Constraint Programming and Database Query Languages

Paris C Kanellakis and Dina Q Goldin

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-94-31
June 1994

Constraint Programming and Database Query Languages

Paris C Kanellakis * and Dina Q Goldin *

Brown University, Providence, RI 02192, USA

Abstract. The declarative programming paradigms used in constraint languages can lead to powerful extensions of Codd's relational data model. The development of constraint database query languages from logical database query languages has many similarities with the development of constraint logic programming from logic programming, but with the additional requirements of data efficient, set-at-a-time, and bottom-up evaluation. In this overview of constraint query languages (CQLs) we first present the framework of [41]. The principal idea is that: "the k-tuple (or record) data type can be generalized by a conjunction of quantifier-free constraints over k variables". The generalization must preserve various language properties of the relational data model, e.g., the calculus/algebra equivalence, and have time complexity polynomial in the size of the data. We next present an algebra for dense order constraints that is simpler to evaluate than the calculus described in [41], and we sharpen some of the related data complexity bounds. We note that CQLs are applicable to spatial databases. This is because these languages have "spatial point set" as the semantics of their record data type and because existing multi-dimensional searching data structures can support I/O efficient access to sets of records. Finally, we observe that CQLs can be augmented with complex object data types, aggregate operations, and null-values, just like the relational data model.

1 From Constraint Programming to Database Querying

1.1 Declarative Programming with Constraints

Constraint programming paradigms are inherently declarative, since they implicitly describe computations by specifying how these computations are constrained. *Programming with constraints as primitives* (or constraint programming) is appealing because constraints are the normal language of discourse for many high-level applications. Pioneering work in constraint programming goes back to the early 1960's, e.g., Sutherland's SKETCHPAD [67]. The theme has been investigated since the 1970's, e.g., in artificial intelligence [31, 54, 55, 66], in graphical-interfaces [10], and in logic programming languages [37, 22, 27]. The relevant literature and contributions are too large to attempt a survey. Instead we will limit our exposition to recent applications in databases.

* Research was supported by ONR grant N00014-91-J-4052 ARPA Order No. 8225.

Perhaps one of the most important advances in constraint programming in the 1980's has been the development of Constraint Logic Programming (CLP) as a general-purpose framework for computations, e.g., in CLP($\Re$) [37], in Prolog III [22], and in CHIP [27, 72]. For more information on CLP see the surveys in [21, 51], the proceedings of logic programming conferences and of recent workshops on the subject, e.g., [42]. In particular, the advances in CLP are very relevant for database applications (which brings us to the subject of this paper). One reason is the many connections between databases and logic programming. Starting with the Japanese 5th Generation Project, these connections were explored in depth during the last decade.

The declarative style of database query languages is an important aspect of database systems, that has been at the core of the relational data model since Codd's pioneering work [20]. Indeed, having such a language for ad-hoc database querying is a requirement in today's commercial relational technology. Querying in Codd's relational calculus is a form of limited (but very successful) constraint programming. It is rather surprising that other advances in constraint programming have not yet influenced database query language design. However, we believe that this will soon change and there will be (in the 1990's) integration of constraint programming and databases.

One intuitive reason for the success of CLP is as follows. A strength of a typical logic programming language (e.g., Prolog) is its top-down, depth-first search strategy. The operation of first-order term unification, at the forefront of this search, is a special form of efficient constraint solving. Additional constraint solving increases the depth of the search and, thus, the effectiveness of the approach. Unfortunately, the bottom-up and set-at-a-time style of evaluation emphasized in relational databases, and more recently in knowledge bases, seems to contradict the top-down, depth-first intuition behind CLP.

The main contribution of the Constraint Query Language (CQL) framework in [41] was to demonstrate that it is possible to bridge the gap between: *bottom-up, efficient, declarative database programming* and *efficient constraint solving*. One key intuition came from CLP: *a conjunction of quantifier-free constraints over k variables, where k depends on the database schema and not the instance, is the correct generalization of the k-tuple or record or ground fact.* The technical tools for this integration were: *data complexity* [17, 73] from database theory, and *quantifier elimination* methods from mathematical logic [28, 29, 68, 23, 8, 48, 60].

More specifically, finite relations are generalized to finitely representable relations and appropriate algebras for their data manipulation can be developed in this framework. In many cases, the algebras have polynomial time data complexity. This use of data complexity (a common tool for studying expressibility in finite model theory) distinguishes the CQL framework from arbitrary (and inherently exponential) theorem proving [30, 9, 15].

Example 1. Let us illustrate database querying and introduce some of its notation. The meaning and use is (or rather should be) intuitively obvious. For formal definitions we refer the reader to [40, 69]. To see that constraints should be part of any database query language consider an example from [41].

This is a query program with *Expenses, Savings* and *Income* input relations, *Balanced* output relation, and a single linear equation constraint: x is user-id, f is amount spent for food, r for rent, m for miscellaneous, s for transfer to savings, w for wages, and i for interest. The intended output of this query is the list of user-ids whose checkbooks balance. As is typical in data processing applications, we assume that the input relations are much larger than the query program and they might reside on secondary storage.

In nonrecursive Datalog notation and using a single linear equation constraint we express the following "balanced checkbook" query.

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

$$C \equiv (f + r + m + s = w + i)$$

In relational technology even a little arithmetic, like the linear equation of this example, is outside the data model and requires special ad-hoc processing. Linear equations, although quite simple theoretically, can be interesting from a practical point of view. Also, as illustrated in [41], the simple case of linear equations can be related to the relational database theory theme of query homomorphisms. □

One point of the above example is that constraints integrate naturally in the core database query syntax (called tableaux in the relational databases of the 1970's and nonrecursive Datalog in the knowledge bases of the 1980's). Another point is that this constraint program has a natural analog in constraint maintenance. We can think of a scenario where the checkbook should always balance, and any update to the database would have to preserve the constraint C.

Constraint Programming vs Maintenance: In databases the term constraints is most often used in connection with "integrity constraints" and with their maintenance. Note the difference between constraint maintenance, where an update might trigger some additional computation (like in spread-sheets) and database retrieval, where a new relation is defined based on constraints. Constraint programming is a high-level programming style, which is often supported by integrity constraint maintenance algorithms. □

The CQL framework of [41] provided a unified view of some previous database research: for example, on the power of constraints for the implicit specification of temporal data [19], for extending relational algebra [33, 46], for magic set evaluation [58], and for logic program analysis [13]. CQLs also extend the approach to safe queries of [4, 35, 44, 58].

The example CQLs of [41] are applicable to spatial databases, where dense order is present. Temporal databases require the development of analogous frameworks for the theory of discrete order with constants see [39].

Complete vs Partial Information: The key concept in CQL, illustrated in the next subsection is that constraints describe *point-sets*, such that *all* their points are in the database. With the appropriate constraint theory these point-sets are accurate (and perhaps the most intuitive) representations of spatial objects. For efficiency reasons we require that these constraints be quantifier-free, just like the CLP axioms are quantifier-free. The [41] framework is thus one of *complete information.*

The problem of managing *partial information* in databases has received a fair amount of attention, see [40] for pointers to the database literature. For example, formulas with variables called *null-values* can be thought of as formulas representing many possible states (of which one is true). They would correspond to constraints that have existential quantifiers. In this sense, constraint programming has already been applied extensively to incomplete information databases. However, it is well known, that the existential quantification results in a number of difficulties with the efficient processing of null-values and partial information in general. It is easy to see that, just like the relational data model, constraint frameworks can be augmented with partial information constructs, such as null-values, but at the expense of data complexity.

Constraint logic programming paradigms are currently attracting a great deal of attention in languages for operations research applications [72] and have also impacted the field of concurrent programming language design [64]. The use of constraints for operations research and for concurrency is sometimes semantically different from their use here. For example: Constraints can be used to represent the many possible states (of which one is true) of a set of concurrent processes. Each individual concurrent process maintains and manipulates constraints that describe the *partial information* it has about the state of all processes. These semantic differences are due to the use of complete vs partial information. □

1.2 Spatial Database Applications

Manipulation of spatial data is an important application area (e.g., spatial or geographic databases) that requires both relational query language techniques and arithmetic calculations. Indexes for range searching and modeling of complex structures have been used to bridge the gap between declarative accessing of large volumes of spatial data and performing common computational geometry tasks. The resulting extended relational systems are somewhat ad-hoc. Object-oriented database technology has also been applied to spatial databases with mixed success. Although, object-oriented databases emphasize rich and flexible data types, they do not incorporate the geometry of point sets in their data models. This geometry is implicit in constraints and a particular strength of CQLs.

