Constraint Query Algebras

by
Dina Q Goldin
B.A. Yale University, 1985
M.S. Brown University, 1987

Thesis
Submitted in partial fulfillment of the requirements for the
Degree of Doctor of Philosophy in the Department of Computer Science
at Brown University

May 1997

This dissertation by Dina Q Goldin
is accepted in its present form
by the Department of Computer Science
as satisfying the dissertation requirement
for the degree of Doctor of Philosophy.

Date 1/22/97

Stanley B. Zdonik
Stan B. Zdonik

Recommended to the Graduate Council

Date 1/22/97

Pascal Van Hentenryck

Date 1/12/97

Gabriel Mark Kuper
Free University of Brussels, Belgium

Approved by the Graduate Council

Date Jan. 23, 1997

Peter J. Denning

Dedication

This thesis is dedicated to Paris C. Kanellakis, who was my thesis advisor until his tragic death on Dec. 20, 1995, in the plane crash in Colombia. I owe a debt of gratitude to Paris for having chosen me to be his student and for all that I have learned from him. Paris saw the researcher in me while I still had my doubts, and he nurtured it till I was ready to stand on my own academic feet.

I deeply regret that Paris is not around now, when the fruits of his labor are finally at hand. His support has been invaluable, and his guidance truely inspiring; he forever remains my advisor in spirit.

Acknowledgements

I thank my co-authors, who validated my research abilities by inviting me to collaborate on joint publications: Gabi Kuper, Jan Chomicki, and Alex Brodsky. I thank people at Brown and elsewhere who took an interest in the direction of my academic career when my advisor's death made it uncertain: Dirk Van Gucht, Pascal Van Hentenryck, and Peter Wegner. I thank Stan Zdonik for taking on the role of my new advisor. But mostly, I thank my extended family, for just being there.

Combining the jobs of raising a family in Boston and studying for a doctorate degree at Brown University, all the while preserving my sanity, would have been impossible without the tremendous support of my extended family. I cannot imagine how I would have done it without them.

Contents

List of Figures

Chapter 1

Introduction and Background Definitions

1.1 Introduction

Traditional database technology is well-suited for handling "administrative data", such as names and telephone numbers. In the 90's, with the proliferation of the Internet and the unprecedented accessibility of desktop computing, the users are quickly accumulating non-traditional data, such as for medical, scientific, or geographic applications. More and more, they are starting to expect the same kind of data manipulation facilities for this data as MIS departments have enjoyed for theirs.

Constraint databases (CDBs) extend the relational database paradigm to allow working with infinite amounts of data, as long as it is finitely representable (via constraints). A perfect example is spatiotemporal data, which consists of points in time and/or in space, typical of the applications mentioned above. Besides having constraints for data representation, constraint databases also have them as query language primitives, with clean integration into all query language paradigms: procedural (algebraic), declarative (formula-based), or deductive (logic programming). Therefore, CDBs provide a strictly more expressive query paradigm than relational databases, even for "administrative" data.

CDB theory is an active area of theoretical database research. Our work concentrates on the "middle layer" of constraint database systems, i.e., the layer underneath the user interface layer and above the disk access layer. In this layer, querying consists

of primitive operations that are equivalent to the operations of the relational query algebra; i.e., any CDB query can be translated into an expression over these primitive operations. For this reason, we will refer to it as Constraint Query Algebra.

The particular issues for CDBs that we will examine include data representation, operator efficiency, and safe use of additional operators (see Section 1.7 for individual topics and our contributions). We consider Constraint Query Algebra to be particularly appropriate for studying these issues, because the algebraic middle-level for relational algebra proved so appropriate for similar issues in the relational database context. Our ultimate goal, as yet unattainable, is commercial success of Constraint Databases, on par with that of Relational Systems.

1.2 Constraint Databases

Our perception of how data management features should be packaged into software systems is largely governed by forty years of experience with mainframe database applications, and is quickly becoming outdated. Today's relational technology is not suited for tomorrow's data intensive applications, such as scientific or geographical or medical data repositories and query systems. Constraint Databases address this shortcoming of the relational paradigm by enriching both the data model and the query primitives with constraints. It is a young area of database research that lies at the intersection of Database Management, Constraint Programming, Computational Geometry and Operations Research.

One can look at the history of Constraint Logic Programming (CLP) to get a glimpse at the potential promise that Constraint Databases hold. Until the mid-'80s, Logic Programming was viewed mostly as an instructional tool, with limited real-world applications. Then, Logic Programming was transformed by applying the following insight:

- 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, i.e., CLP. Now, CLP is finding applications in Operations Research (which has been revolutionized by this technology), in Scientific Problem Solving, and in other areas where problems can be stated declaratively and

can be solved by a combinatorial search approach [VanH].

It was not immediately clear how to apply the same type of insight to Relational Databases, since the bottom-up and set-at-a-time style of database query evaluation emphasized in databases seems to contradict the top-down, depth-first intuition behind Constraint Logic Programming. In [KKR95], the gap between database programming and constraint solving was bridged, with the following key intuition:

- 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.

Therefore, an appropriate generalization of a tuple is a *conjunction of constraints over the tuple attributes.*

The constraint data model is a generalization of the relational data model which allows databases to model relations that include *infinitely* many data points, by replacing the notion of *finite data* with *finitely representable* data. A perfect example is spatiotemporal data, which consists of points in time and/or in space [BJM93, BLLM95, PVV94, VGVG95, Cho94], typical of the applications mentioned at the beginning of the chapter.

Besides extending the data model with constraints, Constraint Databases also integrate constraints into the queries, while preserving the efficient bottom-up declarative semantics that enabled relational databases to become such a success. There has been work on the complexity of constraint database queries ([KKR95, GST94, PVV95]), concentrating on data complexity (i.e. treating the number of variables k in a query as a constant, see [CH82, Var82]). 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 has provided a unified view of some previous database research: for example, on the power of constraints for the implicit specification of temporal data [CI89], on relational query safety [AGSS86], on conjunctive queries with inequalities [Klu88] and on extending magic sets [Ram88]. By widely enriching the types of data that can be managed by Database Management Systems, as well as the types of queries that can be expressed in they System, Constraint Database technology holds a lot of promise for the database users of tomorrow.

1.3 Declarative Querying of CDBs

In this section, we discuss declarative query languages for constraint databases, or Constraint Query Languages (CQLs). 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 [Cod70] 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 [AHV, Kan90, Ull].

CQLs can be viewed as a specialized form of constraint programming, similar to the way that relational query languages can be viewed as a form of first-order theorem proving. 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 [Sut]. The theme has been investigated since the 1970's, e.g., in artificial intelligence [Mon74, Mac77, Fre78, Ste80], in graphical-interfaces [Bor81], and in logic programming languages [JL87, DVSAGB88, Col80].

