even this is more general than typical integrity constraints on relational databases because it assumes that the domain of the second attribute is an interpreted domain with

order. (The usual assumption is an uninterpreted domain with no constraint relations defined on it.)

Aggregation operators [87] have an important use in practical database query languages such as SQL. It is a challenging problem to define meaningful and efficiently evaluable aggregation operators on generalized databases. Maximum, minimum and area were proposed as aggregation operators in [95]. More recent work on this problem appears in [40, 37, 67].

Indefinite information represented by null values is also a practical concern. Null values represent either unknown or unexisting values. Evaluation of constraint queries with null values is considered in [90, 91, 134].

Complex values are important for the convenient representation of many kind of real-life data [3]. Values such as Booleans, strings, integer, real or rational numbers are all scalar values. Complex values are built from scalar values using in a nested way set and tuple constructors. It is a challenge to define constraints and find good constraint solving algorithms for complex objects. The introduction of set constraints and a quantifier elimination algorithm on set variables in [134, 122] can be considered as a preliminary step in that direction. In [134] complex values are further enhanced by object identifiers, leading to an interesting combination of constraint and object-oriented programming. Another proposal to combine constraints and objects is presented in [24]. Nested databases with dense order constraints are considered in [66].

Recent work on spatial and topological databases that extend the constraint data model can be found in [93, 108, 126]. In [108] constraint queries where variables range over regions instead of points are considered. In [126] constraint database and knowledge-base change operators, including revision, update, and arbitration are described. A survey on spatial databases can be found in [110].

8 Prototype Systems

It is encouraging to see that some prototype constraint database systems are just starting to appear. The constraint database system DISCO (short for Datalog with integer and set constraints) [25] implements two data types: integers and (finite and cofinite) sets of integers. In DISCO the following constraint relations are allowed: (1) between integers variables and constants: $=, \neq, <, \leq, >, \geq$ and gap-order constraint $<_k$ (see Section 5.2), (2) between set variables and constants: $=, \subseteq$, and (3) between integer constants c and set variables or constants X: $c \in X$ and $c \notin X$. Seminaive evaluation and projection and selection pushing is implemented in DISCO. The DISCO system was used recently for example in genomic database applications [127].

An implementation of Datalog with periodicity constraints was described re-

cently in [144] . Also, a compile-time constraint-solving method is implemented within the DeCoR database system in [63]. The C^3 constraint object-oriented system is also being implemented recently [28]. These prototype systems are important feasibility demonstrations for the theoretical ideas about constraint databases. They also may give interesting feedback for further theoretical study.

9 Open Problems

There are obviously many research topics that need further consideration. The list below collects only some of the most interesting and it seems difficult open problems concerning constraint query languages and generalized databases.

(1) What is the relative expressive power of relational calculus queries over generalized databases with polynomial inequality constraints over the reals under the unrestricted vs. the active-domain semantics? This problem was studied in [109].

(2) What is the size of the generalized database output for Datalog queries with integer order constraints, i.e. in $Th(Z, <)$? This problem was studied in [121]. Note that without constraints, this problem reduces in an obvious way to the data complexity problem. A similar reduction however still needs to be shown when constraints are present.

(3) What is the data complexity of relational calculus with only linear inequality constraints, i.e., in $Th(R, +, <)$? For *k-bounded* queries, that is, queries where the number of occurrences of the addition symbol in each constraint is bounded, it was shown that the problem is in AC_0 in [65]. Is the data complexity still in AC_0 without this restriction?

(4) Ajtai and Gurevich [9] showed that the queries that are expressible in both Datalog and relational calculus over regular databases are those that are *bounded*. Here bounded means that the number of iterations needed in the naive evaluation of the query is always a constant for any input database [3, 79]. Is this result true also if we allow generalized databases?

(5) What is the upper bound for the computational complexity of testing containment of conjunctive queries with quadratic equation constraints? The lower bound of Π_2^p-hard is shown in [82].

(6) Can we design a data structure that implements insertion, deletion, and range searches with optimal worst-case access to secondary storage? This problem is investigated in [84, 117, 116].

Acknowledgement: I thank the editors, Leonid Libkin and Bernhard Thalheim, for their encouragement in writing this survey. I also thank the referees for numerous helpful comments.