To illustrate our goals for integration of application, language paradigm, and implementation, let us revisit the canonical successful case of integration in the relational data model. In this data model, an important application area (data processing) is described in a declarative style (relational calculus) so that it can be automatically and efficiently translated into procedural style (relational algebra). Program evaluation is bottom-up and set-at-a-time as opposed to top-down and tuple-at-a-time, because the applications involve massive amounts of structured data. This evaluation may be optimized, e.g., via algebraic transformations, selection propagation etc. It may be performed in-core in polynomial time, because of the low complexity of the calculations expressed. Most importantly, it may be implemented efficiently with large amounts of data in secondary storage via indexing and hashing.

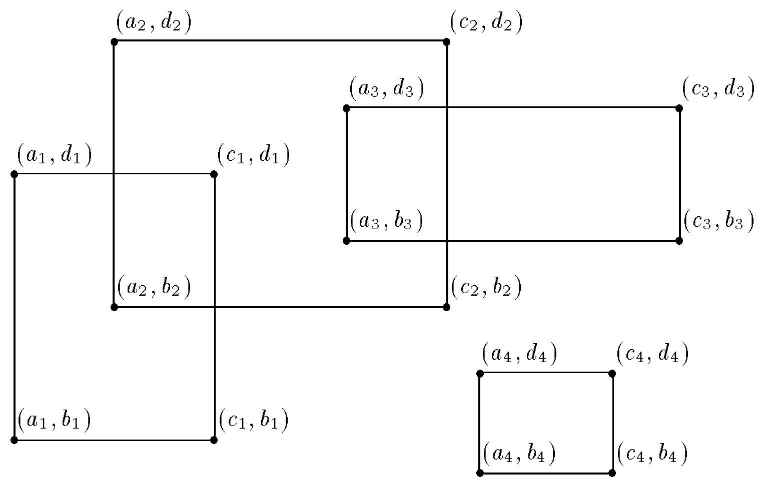

Fig. 1. Rectangle intersection

We believe that by generalizing relational formalisms to constraint formalisms it is, in principle, possible to generalize all the key features of the relational data model to spatial data models. **(1)** The language framework preserves the declarative style and the efficiency of relational database languages. **(2)** The possible applications of constraint databases include both data processing and numerical processing of spatial data. **(3)** The implementation technology of spatial access methods (see [62, 63]) naturally matches the new formalism.

Example 2. Let us try to illustrate these points **(1-3)** with an example. A very common task from computational geometry and spatial databases is the problem of computing all rectangle intersections [57, 63].

The database consists of a set of rectangles in the plane (the four rectangles of Figure 1) and we want to compute all pairs of distinct intersecting rectangles. Note that the theory of constraints used in this simple, but very common, example is the theory of dense order with constants.

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

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

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

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

Let $RC(z, x, y)$ be a ternary relation. Consider a CQL (to be defined more formally in the next subsection) where we interpret $RC(z, x, y)$ to mean that (x, y) is a point in the rectangle with name z. The rectangle that was stored above by (n, a, b, c, d), would now be stored as the generalized tuple $(z = n) \wedge (a \leq x \leq c) \wedge (b \leq y \leq d)$. The meaning of this generalized tuple is an infinite set of points. The set of all intersecting rectangles can now be expressed as:

$$Intersects(n_1, n_2) :\!-\!\!-\; n_1 \neq n_2, RC(n_1, x, y), RC(n_2, x, y)$$

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

Of course the ease of expression in the above example is there because the (implicit) existential quantification of x, y in the above intersection rule is over an infinite domain. But, despite this, there are efficient evaluation techniques.

Example 3. In the above example we used relational calculus notation and non-recursive Datalog notation. Let us continue with a calculation of the pairs of rectangles that are reachable from each other in recursive Datalog notation. Note that again the x, y are quantified over an infinite domain.

$$Reachable(n_1, n_2) :\!-\!\!-\; Reachable(n_1, n_3), Reachable(n_3, n_2)$$

$$Reachable(n_1, n_2) :\!-\!\!-\; RC(n_1, x, y), RC(n_2, x, y)$$

As shown in [41] recursive Datalog with dense order is evaluable in polynomial time. The algorithm used there was a bottom-up version of CLP($\Re$).

In general, more complex constraints and recursion lead to languages that can compute all partial recursive functions. However, one could limit recursion to finite domains as in this example, where recursion happens to be over the finite set of rectangle names. □

The above examples may be complemented with many other computational geometry tasks, such as computing convex hulls and Voronoi diagrams [57]. In these examples the typical computational geometry input of "N arbitrary points" becomes an input database of size N and the task specification becomes a fixed size CQL program.

Most natural computational geometry tasks are expressible declaratively using CQLs that combine first-order logic with simple arithmetic constraints. They can then be evaluated in a procedural fashion using the general-purpose evaluation method that is a CQL *intrepreter*. Special computational geometry algorithms would be used by a CQL *compiler* as optimizations additional to this general-purpose evaluation method.

A very important optimization involves indexing particular variables in the generalized tuples. In the above examples such indexing would correspond to I/O efficient processing of the projections of the rectangles on the x and y dimensions. More specifically, as shown in (the journal version of) [41], if in a CQL the projection of any generalized tuple on x is an interval, then the problem of 1-dimensional searching on x in the constraint data model is a special case of 2-dimensional searching in the relational data model.

We will treat this optimization in detail in Section 4 of this paper because we believe that it is the most practical one for spatial databases. Fortunately, current spatial database access methods are applicable to indexing in the CQL framework. Let us now describe this framework as presented in [41].

1.3 The CQL Framework

(1) *A generalized k-tuple is a quantifier-free conjunction of constraints on k variables, which range over a set D.* There are many kinds of generalized tuples depending on the kind of constraints used. *In all cases equality constraints among individual variables and constants are allowed.* In the relational database model $R(3,4)$ is a tuple of arity 2 or ground fact. It is a single point in 2-dimensional space representable also as $R(x,y)$ with $x = 3$ and $y = 4$, where x, y range over some finite set. $R(x,y)$ with $(x = y \wedge x < 2)$ is a generalized tuple of arity 2 and so is $R(x,y)$ with $x + y = 2.5$, where x, y range over the rational or the real numbers. Hence, a generalized tuple of arity k is a finite representation of a possibly infinite set of tuples of arity k (or points in k-dimensional space D^k).

(2) *A generalized relation of arity k is a finite set of generalized k-tuples, with each k-tuple over the same variables.* It is a first-order formula in disjunctive normal form (DNF) of constraints, which uses at most k variables ranging over set D. Each generalized relation describes a possibly infinite set of arity k tuples (or points in k-dimensional space D^k). *A generalized database is a finite set of generalized relations.*

(3) *The syntax of a CQL is the union of an existing database query language and a decidable logical theory.* For example: Relational calculus [20] + the theory of real closed fields [68, 23, 8, 48, 60]; Relational calculus [20] + the theory of real (or Presburger) addition [29, 30, 9, 15]; Relational calculus [20] + the theory of discrete order with constants. Inflationary Datalog$^\neg$ [3, 32, 47] + the theory of dense order with constants[28]; Inflationary Datalog$^\neg$ + the theory of equality on an infinite domain with constants.

(4) *The semantics of a CQL is based on that of the decidable logical theory, by interpreting database atoms as shorthands for formulas of the theory.* Let $\phi = \phi(x_1, \ldots, x_m)$ be a CQL query program using free variables $x_1, \ldots, x_m$. Let predicate symbols $R_1, \ldots, R_n$ in ϕ name the input generalized relations and let $r_1, \ldots, r_n$ be corresponding input generalized relations. We interpret the program in the context of such an input. Let $\phi[r_1/R_1, \ldots, r_n/R_n]$ be the formula of the theory that is obtained by replacing in ϕ each database atom $R_i(z_1, \ldots, z_k)$ by the DNF formula for input generalized relation r_i, with its variables appropriately renamed to $z_1, \ldots, z_k$. The output is the possibly infinite set of points in m-dimensional space D^m, such that instantiating the free variables $x_1, \ldots, x_m$ of formula $\phi[r_1/R_1, \ldots, r_n/R_n]$ to any one of these points makes the formula true. (Wlog an occurrence of a database atom in ϕ is of the form $R_i(z_1, \ldots, z_k)$ $1 \leq i \leq n$, where R_i is of arity k and $z_1, \ldots, z_k$ are distinct variables; this is because in our framework we can always use equality constraints).