One of the most important advances in constraint programming in the 1980's has been the development of Constraint Logic Programming (CLP) as a general-purpose framework for computations, e.g., in CLP($\Re$) [JL87], in Prolog III [Col80], and in CHIP [DVSAGB88, VanH]. 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 [KKR95], 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.

In [KKR95], finite relations are generalized to finitely representable relations and appropriate calculi for their data manipulation can be developed in this framework. Thus, constraint databases are a natural generalization of the relational model of data by allowing infinite relations that are finitely representable using constraints. The work on the complexity of constraint database queries ([KKR95, GST94, PVV95]) has concentrated on data complexity, treating the number of variables k in a query as a constant. 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.

1.4 Procedural Querying of CDBs

The relational data model was pioneered in [Cod70]. From that work, follows that Relational Calculus on finite sets can be evaluated bottom-up in closed form. *Relational Algebra*, an operator-based query paradigm with bottom-up expression evaluation semantics and LOGSPACE data complexity [Var82], is just the query language whose existence was foreseen in [Cod70].

The algebraic querying paradigm is not declarative, since the algebraic expressions represent a 'plan' or a 'recipe' for evaluating a query. Relational Algebras play a very important role in relational database theory, since they are more useful than the calculi for carrying out query optimization and evaluation. For this reason, it is typical that declarative user queries are translated into algebraic expressions before they are optimized and evaluated by the relational database system.

Constraint Query Algebras (for various constraint classes) are the equivalent of Relational Algebras for the Constraint Database model. No syntax or semantics for CQAs was provided in [KKR95]; however, 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*. Note that it is relatively easy to transform a Constraint Query Calculus into an "naive" algebra, one where operations are defined as syntactic manipulations of constraint formulas. For example, with Fourier-Motzkin elimination [Sch] one easily derives a "naive" algebra for linear constraint databases [GST94].

In this work, we will show that the "naive" approach is not sufficient to define a

good Constraint Query Algebra, i.e. one that preserves the practical advantages of the algebraic querying paradigm. We will provide the syntax and semantics of CQAs, with the goal of preserving these advantages. We expect that CQAs will prove just as ubiquitous in Constraint Database Systems as Relational Query Algebras are in the relational database systems.

1.5 Relational Algebra: Definitions

Since the Constraint Query Algebras are a generalization of Relational Algebras, we need to define the latter before looking at the former. 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 Chapter 3; in this section, we present a definition of Relational Algebra.

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. In the definitions that follow, clause (1) expresses the conditions required of the argument and resulting relations, and clause (2) expresses the semantics of the operation.

Projection: $\pi_Z(r)$ is the projection of r on Z.
(1) $Z \subseteq \alpha(r)$ and $\alpha(\pi_Z(r)) = Z$.
(2) $\pi_Z(r) = \{t[Z] : t \in r\}$, where $t[Z]$ is the restriction of t to the variables in Z.

Selection: $\varsigma_\phi(r)$ is the selection on r by ϕ.
(1) ϕ is a boolean formula over $\alpha(r)$ and $\alpha(\varsigma_\phi(r)) = \alpha(r)$.
(2) $\varsigma_\phi(r) = \{t : t \in r$ and $\phi(t)$ is true$\}$.

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) $r_1 \bowtie r_2 = \{t : t$ is an $\alpha(r_1) \cup \alpha(r_2)$-tuple, such that $t[\alpha(r_1)] \in r_1$ 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) $\alpha(r_1) \cap \alpha(r_2) = \emptyset; \alpha(r_1 \bowtie r_2) = \alpha(r_1) \cup \alpha(r_2)$.
(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: Both Cross-Product and Intersection are special cases of the Natural-Join operation, so the CQA operations we define later in this work will only include an analogue to the Natural-Join. It should be noted that Natural-Join is itself expressible using Cross-Product, Select, and Project; however, Natural-Join is more commonly used than the Cross-Product.

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 [Kan90].

1.6 Principles of Constraint Query Languages

As a background for the rest of the work, we present a summary of the framework from [KKR95].

Definition 1 A generalized k-tuple, *or a* constraint 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.*

For example, in the relational database model, $R(3,4)$ is a tuple of arity 2. 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. All relational tuples, including this one, are generalized tuples over equality constraints. $R(x,y)$ with $(x=y \wedge x<2)$ is also 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).

Definition 2 A generalized relation, *or a* constraint relation *of arity k is a finite set of generalized k-tuples, with each k-tuple over the same variables.* A generalized database *is a finite set of generalized relations.*

A generalized relation 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).

Definition 3 The syntax of a Constraint Query Calculus *is the union of a relational database query language and formulas in a decidable logical theory.*

For example: Relational calculus [Cod70] + the theory of real closed fields [Tar, Ren92]; Relational calculus (or even Inflationary Datalog$^\neg$, [AHV] + the theory of dense order with constants [FG77].

Definition 4 The semantics of CQC *is based on that of the decidable logical theory, by interpreting database atoms as shorthands for formulas of the theory.*

Let $\phi=\phi(x_1,\ldots,x_m)$ be a constraint query program using free variables $x_1,\ldots,x_m$. Let predicate symbols $R_1,\ \ldots,\ R_n$ in ϕ name the input generalized relations and let $r_1,\ldots,r_n$ be corresponding input generalized relations. We interpret the program in the context of such an input. Let $\phi[r_1/R_1,\ldots,r_n/R_n]$ be the formula of the theory that is obtained by replacing in ϕ each database atom $R_i(z_1,\ldots,z_k)$ by the DNF formula for input generalized relation r_i, with its variables appropriately renamed to $z_1,\ \ldots,\ z_k$. The output is the possibly infinite set of points in m-dimensional space D^m, such that instantiating the free variables $x_1,\ldots,x_m$ of formula $\phi[r_1/R_1,\ldots,r_n/R_n]$ to any one of

these points makes the formula true. (We can assume that an occurrence of a database atom in ϕ is of the form $R_i(z_1, \ldots, z_k)$ $1 \leq i \leq n$, where R_i is of arity k and $z_1, \ldots, z_k$ are distinct variables; this is because we can always use equality constraints).

Definition 5 *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.

The framework of [KKR95] imposes two critical requirements on queries:

- *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 [Cod70, Ull]. The precise analogue in relational databases is the notion of weak safety of [AGSS86]. 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 with operator-based expressions, i.e., a *Relational Algebra.*

- *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 [AHU]. It seems reasonable to limit computations to efficient ones, i.e., PTIME manipulations of the data. The constraint query framework is interesting because many combinations of database query languages and decidable theories have PTIME data complexity.

1.7 Overview

Constraint query languages extend the relational paradigm to allow the handling of spatiotemporal data. In this work, we study Constraint Query Algebras, which we believe is the proper query language for considering the implementational issues of constraint query languages.

