

# Constraint Query Algebras

DINA Q GOLDIN AND PARIS C. KANELLAKIS

dgk@cs.brown.edu

*Computer Science Department, Box 1910, Brown University, Providence, RI 02912*

**Abstract.** Constraint query languages are natural extensions of relational database query languages. A framework for their declarative specification (constraint calculi) and efficient implementation (low data complexity and secondary storage indexing) was presented in Kanellakis et al., 1995. *Constraint query algebras* form a procedural language layer between high-level declarative calculi and low-level indexing methods. Just like the relational algebra, this intermediate layer can be very useful for program optimization. In this paper, we study properties of constraint query algebras, which we present through three concrete examples. The *dense order constraint algebra* illustrates how the appropriate canonical form can simplify expensive operations, such as projection, and facilitate interaction with updates. The *monotone two-variable linear constraint algebra* illustrates the concept of strongly polynomial operations. Finally, the *lazy evaluation of (non)linear constraint algebras* illustrates how large numbers of (non)linear constraints could be implemented with only a small amount of costly symbolic processing.

**Keywords:** database queries, constraint databases, data complexity, linear programming, quantifier elimination, relational algebra

## 1. Introduction

**Motivation:** 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 (Sutherland, 1963). The theme has been investigated since the 1970's, e.g., in artificial intelligence (Montanari, 1974; Mackworth, 1977; Freuder, 1978; Steele, 1980), in graphical-interfaces (Borning, 1981), and in logic programming languages (Jaffar and Lassez, 1987; Dincbas et al., 1988; Colmerauer, 1990). The relevant literature and contributions are too large to attempt a survey. Instead we limit our exposition to recent applications in databases.

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 (Codd, 1970) on the declarative relational calculus and its equivalence to the procedural relational algebra. Indeed, having such languages for ad-hoc database querying is a requirement in today's relational technology (see Abiteboul et al., 1994; Kanellakis, 1990; Ullman, 1982). Querying in Codd's relational calculus can be viewed as a form of limited (but very successful) constraint

programming. The basic motivation of this work is to explore the applicability of constraint programming technology to database languages.

**Constraint Databases:** 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( $\mathbb{R}$ ) (Jaffar and Lassez, 1987), in Prolog III (Colmerauer, 1990), and in CHIP (Dincbas et al., 1988; Van Hentenryck, 1989). The insight that led to CLP is: the unification mechanism of standard Logic Programming can be regarded as a trivial constraint solver (for equality constraints only). Expressiveness is therefore gained by replacing unification with constraint solving, and allowing constraints in logic programs. This advance in CLP is very relevant for database applications.

CLP was adapted to database querying by Kanellakis et al., 1995, who proposed a framework for Constraint Database (CDB) queries by combining bottom-up, efficient, declarative database programming with efficient constraint solving. The insight here, borrowed from CLP research, is: a tuple (or a record or ground fact) in standard relational databases can be regarded as a conjunction of equality constraints on the attributes of the tuple. Integrating constraints with databases leads to the following: *a conjunction of quantifier-free constraints over  $k$  variables, where  $k$  depends on the database schema and not the instance, is an appropriate generalization of the  $k$ -tuple.*

The technical tools for this integration were: *data complexity* (Chandra and Harel, 1982; Vardi, 1982) from database theory, *quantifier elimination* methods from mathematical logic, and *multi-dimensional indexing* (for this part more will be said in Section 6). In Kanellakis et al., 1995 finite relations are generalized to finitely representable relations and appropriate calculi for their data manipulation can be developed in this framework. In many cases, the calculi have polynomial time data complexity. This use of data complexity, a common tool for studying expressibility in finite model theory, distinguishes the CDB framework from arbitrary, and inherently exponential, theorem proving.

The CDB framework, while being a strict generalization of the standard relational framework, also provided a unified view of some previous database research: for example, on the power of constraints for the implicit specification of temporal data (Chomicki and Imielinski, 1989), on relational query safety (Aylamazyan et al., 1986), on conjunctive queries with inequalities (Klug, 1988) and on extending magic sets (Ramakrishnan, 1988).

*Constraint query algebras* form a procedural language layer between high-level declarative calculi and low-level access methods. Just like the relational algebra, this intermediate layer can be very useful for program optimization. Here, we study properties of constraint query algebras, which we present through concrete examples. Although rule-based intermediate constraint languages have been examined (Kanellakis et al., 1995) and optimized, (Mumick et al., 1990; Srivastava and Ramakrishnan, 1992; Brodsky et al., 1993; Stuckey and Sudarshan, 1994), much remains to be done. It is relatively easy to transform a calculus into an algebra, e.g., quantifier elimination is essentially this process. For example, with Fourier-

Motzkin elimination (Schrijver, 1986) one easily derives a “naive” algebra for linear constraint databases (Grumbach et al., 1994). What is a “good” algebra is harder to quantify, and this is what we attempt to do here.

The example algebras we consider involve: (1) *dense order constraints* (see Ferrante and Geiser, 1977; Kanellakis et al., 1995; Klug, 1988), (2) *linear inequality constraints* (see the comprehensive survey in Schrijver, 1986), and (3) lazy evaluation of linear and nonlinear constraints (for *real polynomial constraints* see Tarski, 1951; for recent developments and a symbolic computation survey see Renegar, 1992, and for numerical computation see Van Hentenryck et al., 1995).

**Contributions and Overview:** Previous work on the complexity of constraint database queries (Kanellakis et al., 1995; Grumbach et al., 1994; Paredaens et al., 1995) has concentrated on data complexity, treating the number of variables  $k$  in a query as a constant. We review the basic definitions in Section 2.1.

Algebra-calculus equivalence is fundamental to Codd’s relational framework; it needs to be preserved in constraint databases. Constraint algebras typically consist of operations such as *select*, *project*, *natural-join*, *union*, *rename*, *difference* just like Codd’s relational algebra. Theoretically, any set of operators that preserves the algebra-calculus equivalence can be used, though the author of such algebra can no longer make use of existing relational methodologies for proving equivalence, or for doing query translation and optimization. Additional operators may also prove useful, such as the generalized version of the “aggregation” operator (Chomicki et al., 1996).

The specification and implementation requirements of algebraic operations are detailed in Section 2.2. A minimal requirement is that each algebraic operation must be “efficiently implementable”, where a lot depends on what efficient means.

For example, data complexity arguments oversimplify in order to prove that the computations performed are efficient in the database size. Assuming that the number of variables is a constant  $k$  leads to algebras that are efficient in the database size but “naive”. In such algebras, the projection operation (which captures quantifier elimination) is usually complex and costly, namely exponential in the number of variables  $k$ ; data complexity hides this in a large constant. The critical issue in designing a “good” algebra is to make projection simple and cheap. In this paper we present three possible ways of quantifying “simple” and “cheap”.

**(a):** The fastest imaginable projection, given some subset of variables, would be if we could simply collect the constraints involving these variables. This is what happens with relational algebra where projection is *restriction* and (depending on the representation of sets used) *duplicate elimination*. A set of constraints is *globally consistent* (Freuder, 1982; Dechter, 1992) when the projection of the solution set on any subset of the variables can be computed just this way, via restriction.

This approach is studied in Section 3. The example is an algebra for dense order constraints, which was published by Kanellakis and Goldin, 1994 in preliminary form. As explained in Section 2.2, this algebra also has other good properties with respect to update operations.

**(b):** In Artificial Intelligence, the property of *global consistency*, also known as *decomposability* (Montanari, 1974), is desirable because it allows a *backtrack-free search* (Freuder, 1982). The property has been studied for temporal constraints (Dechter et al., 1991) where it was shown that: “a decomposable constraint set equivalent to a given one can be found in time polynomial in the size of the constraints for any number of variables  $k$ ”. (For an extensive treatment of temporal databases using constraint programming see Koubarakis, 1993).

Clearly, if a class of constraints allows us to compute efficiently, for any constraint set, an equivalent globally consistent representation, then it is possible to implement fast projections for this class. A lot depends on what one considers efficient. For example, for  $m$  linear constraints, when data complexity is used, it is possible to show polynomial bounds but only because the number of variables  $k$  is viewed as constant. The situation is much better when there are algorithms polynomial in  $m, k$ , as in the case of dense order and of temporal constraints. We will call algebras with such efficient operations *strongly polynomial*.

This approach is studied in Section 4. The example is an algebra for monotone linear two-variable constraints. We consider these constraints over the rationals where there is a particularly simple algebra that is strongly polynomial. Temporal constraints are a special form of these monotone constraints over the integers. (However, the approach presented here over the rationals is also applicable to temporal constraints over the integers). The more complex integer case for monotone constraints is analyzed by Hochbaum and Naor, 1994.

**(c):** For a set of  $m$  linear constraints with  $k$  variables, elimination of some variables, i.e., existential quantifier elimination for any  $i \leq k$  variables, has a worst-case bound exponential in  $k$ . Elimination could be exponential because of the output size. A good example is given by Yannakakis, 1988, consisting of a linear constraint set for representing a *parity polytope* in  $k$  dimensions. This polytope, though simple to describe, has  $O(2^k)$  facets, requiring an exponential number of constraints (if no existentially quantified variables are involved in the representation). However, with the use of  $k$  additional variables that are existentially quantified, the number of constraints needed to describe the parity polytope is reduced to  $O(k^2)$  (note that using these additional variables is equivalent to delaying the evaluation of quantifier elimination). Also, there might be other cases between exponential and strongly polynomial. When the linear inequalities contain at most two variables each, Nelson, 1978, has shown that the Fourier-Motzkin algorithm can be optimized to reduce the exponent to  $\log k$ .

Variable elimination, which is exponential in  $(m, k)$ , should be contrasted with the *linear programming problem*, where all variables are eliminated. This *satisfiability* problem for linear constraints, of great practical significance, has worst-case polynomial time algorithms (and efficient average-case algorithms, i.e. simplex). However, even for this problem, the polynomials not only depend on  $m$  and  $k$  but also on the coefficient sizes. The existence of an algorithm for linear programming that is *strongly polynomial*, i.e., where the complexity does not depend on the coefficient sizes, is a major open question.

Strongly polynomial bounds have been achieved for the *linear programming problem* over sets of two-variable linear constraints (Hochbaum and Naor, 1994); its time complexity is  $O(mk^2 \log m)$ . They present a modification of the Fourier-Motzkin algorithm, pruning away most of the constraints that are generated, while preserving *equisatisfiability* of the constraint sets. The actual *equivalence* of the sets is not preserved; indeed, it is not needed to determine feasibility. Unfortunately, this makes the optimizations of Hochbaum and Naor, inapplicable to the projection problem. Thus, the problem of a strongly polynomial algebra for two-variable linear constraints is open.

The pragmatic approach in this third case is as follows: *if strongly polynomial operations are not available then quantifier elimination should be delayed until it is really needed and the representation of constraints should permit such lazy evaluation*. This approach is explored in Section 5. It is applicable to linear and nonlinear constraints as long as queries are positive and could be of significance for constraint query algebra implementations.

**I/O efficiency:** The goal of Sections 3-5 is to develop algebras that resemble the relational algebra in simplicity and efficiency. There is one more important requirement: *indexing for efficient I/O*. This is less of an algebraic and more of a data-structure issue. It was addressed by Kanellakis et al., 1995 and recent progress is surveyed in Section 6, together with some open questions.

## 2. The Constraint Database Framework

### 2.1. Constraint Databases Queries

We present a summary of the framework from Kanellakis et al., 1995.

(1) *A generalized  $k$ -tuple is a quantifier-free conjunction of constraints on  $k$  variables, where these variables 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 two-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  $k$ -ary tuples (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 is a *finite representation* of a possibly infinite set of  $k$ -ary tuples (or points in  $k$ -dimensional space  $D^k$ ). *A generalized database is a finite set of generalized relations.*

(3) *The syntax of a CDB calculus is the union of a relational database query language and formulas in a decidable logical theory.* For example: Relational calculus

(Codd, 1970) + the theory of real closed fields (Tarski, 1951; Renegar, 1992); Relational calculus (or even Inflationary Datalog<sup>∇</sup>, Abiteboul et al., 1994) + the theory of dense order with constants (Ferrante and Geiser, 1977).

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

The framework of Kanellakis et al., 1995 imposes two critical requirements on queries:

(a) *For each input, the queries must be evaluable in closed form and bottom-up.* 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 (Codd, 1970; Ullman, 1982). The precise analogue in relational databases is the notion of weak safety of Aylamazyan et al., 1986. 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*. Such evaluation can usually be described as an algebra.

(b) *For each input, the queries must be evaluable efficiently in the input size, i.e., with at most 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 even in LOGSPACE, depending on the constraints considered (for models of efficient algorithms see Aho et al., 1974).

A query  $Q$  has *data complexity* in PTIME (LOGSPACE) if there is a TM 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). We assume a standard binary encoding of generalized relations. 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. The CDB query framework

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

## 2.2. Relational Algebra

From Codd's original work (Codd, 1970) it follows that: *Relational Calculus on finite sets can be evaluated bottom-up in closed form*. This bottom-up evaluation, with LOGSPACE data complexity (Vardi, 1982), is known as the *Relational Algebra*. It serves as a model for the various Constraint Query Algebras we develop.

Relational Algebra is based on a small number of primitive operations on relations, i.e., sets of tuples without duplicates. *Duplicate elimination* depends on the relational algebra implementation and is typically done only when needed, with lazy evaluation. We will revisit lazy evaluation in Section 5; in this section, we present a definition of Relational Algebra and briefly discuss query optimization.

Let  $X$  be a finite set of attributes from an (infinite) set  $U$ . An  $X$ -tuple  $t$  is a mapping from  $X$  into a set  $D$  distinct from  $U$  (of atomic constants of the database). A *relation*  $r$  over attributes  $\alpha(r) = X$  is a finite set of  $X$ -tuples. For each algebraic operation, in clause (1) we have the conditions required of the argument relation arities and the result arity; in clause (2) we have the semantics of the operation.

**Projection:**  $\pi_X(r)$  is the projection of  $r$  on  $X$ .

(1)  $X \subseteq \alpha(r)$  and  $\alpha(\pi_X(r)) = X$ .

(2)  $\pi_X(r) = \{t[X] : t \in r\}$ , where  $t[X]$  is the restriction of the mapping  $t$  on  $Z$ .

**Selection:**  $\varsigma_{A=B}(r)$  is the selection on  $r$  by  $A = B$ .

(1)  $A, B \in \alpha(r)$  and  $\alpha(\varsigma_{A=B}(r)) = \alpha(r)$ .

(2)  $\varsigma_{A=B}(r) = \{t : t \in r \text{ and } t[A] = t[B]\}$ .

*Remark:* Additional syntax for the selection operation can incorporate constants (Chandra and Harel, 1982).

**Natural Join:**  $r_1 \bowtie r_2$  is the (natural) join of  $r_1$  and  $r_2$ .

(1)  $\alpha(r_1 \bowtie r_2) = \alpha(r_1) \cup \alpha(r_2)$ .

(2)  $r_1 \bowtie r_2 = \{t : t \text{ is an } \alpha(r_1) \cup \alpha(r_2)\text{-tuple, such that } t[\alpha(r_1)] \in r_1 \text{ and } t[\alpha(r_2)] \in r_2\}$ .