(5) *For each input, the queries must be evaluable in closed form and bottom-up.* By closed form we mean that the output of any query program applied to any input generalized relations must be a generalized relation. The analogue for the relational model is that relations are finite structures, and queries are supposed to preserve this finiteness. This is a requirement that creates various "safety" problems in relational databases [20, 69]. The precise analogue in relational databases is the notion of weak safety of [4]. Evaluation of a query corresponds to an instance of a decision problem. Quantifier elimination procedures realize the goal of closed form and use induction on the structure of formulas, which leads to bottom-up evaluation.

(6) *For each input, the queries must be evaluable efficiently in the input size, i.e., with at least PTIME data complexity.* Database atomic formulas indicate, in the declarative query language itself, the parts that can grow asymptotically versus the parts that are constant-size. By fixing the program size and letting the database grow, one can prove that the evaluation can be performed in PTIME or in NC or in LOGSPACE, depending on the constraints considered (for the various complexity classes see [38]).

1.4 Languages, efficiency, efficiency, ...

In the remainder of this paper we explore CQLs with an emphasis on quantifying language efficiency.

In Section 2, we present an algebra for dense order constraints (due to Goldin) which is simpler to evaluate than the calculus described in [41]. In Section 3, we sharpen some of the related data complexity bounds and comment on the current state-of-the-art of data complexity analysis. We also observe that complex object data types can easily be added to the CQL framework. In Section 4, we describe how the CQL framework can be supported by existing multi-dimensional searching data structures.

We close, in Section 5, with some open questions and some recent developments.

2 An Algebra for Dense Order Constraints

Dense order inequality constraints are all formulas of the form $x\theta y$ and $x\theta c$, where x, y are variables, c is a constant, and θ is one of $=$, $\leq$, $<$ (or its negation $\neq$, $>$, $\geq$). We assume that these constants are interpreted over a countably infinite set D with a binary relation which is a dense order (e.g., we can take D to be the rationals). Constants, $=$, $\leq$, and $<$ are interpreted respectively as elements, equality, the dense order, and the irreflexive dense order of D. For the first-order theory of dense order see [28] and for its data complexity (with and without recursion) see [41].

In this section we present a first-order algebra for dense order constraints, which is equivalent to the calculus of [41] and is simpler. We first assume that the generalized tuples have a specific form, beyond being conjunctions of constraints, this is Definition 1. It is easy to see that any conjuction of constraints can be transformed into a union of such generalized tuples. In our algebra, we use generalized tuples that are "tighest" or *canonical* (Subsection 2.1). We show that this algebra has the desirable *closure* property (Subsection 2.2).

2.1 Generalized Tuples and Relations in Canonical Form

Definition 1. *A generalized n-tuple* $\overline{t} = (\overline{l}, \overline{u}, \overline{\mu})$ over variables $x_1, \ldots, x_n$ consists of two sequences $\overline{l} = (l_1, \ldots l_n)$ and $\overline{u} = (u_1, \ldots, u_n)$, and an n by n matrix $\overline{\mu} = (\mu_{1,1}, \mu_{1,2}, \ldots, \mu_{n,n})$, where:

(a) the l_i's are constants in $D \cup \{-\infty\}$;
(b) the u_i's are constants in $D \cup \{+\infty\}$;
(c) for all i, $l_i \leq u_i$;
(d) for all i, j, $\mu_{i,j}$ is one of $=$, $<$, $>$, ?.

W.l.o.g., we can assume that the entries of $\overline{\mu}$ along the diagonal are $=$, and $\overline{\mu}$ is symmetric (e.g., if $x < y$ then $y > x$). □

Definition 2. Each generalized n-tuple over variables $X = \{x_1, \ldots, x_n\}$ defines a formula with free variables X, and a set of points in D^n. This point-set is the *semantics* of the generalized tuple.

1. For a generalized n-tuple $\overline{t} = (\overline{l}, \overline{u}, \overline{\mu})$, the *formula* $F(\overline{t})$, with n free variables $\{x_1, \ldots, x_n\}$, is the conjunction of $n^2 + n$ constraints ψ_i and $\xi_{i,j}$:
   (a) for all tuples (l_i, u_i), ψ_i is $(x_i = l_i)$ if $l_i = u_i$, and $(l_i < x_i < u_i)$ otherwise;
   (b) for all entries $\mu_{i,j}$, $\xi_{i,j}$ is $(x_i\ \mu_{i,j}\ x_j)$ if $\mu_{i,j}$ is not ?, and *TRUE* otherwise.
2. For a generalized n-tuple $\overline{t} = (\overline{l}, \overline{u}, \overline{\mu})$, the corresponding *set of points* $P(\overline{t})$ is the set of all points $\overline{x} = (x_1, \ldots, x_n)$ satisfying $F(\overline{t})$. □

Example 4. The following is a tuple $\overline{t}$ on variables (A, B, C) whose formula is

$$(-\infty < A < 5\ \wedge\ 1 < B < 7\ \wedge\ C = 6\ \wedge\ A > B). \ \square$$

In tabular form we represent generalized tuples as follows:

$\overline{t}$	A	B	C
$\overline{l}$	$-\infty$	1	6
$\overline{u}$	5	7	6
$\overline{\mu}$	$=$	$>$	?
	$<$	$=$	?
	?	?	$=$

We can define a partial order on all tuples that share the same set of variables:

Definition 3. Let $\overline{t}$ and $\overline{t}'$ be generalized tuples over variables X, with the ψ's and ξ's as in Definition 2. $\overline{t}'$ is *tighter* than $\overline{t}$ if:

$$\forall i,\ \psi'_i \text{ is } \textit{tighter} \text{ than } \psi_i \text{ and } \forall i, j,\ \xi'_{i,j} \text{ is } \textit{tighter} \text{ than } \xi_{i,j},$$

where constraint θ' is *tighter* than constraint θ iff the universal closure of $(\theta' \Rightarrow \theta)$ is true in D. □

If $\overline{t}'$ is tighter than $\overline{t}$, then the universal closure of $(F(\overline{t}') \Rightarrow F(\overline{t}))$ will also be true, so $P(\overline{t}') \subseteq P(\overline{t})$. We can also define equivalence classes over generalized tuples w.r.t. their sets of points:

Definition 4. Let $\overline{t}_1, \overline{t}_2$ be generalized tuples over variables X. Then, $\overline{t}_1$ and $\overline{t}_2$ belong to the same *equivalence class* E iff $P(\overline{t}_1) = P(\overline{t}_2)$.

Proposition 5. *For each class E (*except *for the class of tuples whose point set is empty) there is a unique member $\overline{t}_c$ such that $\forall \overline{t} \in E$, $\overline{t}_c$ is tighter than $\overline{t}$.*

We will refer to this tuple $\overline{t}_c$ as the *canonical representation* of all members of E. It can be constructed efficiently as follows:

Proposition 6. *Let $\overline{t}$ be a generalized tuple such that $F(\overline{t})$ is satisfiable. Then, $\overline{t}'$ is a canonical representation of $\overline{t}$ if:*

(a) $\forall i,\ \psi'_i$ is the tightest constraint on constants in $\overline{t}$ such that $F(\overline{t}) \Rightarrow \psi'_i$;
(b) $\forall i, j,\ \xi'_{i,j}$ is the tightest constraint such that $F(\overline{t}) \Rightarrow \xi'_{i,j}$. □

It is easy to show that the canonical representation for any generalized n-tuple can be found efficiently:

Proposition 7. *Given a generalized n-tuple $\overline{t}$, it is possible in time $T = O(n^c)$ for some constant c to determine whether $\overline{t}$ is satisfiable, and if so, to find the canonical representation of $\overline{t}$.*

We will call a generalized tuple $\overline{t}$ *canonical* if $P(\overline{t})$ is not empty and $\overline{t}$ is its own canonical representation.

Example 5. The tuple $\overline{t}$ in the previous example is not canonical because:

$(A < 5 \ \wedge \ A > B) \Rightarrow B < 5$, but $l_2 = 7$.

The following tuple $\overline{t}'$ is a canonical representation of $\overline{t}$:

$\overline{t}'$	A B C
$\overline{l}'$	1 1 6
$\overline{u}'$	5 5 6
$\overline{\mu}'$	$=$ $>$ $<$ $<$ $=$ $<$ $>$ $>$ $=$

$F(\overline{t}') = (1 < A < 5 \ \wedge \ 1 < B < 5 \ \wedge \ C = 6 \ \wedge \ A > B \ \wedge \ A < C \ \wedge \ B < C)$. □

Having defined canonical tuples, we are ready to define generalized relations and generalized databases consisting of such tuples.