In Chapter 2, we provide the foundations for the rest of the work, by presenting the issues involved in designing Constraint Query Algebras. We consider the desirable properties for *data representation*, and for *operator implementation*. We examine the notion of duality of representation (syntax) and meaning (semantics), central to the theory of CQAs. We define the principle of *semantic closure*, one which makes explicit the commutativity of the syntactic and semantic interpretations of a CQA.

In Chapter 3, we briefly consider two issues that are crucial to practical implementations of constraint database querying: indexing and optimization. For optimization, we suggest two promising approaches to optimizing CQA queries. The first is *lazy evaluation* of linear and nonlinear constraints (for *real polynomial constraints* see [Tar]; for recent developments and a symbolic computation survey see [Ren92], and for numerical computation see [VMK95]. The second approach is to optimize CQAs by using the indexing information, just as for relational algebras.

In Chapter 4, we present the syntax for data representation for dense-order [FG77, Kan95, Klu88] and temporal constraint tuples, and define the algebraic operations over this representation. We apply the principle of *semantic closure* to establish the correctness of the dense-order algebra. We note that this algebra satisfies all the criteria for a good algebra presented in in Chapter 2. In particular, the *projection* operation is extremely efficient: projection is carried out by restricting the tuples to the desired attributes, just as for the relational model.

In Chapter 5, we consider how the implementation of the PROJECT operation can be made more efficient for a subclass of *linear inequality constraints* (see the comprehensive survey in [Sch]). We prove a new result, which is that the *projective closure* of a given monotone constraint tuple is often strongly polynomial in the size of the tuple (i.e., whenever the *path expression* for the corresponding *monotone network* is of polynomial size). We also provide a concrete algorithm for achieving this polynomial bound.

In order to match the expressiveness of the relational queries, it is necessary to allow *aggregation* operations in CQA expressions. In fact, generalizing aggregation

operators to constraint databases has been identified as one of the most important open research issues in the constraint database area [KG94, Kan95]. In Chapter 6, we provide a natural restriction on Constraint Database schema which guarantees the safety of algebraic expressions involving aggregation. This restriction, called *variable independence*, is a generalization of the assumptions underlying the classical relational model of data.

Finally, in section 7, we show how queries over spatiotemporal data can be defined in a constraint setting, and implemented with a multidimensional indexing structure. We provide both the description and the implementational details for a framework for similarity querying of time-series data. Similarity queries are strictly more expressive than approximate match queries, for which a framework had been supplied in [FRM94].

Chapters 2-7 represent original work, most of which has already been published [KG94, CGK96, GK96, GK95]. Further work is in progress in the areas of aggregation, monotone constraints, and similarity querying.

Chapter 2

Constraint Query Algebras

Constraint Query Algebras are a generalization of Relational Algebra for different classes of constraints. Constraint Query Algebras need to satisfy the requirements on queries defined above (Section 1.6). They should also have the same expressibility as Constraint Query Calculi, i.e. that for any Constraint Query Calculus expression, there exists an equivalent CQA expression. It is also highly desirable that there be an efficient *translation* between the two expressions.

In this chapter, we provide the foundation for the rest of the work by isolating the individual issues that need to be addressed when designing a good Constraint Query Algebra.

2.1 Data Representation and Semantics

The data in Constraint Databases can be viewed both as a constraint formula over k variables (the *generalized tuple*, or *syntax*), and as the set of k-dimensional points that satisfies the formula (the *extension*, or *semantics* of the data). Figure 2.1 illustrates this duality. The concrete format that a Constraint Database may choose for representing the data will lie somewhere between these modalities: some pre-processing of formulas is desirable (at least to eliminate the unsatisfiable ones), yet the actual set of points may be infinite and therefore not directly representable.

The first condition for any choice of data representation is that it have *correct semantics*:

- Given the data representation for a constraint relation R, the (infinite) set of

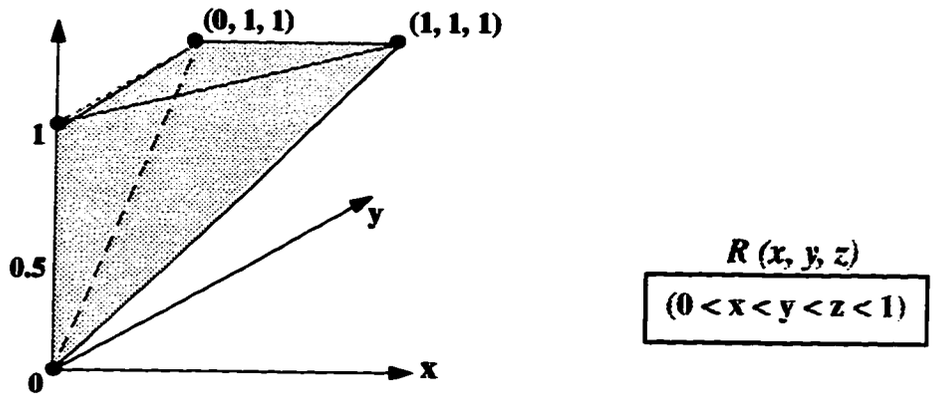


Figure 2.1: The dual nature of constraint data.

tuples $\sigma(R)$ that it represents must be the same as the set of points that satisfies the original constraint formula $\phi(R)$.

Whichever format is chosen, it is assumed that the operators of the Constraint Query Algebra, when implemented, will receive their input data in that format. To be able to compose operations, the output of the operators is also expected to be in the same format. A *pre-processing* mechanism needs to be provided for converting constraint formulas into this format.

There are often implementation reasons to choose a specific format for the generalized tuples in the database. We consider the following characteristics to be desirable for the data representation, when it is considered in the context of a practical Constraint Query Algebra:

- **Efficiency.** This refers both to the space complexity of the data representation, and to the time complexity of evaluating the operators for the given data representation.

- **Succinctness.** 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. For example, if the data is represented by formulae without any post-processing after operations, then the naive implementation of a *join* or an *intersection* by "and"-ing pairs of formulae to create new ones 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.

- **Redundancy.** 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, so this is of less concern. However, a *redundancy removal* procedure akin to rebuilding a database index may need to be done off-line whenever redundancy is somehow determined to reach a certain level.

- **Updating.** Since data needs to be pre-processed, *updates* to the database (i.e., adding, deleting, or modifying data) become an issue. It is highly desirable that updates can be performed in time that depends only on the size of the individual generalized tuple. This means that any representation which requires a scan or search of all tuples when preparing to add a new one is impractical.

- **Indexing.** Given an attribute x to use as a key in a constraint relation r, the indexing structure will typically use intervals for its index, one for each generalized tuple t in r; such an interval represents the range of the values for x within the extension of t. The computation and the updating of these intervals should therefore be very efficient.