References

1. S. Abiteboul, C. Beeri. On the Power of Languages for the Manipulation of Complex Objects. *INRIA Research Report 846*, 1988.
2. S. Abiteboul and P. Kanellakis. Database Theory Column: Query Languages for Complex Object Databases. *SIGACT News*, 21, pp. 9–18, 1990.
3. S. Abiteboul, R. Hull and V. Vianu. Foundations of Databases. *Addison-Wesley*, 1994.
4. S. Abiteboul and V. Vianu. Datalog Extensions for Database Queries and Updates. *J. Comput. System Sci.*, **43** (1991), pp. 62–124.
5. A. Aiken. Set Constraints: Results, Applications and Future Directions. *Proc. 2nd Workshop on Principles and Practice of Constraint Programming*, 171–179, 1994.
6. F. Afrati, S.S. Cosmadakis, S. Grumbach, G.M. Kuper. Linear vs. Polynomial Constraints in Database Query Languages. *Proc. 2nd Workshop on Principles and Practice of Constraint Programming*, 152–160, 1994.
7. A.V. Aho, J.D. Ullman. Universality of Data Retrieval Languages. *Proc. 6th ACM Symp. on Principles of Programming Languages*, 110–117, 1979.
8. H. Aït-Kaci, A. Podelski. Towards a Meaning of LIFE. *Journal of Logic Programming*, 16, 195–234,1993.
9. M. Ajtai, Y. Gurevich. Datalog vs. First Order. *Journal of Computer and Systems Sciences*, 1994.
10. K.R. Apt. Logic Programming. *Handbook of Theoretical Computer Science*, Vol. B, chapter 10, (J. van Leeuwen editor), North-Holland, 1990.
11. K.R. Apt, M.H. van Emden. Contributions to the Theory of Logic Programming. *J. ACM*, Vol. 29 (3), 841–862, 1982.
12. K.R. Apt, H.A. Blair, A. Walker. Towards a Theory of Declarative Knowledge, in: J. Minker, ed., Foundation of Deductive Databases and Logic Programming. Morgan Kaufmann, 1988.
13. 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 (in Russian). *Proc. USSR Acad. of Science (Doklady)*, 286(2):308–311, 1986.
14. D.A. Barrington, N. Immerman, H. Straubing. On Uniformity within NC^1. *Journal of Computer and System Sciences*, 41:274–306,1990.
15. M. Baudinet, J. Chomicki, P. Wolper. Constraint-Generating Dependencies. *Proc. 5th Int. Conf. on Database Theory*, 322–337, 1995.
16. M. Baudinet, M. Niezette, P. Wolper. On the Representation of Infinite Temporal Data and Queries. *Proc. 10th ACM Symp. on Principles of Database Systems*, 280–290, 1991.
17. R. Bayer, E. McCreight. Organization of Large Ordered Indexes. *Acta Informatica*, 1:173–189, 1972.
18. M. Benedikt, G. Dong, L. Libkin, L. Wong. Relational Expressive Power of Constraint Query Languages. *Proc. 15h ACM Symp. on Principles of Database Systems*, 5–16, 1996.
19. M. Ben-Or, D. Kozen, J. Reif. The Complexity of Elementary Algebra and Geometry. *Journal of Computer and System Sciences*, 32:251–264, 1986.
20. C. Bell, A. Nerode, R. Ng, V.S. Subrahmanian. Implementing Deductive Databases by Linear Programming. *Proc. 11h ACM Symp. on Principles of Database Systems*, 283–292, 1992.