**Cross-Product:**  $r_1 \times r_2$  is the cross-product of  $r_1$  and  $r_2$ .

(1)  $\alpha(r_1) \cap \alpha(r_2) = \emptyset$ ;  $\alpha(r_1 \times r_2) = \alpha(r_1) \cup \alpha(r_2)$ .

(2) This is a special case of Natural-Join.

**Intersection:**  $r_1 \cap r_2$  is the intersection of  $r_1$  and  $r_2$ .

(1)  $\alpha(r_1) = \alpha(r_2)$  and  $\alpha(r_1 \cap r_2) = \alpha(r_1)$ .

(2) This is a special case of Natural-Join.

*Remark:* Natural-Join is a very common operation in relational databases. Both Cross-Product and Intersection are special cases of this operation. Note however that Natural-Join is expressible using Project, Select, and Cross-Product. Here, we will provide analogue operations to Natural-Join.

**Union:**  $r_1 \cup r_2$  is the union of  $r_1$  and  $r_2$ .

(1)  $\alpha(r_1) = \alpha(r_2)$  and  $\alpha(r_1 \cup r_2) = \alpha(r_1)$ .

(2)  $r_1 \cup r_2 = \{t : t \in r_1 \text{ or } t \in r_2\}$ .

**Renaming:**  $\varrho_{B|A}(r)$  is the renaming in  $r$  of  $A$  into  $B$ .

(1)  $A \in \alpha(r)$ ,  $B \notin \alpha(r)$  and  $\alpha(\varrho_{B|A}(r)) = (\alpha(r) - \{A\}) \cup \{B\}$ .

(2)  $\varrho_{B|A}(r) = \{t : \text{for some } t' \in r, t[B] = t'[A] \text{ and } t[C] = t'[C] \text{ when } C \neq B \}$ .

**Difference:**  $r_1 - r_2$  is the difference of  $r_1$  and  $r_2$ .

(1)  $\alpha(r_1) = \alpha(r_2)$  and  $\alpha(r_1 - r_2) = \alpha(r_1)$ .

(2)  $r_1 - r_2 = \{t : t \in r_1 \text{ and } t \notin r_2\}$ .

The *positive* fragment of Relational Algebra consists of the above operations except Difference. Note that Relational Algebra is equivalent to the *domain-independent* Relational Calculus for both finite and infinite (i.e., unrestricted) relations (Kanellakis, 1990).

For a given query, there is no unique algebraic expression to evaluate it. *Algebraic transformations* (such as selection propagation and join ordering) are heuristics for transforming algebraic expressions to equivalent ones that are likely to yield faster query evaluation, mostly by reducing the amount of I/O performed.

The use of relational algebra in query optimization is based on two key notions:

- Specifying a search space of semantically equivalent relational algebraic expressions that could have different evaluation costs.
- Estimating the cost of performing an operation (for example, natural join) based on the statistics (for example, cardinality and selectivity) of its operands, and estimating statistics for the result of the operation.

Query optimization via algebraic transformations 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.

### 2.3. Constraint Query Algebras

In Kanellakis et al., 1995, the existence of an efficient bottom-up evaluation strategy for dense order constraints was proven by grouping the tuples in a generalized relation into *r-configurations*. However, they did not provide an algebraic syntax and semantics. Such syntax and semantics are not hard to devise. A typical example is in Grumbach et al., 1994, where a syntax and a semantics is described for linear constraints. In this subsection we examine the issues facing any designer of a *Constraint Query Algebra* (CQA) over some class of constraints  $\mathcal{C}$ .

**CQA Closure:** First, we want to make sure that for any CDB query calculus (see Kanellakis et al., 1995) expression, there exists an equivalent CQA expression. It is also highly desirable that there be an efficient *translation* between the two representations. In most cases, this requirement is satisfied using the closure condition; see (a) of the previous subsection. This is because the closure condition is proved using the relational calculus and the relational algebra over unrestricted, as opposed to finite, relations that also are finitely representable. We refer to Kanellakis et al. for more details on the calculus and in what follows we concentrate on

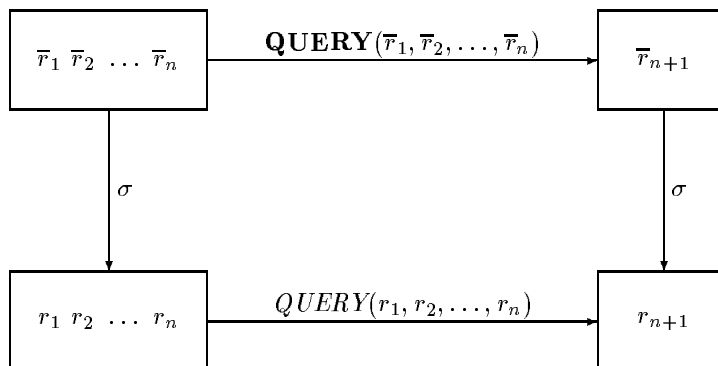


Figure 1. The Closure Condition

showing closure with a set of primitives that capture all the calculus connectives (for a detailed example see Section 3 and the Appendix).

The syntax of the algebraic operators should capture propositional logic (and, or, not) through equality constraints, selection, cross-product, union, and difference, as well as first-order predicate logic (existential quantification) through projection. The semantics of the operations should be such as to provide closure (and equivalence to the calculus typically follows). In particular, if  $\bar{r}_1, \dots, \bar{r}_n$  are generalized relations which describe (possibly infinite) relations  $\sigma(\bar{r}_1), \dots, \sigma(\bar{r}_n)$ , and if **OP** is an operation in a Constraint Query Algebra (where *OP* is the corresponding operation in the Relational Algebra over unrestricted relations) then it should be the case that

$$\text{if } \bar{r}_{n+1} = \mathbf{OP}(\bar{r}_1, \dots, \bar{r}_n), \text{ then } \sigma(\bar{r}_{n+1}) = \mathbf{OP}(\sigma(\bar{r}_1), \dots, \sigma(\bar{r}_n)).$$

We refer to this property as the *closure* of a CQA (see Figure 1).

As a result of the closure condition, CQAs admit the same notion of expression equivalence as relational algebras. It is likely that the heuristics for optimizing algebraic expressions can be generalized to CQA queries. In Section 6, we discuss the issues of implementation that affect these heuristics.

**Generalized Projection:** Generalized to constraint tuples, the definition of projection is as follows:

**Definition.** Let  $\bar{t}$  be a generalized tuple over variables  $X$ , and  $S$  be a subset of  $X$ . A *projection* of  $\bar{t}$  onto  $S$ , denoted by  $\pi_S(\bar{t})$ , is a generalized tuple over  $S$ , such that:

An assignment  $p$  to the variables in  $S$  satisfies  $\pi_S(\bar{t})$  iff there exists an extension of  $p$  to all the variables in  $X$  satisfying  $\bar{t}$ .

The restriction operation of the previous subsection generalizes as well:

**Definition.** A *restriction* of  $\bar{t}$  to  $S$ , denoted by  $\bar{t}[S]$ , is a subset of the constraints in  $\bar{t}$  containing those constraints that do not involve any variables outside of  $S$ .

Note that projection and restriction are not necessarily equivalent in CQAs. A generalized tuple where they are equivalent is *globally consistent*:

**Definition.** A constraint set  $\bar{t}$  over variables  $X$  is *globally consistent* if, for any subset  $S$  of  $X$ ,  $\pi_S(\bar{t}) = E[S]$ .

Clearly, all standard relational tuples are globally consistent.

**Generalized Selection:** Constraints can be added to the syntax of the formula  $F$  in the expression  $\zeta_F(R)$ . At a minimum, the selection operator should admit as  $F$  any constraint from  $\mathcal{C}$ .

**CQA Efficiency Issues:** In addition to the above issues dealing with the syntax and semantics of a CQA, there are issues of efficiency. We examine (1) redundancy removal via satisfiability checks, (2) interaction with dynamic updates, and (3) operation efficiency.

(1) We expect the size of the representation of the constraint database to be somehow correlated to the amount of data stored. In particular, it is unreasonable if many of the generalized tuples have empty extensions, i.e., if the corresponding formula is unsatisfiable. It is also unreasonable to have too much duplication, i.e., if many of the points in the unrestricted database belongs to many generalized tuples at once.

In most applications, the number of tuples intersecting at any one point is likely to be much smaller than the total number of tuples. Given this, the first source of redundancy is of greater concern. For example, whenever we define a *join* or an *intersection* by “and”-ing pairs of tuples, the naive interpretation of this operation would result in many empty tuples.

The implementation of the algebra should have a mechanism for eliminating (or avoiding) empty tuples. Essentially, this means performing a *satisfiability* check for every tuple. To make this task easier, it is reasonable to stipulate some specific *canonical form* for all generalized tuples in the database.

(2) There are often implementation reasons to impose a specific canonical form on the generalized tuples in the database. Moreover, a *pre-processing* mechanism needs to be provided for converting constraint formulas into this format. In this case, *updates* to the database become an issue. It is highly desirable that such *updates* can be performed in time that depends only on the size of the individual generalized tuple.

For example, we may view the *r-configurations* of Kanellakis et al., 1995 as a particular canonical form. There is an explicit algorithm provided there to convert a generalized relation into this format. When a tuple  $t$  is added to an already-converted relation  $R$ , it is necessary to perform work that is at least linear in the size of  $R$  to ensure that  $\{t\} \cup R$  is in the proper format. This renders *r-configurations* impractical for representing generalized databases.

(3) The last issue that we address in this subsection is operation efficiency. We focus on *projection*, one of the basic operations of relational algebra. In linear

programming, projection is known as *variable elimination*, where projecting a generalized relation  $r$  (over variable set  $X$ ) onto variable set  $Y$  is eliminating quantifiers  $X - Y$  from the formula representation of  $r$ . The *Fourier-Motzkin variable elimination method* is the most basic algorithm for variable elimination in linear inequalities.

Projection via restriction (modulo duplicate elimination) is the ideal, realized in the Relational Algebra. Strongly polynomial operations (as described in Section 1) are also desirable. For a set of  $m$  linear constraints, variable elimination has a worst-case bound exponential in the number of variables  $k$ , and is not strongly polynomial. However, there are subclasses of linear constraints where projection can be performed faster. For *temporal* constraints (of the type  $x - y \leq a$ ), which reduce to dense order constraints when  $a = 0$ , computing projections is reducible to computing shortest paths (Dechter et al., 1991), so strong polynomiality follows.

In Section 3, we present an algebra for dense order constraints that addresses all of the issues raised here. This algebra could be extended to handle temporal constraints with no performance penalty. Then, in Section 4, we prove strong polynomiality for projections over monotone two-variable linear constraints, of which dense order constraints are a subclass.

### 3. A Dense Order Constraint Algebra

*Dense order inequality constraints* are all formulas of the form  $x\theta y$  or  $x\theta c$ , where  $x, y$  are variables,  $c$  is a constant, and  $\theta$  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 Ferrante and Geiser, 1977 and for its data complexity (with and without recursion) see Kanellakis et al., 1995.

Most commonly, dense order constraints are used to represent (multi-dimensional) rectangles, or intersections of rectangles and diagonal hyperplanes. Their expressibility suffices in many spatial database and computational geometry applications. Note that the geometry of dense order constraints is richer than that presented in Grumbach and Su, 1995; for example, a rectangle with a cut-off corner, though representable with dense order constraints:  $(3 < x < 5, 1 < y < 4, x > y)$ , was not mentioned there.

In this section we present a CQA for dense order constraints. We first define generalized tuples, and adopt the use of a specific tabular representation for them. We then stipulate that these tuples are put into a *canonical* (or “tightmost”) form. Any conjunction of constraints can be transformed into a union of such canonical tuples. We define algebraic operations over these canonical tuples which satisfy the criteria discussed in Section 2.3. Finally, we show that this algebra has the desirable *closure* property. The equivalence of the dense order constraint algebra with the calculus is a consequence of this property.

### 3.1. Constraint Sets and Generalized Tuples

Let  $X$  be a set of variables. A *constraint set*  $C$  over  $X$  is a finite set of dense order constraints, each of which is of one of the four types:

$$(x \text{ op } y), (x > l), (x < u), (x = c),$$

where  $x, y \in X$ ,  $op \in \{<, >, =\}$  and  $\{c, l, u\} \subseteq D$ . We refer to these types as *two-variable*, *lower bound*, *upper bound*, and *equality* respectively.

We denote the set of all assignments of elements of  $D$  to the variables in  $X$  that satisfy all constraints in  $C$  by  $P(C)$ , the *point set* of  $C$ . We say that  $C$  is *consistent* if  $P(C)$  is not empty. If  $C$  is a consistent constraint set, then the projection of  $P(C)$  onto any variable in  $C$  must either be a single point or an open interval. Two constraint sets  $C_1, C_2$  are *equivalent* iff their point sets are the same:

$$C_1 \equiv C_2 \text{ iff } P(C_1) = P(C_2).$$

We now extend the notation by introducing a new comparison operator  $?$ , where a variable constraint  $(x ? y)$  always evaluates to TRUE; it represents the absence of any constraint between  $x$  and  $y$ . Similarly, we introduce new constants  $-\infty$  and  $+\infty$ , where the lower bound  $(-\infty < x)$  and the upper bound  $(x < +\infty)$  always evaluate to TRUE; they indicate the absence of any upper or lower bounds on  $x$ .

It is easy to see that any constraint set  $C$  can be represented by an equivalent set  $C'$  which has *exactly* one constraint  $\mu_{x,y}$  between any pair of variables, and *exactly* one pair of constants  $(l_x, u_x)$  delimiting any variable  $x$ , where  $l_x \leq u_x$ . We refer to such a constraint set as a *generalized tuple*.

The conjunction of the constraints in a generalized tuple  $\bar{t}$  is the formula  $F(\bar{t})$ :

$$F(\bar{t}) = \bigwedge (c_i) : c_i \in \bar{t}.$$

The terms that are trivially true may be omitted from the formula. Clearly,  $F(\bar{t})$  is satisfiable if and only if  $\bar{t}$  is consistent.

We adopt a tabular representation for generalized tuples, as in the example below. The size of this representation is fixed by the size of  $X$ .

*Example:* The following is a tuple  $\bar{t}$  on variables  $(A, B, C)$  whose formula is

$$(A < 5 \wedge 1 < B \wedge B < 7 \wedge C = 6 \wedge A > B):$$

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

□

From now on, we omit the adjective “generalized” when it is clear from the context.

### 3.2. The Canonical Form

We now turn our attention to sets of constraints that are entailed by a tuple.

**Definition.** A consistent generalized tuple  $\bar{t}_0$  *entails* a constraint  $\theta$  if the universal closure of  $(F(\bar{t}_0) \Rightarrow \theta)$  is true.