We shall use the term *canonical form* for a specific data representation format. For example, we may view the *r-configurations* of [KKR95] 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.

2.2 Algebraic Operations and Semantic Closure

Query algebras typically consist of a few set-based operations, with bottom-up evaluation of the operator-based expressions. 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.

Theoretically, any set of operators that preserves the algebra-calculus equivalence can be used, though the author of such algebra can not make use of existing relational methodologies for doing query translation and optimization. Since we want to build on the existing successes of relational technology, and to be able able to reuse as much machinery as possible in implementing Constraint Databases, we propose that the standard algebraic operations such as SELECT, PROJECT, NATURAL-JOIN, UNION, RENAME, DIFFERENCE are adopted for Constraint Query Algebras, just like Codd's relational algebra (Section 1.5).

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

Definition 6 *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 Section 1.5 generalizes as well:

Definition 7 *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 8 *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 $\varsigma_F(R)$. At a minimum, the selection operator should admit as F any constraint from $\mathcal{C}$.

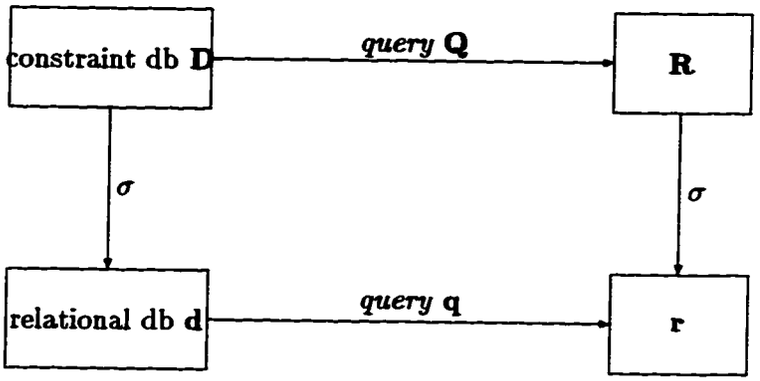


Figure 2.2: The Closure Condition: $\sigma(Q(D)) = q(\sigma(D))$.

2.3 The Closure Principle

Whatever syntax is chosen for a CQA, the semantics of the operations should be such as to provide closure for the semantics of the data (Section 2.1). In particular, if $\overline{r}_1, \ldots, \overline{r}_n$ are generalized relations which describe (possibly infinite) relations $\sigma(\overline{r}_1), \ldots, \sigma(\overline{r}_n)$, 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

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

We refer to this property as the *closure* of a CQA (see Figure 2.3).

This principle of *semantic closure* for the algebraic operators enables us to define algebraic operators syntactically, as concrete operations over some specific data representation, and yet be able to argue about their correctness. That is to say:

> A CQA operator is defined correctly if for any input, the semantics of the output data is the same as it would be for equivalent relational algebra expressions over the corresponding (infinite) sets of points.

Figure 2.3 illustrates operations SELECT and PROJECT over the constraint relation of figure 2.1.

As a result of the closure condition, CQAs admit the same notion of expression equivalence as relational algebras. As a corollary of semantic closure for a given Constraint Algebra, we also prove the equivalence of the Constraint Algebra and Constraint Calculus.

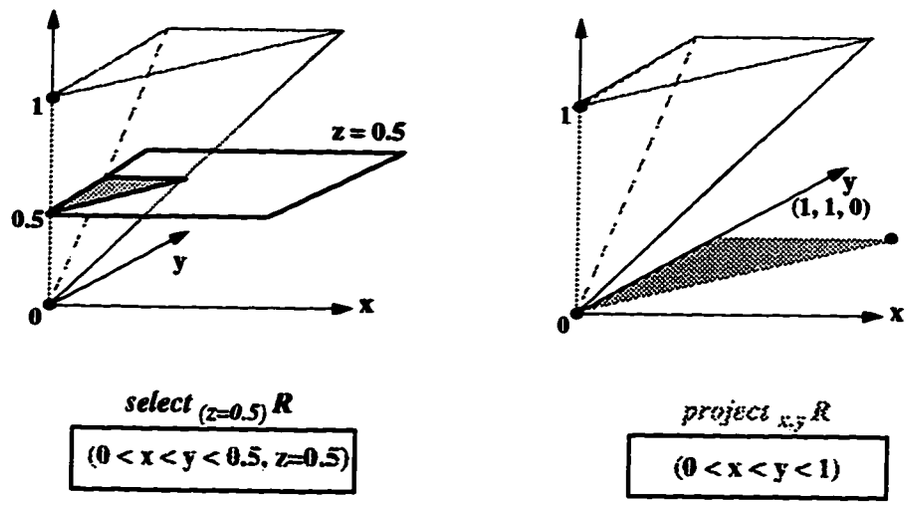


Figure 2.3: Examples of CQA operations over constraint data.

2.4 Operator Efficiency

A minimal requirement for practical Constraint Databases 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.

We believe the critical issue in designing a "good" algebra is to make projection simple and cheap. The approach is different depending on the Constraint Class involved, and both the data representation and the operator implementation have to be tailored in each case.

Projection vs. Restriction: 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* [Fre82, Dec92] 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 Chapter 4. The example is an algebra for dense order constraints, which was published in [KG94] in preliminary form. As explained in Section 2.1, the representation of tuples used in this algebra also has other good properties; this representation also easily extends to temporal constraints.

Strongly Polynomial Projections: 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 in [Yan88], 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, [Nel78], 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 [HN94]; 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.

In Artificial Intelligence, the property of *global consistency*, also known as *decomposability* [Mon74], is desirable because it allows a *backtrack-free search* [Fre82]. The property has been studied for temporal constraints [DMP91] 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 [Kou93]).

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.

This approach is studied in Chapter 5. The example is an algebra for monotone linear two-variable constraints; temporal constraints are a special form of these constraints. We consider monotone constraints over the rationals, and show that in most cases, projection is strongly polynomial. The more complex integer case for monotone constraints is analyzed by Hochbaum and Naor [HN94].

2.5 Additional Algebraic Operators

Aggregation. To be able to express practical queries, some additional operators are needed, just as for relational query languages. One of them is the generalized version of the "aggregation" operator, a general operator that expresses statistical operations such as AVG, MIN, COUNT [Klu82]. Some aggregation operators like **count** are not applicable to infinite relations. On the other hand, new operators like **area** [AS91] (or its generalization, n-dimensional volume [GK94]) occur there quite naturally.

[Kup94] describes a general framework, modeled after [Klu82], for adding aggregate operators to relational algebra and calculus. However, in general, adding aggregates to constraint databases results in langua ges that are not closed, i.e., the result of a query that uses aggregation is not finitely representable using the constraint language of the database.