21. L. Berman. Precise Bounds for Presburger Arithmetic and the Reals with Addition. *Proc. 18th IEEE FOCS*, pp. 95-99, 1977.
22. A.H. Borning. The Programming Language Aspects of ThingLab, A Constraint-Oriented Simulation Laboratory. *ACM TOPLAS* 3:4:353–387, 1981.
23. A. Brodsky, J. Jaffar, M.J. Maher. Toward Practical Constraint Databases. *Proc. 19th VLDB*, 322–331, 1993.
24. A. Brodsky, Y. Kornatzky. The *Lyric* Language: Querying Constraint Objects. *Proc. SIGMOD*, 35–46, 1995.
25. J. Byon, P.Z. Revesz. DISCO: A Constraint Database System with Sets. *Proc. Workshop on Constraint Databases and Applications*, Springer-Verlag LNCS 1034, 68–83, 1995.
26. A. Brodsky, C. Lassez, J.L. Lassez, M.J. Maher. Separability of Polyhedra for Optimal Filtering of Spatial and Constraint Data. *Proc. 14th Symp. on Principles of Database Systems*, 1995.
27. A. Brodsky, Y. Sagiv. Inference of Inequality Constraints in Logic Programs. *Proc. 10th ACM Symp. on Principles of Database Systems*, 227–241, 1991.
28. A. Brodsky and V. Segal. The C^3 Constraint Object-Oriented Database System: An Overview, 134–159, In: [61].
29. A.R. Bruss, A.R. Meyer. On Time-Space Classes and their Relation to the Theory of Real Addition. *Proc. 10th ACM STOC*, pp. 233-239, 1978.
30. S.R. Buss. The Formula Value Problem is in ALOGTIME. *Proc. 19th ACM STOC*, pp. 123-131, 1987.
31. A.K. Chandra, D. Harel. Computable Queries for Relational Data Bases. *Journal of Computer and System Sciences*, vol. 21, 156–178, 1980.
32. A.K. Chandra, D. Harel. Structure and Complexity of Relational Queries. *Journal of Computer and System Sciences*, vol. 25, 99–128, 1982.
33. A.K. Chandra, D. Harel. Horn Clause Queries and Generalizations. *Journal of Logic Programming*, vol. 2, 1–15, 1985.
34. A.K. Chandra, P.M. Merlin. Optimal Implementation of Conjunctive Queries in Relational Databases. *Proc. ACM STOC*, 77–90, 1977.
35. J. Chomicki. Polynomial Time Query Processing in Temporal Deductive Databases. *Proc. 9th ACM Symp. on Principles of Database Systems*, 379–391, 1990.
36. J. Chomicki. *Functional Deductive Databases: Query Processing in the Presence of Limited Function Symbols*, Ph.D. Thesis. Rutgers University, 1990.
37. J. Chomicki, D. Goldin, G. Kuper. Variable Independence and Aggregation Closure. *Proc. 15th ACM Symp. on Principles of Database Systems*, 40–48, 1996.
38. J. Chomicki, T. Imielinski. Relational Specifications of Infinite Query Answers. *Proc. ACM SIGMOD*, 174–183, 1989.
39. J. Chomicki, T. Imielinski. Finite Representation of Infinite Query Answers. *ACM Transactions of Database Systems*, 181–223, vol. 18, no. 2, 1993.
40. J. Chomicki, G. Kuper. Measuring Infinite Relations. *Proc. 14th ACM Symp. on Principles of Database Systems*, 78-85, 1995.
41. E.F. Codd. A Relational Model for Large Shared Data Banks. *CACM*, 13:6:377–387, 1970.
42. J. Cohen. Constraint Logic Programming Languages. *CACM*, 33:7:52–68, 1990.
43. A. Colmerauer. An Introduction to Prolog III. *CACM*, 33:7:69–90, 1990.