The set of constraints entailed by a given consistent tuple could be infinite. We restrict ourselves to a finite subset, by considering only the constraints whose constants appear in the original tuple. We subdivide this finite set, grouping constraints according to the type of the constraint and the variables that appear in it:

**Definition.** Let  $\bar{t}_0$  be a consistent tuple over attributes  $X$ . For all  $x, y \in X$ , we now define the constraint sets  $\mathcal{S}(\bar{t}_0, x, y)$ ,  $\mathcal{S}(\bar{t}_0, x, L)$ , and  $\mathcal{S}(\bar{t}_0, x, U)$ . Let  $\theta$  be a constraint entailed by  $\bar{t}_0$  such that all constants appearing in  $\theta$  also appear in  $\bar{t}_0$ . Then:

$$\begin{aligned} \theta \in \mathcal{S}(\bar{t}_0, x, y) & \text{ if } \theta = (x \text{ op } y); \\ \theta \in \mathcal{S}(\bar{t}_0, x, L) & \text{ if } \theta = (x \text{ op } c), \text{ where } \text{op} \in \{>, =\}; \\ \theta \in \mathcal{S}(\bar{t}_0, i, U) & \text{ if } \theta = (x \text{ op } c), \text{ where } \text{op} \in \{<, =\}. \end{aligned}$$

By default,  $\mathcal{S}(\bar{t}_0, x, y)$  contains the constraint  $(x ? y)$ ,  $\mathcal{S}(\bar{t}_0, x, L)$  contains  $(-\infty < x)$ , and  $\mathcal{S}(\bar{t}_0, x, U)$  contains  $(x < +\infty)$ . These constraint sets  $\mathcal{S}$  are the *entailed constraint sets* of  $\bar{t}_0$ . □

By definition, each entailed constraint set  $\mathcal{S}()$  is non-empty and finite. Also, for any two constraints in  $\mathcal{S}()$ , one of the two entails the other (by properties of dense order). By structural induction, combined with the non-symmetry of entailment, we know that each  $\mathcal{S}()$  has a unique member that entails all other members. We refer to this member as the *tightmost* member of  $\mathcal{S}()$ .

For any generalized tuple  $\bar{t}_0$ , the set of the tightmost members of all the entailed constraint sets of  $\bar{t}_0$  forms another generalized tuple, known as *the canonical form of  $\bar{t}_0$* :

**Definition.** The *canonical form* of  $\bar{t}_0$  is a constraint set  $\bar{t}_c$  consisting of the tightmost members for all the entailed constraint sets of  $\bar{t}_0$ :

- (1) tightmost member of  $\mathcal{S}(\bar{t}_0, x_i, x_j)$  is the two-variable constraint over  $(x_i, x_j)$  in  $\bar{t}_c$ ;
- (2) tightmost member of  $\mathcal{S}(\bar{t}_0, x_i, L)$  is the lower bound over  $x_i$  in  $\bar{t}_c$ ;
- (3) tightmost member of  $\mathcal{S}(\bar{t}_0, x_i, U)$  is the upper bound over  $x_i$  in  $\bar{t}_c$ .

A consistent generalized tuple  $\bar{t}_0$  is *canonical* if it is its own canonical form.

*Example:* The tuple  $\bar{t}$  in the previous example is not canonical because:

$$(A < 5 \wedge A > B) \Rightarrow (B < 5), \text{ but the upper bound on } B \text{ is } (B < 7).$$

The following tuple  $\bar{t}'$  is a canonical form of  $\bar{t}$ :

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

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

□

Next, we show that the canonical form of  $\bar{t}_0$  is equivalent to  $\bar{t}_0$ .

**LEMMA 1** *Let  $\bar{t}_c$  be the canonical form of  $\bar{t}_0$ . Then,  $P(\bar{t}_c) = P(\bar{t}_0)$ .*

**Proof:** All constraints in  $\bar{t}_c$  are entailed by  $\bar{t}_0$ ; therefore,  $F(\bar{t}_0) \Rightarrow F(\bar{t}_c)$ , and  $P(\bar{t}_0) \subseteq P(\bar{t}_c)$ . It remains to show that  $F(\bar{t}_c) \Rightarrow F(\bar{t}_0)$ . Let  $\theta$  be an arbitrary constraint in  $\bar{t}_0$ ;  $\theta$  must belong to some entailed subset of  $\bar{t}_0$ . Let  $\theta'$  be the tightmost member of that subset;  $\theta' \in \bar{t}_c$  and  $\theta' \Rightarrow \theta$ . ■

Furthermore, the canonical form of a tuple has the following interesting property:

**LEMMA 2** *Let  $\bar{t}_c$  be the canonical form of  $\bar{t}_0$ . Let  $\theta_1$  be some constraint entailed by  $\bar{t}_0$ , and  $\theta_2$  be the constraint in  $\bar{t}_c$  of the same type and with the same variables as  $\theta_1$ . Then,  $\theta_2$  entails  $\theta_1$ .*

To prove this theorem, we assume there is a constraint involving a constant  $u \notin D(\bar{t}_0)$  which is tighter than the corresponding entailed constraints over constants  $a, b \in D(\bar{t}_0)$  are the closest to  $u$  from below and above respectively. By considering satisfiability and equivalence of various conjunctions of constraints, we derive a contradiction.

**Proof:**

1. The Proposition is trivially true when  $\theta_1$  belongs to some entailed canonical set of  $\bar{t}_0$ . Therefore, we assume that it does not; i.e.,  $\theta_1$  contains some constant not in  $D(\bar{t}_0)$ , where  $D(\bar{t}_0)$  is the set of constants appearing in  $\bar{t}_0$ . W.l.o.g., let  $\theta_1 = (x_i > u)$ , and  $\theta_2 = (x_i > a)$ .

2. Let us further assume that  $\theta_2$  does not entail  $\theta_1$ . Then, it must be the case that  $\theta_1 \Rightarrow \theta_2$ , and  $u > a$ .
3. Let  $\theta_3 = (x_i > b)$ , where  $b$  is the smallest constant in  $D(\bar{t}_0)$  such that  $b > a$ . Clearly,  $\theta_3 \Rightarrow \theta_2$ ; but  $\theta_2$  is the tightmost member of the entailed subset. This means that  $\theta_3$  is not in the subset, so  $\bar{t}_0 \not\Rightarrow \theta_3$ .
4. If  $b < u$ , then  $\theta_1 \Rightarrow \theta_3$ , and  $\bar{t}_0 \Rightarrow \theta_3$ , which is impossible. So,  $a < u < b$ . To summarize:

$$\begin{aligned} \theta_1 &= (x_i > u), u \notin D(\bar{t}_0), F(\bar{t}_0) \equiv F(\bar{t}_0) \wedge \theta_1; \\ \theta_2 &= (x_i > a), a \in D(\bar{t}_0), F(\bar{t}_0) \equiv F(\bar{t}_0) \wedge \theta_2; \\ \theta_3 &= (x_i > b), b \in D(\bar{t}_0), F(\bar{t}_0) \not\equiv F(\bar{t}_0) \wedge \theta_3, a < u < b, \nexists c \in D(\bar{t}_0) : \\ & a < c < b. \end{aligned}$$

5. Let  $\phi = (a < x_i \leq u)$ ; note that  $\theta_2 \equiv (\theta_1 \vee \phi)$ . By definition,  $(\theta_1 \wedge \phi)$  is unsatisfiable, which means that  $F(\bar{t}_0) \wedge (\theta_1 \wedge \phi)$  is unsatisfiable.  $F(\bar{t}_0) \wedge (\theta_1 \wedge \phi) \equiv (F(\bar{t}_0) \wedge \theta_1) \wedge \phi \equiv F(\bar{t}_0) \wedge \phi$ . Therefore,  $(F(\bar{t}_0) \wedge \phi)$  is unsatisfiable.
6. Let  $\psi = (u < x_i < b)$ . By the work of Kanellakis et al., 1995 on r-configurations,  $F(\bar{t}_0) \wedge \phi \equiv F(\bar{t}_0) \wedge \psi$ . Therefore,  $(F(\bar{t}_0) \wedge \psi)$  is unsatisfiable.
7. Let  $\phi' = (u < x_i \leq b)$ ; note that  $\theta_1 \equiv (\theta_3 \vee \phi')$ . Then:

$$F(\bar{t}_0) \equiv F(\bar{t}_0) \wedge \theta_1 \equiv F(\bar{t}_0) \wedge (\theta_3 \vee \phi') \equiv (F(\bar{t}_0) \wedge \theta_3) \vee (F(\bar{t}_0) \wedge \phi').$$

If  $F(\bar{t}_0) \wedge \phi'$  is unsatisfiable, then  $F(\bar{t}_0) \equiv F(\bar{t}_0) \wedge \theta_3$ . This is a contradiction, so  $(F(\bar{t}_0) \wedge \phi')$  must be satisfiable.

8. Note that  $\phi' \equiv \psi \vee (x_i = b)$ . So,

$$\begin{aligned} F(\bar{t}_0) \wedge \phi' &\equiv F(\bar{t}_0) \wedge (\psi \vee (x_i = b)) \equiv (F(\bar{t}_0) \wedge \psi) \vee (F(\bar{t}_0) \wedge (x_i = b)) \equiv \\ & F(\bar{t}_0) \wedge (x_i = b). \end{aligned}$$

Therefore,  $F(\bar{t}_0) \wedge (x_i = b)$  is satisfiable.

9. So, it must be the case that the projection of  $P(C)$  onto  $x_1$  includes  $b$  and excludes the open interval between  $a$  and  $b$ . The projection of  $P(C)$  onto  $x_i$  must also include values in the open interval between  $b$  and  $\{+\infty\}$ ; otherwise,  $\bar{t}_0 \Rightarrow (x_i = b)$ , and  $\theta_2$  would not be the tightmost subset member.
10. This is impossible, since the projection of  $P(C)$  onto  $x_1$  must either be a single point or an open interval. Our assumption that  $\theta_1 \Rightarrow \theta_2$  has led to a contradiction. Therefore, it must be the case that  $\theta_2 \Rightarrow \theta_1$ .

■

It follows that all tuples that are equivalent have the same canonical form:

**LEMMA 3** *Let  $\bar{t}_1$  be any tuple equivalent to  $\bar{t}_0$ ; let  $\bar{t}_c$  be the canonical form of  $\bar{t}_0$ . Then,  $\bar{t}_c$  is the canonical form of  $\bar{t}_1$ .*

Lemmas 1 and 3 allow us to use canonical tuples as a unique representative of any equivalence class of constraint sets:

**THEOREM 1** *Each equivalence class  $E$  of consistent tuples contains a unique member  $\bar{t}_c$  such that  $\forall \bar{t} \in E$ ,  $\bar{t}_c$  is the canonical form of  $\bar{t}$ .*

Since the definition of a canonical form is constructive, it is easy to check that the canonical form for any generalized  $n$ -tuple can be found efficiently. A naive implementation of the check would rely on the following facts:

1. Given a generalized tuple  $\bar{t}$  over  $n$  attributes, there are  $O(n^2)$  distinct dense order constraints over the same set of constants.
2. Checking each of these constraints for entailment is equivalent to checking its negation for consistency with  $\bar{t}$ .
3. The consistency check for dense order constraints can be implemented via a shortest-path algorithm (Aspvall and Shiloach, 1980).

For a less naive implementation, we can avoid executing the shortest-path algorithm every time by storing its results in a *complete directed graph* (Dechter et al., 1991).

We are now ready to define canonical relations and canonical databases consisting of such canonical tuples:

**Definition.**

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

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

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

$\bar{t}_1$	$A$	$B$	$C$	$\xrightarrow{\pi(A,C)}$	$\bar{t}'_1$	$A$	$C$	$\bar{t}'_2$	$A$	$C$	
$\bar{l}_1$	1	1	6		$\bar{l}_2$	4	1	3	$\bar{l}'_1$	1	6
$\bar{u}_1$	5	5	6		$\bar{u}_2$	7	1	$\infty$	$\bar{u}'_1$	5	6
$\bar{\mu}_1$	=	>	<		$\bar{\mu}_2$	=	>	?	$\bar{\mu}'_1$	=	<
	<	=	<			<	=	<		<	<
	>	>	=			?	>	=		?	=

Figure 2. Projection

### 3.3. The Algebraic Operations

We can now introduce the syntax and the semantics of an algebra over canonical relations. As in Codd's relational algebra (Codd, 1970; Kanellakis, 1990), we define basic algebraic operations *projection* ( $\pi$ ), *selection* ( $\varsigma$ ), *natural join* ( $\bowtie$ ), *union* ( $\cup$ ), *difference* ( $-$ ), and *renaming* ( $\varrho$ ), which map one or more canonical relations to a new one. *Algebraic queries* over canonical databases are composed of these basic operations.

For each operation **OP** on canonical relations, we claim that the result is a canonical relation which has the following closure property:

$$\text{if } \bar{r}' = \mathbf{OP}(\bar{r}_1, \dots, \bar{r}_n), \text{ then } \sigma(\bar{r}') = OP(\sigma(\bar{r}_1), \dots, \sigma(\bar{r}_n)),$$

where  $OP$  is the operation of the same name in the relational algebra in Kanellakis, 1990, and  $\sigma$  is defined in Definition 3.2.

In our definitions of operators, we use the notation  $\alpha(\bar{r})$  for the variables of a canonical relation  $\bar{r}$ . To prove the closure theorem of this section we compose the closure properties of operations. (For the detailed proofs, see Appendix).

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

- (1)  $\alpha(\pi_Z(\bar{r})) = Z$ ,
- (2)  $\pi_Z(\bar{r}) = \{(\bar{t}[Z]) : \bar{t} \in \bar{r}\}$ ,

where  $\bar{t}' = (\bar{t}[Z])$  is the *restriction* of  $\bar{t}$  to variables  $Z$  (Definition ).

Syntactically, the variables in  $\bar{t}'$  have the same bounds and the same constraints as the corresponding variables in  $\bar{t}$ , and all other bounds and constraints are dropped. (Note: Projection is restriction, just as for the standard relational algebra; this corresponds to the notion of *global consistency* (Freuder, 1982; Dechter, 1992) for canonical tuples).

*Example:* Let  $\bar{r} = \{\bar{t}_1, \bar{t}_2\}$  be a canonical relation over variables  $(A, B, C)$ . We compute  $\pi_{(A,C)}(\bar{r}) = \{\bar{t}'_1, \bar{t}'_2\}$  (see Figure 3.3).  $\square$

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

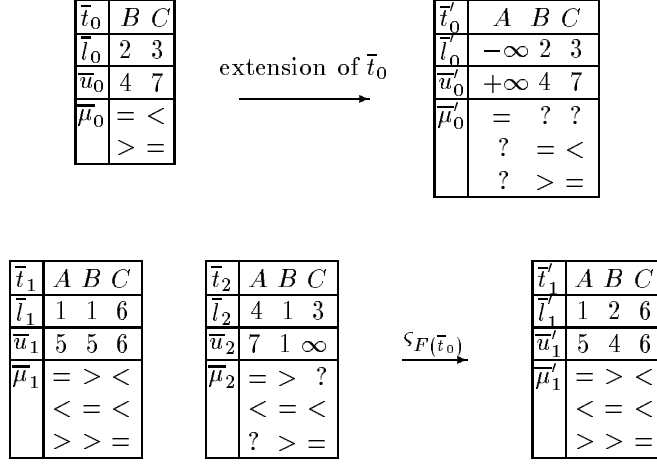


Figure 3. Selection

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

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

**Definition.** Let  $\bar{t}_0$  be a canonical tuple over variables  $Z$ , and  $X$  be a superset of  $Z$ . Then, the *extension* of  $\bar{t}_0$  to  $X$ , written  $(\bar{t}_0 \uparrow X)$  is the most general canonical tuple over  $X$  such that  $\bar{t}_0 = ((\bar{t}_0 \uparrow X) \downarrow Z)$ . In particular,

- (a)  $\forall x \in X$  such that  $x \notin Z$ , the bounds on  $x$  are  $(-\infty < x < +\infty)$ ;
- (b)  $\forall x, y \in X$  such that either  $x \notin Z$  or  $y \notin Z$ ,  $\mu_{x,y} = (x ? y)$ .