In Chapter 6, we will provide a restriction on Constraint Database schema which guarantees the safety of algebraic expressions involving aggregation. This restriction,

called *variable independence*, is a generalization of the assumptions underlying the classical relational model of data.

Recursion. Another operator which we consider essential for expressing practical queries is recursion, which is equivalent to finding paths in a graph whose edges are specified by a relation. Recursive queries are common for databases that hold travel information (such as train or airplane connections), manufacturing information (such as bill-of-materials), or genealogical information (such as family relations). It is well-known that first-order calculi, or equivalently algebras, are not expressive enough to answer such queries [Pa94].

A well-behaved recursion operation for Constraint Databases should have the standard fixpoint semantics of recursion and the queries should be guaranteed to terminate and to have a finitely representable answer. Some progress in this area has been made very recently [GK97]. The operators defined there, the *bounded partial fixpoint* and the *bounded inflationary fixpoint*, lead to some new results in expressiveness and complexity of PTIME boolean queries over linear constraint databases. However, the closure of general queries is not considered, and we believe the operators to be too artificial for practical use.

It is possible that the notion of *variable independence*, defined in Chapter 6, can be applied to provide closure conditions for CQA+fixpoint. However, we will not address this in our present work.

Chapter 3

Towards Practical CQAs

Relational databases would not be such a commercial success if their implementation did not allow such fast ad-hoc querying of such vast amounts of data. This performance is enabled by two technologies: efficient *indexing* of the data, for fast data access, and *optimization* of queries, for selecting good query execution strategies among many alternatives. These issues should prove crucial for constraint databases as well.

Constraint query algebras form a procedural language layer below high-level declarative calculi. Low-level access methods form the next layer of the database, and must be properly interfaced with the algebra. Also, just like the relational algebra, this intermediate layer should prove to play a significant role in query optimization.

In this chapter, we offer a brief look at indexing and optimization issues for constraint databases. We consider two possible approaches to optimizing CQAs. The first is to delay projections by retaining unevaluated variables: *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 is applicable to linear and nonlinear constraints as long as queries are positive and could be of significance for constraint query algebra implementations. The second is to make use of the implementational statistics for the indexing and hashing data structures to guide the optimization heuristics.

3.1 Indexing in Constraint Databases

The language framework of the relational data model does have low data complexity, but does not account for searches that are logarithmic or faster in the sizes of input

relations. Without the ability to perform such searches, relational databases on secondary storage would have been impractical. I/O efficient (i.e., logarithmic or constant) use of secondary storage is an additional requirement, beyond low data complexity 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 [BM72]). 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* $x_1, \ldots, x_k$ 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 [Sam].

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 [PS]. 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 [KRVV93] and for an optimal dynamic one see [AV95].

3.2 Algebraic Expressions and Query Optimization

The use of algebraic expressions 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.

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. An 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 [Ram88, MFPR90, SR92, SS94].

In contrast to algebraic transformations, estimation-based optimization heuristics are implementation-dependent. In fact, the dependency is bi-directional: queries may be made more efficient by using the implementational information (such as the type of indexing available for various attributes), and indexing strategies can be made more efficient by knowing which queries are more likely to be asked. Section 3.1 discusses the implementation of indexing structures for constraint databases. Then, in Section 3.3, we suggest that CQA query optimization can benefit from indexing information in a fashion similar to the approach of System R.

3.3 Optimization in System R

The implementation-related information can be very useful in finding a good query evaluation strategy. An example of a relational database system which uses the implementation-related information for query optimization is *System R* [Ull]. The information includes:

- How many tuples are in a given relation?
  This requires keeping a running count of the number of tuples, updating it upon tuple insertions and deletions.

- What is the average size of a tuple?
  In System R, tuples may have variable size when variable-length strings are present. If tuple size is not constant, an estimation based on a sample of records must be used.

- Which indices exist for a given relation, and what is the image size of each index?
  For an index on a relational attribute X, the *image size* is the number of different values of X appearing in the relation. Assuming that the leaf blocks of the index consist of keys, one per each value of X, the image size is the number of leaf blocks times the number of keys per block. A sampling technique must be used to estimate the latter.

- Which, if any are *clustering indices*?
  For relational databases, a *clustering index* is an index on an attribute (or attributes) such that tuples with the same value for those attributes will tend to be adjacent in memory.

[Ull] has an example of how the above parameters affect the choice of an evaluation strategy for a given query. Some of the issues involved in generalizing the approach of System R to constraint databases are highlighted below.

In constraint databases, tuples cannot be expected to have a fixed size, except for "gap" constraints and their subclasses (dense-order and temporal constraints), where canonical forms allow a fixed-size representation. As a result, average tuple size, can only be determined by an estimate based on sampling. For several reasons discussed in [BJM93], the sampling techniques used in relational databases are inappropriate in the constraint database setting and need to be reconsidered.

Also, the notion of image size is not trivial to generalize. The intuition behind the usefulness of image size in predicting query performance is:

> In a relation R, we can expect that the number of tuples with a given value for an attribute equals the total number of tuples in R divided by the image size.

Due to the infinite semantics of constraint relations, this definition, as well as the way image size is determined, have to be reconsidered to preserve its applicability to query optimization.

We are confident that implementation-based evaluation of query strategies can prove useful in constraint databases, and that the generalizations of the approach of System R will lead to efficient query optimization in constraint databases.

3.4 Lazy Evaluation of (Non)Linear Constraint Algebras

In this section, we consider an alternate approach to optimizing CQA queries, in particular the *positive* fragment consisting of the operations *project*, *select*, *natural-join*, *union*, and *rename*; this is similar to the approach of [BJM93].

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 [VanH]. 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 [VMK95].

Lazy Evaluation: The approach 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 [Sch]. What's worse, variable elimination can be exponential just because of the size of the result [Yan88] 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 3.1). 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 [Sch], 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 $(x_1, \ldots, x_k)$ with extraneous variables $(x_{k+1}, \ldots, x_n)$; 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:** $R = \{\pi_{(x_1,\ldots,x_i)} t_1 : t_1 \in R_1\}$
  Mark $(x_{i+1}, \ldots, x_k)$ as extraneous, so R is a relation over $(x_1, \ldots, x_i)$ with extraneous variables $(x_{i+1}, \ldots, x_n)$. The constraints in t_1 are not changed.

- **Selection:** $R = \{\varsigma_F(t_1) : t_1 \in R_1\}$
  We assume that F is quantifier-free and all its variables are among $(x_1, \ldots, x_n)$. 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.

Chapter 4

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 θ is one of $=$, $\leq$, $<$ (or its negation $\neq$, $>$, $\geq$). We assume that these constants are interpreted over a countably infinite set D with a binary relation which is a dense order (e.g., we can take D to be the rationals). Constants, $=$, $\leq$, and $<$ are interpreted respectively as elements, equality, the dense order, and the irreflexive dense order of D. For the first-order theory of dense order see [FG77] and for its data complexity (with and without recursion) see [KKR95].