44. A. Colmerauer, H. Kanoui, and M. Van Caneghem. Prolog, Bases Th'eoriques et D'evelopements Actuels. *Techniques et Sciences Informatiques*, 2:4:271–311, 1983.
45. G.E. Collins. Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition. *Proc. 2nd GI conference on Automata Theory and Languages*, LNCS 33, pp. 512-532, Springer-Verlag, 1975.
46. D. Comer. The Ubiquitous B-Tree. *Computing Surveys*, 11:2:121–137, 1979.
47. J. Cox, K. McAloon. Decision Procedures for Constraint Based Extensions of Datalog. In: *Constraint Logic Programming*, MIT Press, 1993.
48. J. Cox, K. McAloon, C. Tretkoff. Computational Complexity and Constraint Logic Programming. *Annals of Math. and AI*, 5:163–190, 1992.
49. 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.
50. K. Doets, *From Logic to Logic Programming*. MIT PRess, 1994.
51. A. Ehrenfeucht. An application of games to the completeness problem formalized theories. *Fund. Math*, 49, 1961.
52. M.H. van Emden, R.A. Kowalski. The Semantics of Predicate Logic as a Programming Language. *J. ACM*, Vol. 23 (4), 733–742, 1976.
53. H.B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.
54. J. Ferrante, J.R. Geiser. An Efficient Decision Procedure for the Theory of Rational Order. *Theoretical Computer Science*, 4:227–233, 1977.
55. J. Ferrante, C. Rackoff. A Decision Procedure for the First Order Theory of Real Addition with Order. *SIAM J. Comp*, 4:1:69–76, 1975.
56. J. Ferrante, C.W. Rackoff. *The Computational Complexity of Logical Theories*, Springer-Verlag (No. 718), 1979.
57. M.J. Fischer, M.O. Rabin. Super-Exponential Complexity of Presburger Arithmetic. *SIAM-AMS Proc. volume VII*, American Mathematical Society, 1974.
58. J-B.J. Fourier. Reported in: Analyse des travaux de l'Acadamie Royale des Sciences, pendant l'annee 1824, Partie mathematique, Histoire de l'Academie Royale des Sciences de l'Institut de France, Vol. 7, xlvii-lv, 1827. (Partial English translation in: D.A. Kohler. Translation of a Report by Fourier on his work on Linear Inequalities. Opsearch, Vol. 10, 38–42, 1973.)
59. R. Fraïssé. Sur les classifications des systèmes de relations. *Publ. Sci. Univ Alger*, I:1, 1954.
60. E. Freuder. Synthesizing Constraint Expressions. *CACM*, 21:11, 1978.
61. V. Gaede, A. Brodsky, O. Günther, D. Srivastava, V. Vianu, M. Wallace, (Eds.), *Constraint Databases and Applications*, Proc. Second Int. Workshop on Constraint Database Systems, Delphi, Greece, January 1997, and Workshop on Constraints and Databases, Cambridge, MA August 1996, Springer-Verlag LNCS 1191.
62. K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*. vol. 38, 173–198, 1931.
63. R. Gross, R Marti. Compile-time Constraint Solving in a Constraint Database System. *Proc. Post-ILPS'94 Workshop on Constraints and Databases*, 13–25, 1994.

64. Y. Gurevich, S. Shelah. Fixed-Point Extensions of First-Order Logic. *Annals of Pure and Applied Logic*, 32, 265–280, 1986.
65. S. Grumbach, J Su. Finitely Representable Databases. *Proc. 13th ACM Symp. on Principles of Database Systems*, 289–300, 1994.
66. S. Grumbach, J Su. Dense-Order Constraint Databases. *Proc. 14th ACM Symp. on Principles of Database Systems*, 66-77, 1995.
67. S. Grumbach, J Su. Towards Practical Constraint Databases. *Proc. 15h ACM Symp. on Principles of Database Systems*, 28–39, 1996.
68. S. Grumbach, J Su, C. Tollu. Linear Constraint Databases. *Proc. LCC*, 1994.
69. 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.
70. N. Heintze, J. Jaffar. Set Constraints and Set-Based Analysis. *Proc. 2nd Workshop on Principles and Practice of Constraint Programming*, 1-17, 1994.
71. R. Helm, K. Marriott, M. Odersky. Constraint-based Query Optimization for Spatial Databases. *Proc. 10th ACM Symp. on Principles of Database Systems*, 181–191, 1991.
72. R. Hull, J. Su. Domain Independence and the Relational Calculus. *Acta Informatica*, 31, 513–524, 1994.
73. N. Immerman. Relational Queries Computable in Polynomial Time. *Information and Control*, 68:86-104, 1986.
74. J. Jaffar, J.L. Lassez. Constraint Logic Programming. *Proc. 14th ACM POPL*, 111–119, 1987.
75. J. Jaffar, M.J. Maher. Constraint Logic Programming: A Survey. *J.Logic Programming*, 19 & 20, 503–581, 1994.
76. J. Jaffar, S. Michaylov, P.J. Stuckey, R.H. Yap. The CLP(R) Language and System. *ACM Transactions on Programming Languages and Systems*, 14:3, 339-395, 1992.
77. D.S. Johnson. A Catalogue of Complexity Classes. *Handbook of Theoretical Computer Science*, Vol. A, chapter 2, (J. van Leeuwen editor), North-Holland, 1990.
78. F. Kabanza, J-M. Stevenne, P. Wolper. Handling Infinite Temporal Data. *Proc. 9th ACM Symp. on Principles of Database Systems*, 392–403, 1990. Final version to appear in *Journal of Computer and System Sciences.*
79. P.C. Kanellakis. Elements of Relational Database Theory. *Handbook of Theoretical Computer Science*, Vol. B, chapter 17, (J. van Leeuwen editor), North-Holland, 1990.
80. P.C. Kanellakis. Tutorial: Constraint Programming and Database Languages. *Proc. 14th ACM Symp. on Principles of Database Systems*, 46-53, 1995.
81. P.C. Kanellakis, D.Q. Goldin. Constraint Programming and Database Query Languages. *Proc. 2nd TACS*, 1994.
82. P.C. Kanellakis, G.M. Kuper, P. Z. Revesz. Constraint Query Languages. *Journal of Computer and System Sciences*, vol. 51, no. 1, pp. 26-52, August 1995. (Preliminary version in *Proc. 9th ACM Symp. on Principles of Database Systems*, 299–313, 1990.)
83. P.C. Kanellakis, J.L. Lassez, V.J. Saraswat, eds., *Proc. Workshop on the Principles and Practice of Constraint Programming*, 1993.