Definition 8. 1. A *generalized relation* over variables $X = (x_1, \ldots, x_n)$ is a finite set of canonical generalized tuples over X. There exists a natural mapping ϕ from generalized relations over X to DNF formulas over X, and a corresponding mapping σ from generalized relations to (finite or infinite) relations over D^n:

$$\phi(\overline{r}) = \bigvee_{\overline{t} \in \overline{r}} F(\overline{t}), \quad \sigma(\overline{r}) = \bigcup_{\overline{t} \in \overline{r}} P(\overline{t})$$

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

To conclude this subsection, we would like to compare canonical tuples with the different tuple construct used in [41], called r-configurations. There is a definite similarity in the syntax of canonical tuples here and r-configurations, and in fact:

> For a given finite subset D_ϕ of D, an r-configuration is any canonical generalized tuple $\overline{t} = (\overline{l}, \overline{u}, \overline{\mu})$ where (a) $\forall i$, there does not exist $c \in D_\phi$ such that $l_i < c < u_i$, and (b) $\forall i, j, \ \mu_{i,j}$ is one of $\{<, >, =\}$.

The time of checking whether a generalized tuple is an r-configuration is *not constant* in the size of D_ϕ, whereas it is *constant* for checking whether a tuple is canonical. Therefore an advantage of the canonical tuples here over the r-configurations of [41] is that inserts are more efficient.

Furthermore, for an arbitrary conjunction θ of constraints over free variables X, the size of the smallest set of canonical tuples $\overline{r}$ over X such that $\theta \equiv \bigvee_{\overline{t} \in \overline{r}} F(\overline{t})$ is only dependent on the length of θ, whereas it would grow to be *at least linear* in the size of D_ϕ if $\overline{r}$ was restricted to r-configurations. Nevertheless, if we choose to restrict generalized relations to sets of r-configurations, it is important to note that all of the definitions and lemmas in the next subsection remain valid.

2.2 The Algebraic Operations

We can now introduce the syntax and the semantics of an algebra over generalized relations. As in Codd's relational algebra [20, 40], we will define basic algebraic operations *projection* (π), *selection* (ς), *natural join* ($\bowtie$), *union* ($\cup$), *difference* ($-$), and *renaming* (ϱ), which map one or more generalized relations to a new generalized relation. *Algebraic queries* over generalized databases are composed of these basic operations.

For each operation **OP** on generalized relations, we will claim that the resulting generalized relation has the following closure property:

$$\text{if } \overline{r}' = \mathbf{OP}(\overline{r}_1, \ldots, \overline{r}_n), \text{ then } \sigma(\overline{r}') = \mathrm{OP}(\sigma(\overline{r}_1), \ldots, \sigma(\overline{r}_n)),$$

where OP is the operation of the same name in the relational algebra in [40], and σ is defined in Definition 8.

Remark: In our definitions of operators, we use the notation $\alpha(\overline{r})$ for the variables of a generalized relation $\overline{r}$. To prove the closure theorem of this section (see Figure below) we compose the closure properties of operations.

$$\begin{tikzcd}
\boxed{\overline{r}_1\ \overline{r}_2\ \ldots\ \overline{r}_n} \arrow[r, "\mathbf{QUERY}(\overline{r}_1, \overline{r}_2, \ldots, \overline{r}_n)"] \arrow[d, "\sigma"] & \boxed{\overline{r}_{n+1}} \arrow[d, "\sigma"] \\
\boxed{r_1\ r_2\ \ldots\ r_n} \arrow[r, "\mathrm{QUERY}(r_1, r_2, \ldots, r_n)"] & \boxed{r_{n+1}}
\end{tikzcd}$$

Fig. 2. The Closure Condition

PROJECTION. If $\overline{r}$ is a generalized relation over variables X, and Z is a subset of X, then $\pi_Z(\overline{r})$ is the projection of $\overline{r}$ on Z:

(1) $\alpha(\pi_Z(\overline{r})) = Z$,
(2) $\pi_Z(\overline{r}) = \{(\overline{t} \downarrow Z) : \overline{t} \in \overline{r}\}$,

where $\overline{t}' = (\overline{t} \downarrow Z)$ is the *restriction* of $\overline{t}$ to variables Z, i.e., the variables in $\overline{t}'$ have the same bounds and the same constraints as the corresponding variables in $\overline{t}$, and all other bounds and constraints are dropped.

Lemma 9. *Let* $\overline{r}' = \pi_Z(\overline{r})$*. Then,* $\sigma(\overline{r}') = \pi_Z(\sigma(\overline{r}))$*.*

Example 6. Let $\overline{r} = \{\overline{t}_1, \overline{t}_2\}$ be a generalized relation over variables (A, B, C). We compute $\pi_{(A,C)}(\overline{r}) = \{\overline{t}'_1, \overline{t}'_2\}$ (see figure on Projection). □

$\overline{t}_1$	A	B	C
$\overline{l}_1$	1	1	6
$\overline{u}_1$	5	5	6
$\overline{\mu}_1$	$=$	$>$	$<$
	$<$	$=$	$<$
	$>$	$>$	$=$

$\overline{t}_2$	A	B	C
$\overline{l}_2$	4	1	3
$\overline{u}_2$	7	1	∞
$\overline{\mu}_2$	$=$	$>$	?
	$<$	$=$	$<$
	?	$>$	$=$

$\xrightarrow{\pi_{(A,C)}}$

$\overline{t}'_1$	A	C
$\overline{l}'_1$	1	6
$\overline{u}'_1$	5	6
$\overline{\mu}'_1$	$=$	$<$
	$>$	$=$

$\overline{t}'_2$	A	C
$\overline{l}'_2$	4	3
$\overline{u}'_2$	7	∞
$\overline{\mu}'_2$	$=$	?
	?	$=$

Fig. 3. Projection

SELECTION. If $\overline{r}$ is a generalized relation over variables X, Z is a subset of X, and $\overline{t}_0$ is a generalized tuple over variables Z, then $\varsigma_{F(\overline{t}_0)}(\overline{r})$ is the selection on $\overline{r}$ by $F(\overline{t}_0)$:

(1) $\alpha(\varsigma_{F(\overline{t}_0)}(\overline{r})) = X$,
(2) $\varsigma_{F(\overline{t}_0)}(\overline{r}) =$
$\{\overline{t}$: for some $\overline{t}' \in \overline{r}$, $\overline{t}$ is the common tuple of $(\overline{t}_0 \uparrow X)$ and $\overline{t}'\}$,

where $(\overline{t}_0 \uparrow X)$ and the notion of common tuple are defined below.

Definition 10. Let $\overline{t}$ be a canonical generalized tuple over variables X, and Z be a superset of X. Then, $\overline{t}'$ is the *extension* of $\overline{t}$ to variables Z, written $(\overline{t} \uparrow Z)$ if $\overline{t}'$ is the most general tuple over variables Z such that $\overline{t} = (\overline{t}' \downarrow X)$. In particular,

(a) $\forall i$ such that $z_i \notin X$, $\psi'_i = (-\infty < x_i < +\infty)$;
(b) $\forall i, j$ such that either $z_i \notin X$ or $z_j \notin X$, $\xi'_{i,j} = TRUE$. □

Definition 11. Let $\overline{t}^1$ and $\overline{t}^2$ be canonical generalized tuples over variables X. Then, $\overline{t}$ is the *common tuple* of $\overline{t}^1$ and $\overline{t}^2$ if $\overline{t}$ is a canonical tuple over variables X such that $F(\overline{t}) \equiv (F(\overline{t}^1) \wedge F(\overline{t}^2))$, or alternately, $P(\overline{t}) = P(\overline{t}^1) \cap P(\overline{t}^2)$. In particular, $\overline{t}$ is the canonical representation of $\overline{t}'$, where:

$\forall i,\ \psi'_i \equiv (\psi^1_i \wedge \psi^2_i)$, and $\forall i, j,\ \xi'_{i,j} \equiv (\xi^1_{i,j} \wedge \xi^2_{i,j})$.