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 applications. Note that the geometry of dense order constraints is richer than that presented in [GS95]; for example, some rectangles 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 Chapter 2. 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.

4.1 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:

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

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 1 *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.

4.2 The Canonical Form

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

Definition 9 *A consistent generalized tuple $\bar{t}_0$* entails *a constraint θ 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 10 *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 θ be a constraint entailed by $\bar{t}_0$ such that all constants appearing in θ also appear in $\bar{t}_0$. Then:*

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

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-symmetricity 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 11 *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 2 *The tuple* $\bar{t}$ *in the previous example is not canonical because:*

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

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

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

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

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 θ be an arbitrary constraint in $\bar{t}_0$; θ must belong to some entailed subset of $\bar{t}_0$. Let θ' 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 θ_1 be some constraint entailed by $\bar{t}_0$, and θ_2 be the constraint in $\bar{t}_c$ of the same type and with the same variables as θ_1. Then, θ_2 entails θ_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 θ_1 belongs to some entailed canonical set of $\bar{t}_0$. Therefore, we assume that it does not; i.e., θ_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 θ_2 does not entail θ_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 θ_2 is the tightmost member of the entailed subset. This means that θ_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:

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

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 [KKR95] 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,

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

   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 θ_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 12 *1. A* canonical relation *over variables $X = (x_1, \ldots, x_n)$ is a finite set of canonical tuples over X. There exists a natural mapping ϕ from canonical relations over X to DNF formulas over X, and a corresponding mapping σ 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.*

4.3 The Algebraic Operations

We can now introduce the syntax and the semantics of an algebra over canonical relations. As in Codd's relational algebra [Cod70, Kan90], we define basic algebraic operations *projection* (π), *selection* (ς), *natural join* ($\bowtie$), *union* ($\cup$), *difference* ($-$), and *renaming* (ϱ), which map one or more canonical relations to a new one. *Algebraic queries* over canonical databases are composed of these basic operations.

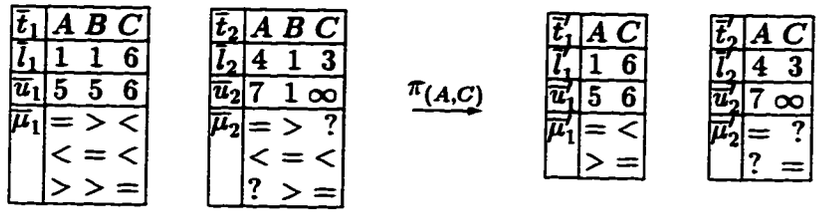


Figure 4.1: Projection

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

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

where OP is the operation of the same name in the relational algebra in [Kan90], and σ is defined in Definition 12.

In our definitions of operators, we use the notation $\alpha(\bar{r})$ for the variables of a canonical relation $\bar{r}$.

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 7).

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* [Fre82, Dec92] for canonical tuples).

Example 3 *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 4.1).*

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 $F(\bar{t}_0)$:

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

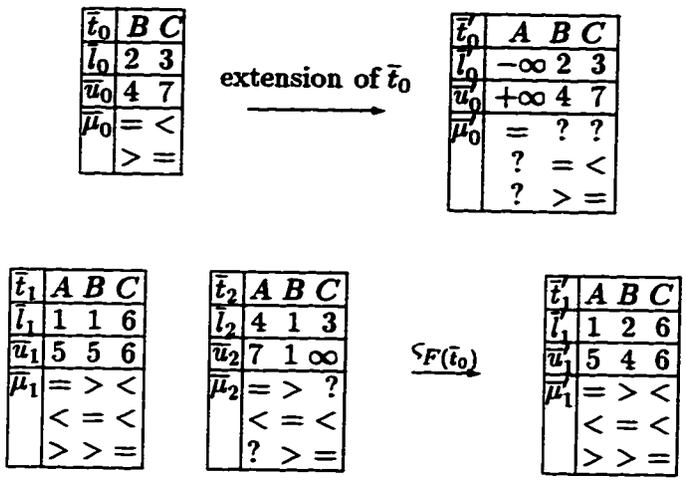


Figure 4.2: Selection

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

Definition 13 *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 4 *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 4.2). 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.*

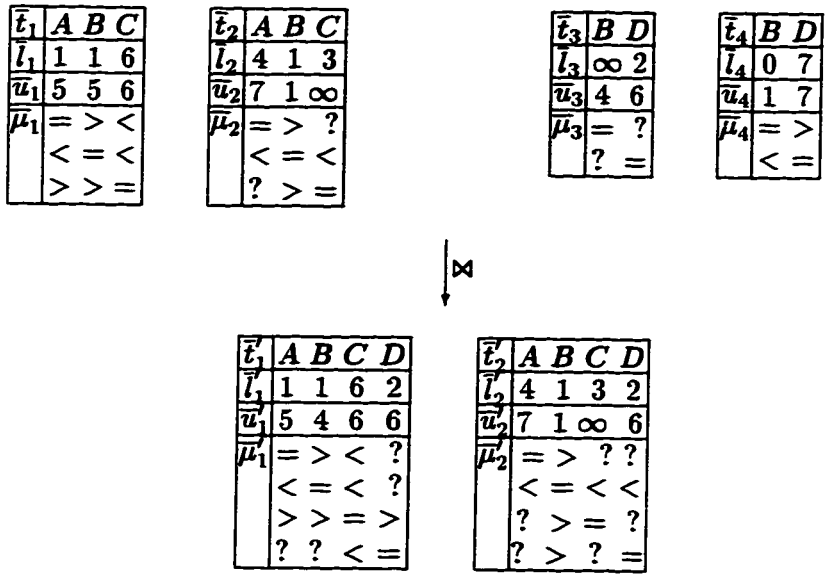


Figure 4.3: Join

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

Example 5 *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 4.3). 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 $\bar{t}_3$.*

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

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

Definition 14 *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 ϕ_0 be the constraint such that $(\theta_1 \wedge \theta_2) \equiv \phi_0$. Let $\{\phi_1, \ldots, \phi_k\}$ be the smallest set of disjoint constraints such that $(\theta_1 \wedge \neg\theta_2) \equiv \vee(\phi_1, \ldots, \phi_k)$; $k \geq 1$.*

4. *Let $\bar{t}_1^0$ ($\bar{t}_2^0$) be obtained by replacing θ_1 in $\bar{t}_1$ (θ_2 in $\bar{t}_2$) by ϕ_0 and putting the result into canonical form. Similarly, let $\{\bar{t}_1^1, \ldots, \bar{t}_1^k\}$ be obtained by replacing θ_1 in $\bar{t}_1$ by each ϕ_i, $1 \leq i \leq k$, and putting the result into canonical form.*