The *common tuple* of two canonical tuples over the same variables is formed by taking a union of the constraint sets for the two tuples, and putting it into canonical form. If the union is inconsistent, we say that there is no common tuple.

Let  $\bar{t}$  be the common tuple of  $\bar{t}_1$  and  $\bar{t}_2$ . Each entry in the table for  $\bar{t}$  is obtained by considering the pair of corresponding entries in the tables for  $\bar{t}_1$  and  $\bar{t}_2$ , and taking the tighter one. The resulting tuple is then put into canonical form. Clearly,  $F(\bar{t}) \equiv (F(\bar{t}_1) \wedge F(\bar{t}_2))$ , and  $P(\bar{t}) = P(\bar{t}_1) \cap P(\bar{t}_2)$ .

The following is an example of a Selection operation:

*Example:* Let  $\bar{r} = \{\bar{t}_1, \bar{t}_2\}$  be a canonical relation over variables  $(A, B, C)$ , and let  $\bar{t}_0$  be a canonical tuple over variables  $(B, C)$ , where  $F(\bar{t}_0) = (2 < B < 4 \wedge 3 < C < 7)$ . We compute  $\varsigma_{F(\bar{t}_0)}(\bar{r})$  (see Figure ). The result is the common tuple of  $\bar{t}_1$  and  $\bar{t}_0 \uparrow (A, B, C)$ , since the other pair of tuples has no common tuple.  $\square$

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

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

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

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

$\downarrow \bowtie$

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

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

Figure 4. Join

**NATURAL JOIN.** If  $\bar{r}_1$  is a canonical relation over variables  $X$ ,  $\bar{r}_2$  is a canonical relation over variables  $Y$ , and  $Z = X \cup Y$ , then  $\bar{r}_1 \bowtie \bar{r}_2$  is the natural join of  $\bar{r}_1$  and  $\bar{r}_2$ :

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

*Example:* Let  $\bar{r}_1 = \{\bar{t}_1, \bar{t}_2\}$  be a canonical relation over variables  $(A, B, C)$ , and  $\bar{r}_2 = \{\bar{t}_3, \bar{t}_4\}$  be a canonical relation over variables  $(B, D)$ . We compute  $\bar{r}_1 \bowtie \bar{r}_2$  (see Figure ). It consists of two tuples,  $\bar{t}'_1$  and  $\bar{t}'_2$ , where  $\bar{t}'_1$  is the common tuple of  $\bar{t}_1$  and  $\bar{t}_3$ , and  $\bar{t}'_2$  is the common tuple of  $\bar{t}_2$  and  $\bar{t}_3$ .  $\square$

**UNION.** If  $\bar{r}_1$  and  $\bar{r}_2$  are canonical relations over variables  $X$ , then  $\bar{r}_1 \cup \bar{r}_2$  is the union of  $\bar{r}_1$  and  $\bar{r}_2$ :

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

**DIFFERENCE.** If  $\bar{r}_1$  and  $\bar{r}_2$  are canonical relations over variables  $X$ , then  $\bar{r}_1 - \bar{r}_2$  is the difference of  $\bar{r}_1$  and  $\bar{r}_2$ :

- (1)  $\alpha(\bar{r}_1 - \bar{r}_2) = X$ ;
- (2) If size of  $\bar{r}_2 = 0$ ,  $\bar{r}_1 - \bar{r}_2 = \bar{r}_1$ ; otherwise, the definition is recursive:  
 $\bar{r}_1 - \bar{r}_2 = \cup_{\bar{t}_1 \in \bar{r}_1} \{ \text{tupdif}(\bar{t}_1, \bar{t}_2) - \text{setdif}(\bar{r}_2, \{\bar{t}_2\}) : \bar{t}_2 \in \bar{r}_2 \}$ ,

where *setdif* is the standard set difference operator and *tupdif* is *tuple difference*, defined next.

**Definition.** In the following definition, the word *constraint* will refer either to a *variable constraint*, or to the conjunction of a lower bound and an upper bound (an *interval constraint*). Let  $\bar{t}_1$  and  $\bar{t}_2$  be canonical tuples over variables  $X$ . Their *tuple difference*,  $tupdif(\bar{t}_1, \bar{t}_2)$  is a set of tuples:

1. If there exists a pair of constraints such that  $\theta_1 \wedge \theta_2$  is not satisfiable, then  $tupdif(\bar{t}_1, \bar{t}_2) = \{\bar{t}_1\}$ .
2. Otherwise, let  $\theta_1 \in \bar{t}_1$  and  $\theta_2 \in \bar{t}_2$  be a pair of *distinct* corresponding constraints such that  $\theta_1 \wedge \neg\theta_2$  is satisfiable; if such a pair does not exist,  $tupdif(\bar{t}_1, \bar{t}_2) = \emptyset$ .
3. Let  $\phi_0$  be the constraint such that  $(\theta_1 \wedge \theta_2) \equiv \phi_0$ . Let  $\{\phi_1, \dots, \phi_k\}$  be the smallest set of disjoint constraints such that  $(\theta_1 \wedge \neg\theta_2) \equiv \vee(\phi_1, \dots, \phi_k)$ ;  $k \geq 1$ .
4. Let  $\bar{t}_1^0$  ( $\bar{t}_2^0$ ) be obtained by replacing  $\theta_1$  in  $\bar{t}_1$  ( $\theta_2$  in  $\bar{t}_2$ ) by  $\phi_0$  and putting the result into canonical form. Similarly, let  $\{\bar{t}_1^1, \dots, \bar{t}_1^k\}$  be obtained by replacing  $\theta_1$  in  $\bar{t}_1$  by each  $\phi_i$ ,  $1 \leq i \leq k$ , and putting the result into canonical form.
5. Then  $tupdif(\bar{t}_1, \bar{t}_2) = tupdif(\bar{t}_1^0, \bar{t}_2^0) \cup \{\bar{t}_1^1, \dots, \bar{t}_1^k\}$ .

*Example:* The difference of  $R_1 = \{\bar{t}_1, \bar{t}_2\}$  and  $R_2 = \{\bar{t}_3, \bar{t}_4\}$ , shaded in Figure 5, is computed as follows:

1.  $R_1 - R_2 = (tupdif(\bar{t}_1, \bar{t}_3) - \{\bar{t}_4\}) \cup (tupdif(\bar{t}_2, \bar{t}_3) - \{\bar{t}_4\})$ .
2. By item (2) of Definition,  $tupdif(\bar{t}_1, \bar{t}_3) = \emptyset$ . By item (1) of Definition,  $tupdif(\bar{t}_2, \bar{t}_3) = \{\bar{t}_2\}$ . So,  $R_1 - R_2 = (\emptyset - \{\bar{t}_4\}) \cup (\{\bar{t}_2\} - \{\bar{t}_4\}) = tupdif(\bar{t}_2, \bar{t}_4)$ .
3. By item (2) of Definition,  $\theta_1$  is  $(3 < x < 8)$ , and  $\theta_2$  is  $(7 < x < 15)$ . Then, by item(3),  $\phi_0$  is  $(7 < x < 8)$ , and  $k = 2$ , where  $\phi_1$  is  $(3 < x < 7)$ ,  $\phi_2$  is  $(x = 7)$ .
4. By items (4) and (5) of Definition,  $\bar{t}_2^0$  is  $(7 < x < 8 \wedge 1 < y < 4)$ ,  $\bar{t}_2^1$  is  $(3 < x < 7 \wedge 1 < y < 4)$ ,  $\bar{t}_2^2$  is  $(x = 7 \wedge 1 < y < 4)$ ,  $\bar{t}_4^0$  is  $(7 < x < 8 \wedge 2 < y < 5)$ , and  $tupdif(\bar{t}_2, \bar{t}_4) = tupdif(\bar{t}_2^0, \bar{t}_4^0) \cup \{\bar{t}_2^1, \bar{t}_2^2\}$ .
5. At the end, the following four tuples will be returned:  
 $(3 < x < 7 \wedge 1 < y < 4)$ ,  $(x = 7 \wedge 1 < y < 4)$ ,  $(7 < x < 8 \wedge 1 < y < 2)$ ,  $(7 < x < 8 \wedge y = 2)$ .

□

RENAMING If  $\bar{r}$  is a canonical relation over variables  $X$ ,  $x \in X$ ,  $y \notin X$ , then  $\rho_{y|x}(\bar{r})$  is the renaming in  $\bar{r}$  of  $x$  to  $y$ :

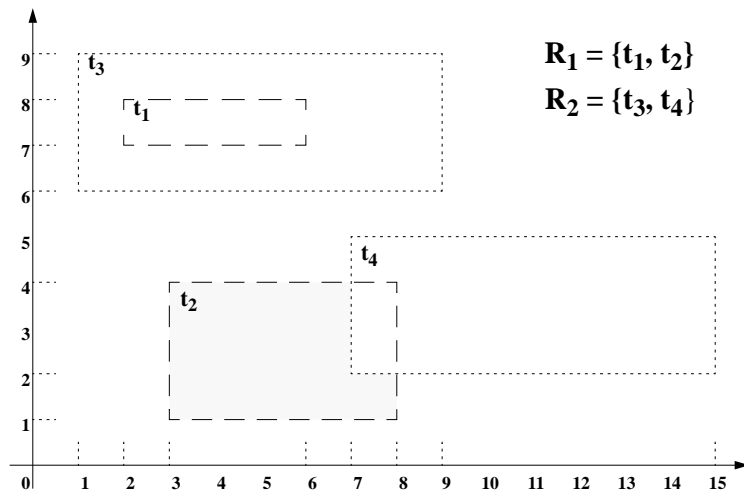


Figure 5. The difference of  $R_1$  and  $R_2$ .

- (1)  $\alpha(\varrho_{y|x}(\bar{r})) = (X - \{x\}) \cup \{y\}$ ,
- (2)  $\varrho_{y|x}(\bar{r}) = \{\bar{t} : \text{for some } \bar{t}' \in \bar{r}, \bar{t} = \bar{t}' \text{ with } x \text{ replaced by } y\}$ .

The first-order CDB calculus for dense order is relational calculus combined with the decidable theory of dense order with constants (Kanellakis et al., 1995). 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 compose to give us the following theorem:

**THEOREM 2** *For every relational algebra QUERY on unrestricted finitely representable relations over  $D^n$ , the constraint algebra **QUERY** that uses **OPs** instead of **OPs** has the property that:*

$$\sigma(\mathbf{QUERY}(\bar{r}_1, \dots, \bar{r}_n)) = \mathbf{QUERY}(\sigma(\bar{r}_1), \dots, \sigma(\bar{r}_n))$$

**Proof:** See Appendix. ■

### 3.4. Remarks on the Syntax of Dense Order Algebra

The tabular format introduced for the dense order constraints above was kept deliberately simple. This syntax was inspired by the r-configurations, discussed below. It is important to note that this syntax can be adapted to richer classes of constraints while preserving its elegance and without incurring any performance penalties.

Two such classes are dense order constraints with “less-than-or-equal” and temporal constraints:

- *Dense order constraints with “less-than-or-equal”* are all formulas of the form  $x\theta y$  or  $x\theta c$ , where  $x, y$  are variables,  $c$  is a constant, and  $\theta$  is one of  $=, \leq, <, >, \geq$ . To incorporate the  $\leq, \geq$  operators into the bounds, we tag each constant with a flag that indicates whether the bound is open or closed. For example, the lower bound for  $x$  in the constraint  $(x > c)$  is  $c^-$ , whereas for the constraint  $(x \geq c)$  it is  $c^+$ . Also, the operators  $\leq, \geq$  need to be added to the list of permissible binary operators in the body of the table.
- The *temporal constraints* consist of the same types of constraints as dense order constraints, except that the two-variable constraints specify the *distance*  $a$  between variables:  $(x < y + a)$ ,  $(x = y + a)$ , where  $a \geq 0$ . Thus, the dense order constraints are a subclass of temporal constraints where  $a$  is always 0. To use the dense order tabular format for the temporal constraints, we add the distance constant into the table as a tag for the binary operators. For example, the above constraints correspond to  $(x <_a y)$ ,  $(x =_a y)$ .

Note that the above two approaches can be combined, so that the operators include  $\leq, \geq$ , and both the bounds and the operators are tagged. With appropriate adjustments to the definitions, all algorithms and results hold for these new tabular formats.

To conclude this section, we would like to compare canonical tuples with a different tuple construct used in Kanellakis et al., 1995, called *r-configurations*. There is a deliberate similarity in the syntax of canonical tuples here and r-configurations, and in fact:

For a given finite subset  $D_\phi$  of  $D$ , an r-configuration is any canonical tuple  $\bar{t}$  over variables  $X$  where (a)  $\forall x \in X, l_x, u_x \in D_\phi$ , and there does not exist  $c \in D_\phi$  such that  $l_x < c < u_x$ ; and (b)  $\forall x, y \in X, \mu_{x,y} \neq (x ? y)$ .

The time of checking whether a canonical tuple is an r-configuration is *not constant* in the size of  $D_\phi$ , whereas it is *constant* for checking whether a tuple is canonical. Therefore an advantage of the canonical tuples here over the r-configurations of Kanellakis et al. is that insertions are more efficient.

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

#### 4. A Monotone Two-Variable Linear Constraint Algebra

In this section, we consider the class of *monotone two-variable linear constraints*; the dense order inequalities are a subclass of monotone constraints. We show that for sets of these constraints, projections can be found in strongly polynomial time.

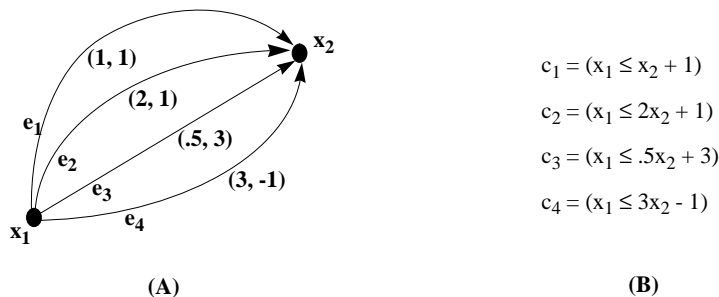


Figure 6. The set of edges (A) corresponds to the set of constraints (B).

Also, we show that for monotone constraints, a globally consistent representation can be found in polynomial time.