If some ψ'_i or $\xi'_{i,j}$ does not exist, then $P(\overline{t}^1) \cap P(\overline{t}^2)$ is empty and there is no common tuple. □

Lemma 12. *Let* $\overline{r}' = \varsigma_{F(\overline{t}_0)}(\overline{r})$*. Then,* $\sigma(\overline{r}') = \varsigma_{F(\overline{t}_0)}(\sigma(\overline{r}))$*.*

Example 7. Let $\overline{r} = \{\overline{t}_1, \overline{t}_2\}$ be a generalized relation over variables (A, B, C), and let $\overline{t}_0$ be a canonical tuple over variables (B, C), where $F(\overline{t}_0) = (2 < B < 4 \ \wedge\ 3 < C < 7)$. We compute $\varsigma_{F(\overline{t}_0)}(\overline{r}) = \{\overline{t}'_1\}$ (see figure on Selection). □

$\overline{t}_1$	A	B	C
$\overline{l}_1$	1	1	6
$\overline{u}_1$	5	5	6
$\overline{\mu}_1$	$=$	$>$	$<$
	$<$	$=$	$<$
	$>$	$>$	$=$

$\overline{t}_2$	A	B	C
$\overline{l}_2$	4	1	3
$\overline{u}_2$	7	1	∞
$\overline{\mu}_2$	$=$	$>$	?
	$<$	$=$	$<$
	?	$>$	$=$

$$\xrightarrow{\varsigma_{F(\overline{t}_0)}}$$

$\overline{t}'_1$	A	B	C
$\overline{l}'_1$	1	2	6
$\overline{u}'_1$	5	4	6
$\overline{\mu}'_1$	$=$	$>$	$<$
	$<$	$=$	$<$
	$>$	$>$	$=$

Fig. 4. Selection

Natural Join. If $\overline{r}_1$ is a generalized relation over variables X, $\overline{r}_2$ is a generalized relation over variables Y, and $Z = X \cup Y$, then $\overline{r}_1 \bowtie \overline{r}_2$ is the natural join of $\overline{r}_1$ and $\overline{r}_2$:

(1) $\alpha(\overline{r}_1 \bowtie \overline{r}_2) = Z$,
(2) $\overline{r}_1 \bowtie \overline{r}_2 = \{\overline{t} :$ for some $\overline{t}_1 \in \overline{r}_1, \overline{t}_2 \in \overline{r}_2$, $\overline{t}$ is the common tuple of $(\overline{t}_1 \uparrow Z)$ and $(\overline{t}_2 \uparrow Z)\}$.

Lemma 13. *Let $\overline{r}' = \overline{r}_1 \bowtie \overline{r}_2$. Then, $\sigma(\overline{r}') = \sigma(\overline{r}_1) \bowtie \sigma(\overline{r}_2)$.*

Example 8. Let $\overline{r}_1 = \{\overline{t}_1, \overline{t}_2\}$ be a generalized relation over variables (A, B, C), and $\overline{r}_2 = \{\overline{t}_3, \overline{t}_4\}$ be a generalized relation over variables (B, D). We compute $\overline{r}_1 \bowtie \overline{r}_2$ (see figure on Join). □

$\overline{t}_1$	A	B	C
$\overline{l}_1$	1	1	6
$\overline{u}_1$	5	5	6
$\overline{\mu}_1$	$=$	$>$	$<$
	$<$	$=$	$<$
	$>$	$>$	$=$

$\overline{t}_2$	A	B	C
$\overline{l}_2$	4	1	3
$\overline{u}_2$	7	1	∞
$\overline{\mu}_2$	$=$	$>$	?
	$<$	$=$	$<$
	?	$>$	$=$

$\overline{t}_3$	B	D
$\overline{l}_3$	∞	2
$\overline{u}_3$	4	6
$\overline{\mu}_3$	$=$	?
	?	$=$

$\overline{t}_4$	B	D
$\overline{l}_4$	0	7
$\overline{u}_4$	1	7
$\overline{\mu}_4$	$=$	$>$
	$<$	$=$

$$\downarrow \bowtie$$

$\overline{t}'_1$	A	B	C	D
$\overline{l}'_1$	1	1	6	2
$\overline{u}'_1$	5	4	6	6
$\overline{\mu}'_1$	$=$	$>$	$<$	?
	$<$	$=$	$<$	?
	$>$	$>$	$=$	$>$
	?	?	$<$	$=$

$\overline{t}'_2$	A	B	C	D
$\overline{l}'_2$	4	1	3	2
$\overline{u}'_2$	7	1	∞	6
$\overline{\mu}'_2$	$=$	$>$	?	$>$
	$<$	$=$	$<$	$<$
	?	$>$	$=$	?
	$<$	$>$	?	$=$

Fig. 5. Join

Union. If $\overline{r}_1$ and $\overline{r}_2$ are generalized relation over variables X, then $\overline{r}_1 \overline{\cup} \overline{r}_2$ is the union of $\overline{r}_1$ and $\overline{r}_2$:

(1) $\alpha(\overline{r}_1 \overline{\cup} \overline{r}_2) = X$,
(2) $\overline{r}_1 \overline{\cup} \overline{r}_2 = \{\overline{t} : \overline{t} \in \overline{r}_1 \text{ or } \overline{t} \in \overline{r}_2\}$.

Lemma 14. *Let $\overline{r}' = \overline{r}_1 \overline{\cup} \overline{r}_2$. Then, $\sigma(\overline{r}') = \sigma(\overline{r}_1) \cup \sigma(\overline{r}_2)$.*

Difference. If $\overline{r}_1$ and $\overline{r}_2$ are generalized relations over variables X, then $\overline{r}_1 \overline{-} \overline{r}_2$ is the difference of $\overline{r}_1$ and $\overline{r}_2$. The definition of $\overline{r}_1 \overline{-} \overline{r}_2$ is recursive, as follows:

(1) $\alpha(\overline{r}_1 \overline{-} \overline{r}_2) = X$,
(2) If size of $\overline{r}_1 > 1$, $\overline{r}_1 \overline{-} \overline{r}_2 = \overline{\cup}_{\overline{t} \in \overline{r}_1} (\{\overline{t}\} \overline{-} \overline{r}_2)$,
(3) If size of $\overline{r}_1 = 0$, $\overline{r}_1 \overline{-} \overline{r}_2 = \emptyset$,
(4) If size of $\overline{r}_1 = 1$ (i.e., $\overline{r}_1 = \{\overline{t}\}$), $\overline{r}_1 \overline{-} \overline{r}_2 =$
(for some $\overline{t}' \in \overline{r}_2$, (tuple difference of $\overline{t}$ and $\overline{t}'$) $-$ $(\overline{r}_2 - \{\overline{t}'\})$),

where tuple difference is a set of tuples defined below.

Definition 15. Let $\overline{t}$ and $\overline{t}'$ be canonical generalized tuples over variables X.

1. If $\overline{t} = \overline{t}'$, *tuple difference* of $\overline{t}$ and $\overline{t}'$ is $\emptyset$.
2. If $\exists i$ such that $\psi_i \wedge \psi_i'$ is not satisfiable, or $\exists i, j$ such that $\xi_{i,j} \wedge \xi_{i,j}'$ is not satisfiable, *tuple difference* of $\overline{t}$ and $\overline{t}'$ is $\{\overline{t}\}$.
3. Let θ be a constraint in $F(\overline{t})$, either ψ or ξ, such that $\theta \neq \theta'$, where θ' is the corresponding constraint in $F(\overline{t}')$.
4. Let $\theta \equiv \vee(\theta_1, \ldots, \theta_k)$, where $\theta_1 \equiv (\theta \wedge \theta')$, all θ_i's are disjoint, and $k \geq 1$ is the smallest possible.
5. Let $(\overline{t}_1, \ldots, \overline{t}_k)$ be obtained by replacing θ in $\overline{t}$ by each θ_i and putting the result into canonical form. Let $\overline{t}_1'$ be obtained by replacing θ' in $\overline{t}'$ by θ_1, and putting the result into canonical form.
6. Then, *tuple difference* of $\overline{t}$ and $\overline{t}'$ is defined recursively as $\{\overline{t}_2, \ldots, \overline{t}_k\} \overline{\cup}$ (tuple difference of $\overline{t}_1$ and $\overline{t}_1'$).

Lemma 16. *Let $\overline{r}' = \overline{r}_1 \overline{-} \overline{r}_2$. Then, $\sigma(\overline{r}') = \sigma(\overline{r}_1) - \sigma(\overline{r}_2)$.*

Renaming. If $\overline{r}$ is a generalized relation over variables $X = (x_1, \ldots, x_n)$, and y is a variable not in X, then $\varrho_{x_i|y}(\overline{r})$ is the renaming in $\overline{r}$ of x_i to y:

(1) $\alpha(\varrho_{x_i|y}(\overline{r})) = (X - \{x_i\}) \cup \{y\}$,
(2) $\varrho_{x_i|y}(\overline{r}) = \overline{r}$.