5. *Then $tupdif(\bar{t}_1, \bar{t}_2) = tupdif(\bar{t}_1^0, \bar{t}_2^0) \cup \{\bar{t}_1^1, \ldots, \bar{t}_1^k\}$.*

Example 6 *The difference of $R_1 = \{\bar{t}_1, \bar{t}_2\}$ and $R_2 = \{\bar{t}_3, \bar{t}_4\}$, shaded in Figure 4.4, 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 14, $tupdif(\bar{t}_1, \bar{t}_3) = \emptyset$. By item (1) of Definition 14, $tupdif(\bar{t}_2, \bar{t}_3) = \{\bar{t}_2\}$. 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 14, θ_1 is $(3 < x < 8)$, and θ_2 is $(7 < x < 15)$. Then, by item(3), ϕ_0 is $(7 < x < 8)$, and $k = 2$, where ϕ_1 is $(3 < x < 7)$, ϕ_2 is $(x = 7)$.*

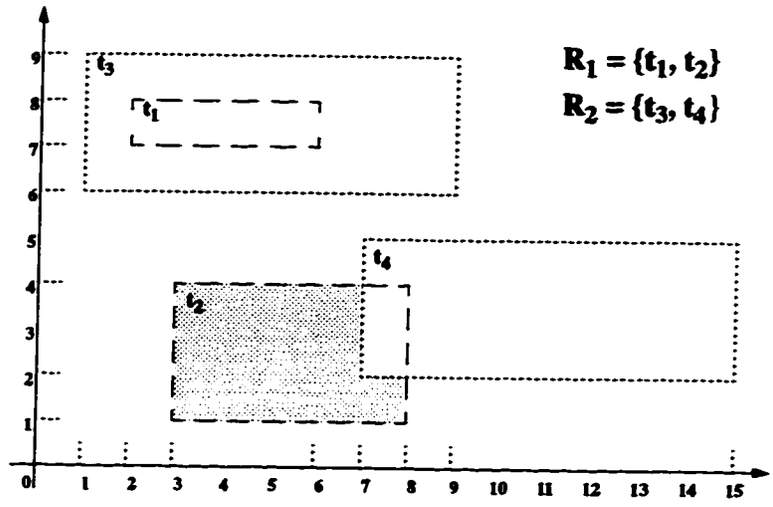


Figure 4.4: The difference of R_1 and R_2.

4. *By items (4) and (5) of Definition 14,* $\bar{t}_2^0$ *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 $\varrho_{y|x}(\bar{r})$ is the renaming in $\bar{r}$ of x to y:

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

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

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

where OP is the operation of the same name in the relational algebra and σ is defined in Definition 12.

The closure properties of the operations are composed to obtain the closure of the dense-order algebra:

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

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

Proof: See Appendix. □

4.4 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 θ 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 [KKR95], 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_ϕ 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_ϕ, whereas it is *constant* for checking whether a tuple is canonical. Therefore an advantage of the canonical tuples here over the r-configurations of Kanellakis et al. is that insertions are more efficient.

Furthermore, for an arbitrary conjunction θ 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 θ, whereas it would grow to be *at least linear* in the size of D_ϕ 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 section remain valid.

Chapter 5

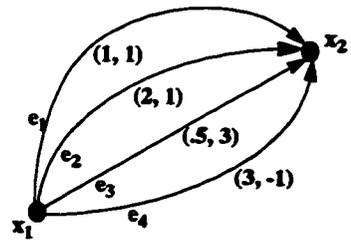
A Monotone Two-Variable Linear Constraint Algebra

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. 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 6) 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 stage of variable elimination. We use graph parsing techniques for regular path expressions to show that the performance of the resulting algorithms is strongly polynomial.

5.1 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.



(A)

$c_1 = (x_1 \leq x_2 + 1)$

$c_2 = (x_1 \leq 2x_2 + 1)$

$c_3 = (x_1 \leq .5x_2 + 3)$

$c_4 = (x_1 \leq 3x_2 - 1)$

(B)

Figure 5.1: The set of edges (A) corresponds to the set of constraints (B).

For the rest of this chapter, we assume that θ 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 5.1); $a > 0$.

Each node $x_i \in X$ is labeled with the upper and lower *bounds* for the corresponding variable: low(x_i) and high(x_i). If $\{(l_1 \leq x_i), \ldots, (l_k \leq x_i)\} \subseteq M$, then low($x_i$) is the minimum of l_j's, with the default value of $-\infty$. Similarly, if $\{(h_1 \geq x_i), \ldots, (h_k \geq x_i)\} \subseteq M$, then high($x_i$) is the maximum of h_j's, with the default value of $+\infty$.

Definition 15 *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 16 *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) $g > 0$: $C \equiv (x_1 \leq h/g)$; high(x_1) can be updated to h/g.
(d) $g < 0$: $C \equiv (x_1 \geq h/g)$; 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 [Sch] 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 stage i is denoted by M_i, where $M_1 = M$. At stage i, the i'th member of $X - S$ is eliminated; call it x_i:

1. All constraints in which x_i participates are partitioned into two sets, H and L. H contains the constraints of the form $x_i \leq h$, and L contains those of the form $x_i \geq l$, 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. The computation proceeds by eliminating, one by one, all variables that are in $X - S$, while updating

bounds and removing redundant constraints at each stage. If at any point we obtain a node whose lower bound exceeds its upper bound, M is known to be unsatisfiable.

Here is the procedure for eliminating variable x_i:

1. 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

2. 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 $f(h_i)$.

3. For each edge e leaving x_i (labeled f), where x_k is the tail of e:

   if $l_i \geq f(l_k)$, replace l_k by $f^{-1}(l_i)$

4. Remove from $\mathcal{M}$ the node x_i and all edges adjacent on x_i. □

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 stage of the node elimination algorithm, while preserving network equivalence.

This is similar to the approach of [HN94], where the Fourier-Motzkin algorithm is modified to solve the feasibility problem efficiently. There, the *equisatisfiability* of the constraint sets is preserved at each stage; 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.

5.2 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 stage of variable elimination, are used to reject most of the new edges produced during the

stage. This modification to the variable elimination algorithm allows us to achieve strongly polynomial performance.

Since there is a one-to-one correspondence between binary monotone constraints and edges in a monotone network, we will use these terms interchangeably.

5.2.1 Domains and Ranges of Monotone Constraints

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 5.1; $|\mathcal{E}| = k > 0$. The labels of the edges are linear functions $\{f_1, \ldots, f_k\}$. 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 17 *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 there are no values on which f_i is minimal, the domain of e_i is empty (denoted by ϵ). 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.