Projecting a constraint set  $\bar{t}$  over variables  $X$  onto  $S \subset X$  (see Definition ) is also referred to as *eliminating  $X - S$*  from  $\bar{t}$ . We use a variant of the *Fourier-Motzkin variable elimination method* to perform this operation. First, we restate the variable elimination method as a graph algorithm. Then, we modify the algorithm by pruning certain edges from the graph at each step of variable elimination. We use graph parsing techniques for regular path expressions to show that the performance of the resulting algorithms is strongly polynomial.

#### 4.1. Variable Elimination over Monotone Two-Variable Constraints

*Monotone two-variable linear constraints* are all formulas of the form  $x_1 \theta ax_2 + b$ , where  $\theta \in \{<, =, \leq\}$ ,  $x_1, x_2$  are variables and  $a, b$  are rationals;  $a$  must be non-negative. We will simply say “monotone constraints” when referring to monotone two-variable linear constraints. The dense order inequalities are a special form of monotone constraints. For the rest of this chapter, we assume that  $\theta$  is  $\leq$ , though all results generalize to other binary operations.

A set  $M$  of monotone constraints over a set of variables  $X$  can be represented by a directed labeled multigraph  $\mathcal{M} = (X, E)$ , which we call a *monotone constraint network*. There is a *node* in  $X$  for every variable in  $M$ , and an *edge* in  $E$  for every two-variable constraint in  $M$ ; there are no reflexive edges.

Each edge in  $E$  is labeled with a *monotone function*: there is an edge  $e_i$  from  $x_1$  to  $x_2$  labeled  $f$  if and only if there is a constraint  $c_i = (x_1 \leq f(x_2))$  in  $M$ . We represent  $f$  by the pair of its coefficients  $(a, b)$ , where  $f(x) = ax + b$  (see Figure 6);  $a > 0$ .

Each node  $x_i \in X$  is labeled with the upper and lower *bounds* for the corresponding variable; i.e., if  $l_i \leq x_i \leq h_i$  is in  $M$ , then  $\text{low}(x_i)$  is  $l_i$  and  $\text{high}(x_i)$  is  $h_i$ . The default value for  $\text{low}(x_i)$  is  $-\infty$ ; the default value for  $\text{high}(x_i)$  is  $+\infty$ .

**Definition.** Two networks  $\mathcal{M}_1$  and  $\mathcal{M}_2$  are *equivalent* if the corresponding constraint sets are equivalent.

We define a *composition* operation on the edges of  $\mathcal{M}$  as follows:

**Definition.** Let  $e_1$  be an edge in  $\mathcal{M}$  from  $x_1$  to  $x_2$  labeled  $f_1$ , where  $f_1(x) = ax + b$ . Let  $e_2$  be an edge in  $\mathcal{M}$  from  $x_2$  to  $x_3$  labeled  $f_2$ , where  $f_2(x) = cx + d$ . Then,  $e_1 \otimes e_2$  is an edge from  $x_1$  to  $x_3$  labeled  $f_1 \otimes f_2$ , where:

$$(f_1 \otimes f_2)(x) = f_1(f_2(x)) = (ac)x + (ad + b).$$

The constraint corresponding to  $e_1 \otimes e_2$  is entailed by the conjunction of the constraints corresponding to  $e_1$  and  $e_2$ :  $(x_1 \leq f_1(x_2)) \wedge (x_2 \leq f_2(x_3)) \Rightarrow (x_1 \leq (f_1 \otimes f_2)(x_3))$ . Therefore, adding  $e_1 \otimes e_2$  to  $\mathcal{M}$  results in an equivalent network. Note that  $e_1 \otimes e_2$  cannot be added to  $\mathcal{M}$  when  $x_1 = x_3$ , since  $\mathcal{M}$  may not have reflexive edges. In this case, however,  $(x_1 \leq (f_1 \otimes f_2)(x_3)) \equiv (gx_1 \leq h)$ , where  $g = 1 - ac$  and  $h = ac + d$ ; let us call this constraint  $C$ .

There are four possible cases for  $C$ , depending on the values of  $g, h$ :

- (a)  $g = 0, h < 0$ :  $C$  is unsatisfiable and  $\mathcal{M}$  is unsatisfiable.
- (b)  $g = 0, h > 0$ :  $C$  is trivially true and can be ignored.
- (c)  $g > 0$ :  $C \equiv (x_1 \leq h/g)$ ;  $\text{high}(x_1)$  can be updated to  $h/g$ .
- (d)  $g < 0$ :  $C \equiv (x_1 \geq h/g)$ ;  $\text{low}(x_1)$  can be updated to  $h/g$ .

The updates described above correspond to adding entailed constraints to the network, so network equivalence is preserved.

We next show how to use the network representation of  $M$  to implement the Fourier-Motzkin Elimination algorithm for computing the projection of  $M$  onto some subset  $S$  of  $X$ , where  $|S| \geq 1$ . We begin with an informal description of the original algorithm; see Schrijver, 1986 for a formal discussion.

**Fourier-Motzkin Elimination** The variables in  $X - S$  are eliminated one by one. The set of constraints at the beginning of step  $i$  is denoted by  $M_i$ , where  $M_1 = M$ . At step  $i$ , the  $i$ th member of  $X - S$  is eliminated; call it  $x$ :

1. All constraints in which  $x$  participates are partitioned into two sets,  $H$  and  $L$ .  $H$  contains the constraints of the form  $x \leq h$ , and  $L$  contains those of the form  $x \geq l$ , where  $h, l$  are either constants or monotone functions.
2. A new set of constraints  $LH$  is created as follows: for all  $l \in L$  and  $h \in H$ , a new inequality  $l \leq h$  is added to  $LH$ . The size of  $LH$  is  $|L| \cdot |H|$ .
3.  $M_{i+1} = (M_i \cup LH) - (L \cup H)$ .

If  $S = k$ , then  $M_{k+1}$  is the desired set of constraints.

When implemented via monotone networks, the same algorithm is as follows:

**Algorithm I.**  $M$  is represented by a network  $\mathcal{M}$ , where each node  $x_i$  is tagged with the constant bounds  $(l_i, h_i)$  and each edge  $e_i$  is labeled with  $f_i$ . All nodes corresponding to variables  $X - S$  are eliminated from  $\mathcal{M}$  one by one, while updating bounds and adding new composed edges at each step. Here is the procedure for eliminating variable  $x_i$ :

For each pair of edges  $(e_j, e_k)$ , where  $e_j$  ends at  $x_i$  and  $e_k$  starts at  $x_i$ :

if the head of  $e_j$  is distinct from the tail of  $e_k$ , add  $e_j \otimes e_k$  to  $\mathcal{M}$ ;  
 otherwise, use  $e_j \otimes e_k$  to update the bounds on the head of  $e_j$

1. For each edge  $e$  entering  $x_i$  (labeled  $f$ ), where  $x_j$  is the head of  $e$ :  
 if  $f(h_i) \leq h_j$ , replace  $h_j$  by  $f(h_i)$ .
2. For each edge  $e$  leaving  $x_i$  (labeled  $f$ ), where  $x_j$  is the tail of  $e$ :  
 if  $l_i \geq f(l_j)$ , replace  $l_j$  by  $f^{-1}(l_i)$
3. Remove from  $\mathcal{M}$  the node  $x_i$  and all edges adjacent on  $x_i$ .  $\square$

The edges that are in  $\mathcal{M}$  at the end of variable elimination correspond to all *simple* paths  $p$  in the original network such that:

$p$  starts and ends at nodes from  $X - S$ ; all other nodes in  $p$  are in  $S$ .

Note that the number of such paths can be exponential in the size of  $\mathcal{M}$ . We will show that we can prune most of the edges created at each step of the node elimination algorithm, while preserving network equivalence.

This is similar to the approach of Hochbaum and Naor, 1994, where the Fourier-Motzkin algorithm is modified to solve the feasibility problem efficiently. There, the *equisatisfiability* of the constraint sets is preserved at each step; this condition is necessary to guarantee correctness of the feasibility algorithm. However, *equivalence* of the sets is not preserved, making the optimizations of Hochbaum and Naor inapplicable to the projection problem.

## 4.2. Modifying the Variable Elimination Algorithm

In this section, we tag each edge of the monotone network with intervals, called a *domain* and a *range*. These intervals, computed at the beginning of each step of variable elimination, are used to reject most of the new edges produced during the step. This modification to the variable elimination algorithm allows us to achieve strongly polynomial performance.

Let us consider, for two arbitrary nodes  $x_1$  and  $x_2$  in  $\mathcal{M}$ , the set of all edges (*cluster*)  $\mathcal{E}$  from  $x_1$  to  $x_2$ , as in Figure 6;  $|\mathcal{E}| = k > 0$ . The labels of the edges are linear functions  $\{f_1, \dots, f_k\}$ . We assume they are distinct, so as to ensure that the corresponding constraints are not equivalent. (Note that the  $f$ 's need not be independent).

Given an edge  $e_i$  labeled  $f_i$ , we define the *domain of  $e_i$*  to be the set of values for  $x_2$  over which  $f_i$  is the minimal function:

**Definition.** Let  $e_i$  be an arbitrary edge in  $\mathcal{E}$ , and  $f_i$  be its label. A value  $v$  is in the *domain* of  $e_i$  if and only if for every edge  $e' \in \mathcal{E}$  (with label  $f'$ ),  $f(v) \leq f'(v)$ . If

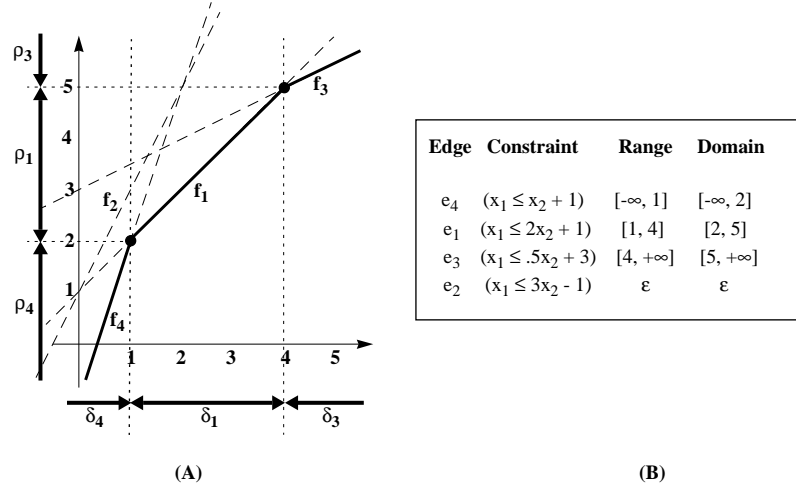


Figure 7. Domains and ranges: graphical (A) and tabular (B) representations.

there are no values on which  $f_i$  is minimal, the domain of  $e_i$  is empty (denoted by  $\epsilon$ ). We say that a domain is *trivial* if it is either empty or its upper bound equals its lower bound (i.e., it consists of one value); otherwise, it is *non-trivial*.

**LEMMA 4** *The non-trivial domain for the edges in  $\mathcal{E}$  form a set of consecutive intervals covering the whole  $x_2$ -axis.*

**Proof:** Due to the linearity of  $f_i$ 's, any arbitrary pair of distinct functions  $f_1$  and  $f_2$  can intersect in at most one point. This means that all the domain are disjoint, except at the endpoints. Due to the monotonicity of  $f_i$ 's, every value  $v$  belongs to at least one domain. ■

We also associate a *range* with each edge:

**Definition.** Let  $e_i$  be an arbitrary edge in  $\mathcal{E}$ , and  $f_i$  be its label. A value  $v$  is in the *range* of  $e_i$  if and only if  $f_i^{-1}(v)$  is in the domain of  $e_i$ .

It is easy to see that the ranges form a set of consecutive intervals covering the whole  $x_1$ -axis, and that  $\text{range}(e_1) < \text{range}(e_2)$  if and only if  $\text{domain}(e_1) < \text{domain}(e_2)$ , where ' $>$ ' designates a natural ordering of intervals (Figure 7).

**PROPOSITION 1** *Removing any edge with a trivial domain from the network at any point in Algorithm I results in an equivalent network.*

**Proof:** When we have an edge  $e$  in  $\mathcal{M}$  whose domain is trivial,  $e$  corresponds to a redundant constraint, and  $\mathcal{M}$  is equivalent to  $\mathcal{M} - e$ . ■

Let us now consider what happens when we eliminate some node  $x_2$ . Then, for all pairs of nodes  $x_1$  and  $x_3$ , we compose each edge from  $x_1$  to  $x_2$  with each edge from  $x_2$  to  $x_3$ , and add the resulting edge to  $\mathcal{M}$ . Let  $\mathcal{E}_1$  be a cluster of edges from  $x_1$  to  $x_2$ , with  $|\mathcal{E}_1| = j$ , and  $\mathcal{E}_2$  be a cluster of edges from  $x_2$  to  $x_3$ , with  $|\mathcal{E}_2| = k$ . We now show that at most  $j + k - 1$  of the  $jk$  edges formed by pairwise composition are not redundant.

**LEMMA 5** *Let  $e_1 = (x_1, x_2, f_1)$  and  $e_2 = (x_2, x_3, f_2)$  be two arbitrary adjacent edges; let  $v$  be some real number, and let  $u = f_2(v)$ .  $v$  belongs to the domain of  $e_1 \otimes e_2$  only if  $u$  belongs to the intersection of the domain of  $e_1$  and the range of  $e_2$ .*

**Proof:** **Case 1:**  $u$  is not in the domain of  $e_1$ . From the definition of domain, there exists an edge  $e_3 = (x_1, x_2, f_3)$  such that  $f_3(u) < f_1(u)$ . Then,  $(f_1 \otimes f_2)(v) = f_1(f_2(v)) = f_1(u) > f_3(u) = f_3(f_2(v)) = (f_3 \otimes f_2)(v)$ . So,  $v$  is not in the domain of  $(e_1 \otimes e_2)$ . **Case 2:**  $u$  is not in the range of  $e_2$ . From the definition of range, we know that  $v$  is not in the domain of  $e_2$ , and there exists an edge  $e_4 = (x_2, x_3, f_4)$  such that  $f_4(v) < f_2(v)$ . Then,  $(f_1 \otimes f_2)(v) = f_1(f_2(v)) > f_1(f_4(v)) = (f_1 \otimes f_4)(v)$ . So,  $v$  is not in the domain of  $(e_1 \otimes e_2)$ . ■

**COROLLARY 1** *Let  $e_1 = (x_1, x_2, f_1)$  and  $e_2 = (x_2, x_3, f_2)$  be two arbitrary adjacent edges.  $e_1 \otimes e_2$  is not redundant if and only if the intersection of the domain of  $e_1$  and the range of  $e_2$  is non-trivial.*

We are now ready to assert that most of the edges can be pruned:

**THEOREM 3** *Let  $\mathcal{E}_1$  be a cluster of edges from  $x_1$  to  $x_2$ , with  $|\mathcal{E}_1| = j$ ; let  $\mathcal{E}_2$  be a cluster of edges from  $x_2$  to  $x_3$ , with  $|\mathcal{E}_2| = k$ . Less than  $j + k$  of the  $jk$  edges formed by pairwise composition of  $\mathcal{E}_1$  and  $\mathcal{E}_2$  are not redundant.*

**Proof:** In this proof, *boundary point* denotes a startpoint or an endpoint of an interval on the real axis, excluding infinity. Note that a set of  $n$  non-trivial disjoint intervals covering the real axis has  $n - 1$  boundary points; if some intervals are trivial, there are even fewer points.