84. P. C. Kanellakis, S. Ramaswamy, D. E. Vengroff, J. S. Vitter. Indexing for Data Models with Constraints and Classes. *Proc. 12th ACM Symp. on Principles of Database Systems*, 233–243, 1993.
85. L.G. Khachian. A Polynomial Algorithm in Linear Programming. *Soviet Math. Dokl.*, 20(1), 191-194, 1979.
86. M. Kifer. On Safety, Domain Independence, and Capturability of Database Queries. *Proc. International Conference on Databases and Knowledge Bases*, Jerusalem Israel, 1988.
87. A. Klug. Equivalence of Relational Algebra and Relational Calculus Query Languages having Aggregate Functions. *JACM*, 29:3:699–717, 1982.
88. A. Klug. On Conjunctive Queries Containing Inequalities. *JACM*, 35:1:146–160, 1988.
89. P. Kolaitis, C.H. Papadimitriou. Why not Negation by Fixpoint? *Proc. 7th ACM Symp. on Principles of Database Systems*, 231–239, 1988.
90. M. Koubarakis. Representing and Querying in Temporal Databases: the Power of Temporal Constraints. *Proc. Ninth International Conference on Data Engineering*, 1993.
91. M. Koubarakis. Complexity Results for First-Order Theories of Temporal Constraints. *Int. Conf. on Knowledge Representation and Reasoning*, 1994.
92. D. Kozen, C. Yap. Algebraic Cell Decomposition in NC. *Proc. 26th IEEE FOCS*, 515–521, 1985.
93. B. Kuipers, J. Paredaens, Jan Van den Bussche. On Topological Elementary Equivalence of Spatial Databases. *Proc. 6th Int. Conf. on Database Theory*, 432–446, Springer-Verlag LNCS 1186, 1997.
94. G.M. Kuper. On the Expressive Power of the Relational Calculus with Arithmetic Constraints. *Proc. 3rd Int. Conf. on Database Theory*, 202–211, 1990.
95. G.M. Kuper. Aggregation in Constraint Databases. *Proc. Workshop on the Principles and Practice of Constraint Programming*, 176–183, 1993.
96. W. Leler. *Constraint Programming Languages.* Addison Wesley, 1987.
97. A. Levy, I.S. Mumick, Y. Sagiv, O. Shmueli. Equivalence, Query Reachability and Satisfiability in Datalog Extensions. *Proc. 12h ACM Symp. on Principles of Database Systems*, 109–122, 1993.
98. A. Levy, Y. Sagiv. Constraints and Redundancy in Datalog. *Proc. 11h ACM Symp. on Principles of Database Systems*, 67–80, 1992.
99. C. Langford. Some Theorems on Deducibility. *Annals of Mathematics.* vol. 28, 16–40, 459–471, 1927.
100. L. Libkin, L. Wong. New Techniques for Studying Set Languages, Bag Languages and Aggregate Functions. *Proc. 13h ACM Symp. on Principles of Database Systems*, 155–166, 1994.
101. J.W. Lloyd. *Foundations of Logic Programming.* Spring, Berlin, 2nd ed., 1987
102. A.K. Mackworth. Consistency in Networks of Relations. *AI*, 8:1, 1977.
103. M. J. Maher and D. Srivastava. Chasing Constraint-Tuple Generating Dependencies. *Proc. 15h ACM Symp. on Principles of Database Systems*, 128–138, 1996.
104. Y. Matiyasevich. Enumerable Sets are Diophantine. *Doklady Akademii Nauk SSR.* vol. 191, 279–282, 1970.
105. U. Montanari. Networks of Constraints: Fundamental Properties and Application to Picture Processing. *Information Science*, 7, 1974.