Lemma 17. *Let $\overline{r}' = \varrho_{x_i|y}(\overline{r})$. Then, $\sigma(\overline{r}') = \varrho_{x_i|y}(\sigma(\overline{r}))$.*

Let us now consider the first-order calculus CQL of [41] for dense order. This is relational calculus combined with the decidable theory of dense order with constants. Every query can be expressed semantically as a relational calculus or algebra query on *unrestricted* (finite or infinite) relations over D^n. The above definitions and lemmas compose to prove the following closure property:

Theorem 18. *For every relational algebra QUERY on unrestricted relations over D^n, that are finitely representable, the constraint algebra* **QUERY**, *that uses* **OP***s instead of OPs, has the property that:*

$$\sigma(\mathbf{QUERY}(\overline{r}_1, \ldots, \overline{r}_n)) = QUERY(\sigma(\overline{r}_1), \ldots, \sigma(\overline{r}_n))$$

.

3 PTIME Expressibility

The notion of data complexity arose from a study of the expressibility of computations over finite structures. It corresponds nicely to the intuition that the database size is much larger than the query program size. It seems reasonable to limit computations to efficient ones, i.e., PTIME manipulations of the data. There are characterizations of PTIME [36, 73] using Datalog$^\neg$ on finite, ordered structures. So in terms of expressibility , Datalog$^\neg$ would suffice. However, various other CQLs allow for more "natural" and "flexible" expression of queries.

3.1 Data Complexity

A query Q has *data complexity* in PTIME (resp. LOGSPACE, NC) if there is a TM (resp. TM, PRAM) which given input generalized relations d produces some generalized relation representing the output of $Q(d)$ and uses polynomial time (resp. logarithmic space on the work tape, polynomial number of processors running in polylogarithmic parallel time). See [38] for definitions of TMs and PRAMs. We assume a standard binary encoding of generalized relations.

The CQL framework is interesting because many combinations of database query languages and decidable theories have PTIME data complexity.

¿From Codd's original work [20] it follows that: *relational calculus on finite sets can be evaluated bottom-up in closed form and LOGSPACE data complexity.* The LOGSPACE data complexity analysis is from [17]. In the following table we list some theories with upper bounds on their data complexities.

	Polynomial	**Dense Order**	**Equality**
Relational Calculus	NC	AC^0	AC^0
Datalog$^\neg$	Not closed	=PTIME	PTIME

In [41] it was shown that relational calculus (Inflationary Datalog$^\neg$) with dense order constraints has LOGSPACE (PTIME) data complexity. Also, by a slight modification of [36, 73] Inflationary Datalog$^\neg$ with dense linear order expresses *exactly* PTIME (this is the = in the table above).

It is possible to strengthen the LOGSPACE bounds to uniform AC^0, using random-access Alternating TMs [5, 16] with a fixed number of alternations and a logarithmic time bound. The same applies to relational calculus (Inflationary Datalog$^\neg$) with equality constraints over an infinite domain.

Let us now look at a fairly rich constraint theory. Relational calculus with real polynomial inequality constraints can be evaluated bottom-up in closed form and PTIME data complexity. This is a direct consequence of [23]. More recent results [8, 48, 60] place the data complexity in NC. The precise expressibility (in terms of data complexity within NC) for real polynomial constraints is still open. Clarifying the relationship with the data complexity of weaker theories involving addition and multiplication by constants is a topic of interest [25].

Unfortunately, arithmetic (even just integer order) with recursion easily leads to computability of all partial recursive functions. One implication is that there are qualitatively multiple ways of encoding computations in CLP (e.g., [26].

Sufficient conditions that allow closure under recursion are an interesting topic [61]. Note that, non-closure is very easy to show: Let π be the query program that consists of the rules $S(x, y)$:— $R(x, y)$ and $S(x, y)$:— $R(x, z), S(z, y)$ (i.e., S is the transitive closure of R). If the input r for R consists of the generalized relation $y = 2 * x$, then the result of the query is the set of all points (x, y) that satisfy $y = 2^i * x$ for some $i > 0$. This set is not finitely representable in the language of polynomial inequality constraints.

3.2 Language Extensions

The relational data model has been extended with a number of useful features, e.g., aggregates and complex objects. Are these extensions possible for CQLs and how complicated are they?

First, we would like to observe that the CQL framework allows us (via equality constraints) to embed relational and complex object databases directly in constraint databases. Consider the rectangle intersection example from Section 1. In a CQL we can have both a relation of 5-tuples ranging over a finite set and denoting endpoints of rectangles and a genaralized relation of 3-tuples ranging over an infinite set of named rectangle points. We can even link the two relations together with an "integrity constraint" that relates every 5-tuple with a generalized 3-tuple and vice-versa.

This "naive" solution can be used to add complex objects or aggregates to a CQL by just adding them to the relational data model embedded in the CQL. It is useful because it allows posing queries about the representation itself and not only about the infinite relations represented. There are however some subtle issues that should be researched further.

Aggregates: The first important extension involves aggregate operations (e.g., maximum, average etc), see [45]. Introducing aggregates in a CQL is identical with the relational data model if an attribute ranges over a finite set. Many aggregate primitives have low data complexity [5]. Introducing aggregates in a CQL for attributes ranging over an infinite set is harder, since it typically involves integration, and is a topic of current research [50].

Complex Objects: The second extension involves complex objects. As described in [1, 2] complex objects are built using **tuple** and **finite-set** data type constructors. Generalized tuples are **tuple** constructors, but they also have some of the properties of **finite-set** constructors. This subtelty is best illustrated by an example.

Example 9. Consider a convex polygon in 2-dimensional space. It can be characterized by its vertices or by its facets, which are linear inequalities of the form $a * x + b * y \leq c$.

In the relational data model extended with complex objects one named convex polygon is a object of type:
$\mathbf{tuple}(D,\mathbf{finite\text{-}set}(\mathbf{tuple}(D,D,D)))$
In a CQL the same named convex polygon is a single generalized tuple, with one equality constraint for the name and linear inequality constraints for the facets.

4 I/O Efficiency in Constraint Databases

4.1 Indexing Relational Databases

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

B-trees and their variants B^+-trees, [7, 24], are examples of important data structures for implementing relational databases. In particular, let each secondary memory access transmit B units of data, let r be a relation with N tuples, and let us have a B^+-tree on the attribute x of r. The space used in this case is $O(N/B)$.

The following operations define (dynamic) *1-dimensional searching on relational database attribute* x, with the corresponding performance bounds using a B^+-tree on x: (i) Find all tuples such that for their x attribute ($a_1 \leq x \leq a_2$). If the output size is K tuples, then this *range searching* is in worst-case $O(\log_B N + K/B)$ secondary memory accesses. If $a_1 = a_2$ and x is a key, then this is *key-based searching.* (ii) Insert or delete a given tuple. These are in worst-case $O(\log_B N)$ secondary memory accesses.

The problem of *k-dimensional searching on relational database attributes* $x_1, \ldots, x_k$ generalizes 1-dimensional searching to k attributes, with range searching on k-dimensional intervals. It is a central problem in spatial databases for which there are many solutions with good secondary memory access performance, e.g., grid-files, quad-trees, R-trees, hB-trees, k-d-B-trees etc (see the surveys [62, 63]).

4.2 Indexing Constraint Databases

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

If $(a_1 \leq x \leq a_2)$ is a constraint of our CQL then there is a trivial, but inefficient, solution to the problem of 1-dimensional searching on generalized database attribute x. One can add constraint $(a_1 \leq x \leq a_2)$ to every generalized tuple (i.e., conjunction of constraints) and naively insert or delete generalized tuples in a table. This would involve a linear scan of the generalized relation and introduces a lot of redundancy in the representation. In many cases, the projection of any generalized tuple on x is one interval $(a \leq x \leq a')$. This is true for Example 2, for our CQL's with dense order, for relational calculus with linear inequalities over the reals, and in general when a generalized tuple represents a convex set. Under such natural assumptions, there is a better solution for 1-dimensional searching on generalized database attribute x.

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

The use of generalized 1-dimensional indexes reduces redundancy of representation and transforms 1-dimensional searching on generalized database attribute x into the problem of *dynamic interval management*.