Let  $\mathcal{I}_1$  ( $\mathcal{I}_2$ ) be the set of intervals corresponding to the domains (ranges) of the edges in  $\mathcal{E}_1$  ( $\mathcal{E}_2$ ); let  $S_1$  ( $S_2$ ) be the set of its boundary points. Let  $\mathcal{I}_3$  be the pairwise non-trivial intersection of  $\mathcal{I}_1$  and  $\mathcal{I}_2$ :

$$\mathcal{I}_3 = \{I \cap I' : I \in \mathcal{I}_1, I' \in \mathcal{I}_2, |I \cap I'| > 1\}.$$

$\mathcal{I}_3$  is a set of disjoint non-trivial intervals covering the real axis; let  $S_3$  be the set of its boundary points.

Given any two intersecting intervals  $I = (v_1, v_2)$  and  $I' = (u_1, u_2)$ , the boundary points of  $I \cap I'$  will be among  $\{v_1, v_2, u_1, u_2\}$ . Therefore,  $S_3 \subseteq S_1 \cup S_2$ , and  $|S_3| \leq |S_1| + |S_2| \leq (j - 1) + (k - 1)$ . This means that  $\mathcal{I}_3$  can have at most  $j + k - 1$  intervals; combined with Corollary 1, this completes the proof. ■

We now describe the final algorithm for computing the projection of some monotone constraint set  $M$  onto some subset  $S$  of variables,  $|S| \geq 1$ . Note its similarity to the Algorithm I.

**Algorithm II.** First, represent  $M$  by a network  $\mathcal{M}$ ; tag each edge  $e_i$  with the corresponding label  $f_i$ , and each node by the variable name  $x_i$  and the bounds  $(l_i, h_i)$  on the variable. The computation proceeds by eliminating, one by one, all variables that are in  $X - S$ , while updating bounds and removing redundant constraints at each step. If at any point we obtain a node whose lower bound exceeds its upper bound,  $M$  is known to be unsatisfiable.

Eliminating variable  $x_i$ :

1. For each variable  $x_j$  distinct from  $x_i$ :
  - compute the domains and ranges for the clusters  $\mathcal{E}_{(x_i, x_j)}$  and  $\mathcal{E}_{(x_j, x_i)}$
2. For each pair of edges  $(e_j, e_k)$ , where  $e_j$  ends at  $x_i$  and  $e_k$  starts at  $x_i$ , and the intersection of  $\text{domain}(e_j)$  and  $\text{range}(e_k)$  is non-trivial:
  - if the head of  $e_j$  is distinct from the tail of  $e_k$ , add  $e_j \otimes e_k$  to  $\mathcal{M}$ ;
  - otherwise, use  $e_j \otimes e_k$  to update the bounds on the head of  $e_j$
3. For each edge  $e$  entering  $x_i$  (labeled  $f$ ), where  $x_j$  is the head of  $e$ :
  - if  $f(h_i) \leq h_j$ , replace  $h_j$  by  $f(h_i)$ .
4. For each edge  $e$  leaving  $x_i$  (labeled  $f$ ), where  $x_j$  is the tail of  $e$ :
  - if  $l_i \geq f(l_j)$ , replace  $l_j$  by  $f^{-1}(l_i)$
5. Remove from  $\mathcal{M}$  the node  $x_i$  and all edges adjacent on  $x_i$ .  $\square$

**THEOREM 4** *Algorithm II produces a network equivalent to that of Algorithm I.*

**Proof:** Except for the domain and range check, the two algorithms are the same. This, combined with Proposition 1 and Corollary 1, gives us the theorem.  $\blacksquare$

### 4.3. Monotone Projection Complexity

Algorithm II finds the projection of  $M$  onto some subset  $S$  of its variables. As mentioned earlier, it is equivalent to finding some set of simple *non-trivial* paths in  $\mathcal{M}$ , where  $e_1 e_2 \dots e_k$  is non-trivial iff the domain of  $e_1 \otimes \dots \otimes e_k$  is non-trivial. In this subsection, we analyze the complexity of the algorithm.

We start with an introduction to path expressions (Tarjan, 1981) and show a connection between the size of the path expression and the number of simple non-trivial paths in the corresponding set. We then show that Algorithm II is strongly polynomial.

$\mathcal{M} = (X, E)$  is a directed multigraph; any path in  $\mathcal{M}$  can be regarded as a string over  $E$ . A *path expression*  $P$  of type  $(v, w)$  is a regular expression over  $E$  (see Aho

et al., 1974 for an introduction to regular expressions), where every string in the regular language defined by  $P$  is a path from  $v$  to  $w$ .  $P$  is represented by a tree  $T_P$ . Each node  $n$  in  $T_P$  is associated with a subexpression  $P(n)$  of  $P$  defining a regular language  $\mathcal{L}(n)$ ; the expression at the root is  $P$ .

There are four types of nodes:

1.  $n = e = (x_i, x_j)$ , with no children:  
 $P(n) = e$ , of type  $(x_i, x_j)$ ;
2.  $n = \cdot$ , with children  $n_1$  of type  $(x_i, x_j)$  and  $n_2$  of type  $(x_j, x_k)$  for some  $i, j, k$ :  
 $P(n) = P(n_1) \cdot P(n_2)$ , of type  $(x_i, x_k)$ ;
3.  $n = \cup$ , with children  $n_1$  and  $n_2$ , both of type  $(x_i, x_j)$ :  
 $P(n) = P(n_1) \cup P(n_2)$ , of type  $(x_i, x_j)$ ;
4.  $n = *$ , with child  $n_1$  of type  $(x_i, x_j)$ :  
 $P(n) = P(n_1)^*$ , of type  $(x_i, x_j)$ .

The *size* of  $P$  is the number of nodes in  $T_P$ ; the following lemma establishes a connection between sizes of path expressions of type  $(u, v)$  and the number of simple non-trivial paths from  $u$  to  $v$ :

**LEMMA 6** *Let  $\mathcal{P}(u, v)$  be the path expression representing the set of all paths in  $\mathcal{M}$  from  $u$  to  $v$ . Then, the number of simple non-trivial paths from  $u$  to  $v$  is not greater than the size of  $\mathcal{P}(u, v)$ .*

**Proof:** At each node  $n$  of  $T_{\mathcal{P}(u, v)}$ , the size of  $P(n)$  is defined recursively:

$$\text{size}(P(n)) = 1 + \Sigma(\text{size}(P(n_i)) : n_i \text{ is a child of } n).$$

Let  $\mathcal{N}(n)$  be the subset of  $\mathcal{L}(n)$  which consists only of simple non-trivial paths. We claim that for all  $n$ ,  $|\mathcal{N}(n)| \leq \text{size}(P(n))$ . This is proven by structural induction on  $T_{\mathcal{P}(u, v)}$ , by the use of Theorem 3. At the root node  $r$ ,  $\text{size}(r)$  is the size of  $\mathcal{P}(u, v)$ , and  $\mathcal{N}(r)$  is the set of all simple non-trivial paths from  $u$  to  $v$ . ■

Given two nodes  $u, v$  in  $\mathcal{M}$ , the problem of finding the path expression representing the set of all paths from  $u$  to  $v$  is known as the *path expression problem*. Solving this problem is equivalent to deriving a regular expression for a given finite state automaton; the size of the resulting expression is polynomial in the size of  $\mathcal{M}$  (see Aho et al., 1974). It follows from Lemma 6 that the number of simple non-trivial paths from  $u$  to  $v$  is polynomial in the size of  $\mathcal{M}$ .

We are now ready to prove the following theorem:

**THEOREM 5** *The performance of Algorithm II is strongly polynomial in the size of  $M$ .*

**Proof:** Algorithm II does not depend on the values of the coefficients in  $M$ . It consists of  $O(k)$  steps, where  $k$  is the number of variables:  $X = \{x_1, \dots, x_k\}$ . Let the constant  $Q$  be the maximal number of simple non-trivial paths from  $x_i$  to  $x_j$  in  $\mathcal{M}$ , for all ordered pairs  $(x_i, x_j)$ . We consider one step of the algorithm, eliminating some variable  $x$  from  $M$ :

1. For a cluster of  $m$  edges, the time to compute the domains and ranges of the edges is  $O(m^2)$ . There are  $O(k)$  clusters in step (1) of Algorithm II, and each one has  $O(Q)$  edges, so the total time is  $O(kQ^2)$ .
2. There are  $O(kQ)$  edges entering or leaving  $x$ , forming  $O((kQ)^2)$  pairs. For each such edge or pair of edges, steps (2)-(5) can be done in constant time.

Therefore, each step of the algorithm takes time  $O(kQ^2)$ . There are  $O(k)$  such steps, for a total running time of  $O(k^3Q^2)$ , which is polynomial in the size of  $\mathcal{M}$ . ■

Note that the main result here is *strong polynomiality* of projection for monotone constraints; no claim is being made that the time bounds for the algorithm are tight.

#### 4.4. Globally Consistent Constraint Sets

We have shown in the previous section that computing projections of monotone constraint sets is polynomial with respect to the size of both the query and the data, where the polynomials do not depend on the coefficient sizes, i.e., strongly polynomial. In this section, we show that it is possible, in strongly polynomial time, to obtain a *globally consistent* (Definition ) monotone constraint set equivalent to a given one:

If we are given a constraint set  $E$  over variables  $X$ , we want to somehow preprocess  $E$  so that for the resulting set  $E'$ , the projection onto any subset of  $X$  can be computed by simply collecting the constraints involving these variables. It is trivial to construct  $E'$  by taking the union of the projections of  $E$  onto all subsets of  $X$ ; however, this is an exponential-time procedure. We now show that for two-variable constraints (where each constraint involves *at most two* variables), it suffices to project the constraint set onto the subsets of size 1 and 2.

Though  $E[S]$  is fixed for a given  $E$  and  $S$  (Definition ), there may be different ways to represent  $\pi_S(E)$  (Definition ). We will abuse the notation and assume some arbitrary fixed representation for  $\pi_S(E)$ , perhaps by fixing the algorithm that computes it. It is easy to see that whenever  $E_1 \equiv E_2$  (i.e., their solution sets are equal),  $\pi_S(E_1) \equiv \pi_S(E_2)$  for all  $S$ . It is also easy to see that for all  $E$  and all  $S$ , any solution to  $\pi_S(E)$  satisfies  $E[S]$ : we know that some extension of this solution will satisfy  $E$ , and  $E[S]$  is a subset of  $E$ .

**LEMMA 7** *Let  $E$  be a set of two-variable constraints, and  $E'$  be constructed from  $E$  as follows:*

$$E' = \cup(\pi_S(E)), \text{ for all subsets } S \text{ of } X \text{ where } |S| \leq 2.$$

*Then, the following is true:*

1.  $E' \equiv E$  (i.e., their solution sets are equal).

2. For all subsets  $S$  of  $X$ ,  $\pi_S(E') \equiv E'[S]$ .

**Proof:** First, we show that any assignment satisfying  $E$  satisfies  $E'$ . Let  $C$  be an arbitrary constraint in  $E'$ ; w.l.o.g., let  $C$  involve the variables  $x_1$  and  $x_2$ . By construction, it must be the case that  $C \in \pi_{x_1x_2}(E)$ . Let  $p$  be an arbitrary assignment satisfying  $E$ ; by definition, the restriction of  $p$  to  $\{x_1, x_2\}$  satisfies  $\pi_{x_1x_2}(E)$ . Therefore,  $p$  satisfies  $C$ .

Next, we show that any assignment that does not satisfy  $E$  also violates  $E'$ . Let  $p$  be an arbitrary assignment that does not satisfy  $E$ ; then, there exists a specific constraint  $C \in E$  such that  $p$  satisfies  $\neg C$ . W.l.o.g., let  $C$  involve 2 variables,  $x_1$  and  $x_2$ ; by definition,  $p$  does not satisfy  $\pi_{x_1x_2}(E)$ . Therefore,  $p$  cannot satisfy  $E'$ .

We know from earlier in the subsection that any assignment satisfying  $\pi_S(E')$  satisfies  $E'[S]$ ; it remains to prove the other direction. Let  $S$  be an arbitrary subset of  $X$ . By construction,  $E'[S] = \cup(\pi_{S_i}(E))$ , for all subsets  $S_i$  of  $S$  where  $|S_i| \leq 2$ . Let  $p$  be any assignment satisfying  $E'[S]$ ; then,  $p$  satisfies  $\pi_{S_i}(E)$  for all  $S_i \subseteq S$  with  $|S_i| \leq 2$ . Let  $C$  be a constraint in  $E$  that does not involve any variables outside of  $S$ ; w.l.o.g., let  $C$  involve 2 variables,  $x_1$  and  $x_2$ . We know that  $p$  satisfies  $\pi_{x_1x_2}(E)$ ; therefore,  $p$  satisfies  $C$ . Since we've just shown that  $p$  satisfies all constraints in  $E$  that involve only  $S$ , it follows that  $p$  satisfies  $\pi_S(E)$ . And since  $E' \equiv E$ , we can conclude that  $p$  satisfies  $\pi_S(E')$ . ■

We can now assert the following:

**THEOREM 6** For any monotone two-variable linear constraint set, a globally consistent representation can be found in strongly polynomial time.

**Proof:** Given a set  $X$  of  $k$  variables, there are  $O(k^2)$  subsets  $S \in X$  where  $|S| \leq 2$ . By Lemma 5, we can find the projection of the constraint set onto each  $S$  in strongly polynomial time. By Lemma 7, the union of the resulting sets is globally consistent. ■

## 5. Lazy Evaluation of (Non)Linear Constraint Algebras

In this section, we discuss an alternate way of implementing constraint query algebras, in particular the *positive* fragment consisting of the operations *project*, *select*, *natural-join*, *union*, and *rename*; this is similar to the approach of Brodsky et al., 1993. We suggest that for (general) linear constraints, as well as for nonlinear constraints, it is worthwhile to leverage the performance on a tuple representation that contains existentially quantified (but not eliminated) variables. We call such variables *extraneous*, as opposed to *essential*.

The approach is akin to the CLP approach to constraint stores, where many variables will not participate in the output (Van Hentenryck, 1989). What matters is that the constraints are *satisfiable*, i.e., for some assignment of values to the extraneous variables, the essential variables will form a tuple in the constraint store. For

linear constraints, satisfiability can be guaranteed using linear programming. For nonlinear constraints, satisfiability can be implemented efficiently using numerical methods (Van Hentenryck et al., 1995).

**Lazy Evaluation:** The approach that we propose involves delaying the symbolic processing (quantifier/variable elimination) whenever possible. We call this approach *lazy evaluation*, borrowing this terminology from functional programming. The resulting representation of generalized relations, containing unevaluated existentially quantified variables, is called the *lazy representation*.