106. I. S. Mumick, S. J. Finkelstein, H. Pirahesh, R. Ramakrishnan. Magic Conditions. *Proc. 9th ACM Symp. on Principles of Database Systems*, 314–330, 1990.
107. I.S. Mumick, O. Shmueli. Universal Finiteness and Satisfiability. *Proc. 13h ACM Symp. on Principles of Database Systems*, 190–200, 1994.
108. C.H. Papadimitriou, D. Suciu, V. Vianu. Topological Queries in Spatial Databases. *Proc. 15h ACM Symp. on Principles of Database Systems*, 81–92, 1996.
109. J. Paredaens, J.V.D. Bussche, D.V. Gucht. Towards A Theory of Spatial Database Queries. *Proc. 13h ACM Symp. on Principles of Database Systems*, 279–288, 1994.
110. J. Paredaens. Spatial Databases: The Final Frontier. *Proc. 5th Int. Conf. on Database Theory*, Springer-Verlag LNCS 893, 1995.
111. J. Paredaens, J.V.D. Bussche, D.V. Gucht. First-Order Queries on Finite Structures over the Reals. *Proc. LICS*, 1995.
112. F.P. Preparata, M.I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, 1985.
113. M. Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. Comptes Rendus, I. Congrès des Math. des Pays Slaves, Warsaw, 192–201, 395, 1929.
114. R. Ramakrishnan. Magic Templates: A Spellbinding Approach to Logic Programs. *Proc. 5th International Conference on Logic Programming*, 141–159, 1988.
115. R. Ramakrishnan, D. Srivastava, S. Sudarshan. CORAL: Control, Relations and Logic. *Proc. VLDB*, 1992.
116. R. Ramaswamy. Efficient Indexing for Constraint and Temporal Databases. *Proc. 6th Int. Conf. on Database Theory*, 419–431, Springer-Verlag LNCS 1186, 1997.
117. R. Ramaswamy, S. Subramanian. Path Caching: A Technique for Optimal External Searching *Proc. 13h ACM Symp. on Principles of Database Systems*, 25–35, 1994.
118. C.R. Reddy, D.W. Loveland. Presburger Arithmetic with Bounded Quantifier Alternation. *Proc. ACM Symp. on Theory of Comp.*, 320-325, 1978.
119. 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.
120. P. Z. Revesz. *Constraint Query Languages.* Ph.D. Thesis. Brown University, 1991.
121. P. Z. Revesz. A Closed Form Evaluation for Datalog Queries with Integer (Gap)-Order Constraints, *Theoretical Computer Science*, vol. 116, no. 1, 117-149, 1993. (Preliminary version in *Proc. Third International Conference on Database Theory*, Springer-Verlag LNCS 470, 187–201, 1990.)
122. P. Z. Revesz. Datalog Queries of Set Constraint Databases. *Proc. Fifth International Conference on Database Theory*, Springer-Verlag LNCS 893, 425–438, 1995.
123. P. Z. Revesz. Safe Stratified Datalog with Integer Order Programs. *Proc. First International Conference on Principles and Practice of Constraint Programming*, Springer-Verlag LNCS 976, 154-169, 1995.