This is a well-known problem with many elegant solutions from computational geometry [57]. It is a special case of 2-dimensional searching in relational databases, called 1.5-dimensional searching in [53]. For example, the priority search trees of [53] are a linear space data structure with logarithmic-time update and search algorithms for in-core processing. Grid-files, R-trees, quad-trees etc, have all been used for solving this problem with good secondary memory access performance.

In summary: generalized 1-dimensional indexing is dynamic interval management on secondary storage. It can be implemented with good average I/O efficiency using existing general-purpose spatial data strutures.

Unlike B-trees that offer worst-case guarantees for their performance, many data structures (like the grid-file, R-tree, hB-tree, k-d-B-tree, quad-tree etc) that have been used to solve the problem of dynamic interval management in secondary storage don't offer optimal worst-case times. These data structures work reasonably well on the average, but with certain simple cases, their performance can deteriorate badly. We would like to offer worst-case guarantees like the B-trees do for this problem.

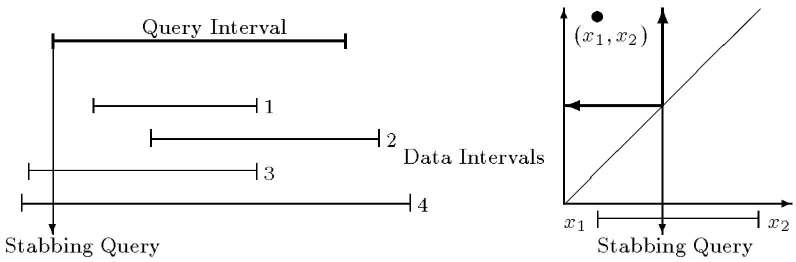

Fig. 6. Showing the relation between interval management and 2-dimensional range searching

4.3 Optimal Static Indexing for Constraints

The first I/O optimal solution for the problem of static interval management in secondary memory was published in [43]. Optimal in-core dynamic interval management is one of the basic tools of computational geometry. However, I/O optimal solutions are non-trivial, even for the static case. Let us describe the relationship of interval management on secondary storage and 2-dimensional searching.

Intervals that intersect a query interval fall into four categories as shown in Figure 6. Categories (1) and (2) can be easily located by sorting all the intervals on their first endpoint and using a B$^+$-tree to locate all intervals whose first endpoint lies in the query interval. So the main difficulty is handling categories (3) and (4). As shown in the figure, categories (3) and (4) can be located by finding all data intervals which contain the first endpoint of the query interval. (This is called a stabbing query at that point.)

Stabbing queries can be mapped to a special case of 2-dimensional range searching. This is done by mapping an interval $[x_1, x_2]$ to the point (x_1, x_2) in two dimensions X, Y. It is easy to see that all points corresponding to intervals will lie above the line $X = Y$. A stabbing query now translates into a two sided query. That is, an interval $[x_1, x_2]$ belongs to a stabbing query at x if and only if the corresponding point (x_1, x_2) is inside the box generated by the lines $X = 0$, $X = x$, $Y = x$, and $Y = \infty$. In fact, such two sided queries will always have their corner on the line $X = Y$. Such queries are called *diagonal corner queries* in [43].

An optimal solution for answering diagonal corner queries in secondary storage is presented in [43]. [43] presents a semi-dynamic data structure for this problem called the metablock tree that has optimal query I/O time $O(\log_B N + K/B)$, uses optimal storage $O(N/B)$. The metablock data structure also allows inserts in $O(\log_B N + (\log_B^2 N)/B)$ amortized I/O time; it is fairly involved and does not allow data points to be deleted.

Space-time tradeoffs for this and other related 2-dimensional range searching problems are examined in [59]. In this paper it is shown that many elegant data structures like the priority search tree, segment tree and the interval tree—all of which have been used for interval management in-core —can be implemented in secondary memory with small space overheads. In particular, two sided queries (a generalization of diagonal queries) can be answered in optimal I/O time $O(\log_B N + K/B)$, using space $O(\frac{N}{B}\log_2\log_2 B)$. This solution can bemade fully dynamic in optimal $O(\log_B N)$ amortized time.

5 Conclusions and Open Questions

The appeal of constraint programming in the 1970's and early 1980's was tempered by the cost of storing and solving constraints, and by the lack of efficient, robust, general-purpose implementations. Advances in hardware, graphical user interfaces, and programming language technology (mainly CLP in the 1980's) have brought constraint-based computing closer to the efficiency of more conventional computing formalisms. The combination of constraint programming and database querying is one area where significant progress can be made in the 1990's. Efficient declarative programming, both in terms of languages and of grahical user interface, can be achieved for an extensive range of applications.

The most challenging task is the implementation and testing of declarative and efficiently evaluable CQLs for spatial applications. The results presented here should be viewed as positive arguments for the feasibility of such an effort. In the context of such a software systems effort, many interesting theoretical questions remain:

(1) Dynamic interval management on secondary storage is the key algorithmic problem for backend support of constraint databases. Existing spatial data structures solve this problem with reasonable efficiency, but there is much room for improvement. While [43] and [59] represent advances in this context, the truly elegant and seemingly difficult problem remains open. Can dynamic interval management on secondary storage be achieved optimally in $O(N/B)$ pages, dynamic query I/O time $O(\log_B N + t/B)$ and worst-case update time $O(\log_B N)$?

(2) Indexing is only one possible optimization. How do various optimization methods (such as selection propagation, join ordering and other algebraic transformations, magic sets etc) combine with CQLs? This would involve extending [58]. For some recent research in this direction we refer to [34, 52, 56, 65, 12]. Constraint use for database logic program analysis is also relevant here, e.g., [13, 14, 70, 71].

(3) We have addressed the spatial applications of CQLs. Constraints offer useful approaches to temporal databases as well. For recent developments in temporal databases we refer to [6, 18, 61]. The subject of temporal constraints has also been addressed extensively in the artificial intelligence literature (which we unfortunately do not have the space or expertise to survey here). For some recent work that connects CQLs to artificial intelligence approaches see [49].

(4) As we argued in this paper, extensions to the relational model, involving partial information and complex object data types, can also be applied to CQLs. For example, null values can be introduced with existential quantification in constraints. Finite complex objects [1, 2] and aggregation [45] over finite sets can co-exist with infinite relations. Are there more subtle ways of adding these

extensions to CQLs? For example, adding aggregation operations over infinite sets is motivated by querying about area size and is a challenging problem [50].

(5) Linear constraint databases [11] are among the most applicable CQLs, and will certainly be among the first to be implemented. Even for this theoretically simple case the expressive power is not fully understood [25]. What are the right linear programming algorithms for these CQLs? Both linear and integer programming with a fixed number of variables are relevant combinatorial problems in this context. How can recursion be combined with linear constraints in a safe fashion?

(6) The technology of algorithms for logical theories is still rather complex, but much progress has been accomplished in recent years. For example, see [8, 48, 60] for the state-of-the-art in real closed fields. What is the precise data complexity of CQLs for the various decidable logical theories? Are there interesting syntactic subsets of these theories that are motivated by constraint databases and for which simpler algorithmic techniques can be used? These would be analogous to project-select-join query programs in the relational model.

(7) Finally, can constraint query languages be designed in an extensible fashion? Are there object-oriented CQLs? An example use of such extensibility, would be calling computational geometry algorithms from a CQL interpreter.

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 and V. Vianu. Datalog Extensions for Database Queries and Updates. *J. Comput. System Sci.*, **43** (1991), pp. 62–124.
4. A.K. Aylamazyan, M.M. Gilula, A.P. Stolboushkin, G.F. Schwartz. Reduction of the Relational Model with Infinite Domain to the Case of Finite Domains. *Proc. USSR Acad. of Science (Doklady)*, 286(2):308–311, 1986.
5. D.A. Barrington, N. Immerman, H. Straubing. On Uniformity within NC^1. *JCSS*, 41:274–306,1990.
6. M. Baudinet, M. Niezette, P. Wolper. On the Representation of Infinite Temporal Data and Queries. *Proc. 10th ACM PODS*, 280–290, 1991.
7. R. Bayer, E. McCreight. Organization of Large Ordered Indexes. *Acta Informatica*, 1:173–189, 1972.
8. M. Ben-Or, D. Kozen, J. Reif. The Complexity of Elementary Algebra and Geometry. *JCSS*, 32:251–264, 1986.
9. L. Berman. Precise Bounds for Presburger Arithmetic and the Reals with Addition. *Proc. 18th IEEE FOCS*, pp. 95-99, 1977.
10. A.H. Borning. The Programming Language Aspects of ThingLab, A Constraint-Oriented Simulation Laboratory. *ACM TOPLAS* 3:4:353–387, 1981.
11. A. Brodsky, J. Jaffar, M.J. Maher. Toward Practical Constraint Databases. *Proc. 19th VLDB*, 322–331, 1993.
12. A. Brodsky, C. Lassez. Separability of Polyhedra and a New Approach to Spatial Storage. in [42].