The argument in favor of lazy evaluation hinges on the complexity gap between performing satisfiability checks and performing variable elimination for sets of linear constraints. The former is polynomial whereas the latter is exponential (Schrijver, 1986). What’s worse, variable elimination can be exponential just because of the size of the result (Yannakakis, 1988) if a non-lazy, or “eager”, representation is used.

Therefore, we propose to use the lazy methodology not just for the intermediate representation during the evaluation of positive algebraic operations, but also for the internal representation when storing generalized relations. Fortunately, storing generalized relation with unevaluated variables still permits us the use of indexing methodologies for constraints (Section 6). This is due to the fact that computing projections onto a single variable  $x$  is equivalent to the *optimization problem* for a set of linear constraints, where we seek the minimum and the maximum allowable values for  $x$ . These optimization problems have polynomial complexity (Schrijver, 1986), giving us an interval over  $x$  that is used in the indexing scheme.

The extraneous variables in the lazy representation will of course need to be removed (via projection) when the relation is output to the user. This is done either by the query processor or by some other system that processes the query output, such a graphical display mechanism. So, lazy evaluation can also be viewed as *delayed projection*, and approached as an optimization step to constraint query algebras.

**A Positive CQA:** Let  $R_1$  be a generalized relation over  $(x_1, \dots, x_k)$  with extraneous variables  $(x_{k+1}, \dots, x_n)$ ; let  $R_2$  be another such generalized relation. The extraneous variables in the two relations are different from the essential variables and local to the relations; this can be ensured through local naming of such variables. The two relations can share essential variables; for Union they have to, for Natural-Join the degree of sharing determines the special cases of Cross-Product and Intersection.

We assume that for any tuple  $t$  of  $R_1, R_2$ ,  $t$  is not empty; this can be accomplished using satisfiability checks. We now describe the positive algebraic operations on  $R_1, R_2$ .

- **Projection:**  $R = \{\pi_{(x_1, \dots, x_i)} t_1 : t_1 \in R_1\}$   
Mark  $(x_{i+1}, \dots, x_k)$  as extraneous, so  $R$  is a relation over  $(x_1, \dots, x_i)$  with extraneous variables  $(x_{i+1}, \dots, x_n)$ . The constraints in  $t_1$  are not changed.

- **Selection:**  $R = \{\zeta_F(t_1) : t_1 \in R_1\}$   
We assume that  $F$  is quantifier-free and all its variables are among  $(x_1, \dots, x_n)$ . We perform a satisfiability check on the conjunction  $F \wedge F(t_1)$ , where  $F(t_1)$  is the formula corresponding to  $t_1$ ; if it is not satisfiable, the *selection* does not produce anything. Otherwise, the constraints in  $F$  are added to the constraints in  $t_1$ ; the sets of essential and extraneous variables remain the same.
- **Natural Join:**  $R = \{t_1 \bowtie t_2 : t_1 \in R_1, t_2 \in R_2\}$ .  
We perform a satisfiability check on  $F(t_1) \wedge F(t_2)$ ; if it is not satisfiable, the *join* does not produce anything. Otherwise, the constraints from  $t_1$  and  $t_2$  are combined. The essential variables are the intersection of the two sets of essential variables. All other variables from either  $R_1$  or  $R_2$  are marked as extraneous.
- **Union:**  $R = \{t : t \in R_1 \vee t \in R_2\}$ , where  $R_2$  has the same essential variables as  $R_1$ .  
The essential variables remain the same; the extraneous variables are the union of the two sets of extraneous variables. The constraints in  $t$  remain unchanged.
- **Rename:** As in the Relational Algebra case.

Of course, delaying projections during the positive fragment of Constraint Algebra forces us to pay for it (in the form of variable elimination) during the operations of *difference* and *duplicate elimination*, and when final output is requested. However, there are good arguments why this is not a significant problem:

- difference is an uncommon operation;
- duplicate elimination is usually performed lazily even in the Relational Algebra;
- the final output is often quite small in practice.

## 6. Discussion: I/O Efficient Algebraic Operations

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 or strong polynomiality of operations, whose satisfaction greatly contributes to relational technology.

B-trees (and their variants  $B^+$ -trees) are examples of important data structures for implementing relational databases (see Bayer and McCreight, 1972). 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) *one-dimensional searching on relational attribute  $x$* , with the corresponding performance bounds using a  $B^+$ -tree on  $x$ :

- 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*.
- 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 attributes*  $x_1, \dots, x_k$  generalizes one 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 – for a survey, see Samet, 1990.

For generalized databases we can define an analogous problem of *one-dimensional searching on generalized relational attribute*  $x$  using the operations:

- Find a generalized relation that represents all tuples of the input generalized relation such that their  $x$  attribute satisfies  $(a_1 \leq x \leq a_2)$ .
- Insert or delete a given generalized tuple.

If  $(a_1 \leq x \leq a_2)$  is a constraint of our CDB query, then there is a trivial, but inefficient, solution to the problem of one-dimensional searching on generalized relational attribute  $x$ . One can add a 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 (i.e., without a range check for the input tuple, or a satisfiability check for the output tuple). 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 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:

- A *generalized one-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 relation that represents all tuples of the input generalized relation whose  $x$  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 is performed by computing its projection and inserting or deleting intervals from a set of intervals.

The use of generalized one-dimensional indexes reduces redundancy of representation and transforms one-dimensional searching on generalized relational attribute

$x$  into the problem of *dynamic interval management*. This is a well-known problem with many elegant solutions from computational geometry (Preparata and Shamos, 1985). 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. For the first optimal static solution see Kanellakis et al., 1993 and for an optimal dynamic one see Agre and Vitter, 1995.

Dynamic interval management is a special case of two-dimensional searching in relational databases, so it can be implemented with good average I/O efficiency using existing general-purpose spatial data structures. Worst-case I/O tradeoffs for this and other related two-dimensional range searching problems are examined in Ramaswamy and Subramanian, 1994.

In this discussion we have focused on the question of efficient indexing. In addition to the search for new indexing methods, it would be interesting to study in more detail how algebraic operations interact with the existing technology. Another important question is: how do various *optimization methods* (such as selection propagation, join ordering and other algebraic transformations, magic sets, etc.) combine with constraint query algebras? For some recent research in this direction we refer to Ramakrishnan, 1988; Mumick et al., 1990; Srivastava and Ramakrishnan, 1992; Stuckey and Sudarshan, 1994.

Extensions to the relational model, involving partial information and complex object data types, can also be applied to CDBs. For example, null values can be introduced with existential quantification in constraints. Finite complex objects (Grumbach and Su, 1995) and aggregation over finite sets can co-exist with infinite relations (Chomicki et al., 1996). Are there efficient ways of adding these extensions to constraint query algebras?

## Acknowledgments

Paris C. Kanellakis, one of the authors of this paper, died in a tragic accident shortly after the completion of the first draft. We thank all the reviewers, whose comments were invaluable in helping us complete the work. We also thank Raghu Ramakrishnan for making a last-minute review of the final draft.

Research supported by ONR Contracts N00014-94-1-1153 and N00014-91-J-4052, ARPA Order 8225, and by NSF Grant IRI-9509933.

## References

- S. Abiteboul, R. Hull, V. Vianu. *Foundations of Databases*. Addison-Wesley, 1994.
- L. Agre, J. S. Vitter. Optimal Interval Management in External Memory. Manuscript, November 1995.
- A.V. Aho, J.E. Hopcroft, J.D. Ullman. *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Co., 1974.
- B. Aspvall, Y. Shiloach. A polynomial time algorithm for solving systems of linear inequalities with two variables per inequality. *SIAM J. Comput.*, 9:4:827-845, 1980.

- 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.
- R. Bayer, E. McCreight. Organization of Large Ordered Indexes. *Acta Informatica*, 1:173–189, 1972.
- A.H. Borning. The Programming Language Aspects of ThingLab, A Constraint-Oriented Simulation Laboratory. *ACM TOPLAS* 3:4:353–387, 1981.
- A. Brodsky, J. Jaffar, M.J. Maher. Toward Practical Constraint Databases. *Proc. 19th VLDB*, 322–331, 1993.
- A. K. Chandra and D. Harel. Structure and Complexity of Relational Queries. *J. Comp. System Sci.*, 25:99–128, 1982.
- J. Chomicki, D. Goldin, G. Kuper. Variable Independence and Aggregation Closure. To appear in *Proc. 15th ACM PODS*, June 1996.
- J. Chomicki, T. Imielinski. Relational Specifications of Infinite Query Answers. *Proc. ACM SIGMOD*, 174–183, 1989.
- E.F. Codd. A Relational Model for Large Shared Data Banks. *CACM*, 13:6:377–387, 1970.
- A. Colmerauer. An Introduction to Prolog III. *CACM*, 33:7:69–90, 1990.
- R. Dechter. From Local to Global Consistency. *Artificial Intelligence*, 55:87–107, 1992.
- R. Dechter, I. Meiri, J. Pearl. Temporal Constraint Networks. *Artificial Intelligence*, 49:61–95, 1991.
- 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.
- J. Ferrante, J.R. Geiser. An Efficient Decision Procedure for the Theory of Rational Order. *Theoretical Computer Science*, 4:227–233, 1977.
- E. Freuder. Synthesizing Constraint Expressions. *CACM*, 21:11, 1978.
- E. Freuder. A sufficient condition for backtrack-free search. *CACM*, 29:1, 1982.
- S. Grumbach, J. Su. Dense Order Constraint Databases. *Proc. 14th ACM PODS*, May 1995, pp. 66–77.
- S. Grumbach, J. Su, C. Tollu. Linear Constraint Query Languages: Expressive Power and Complexity. *Proc. Workshop on Finite Model Theory*, Indiana, Fall 1994.
- D. S. Hochbaum, J. Naor. Simple and Fast Algorithms for Linear and Integer Programs with two Variables per Inequality. *SIAM J. Comput.*, 23:6:1179–1192, 1994.
- J. Jaffar, J.L. Lassez. Constraint Logic Programming. *Proc. 14th ACM POPL*, 111–119, 1987.
- P.C. Kanellakis. Elements of Relational Database Theory. *Handbook of Theoretical Computer Science*, Vol. B, chapter 17, (J. van Leeuwen editor), North-Holland, 1990.
- P.C. Kanellakis and D.Q. Goldin. Constraint Programming and Database Query Languages. *Symposium on Theoretical Aspects of Computer Software*, LNCS 789, pp. 96–120, Sendai Japan, April 1994.
- P. C. Kanellakis, G. M. Kuper, P. Z. Revesz. Constraint Query Languages. *JCSS*, vol. 51, no.1, pp. 26–52, August 1995.
- 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. To appear in *JCSS*.
- A. Klug. On Conjunctive Queries Containing Inequalities. *JACM*, 35:1:146–160, 1988.
- M. Koubarakis. *Foundations of Temporal Constraint Databases*. PhD Thesis. Nat. Tech. Univ. of Athens and Imperial College. 1993.
- A.K. Mackworth. Consistency in Networks of Relations. *AI*, 8:1, 1977.
- U. Montanari. Networks of Constraints: Fundamental Properties and Application to Picture Processing. *Information Science*, 7, 1974.
- I. S. Mumick, S. J. Finkelstein, H. Pirahesh, R. Ramakrishnan. Magic Conditions. *Proc. 9th ACM PODS*, 314–330, 1990.
- G. Nelson. An  $n^{\log n}$  algorithm for the two-variable-per-constraint linear programming satisfiability problem. Technical Report AIM-319, Stanford University, 1978.
- J. Paredaens, J. van den Bussche, D. Van Gucht. First-order Queries on Finite Structures over the Reals. *Proc. IEEE LICS*, 1995.
- F.P. Preparata, M.I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

- R. Ramakrishnan. Magic Templates: A Spellbinding Approach to Logic Programs. *Proc. 5th International Conference on Logic Programming*, 141–159, 1988.
- S. Ramaswamy, S. Subramanian. Path Caching: A Technique for Optimal External Searching. *Proc. 13th ACM PODS*, 14–25, 1994.
- 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.
- H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading MA, 1990.
- A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley and Sons, 1986.
- D. Srivastava, R. Ramakrishnan. Pushing Constraint Selections. *Proc. 11th ACM PODS*, 301–316, 1992.
- G.L. Steele. The Definition and Implementation of a Computer Programming Language Based on Constraints. Ph.D. thesis, MIT, AI-TR 595, 1980.
- P.J. Stuckey, S. Sudarshan. Compiling Query Constraints. *Proc. 13th ACM PODS*, 56–68, 1994.
- I.E. Sutherland. *SKETCHPAD: A Man-Machine Graphical Communication System*. Spartan Books, 1963.
- R.E. Tarjan. A Unified Approach to Path Problems. *JACM*, 28:3:577–593, 1981.
- A. Tarski. *A Decision Method for Elementary Algebra and Geometry*. University of California Press, Berkeley, California, 1951.
- J. D. Ullman. *Principles of Database Systems*, 2<sup>nd</sup> edition. Computer Science Press, 1982.
- P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.
- P. Van Hentenryck, D. McAllester, D. Kapur. Solving Polynomial Systems using a Branch and Prune Approach. Brown CS Tech. Rep. CS-95-01, 1995. To appear in the SIAM J. of Numerical Analysis.
- M. Y. Vardi. The Complexity of Relational Query Languages. *Proc. 14th ACM STOC*, 137–146, 1982.
- M. Yannakakis. Expressing Combinatorial Optimization Problems by Linear Programs. *Proc. 20th ACM STOC*, 223–228, 1988.

## 7. Appendix

In the Appendix, we present the proof of the closure theorem for Dense Order Constraint Algebra operations:

**Theorem 2** For every relational algebra *QUERY* on unrestricted finitely representable relations over  $D^n$ , the constraint algebra **QUERY** that uses **OPs** instead of *OPs* has the property that:

$$\sigma(\mathbf{QUERY}(\bar{r}_1, \dots, \bar{r}_n)) = \mathbf{QUERY}(\sigma(\bar{r}_1), \dots, \sigma(\bar{r}_n))$$

The first three lemmas involve projection.