124. P. Z. Revesz. Safe Query Languages for Constraint Databases. *ACM Transactions on Database Systems*, March, 1998, to appear.
125. P. Z. Revesz. The Evaluation and the Computational Complexity of Datalog Queries of Boolean Constraint Databases, *International Journal of Algebra and Computation*, to appear.
126. P. Z. Revesz. Model-Theoretic Minimal Change Operators for Constraint Databases. *Proc. 6th Int. Conf. on Database Theory*, 447–460, Springer-Verlag LNCS 1186, 1997.
127. P. Z. Revesz. Refining Restriction Enzyme Genome Maps. *Constraints*, vol. 2, no. 3-4, pp. 361-375, December 1997. (Preliminary version in [61].)
128. J. Robinson. Definability and Decision Problems in Arithmetic. *Journal of Symbolic Logic*, Vol. 14, pp. 98–114, 1949.
129. H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS.* Addison-Wesley, Reading MA, 1990.
130. H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, Reading MA, 1990.
131. V.A. Saraswat. *Concurrent Constraint Programming Languages.* Ph.D. thesis, Carnegie Mellon University, 1989.
132. D. Srivastava. Subsumption and Indexing in Constraint Query Languages with Linear Arithmetic Constraints. *Proc. 2nd International Symposium on Artificial Intelligence and Mathematics*, 1992.
133. D. Srivastava, R. Ramakrishnan. Pushing Constraint Selections. *Proc. 11h ACM Symp. on Principles of Database Systems*, 301–315, 1992.
134. D. Srivastava, R. Ramakrishnan, P.Z. Revesz. Constraint Objects. *Proc. 2nd Workshop on Principles and Practice of Constraint Programming*, Springer-Verlag LNCS 874, 274–284, 1994.
135. G.L. Steele. *The Definition and Implementation of a Computer Programming Language Based on Constraints.* Ph.D. Thesis, MIT, AI-TR 595, 1980.
136. A. Stolboushkin and M.A. Taitslin. Finite Queries do not have Effective Syntax. *Proc. 14th ACM Symp. on Principles of Database Systems*, 277–285, 1995.
137. A. Stolboushkin and M.A. Taitslin. Linear vs. Order Constraint Queries over Rational Databases. *Proc. 15h ACM Symp. on Principles of Database Systems*, 17–27, 1996.
138. P.J. Stuckey, S. Sudarshan. Compiling Query Constraints. *Proc. 13h ACM Symp. on Principles of Database Systems*, 56–67, 1994.
139. I.E. Sutherland. *SKETCHPAD: A Man-Machine Graphical Communication System.* Spartan Books, 1963.
140. B.A. Trakhtenbrot. The Impossibility of an Algorithm for the Decision Problem on Finite Models. (In Russian) *Doklady Akademii Nauk SSR*, 70, 569–572, 1950.
141. A. Tarski. *A Decision Method for Elementary Algebra and Geometry.* University of California Press, Berkeley, California, 1951.
142. A. Tarski, F.B. Thompson. Some General Properties of Cylindrical Algebras. *Bulletin of the AMS*, 58:65, 1952.
143. B. Thalheim. *Dependencies in Relational Databases.* Teubner Verlagsgesellschaft, Stuttgart and Leipzig, 1991.
144. D. Toman. Top-Down Beats Bottom-Up for Constraint Based Extensions of Datalog. *Proc. ILPS*, 98–112, 1995.

145. D. Toman. *Foundations of Temporal Query Languages.*, Ph.D. Thesis. Kansas State University, 1995.
146. D. Toman. Point vs. Interval-Based Query Languages for Temporal Databases. *Proc. 15h ACM Symp. on Principles of Database Systems*, 58–67, 1996.
147. D. Toman, J. Chomicki, D.S. Rogers. Datalog with Integer Periodicity Constraints. *Proc. ILPS*, 1994.
148. S. Tsur and C. Zaniolo. LDL: A Logic-Based Data-Language. *Proc. VLDB*, pp 33-41, 1986.
149. J.D. Ullman. *Principles of Database and Knowledge-Base Systems.* Computer Science Press, vol. 1&2, 1989.
150. R. van den Dries. Remarks on Tarski's problem concerning $(\mathbf{R}, +, *, exp)$. In *Logic Colloquium*, North-Holland, 1982. Elsevier.
151. R. van der Meyden. The Complexity of Querying Indefinite Data about Linearly Ordered Domains. *Proc. 11th ACM Symp. on Principles of Database Systems*, 331–346, 1992.
152. A. Van Gelder. Deriving Constraints among Argument Sizes in Logic Programs. *Proc. 9th ACM Symp. on Principles of Database Systems*, 47–60, 1990.
153. P. Van Hentenryck. Constraint Logic Programming, *The Knowledge Engineering Review*, 6, 165–180,1989.
154. P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.
155. M.Y. Vardi. The Complexity of Relational Query Languages. *Proc. 14th ACM STOC*, 137–146, 1982.
156. P. Voda. Types of Trilogy. *Proc. 5th International Conference on Logic Programming*, 580–589, 1988.
157. H.P. Williams. Fourier-Motzkin Elimination Extension to Integer Programming Problems. In *Journal of Combinatorial Theory (A).* vol. 21, 118–123, 1976.

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