13. A. Brodsky, Y. Sagiv. Inference of Monotonicity Constraints in Datalog Programs. *Proc. 8th ACM PODS*, 190–200, 1989.
14. A. Brodsky, Y. Sagiv. Inference of Inequality Constraints in Logic Programs. *Proc. 10th ACM PODS*, 227–241, 1991.
15. 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.
16. S.R. Buss. The Formula Value Problem is in ALOGTIME. *Proc. 19th ACM STOC*, pp. 123-131, 1987.
17. A.K. Chandra, D. Harel. Structure and Complexity of Relational Queries. *JCSS*, 25:1:99–128, 1982.
18. J. Chomicki. Ptime Query Processing in Temporal Deductive Databases. *Proc. 9th ACM PODS*, 379–391, 1990.
19. J. Chomicki, T. Imielinski. Relational Specifications of Infinite Query Answers. *Proc. ACM SIGMOD*, 174–183, 1989.
20. E.F. Codd. A Relational Model for Large Shared Data Banks. *CACM*, 13:6:377–387, 1970.
21. J. Cohen. Constraint Logic Programming Languages. *CACM*, 33:7:52–68, 1990.
22. A. Colmerauer. An Introduction to Prolog III. *CACM*, 33:7:69–90, 1990.
23. 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.
24. D. Comer. The Ubiquitous B-Tree. *Computing Surveys*, 11:2:121–137, 1979.
25. S.S. Cosmadakis, G.M. Kuper Expressiveness of First-Order Constraint Languages. Manuscript, December 1993.
26. J. Cox, K. McAloon, C. Tretkoff. Computational Complexity and Constraint Logic Programming. *Annals of Math. and AI*, 5:163–190, 1992.
27. 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.
28. J. Ferrante, J.R. Geiser. An Efficient Decision Procedure for the Theory of Rational Order. *Theoretical Computer Science*, 4:227–233, 1977.
29. J. Ferrante, C. Rackoff. A Decision Procedure for the First Order Theory of Real Addition with Order. *SICOMP*, 4:1:69–76, 1975.
30. M.J. Fischer, M.O. Rabin. Super-Exponential Complexity of Presburger Arithmetic. *SIAM-AMS Proc. volume VII*, American Mathematical Society, 1974.
31. E. Freuder. Synthesizing Constraint Expressions. *CACM*, 21:11, 1978.
32. Y. Gurevich, S. Shelah. Fixed-Point Extensions of First-Order Logic. *Annals of Pure and Applied Logic*, 32, 265–280, 1986.
33. 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.
34. R. Helm, K. Marriott, M. Odersky. Constraint-based Query Optimization for Spatial Databases. *Proc. 10th ACM PODS*, 181–191, 1991.
35. R. Hull, J. Su. Domain Independence and the Relational Calculus. *Technical Report* 88–64, University of Southern California.
36. N. Immerman. Relational Queries Computable in Polynomial Time. *Information and Control*, 68:86-104, 1986.
37. J. Jaffar, J.L. Lassez. Constraint Logic Programming. *Proc. 14th ACM POPL*, 111–119, 1987.
38. D.S. Johnson. A Catalogue of Complexity Classes. *Handbook of Theoretical Computer Science*, Vol. A, chapter 2, (J. van Leeuwen editor), North-Holland, 1990.

39. F. Kabanza, J-M. Stevenne, P. Wolper. Handling Infinite Temporal Data. *Proc. 9th ACM PODS*, 392–403, 1990.
40. P.C. Kanellakis. Elements of Relational Database Theory. *Handbook of Theoretical Computer Science*, Vol. B, chapter 17, (J. van Leeuwen editor), North-Holland, 1990.
41. P. C. Kanellakis, G. M. Kuper, P. Z. Revesz. Constraint Query Languages. *Proc. 9th ACM PODS*, 299–313, 1990. Full version available as Brown Univ. Tech. Rep. CS-92-50. To appear in JCSS.
42. P.C. Kanellakis, J.L. Lassez, V.J. Saraswat. *Proceedings of the Workshop on the Principles and Practice of Constraint Programming (PPCP93).* Newport RI, April 1993. Preliminary version available via anonymous ftp from wilma.cs.brown.edu (128.148.33.66) in the directory /pub/ppcp93. To appear from MIT Press.
43. P. C. Kanellakis, S. Ramaswamy, D. E. Vengroff, J. S. Vitter. Indexing for Data Models with Constraints and Classes. *Proc. 12th ACM PODS*, 233–243, 1993.
44. M. Kifer. On Safety, Domain Independence, and Capturability of Database Queries. *Proc. International Conference on Databases and Knowledge Bases*, Jerusalem Israel, 1988.
45. A. Klug. Equivalence of Relational Algebra and Relational Calculus Query Languages having Aggregate Functions. *JACM*, 29:3:699–717, 1982.
46. A. Klug. On Conjunctive Queries Containing Inequalities. *JACM*, 35:1:146–160, 1988.
47. P. Kolaitis, C.H. Papadimitriou. Why not Negation by Fixpoint? *Proc. 7th ACM PODS*, 231–239, 1988.
48. D. Kozen, C. Yap. Algebraic Cell Decomposition in NC. *Proc. 26th IEEE FOCS*, 515–521, 1985.
49. M. Koubarakis. *Foundations of Temporal Constraint Databases.* PhD Thesis. Nat. Tech. Univ. of Athens and Imperial College. 1993.
50. G.M. Kuper. Aggregation in Constraint Databases. in [42].
51. W. Leler. *Constraint Programming Languages.* Addison Wesley, 1987.
52. A. Levy, Y. Sagiv. Constraints and Redundancy in Datalog. *Proc. 11th ACM PODS*, 67–81, 1992.
53. E. McCreight. Priority Search Trees. *SIAM J. Computing*, 14:257–276, 1985.
54. A.K. Mackworth. Consistency in Networks of Relations. *AI*, 8:1, 1977.
55. U. Montanari. Networks of Constraints: Fundamental Properties and Application to Picture Processing. *Information Science*, 7, 1974.
56. I. S. Mumick, S. J. Finkelstein, H. Pirahesh, R. Ramakrishnan. Magic Conditions. *Proc. 9th ACM PODS*, 314–330, 1990.
57. F.P. Preparata, M.I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, 1985.
58. R. Ramakrishnan. Magic Templates: A Spellbinding Approach to Logic Programs. *Proc. 5th International Conference on Logic Programming*, 141–159, 1988.
59. S. Ramaswamy, S. Subramanian. Path Caching: A Technique for Optimal External Searching. Manuscript submitted for publication.
60. 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.
61. P.Z. Revesz. A Closed Form for Datalog Queries with Integer Order. *Proc. 3rd International Conference on Database Theory*, 1990, (to appear in TCS).
62. H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS.* Addison-Wesley, Reading MA, 1990.

63. H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, Reading MA, 1990.
64. V.A. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie Mellon University, 1989.
65. D. Srivastava, R. Ramakrishnan. Pushing Constraint Selections. *Proc. 11th ACM PODS*, 301–316, 1992.
66. G.L. Steele. The Definition and Implementation of a Computer Programming Language Based on Constraints. Ph.D. thesis, MIT, AI-TR 595, 1980.
67. I.E. Sutherland. *SKETCHPAD: A Man-Machine Graphical Communication System.* Spartan Books, 1963.
68. A. Tarski. *A Decision Method for Elementary Algebra and Geometry.* University of California Press, Berkeley, California, 1951.
69. J.D. Ullman. *Principles of Database Systems.* Computer Science Press, 2^{nd} Ed., 1982.
70. R. van der Meyden. The Complexity of Querying Indefinite Data about Linearly Ordered Domains. *Proc. 11th ACM PODS*, 331–346, 1992.
71. A. Van Gelder. Deriving Constraints among Argument Sizes in Logic Programs. *Proc. 9th ACM PODS*, 47–60, 1990.
72. P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.
73. M.Y. Vardi. The Complexity of Relational Query Languages. *Proc. 14th ACM STOC*, 137–146, 1982.