LEMMA 8 *If  $\bar{t}$  is canonical and  $\bar{t}' = (\bar{t} \downarrow Z)$ , then  $\bar{t}'$  is canonical.*

**Proof:**

1. Assume that  $\bar{t}'$  is not canonical. Then, there exists a constraint  $\theta$  such that  $\bar{t}' \models \theta$ , but  $\theta$  is strictly tighter than the corresponding constraint in  $\bar{t}'$ .
2. Since the constraints in  $\bar{t}'$  are a subset of the constraints in  $\bar{t}$ ,  $\theta$  must also be strictly tighter than the corresponding constraint in  $\bar{t}$ . Furthermore, since  $\bar{t}$  is a union of  $\bar{t}'$  and some additional constraints, if  $\bar{t}' \models \theta$ , then  $\bar{t} \models \theta$ .
3. Therefore,  $\bar{t} \models \theta$  but  $\theta$  is strictly tighter than the corresponding constraint in  $\bar{t}$ . This is impossible since  $\bar{t}$  is canonical. Therefore,  $\bar{t}'$  must also be canonical. ■

LEMMA 9 *If  $\bar{t}$  is a canonical  $n$ -tuple and  $\bar{t}' = (\bar{t} \downarrow Z)$ , then  $P(\bar{t}') = \pi_Z(P(\bar{t}))$*

**Proof:**

1. Let  $p \in P(\bar{t})$ , i.e.,  $p$  is an assignment over  $X$  satisfying  $\bar{t}$ . Since  $\bar{t}' \subseteq \bar{t}$ , restricting  $p$  to  $Z$  must satisfy all the constraints in  $\bar{t}'$ . Therefore,  $\pi_Z(P(\bar{t})) \subseteq P(\bar{t}')$ .
2. For the other direction, it suffices to show that when  $|Z| = |X| - 1$ , if  $p \in P(\bar{t}')$  is an assignment over  $Z$ , then there exists an extension of  $p$  to  $X$  satisfying  $\bar{t}$ . The remainder of the proof easily follows by induction on  $|X - Z|$ .
3. Let  $X = \{x_1, \dots, x_n\}$ ,  $Z = X - \{x_n\}$ . Let  $p$  be an assignment satisfying  $\bar{t}'$ . We define a sequence of constraint sets  $\{C_1, \dots, C_n\}$  as follows:

$$\begin{aligned} C_0 &= \bar{t}' \cup (\bigcup_{1 \leq i \leq n-1} (x_i = p(x_i))), \text{ where } p(x_i) \text{ is the constant assigned} \\ &\text{to } x_i \text{ in } p; \\ C_1 &= C_0 \cup \theta, \text{ where } \theta \equiv (l_n = x_n) \text{ if } (l_n = u_n), \text{ and } \theta \equiv (l_n \leq x_n \leq u_n) \\ &\text{otherwise;} \\ C_2 &= C_1 \cup \xi_1, \text{ where } \xi_1 \text{ is the two-variable constraint over } (x_n, x_1) \text{ in } \bar{t}; \end{aligned}$$

...  
 $C_n = C_{n-1} \cup \xi_{n-1}$ , where  $\xi_{n-1}$  is the two-variable constraint over  $(x_n, x_{n-1})$  in  $\bar{t}$ .

4. Note that  $C_n \supset \bar{t}$ . In the remainder of the proof, we show that each  $C_i$ ,  $1 \leq i \leq n$ , is satisfiable. Since any  $p'$  satisfying  $C_n$  is an extension of  $p$ , we can conclude that there exists an extension of  $p$  to  $X$  satisfying  $\bar{t}$ .
5. *Inductive Assumption (IA)*. The actual statement about each  $C_i$ , proven by induction on  $i$ , is that  $C_i$  is satisfiable *and* one of the following is true:
  - (a)  $C_i \equiv C_0 \cup \{(x_n = c_e)\}$ , where either  $c_e = l_n = u_n$ , the bounds of  $x_n$  in  $\bar{t}$ , or there exists  $k \leq i$  such that  $c_e = p(x_k)$  and  $\xi_k = (x_n = x_k)$ ;
  - (b)  $C_i \equiv C_0 \cup \{(x_n > c_l), (x_n < c_u)\}$ , where:
 

either  $c_l = l_n$ , or there exists  $k \leq i$  s.t.  $c_l = p(x_k)$  and  $\xi_k = (x_n > x_k)$ ; and  
 either  $c_u = u_n$ , or there exists  $k \leq i$  s.t.  $c_u = p(x_k)$  and  $\xi_k = (x_n < x_m)$ .
6. *Base Case*:  $C_1 = \bar{t}' \cup (\bigcup_{1 \leq i \leq n-1} (x_i = p(x_i))) \cup \theta$ , where  $\theta$  is either  $(l_n < x_n < u_n)$  or  $(l_n = x_n)$ . It is easy to see that *IA* holds for  $C_1$ .
7. *Inductive Step*. Let us now assume that for some  $j \leq n-1$ , *IA* holds for all  $C_i$ ,  $1 \leq i \leq j$ . By definition,  $C_{j+1} = C_j \cup \xi_j$ , where  $\xi_j$  is one of:

$$(x_n = x_j), (x_n < x_j), (x_n > x_j).$$

By inductive assumption, there are two possibilities for  $C_j$ :

$$C_j \equiv C_0 \cup (x_n = c_e) \text{ or } C_j \equiv C_0 \cup \{(x_n > c_l), (x_n < c_u)\}.$$

It remains to consider the different combinations of  $C_j$  and  $\xi_j$ , making use of the following facts: (a)  $\bar{t}$  is canonical, (b) each  $C_i$  is a subset of  $\bar{t}$  and a superset of  $C_0$ . For each combination, it is easy to see that *IA* holds for  $C_j$ . ■

LEMMA 10 *Let  $\bar{r}' = \pi_Z(\bar{r})$ . Then,  $\bar{r}'$  is a canonical relation, and  $\sigma(\bar{r}') = \pi_Z(\sigma(\bar{r}))$ .*

**Proof:** This follows from Lemmas 8 and 9. ■

The next four lemmas involve selection, natural-join, union, and renaming.

LEMMA 11 *Let  $\bar{r}' = \varsigma_{F(\bar{t}_0)}(\bar{r})$ . Then,  $\bar{r}'$  is a canonical relation, and  $\sigma(\bar{r}') = \varsigma_{F(\bar{t}_0)}(\sigma(\bar{r}))$ .*

**Proof:**

1. According to the definition, all tuples in  $\bar{r}'$  are canonical.

2. Let  $p$  be an assignment to the variables in  $X$ .  $p \in \sigma(\bar{r}')$  iff there exists  $\bar{t}' \in \bar{r}'$  such that  $p \in P(\bar{t}')$  iff, by definition of generalized selection, there exists  $\bar{t} \in \bar{r}$  such that  $p \in P(\bar{t}_0 \uparrow X) \cap P(\bar{t})$ .
3.  $p \in P(\bar{t}_0 \uparrow X)$  iff  $p$  satisfies  $F(\bar{t}_0)$ . Therefore,  $p \in \sigma(\bar{r}')$  iff  $p$  satisfies  $F(\bar{t}_0)$  and there exists  $\bar{t} \in \bar{r}$  such that  $p \in P(\bar{t})$ .
4. This is equivalent to stating that  $p \in \sigma(\bar{r}')$  iff  $p \in \varsigma_{F(\bar{t}_0)}(\sigma(\bar{r}))$ .

■

LEMMA 12 *Let  $\bar{r}' = \bar{r}_1 \bowtie \bar{r}_2$ . Then,  $\bar{r}'$  is a canonical relation, and  $\sigma(\bar{r}') = \sigma(\bar{r}_1) \bowtie \sigma(\bar{r}_2)$ .*

**Proof:**

1. According to the definition, all tuples in  $\bar{r}'$  are canonical.
2. Let  $p$  be an assignment to the variables in  $Z$ .  $p \in \sigma(\bar{r}')$  iff there exists  $\bar{t}' \in \bar{r}'$  such that  $p \in P(\bar{t}')$  iff, by definition of generalized join, there exists some  $\bar{t}_1 \in \bar{r}_1$  and some  $\bar{t}_2 \in \bar{r}_2$  such that  $\bar{t}'$  is the common tuple of  $(\bar{t}_1 \uparrow Z)$  and  $(\bar{t}_2 \uparrow Z)$ .
3. This is equivalent to stating that  $p \in \sigma(\bar{r}')$  iff  $p$  satisfies  $F(\bar{t}_1 \uparrow Z) \wedge F(\bar{t}_2 \uparrow Z)$ .
4. By definition of common tuple,  $p$  satisfies  $F(\bar{t}_1 \uparrow Z)$  iff  $\pi_X(p) \in P(\bar{t}_1)$ ; i.e.,  $\pi_X(p) \in \sigma(\bar{r}_1)$ . Likewise,  $p$  satisfies  $F(\bar{t}_2 \uparrow Z)$ , iff  $\pi_Y(p) \in \sigma(\bar{r}_2)$ . Therefore,  $p \in \sigma(\bar{r}')$  iff  $p \in \sigma(\bar{r}_1) \bowtie \sigma(\bar{r}_2)$ .

■

LEMMA 13 *Let  $\bar{r}' = \bar{r}_1 \cup \bar{r}_2$ . Then,  $\bar{r}'$  is a canonical relation, and  $\sigma(\bar{r}') = \sigma(\bar{r}_1) \cup \sigma(\bar{r}_2)$ .*

**Proof:** This follows from the definitions. ■

LEMMA 14 *Let  $\bar{r}' = \varrho_{x_i|y}(\bar{r})$ . Then,  $\bar{r}'$  is a canonical relation, and  $\sigma(\bar{r}') = \varrho_{x_i|y}(\sigma(\bar{r}))$ .*

**Proof:** This follows from the definitions. ■

The last two Lemmas involve the difference operator; the notation here is taken from the definitions.

LEMMA 15  $\sigma(\text{tupdif}(\bar{t}_1, \bar{t}_2)) = P(\bar{t}_1) - P(\bar{t}_2)$ .

**Proof:** The proof proceeds by induction on  $m$ , the number of constraints that are different in  $\bar{t}_1$  and  $\bar{t}_2$ . The inductive assumption is that whenever the number of different constraints is less than  $m$ ,  $\text{tupdif}(\bar{t}_1, \bar{t}_2) = P(\bar{t}_1) - P(\bar{t}_2)$ .

**Base Case:**  $m = 0$ .

If  $m = 0$ , then  $\bar{t}_1 = \bar{t}_2$ , and  $P(\bar{t}_1) = P(\bar{t}_2)$ . Therefore,  $\text{tupdif}(\bar{t}_1, \bar{t}_2) = \emptyset = P(\bar{t}_1) - P(\bar{t}_2)$ .

**Inductive Step:**

(a) By definition,  $\text{tupdif}(\bar{t}_1, \bar{t}_2) = \text{tupdif}(\bar{t}_1^0, \bar{t}_2^0) \cup \{\bar{t}_1^1, \dots, \bar{t}_1^k\}$ , so

$\sigma(\text{tupdif}(\bar{t}_1, \bar{t}_2)) = \sigma(\text{tupdif}(\bar{t}_1^0, \bar{t}_2^0)) \cup (\cup_{1 \leq i \leq k} P(\bar{t}_1^i))$ .

(b)  $\bar{t}_1^0$  and  $\bar{t}_2^0$  have one more constraint in common than did  $\bar{t}_1$  and  $\bar{t}_2$  ( $\phi_0$  is that constraint), so by inductive assumption,  $\sigma(\text{tupdif}(\bar{t}_1^0, \bar{t}_2^0)) = P(\bar{t}_1^0) - P(\bar{t}_2^0)$ . Therefore,

$\sigma(\text{tupdif}(\bar{t}_1, \bar{t}_2)) = (P(\bar{t}_1^0) - P(\bar{t}_2^0)) \cup (\cup_{1 \leq i \leq k} P(\bar{t}_1^i))$ .

(c) For  $1 \leq i \leq k$ ,  $(\theta_2 \wedge \phi_i)$  is not satisfiable, so  $P(\bar{t}_2)$  and  $P(\bar{t}_1^i)$  are disjoint. This means that  $P(\bar{t}_1) - P(\bar{t}_2) = P(\bar{t}_1) - P(\bar{t}_2^0)$ .

(d)  $P(\bar{t}_1) = \cup_{0 \leq i \leq k} P(\bar{t}_1^i)$ ; also,  $P(\bar{t}_2)$  and  $P(\bar{t}_1^i)$  are disjoint for  $1 \leq i \leq k$ . So,  $P(\bar{t}_1) - P(\bar{t}_2) = \cup_{0 \leq i \leq k} P(\bar{t}_1^i) - P(\bar{t}_2) = (\cup_{1 \leq i \leq k} P(\bar{t}_1^i) \cup P(\bar{t}_1^0)) - P(\bar{t}_2) = (P(\bar{t}_1^0) - P(\bar{t}_2^0)) \cup (\cup_{1 \leq i \leq k} P(\bar{t}_1^i)) = \text{tupdif}(\bar{t}_1^0, \bar{t}_2^0)$  (by the equality of (b)). ■

**LEMMA 16** *Let  $\bar{r}' = \bar{r}_1 - \bar{r}_2$ . Then,  $\bar{r}'$  is a canonical relation, and  $\sigma(\bar{r}') = \sigma(\bar{r}_1) - \sigma(\bar{r}_2)$ .*

**Proof:** Since all tuples in a tuple difference are canonical, all tuples in  $\bar{r}'$  are canonical. The proof proceeds by induction on  $m$ , the size of  $\bar{r}_2$ . The inductive assumption is that whenever  $|\bar{r}_2| < m$ ,  $\sigma(\bar{r}_1 - \bar{r}_2) = \sigma(\bar{r}_1) - \sigma(\bar{r}_2)$ .

**Base case:**  $m = 0$ .

Since  $\bar{r}_2 = \emptyset$ ,  $\sigma(\bar{r}_2) = \emptyset$ . By definition,  $\bar{r}_1 - \bar{r}_2 = \bar{r}_1$ ; therefore,  $\sigma(\bar{r}_1 - \bar{r}_2) = \sigma(\bar{r}_1) = \sigma(\bar{r}_1) - \sigma(\bar{r}_2)$ .

**Inductive step:**  $|\bar{r}_2| = m$ .

(a)  $\bar{r}_1 - \bar{r}_2 = \cup_{\bar{t}_1 \in \bar{r}_1} \{\text{tupdif}(\bar{t}_1, \bar{t}_2) - \text{setdif}(\bar{r}_2, \{\bar{t}_2\}) : \bar{t}_2 \in \bar{r}_2\}$ .

(b) Let  $\bar{r}_3$  be  $\text{tupdif}(\bar{t}_1, \bar{t}_2)$ . By Lemma 15,  $\sigma(\bar{r}_3) = P(\bar{t}_1) - P(\bar{t}_2)$ .

(c) Let  $\bar{r}_4$  be  $\text{setdif}(\bar{r}_2, \{\bar{t}_2\})$ ;  $\sigma(\bar{r}_2) = P(\bar{t}_2) \cup \sigma(\bar{r}_4)$ .  $|\bar{r}_4| = m - 1$ , so by inductive assumption,  $\sigma(\bar{r}_3 - \bar{r}_4) = \sigma(\bar{r}_3) - \sigma(\bar{r}_4)$ .

(d) So,  $\sigma(\bar{r}_1 - \bar{r}_2) = \sigma(\cup_{\bar{t}_1 \in \bar{r}_1} : \bar{r}_3 - \bar{r}_4) = \cup_{\bar{t}_1 \in \bar{r}_1} \sigma(\bar{r}_3 - \bar{r}_4) = \cup_{\bar{t}_1 \in \bar{r}_1} (\sigma(\bar{r}_3) - \sigma(\bar{r}_4)) = \cup_{\bar{t}_1 \in \bar{r}_1} (P(\bar{t}_1) - P(\bar{t}_2) - \sigma(\bar{r}_4)) = \cup_{\bar{t}_1 \in \bar{r}_1} (P(\bar{t}_1) - \sigma(\bar{r}_2)) = (\cup_{\bar{t}_1 \in \bar{r}_1} P(\bar{t}_1)) - \sigma(\bar{r}_2) = \sigma(\bar{r}_1) - \sigma(\bar{r}_2)$ . ■