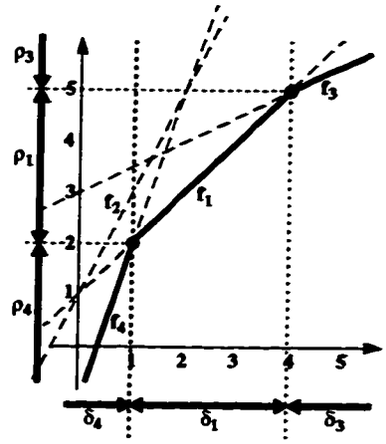
Lemma 4 *The non-trivial domains for the edges in $\mathcal{E}$ form a set of consecutive intervals covering the whole x_2-axis.*

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 18 *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 range(e_1) < range(e_2) if and only if domain(e_1) < domain(e_2), where $'>'$ designates a natural ordering of intervals (Figure 5.2).



(A)

Edge	Constraint	Range	Domain
e_4	$(x_1 \leq x_2 + 1)$	$[-\infty, 1]$	$[-\infty, 2]$
e_1	$(x_1 \leq 2x_2 + 1)$	$[1, 4]$	$[2, 5]$
e_3	$(x_1 \leq .5x_2 + 3)$	$[4, +\infty]$	$[5, +\infty]$
e_2	$(x_1 \leq 3x_2 - 1)$	ε	ε

(B)

Figure 5.2: Domains and ranges: graphical (A) and tabular (B) representations.

5.2.2 Redundant Constraints

When we have an edge e in $\mathcal{M}$ whose domain is trivial, e corresponds to a redundant constraint:

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 en 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 en 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 $\{v_1, v2, u_1, u_2\}$. 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. □

5.2.3 Monotone Projection Algorithm

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

Algorithm II. 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. The computation proceeds by eliminating, one by one, all variables that are in $X - S$, while updating bounds and removing redundant constraints at each stage. If at any point we obtain a node whose lower bound exceeds its upper bound, M is known to be unsatisfiable.

Here is the procedure for 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, *such that the intersection of domain(e_j) and 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_k is the tail of e:

   if $l_i \geq f(l_k)$, replace l_k by $f^{-1}(l_i)$

5. Remove from $\mathcal{M}$ the node x_i and all edges adjacent on x_i. □

Except for the computation in Step 1 and the check of domains and ranges in Step 2, this algorithm is the same as Algorithm I. This, combined with Proposition 1 and Corollary 1, gives us the following theorem:

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

Proof: Algorithm II produces the same constraints as Algorithm I, except for the constraints that result from the composition of two edges whose domain and range respectively do not intersect. This, combined with Proposition 1 and Corollary 1, gives us the theorem. □

5.3 Strongly Polynomial Time Complexity

In this section, we show that the time complexity of monotone projections is strongly polynomial. Furthermore, we show that the time complexity of computing *globally consistent* sets of monotone constraints is also strongly polynomial.

5.3.1 Time Complexity of Eliminating One Variable

In this subsection, we consider the time complexity of performing one stage of Algorithm II in Section 5.2.3. We assume that we are eliminating x_i, and that there are n edges incident on x_i. By making use of *convex hulls* algorithms from computational geometry [PS], we show that the time complexity is $O(n \log n)$.

We start by observing the following:

> *the non-trivial edges of a given cluster correspond to the facets of the convex hull for the set of all points defined by the cluster.*

Therefore, the problem of determining which of the edges in a cluster are non-trivial is equivalent to the problem of computing the convex hull for a set of points defined as an intersection of half-planes. For k half-planes, such algorithms have time complexity $O(k \log k)$ [PS].

Each facet of the convex hull is defined by the pair of vertices that bound it. Given a non-trivial edge, its domain and range can be obtained directly from the coordinates of the vertices bounding the corresponding facet (see Figure 5.1). This gives us the following:

Proposition 2 *In Step 1 of Algorithm II, the domains and ranges can be computed in time $O(n \log n)$.*

The set of domains of all edges entering x_i (of size k_1) can be sorted in time $O(k_1 \log k_1)$; similarly with the set of ranges of all edges leaving x_i (of size k_2). It remains to check which of the domain-range pairs have a non-trivial intersection, which is equivalent to a *merge* of the two sorted lists, and can be done in $O(k_1 + k_2) = O(n)$ time. This gives us the following:

Proposition 3 *In Step 2 of Algorithm II, the non-redundant edges can be determined in time $O(n \log n)$.*

We are now ready for the main result:

Lemma 6 *The time complexity of eliminating a node with n incident edges is $O(n \log n)$.*

Proof: It is clear that the time complexity of Steps 3 through 5 is $O(n)$. By summing the time complexities of all steps (Propositions 2 and 3), we obtain the desired result. □

5.3.2 Time Complexity of Monotone Projection

Algorithm II finds the projection of M onto some subset S of its variables. This is equivalent to finding some set of simple *non-trivial* paths in $\mathcal{M}$;

Definition 19 *Let p be a path in $\mathcal{M}$; $p = (e_1 \ldots e_k)$. The domain of p is defined as the domain of $e_1 \otimes \ldots \otimes e_k$; similarly, the range of p is defined as the range of $e_1 \otimes \ldots \otimes e_k$. We say that p is non-trivial iff its domain is non-trivial.*

In this subsection, we show the connection between simple non-trivial paths and *path expressions* [Tar81a, Tar81b]. We start with an introduction to path expressions [Tar81a, Tar81b].

Let $\mathcal{M} = (X, E)$ be a directed multigraph; any path in $\mathcal{M}$ can be regarded as a string over the alphabet E. The set of all paths in $\mathcal{M}$ forms a regular language (see [AHU] for an introduction to regular languages). The *path expression* over $\mathcal{M}$ is a regular expression that defines this regular language; its *size* is the length of the expression, with all subexpressions unfolded.

Lemma 7 *Let $\mathcal{P}$ be the path expression over $\mathcal{M}$. Then, the number of simple non-trivial paths in $\mathcal{M}$ is not greater than the size of $\mathcal{P}$.*

Proof: Let $paths(\mathcal{P})$ represent the set of simple non-trivial paths in the language of $\mathcal{P}$; let $size(\mathcal{P})$ represent the size of $\mathcal{P}$. We claim that $|paths(\mathcal{P})| \leq size(\mathcal{P})$; the proof is by structural recursion on $\mathcal{P}$.
Case 1 ($\mathcal{P} = e_i$, for some edge e_i in $\mathcal{M}$): $size(\mathcal{P}) = 1$, and $paths(\mathcal{P}) = \{e_i\}$.
Therefore, $|paths(\mathcal{P})| = 1$, so the claim holds.
Case 2 ($\mathcal{P} = \mathcal{P}_1^*$, for some regular expression $\mathcal{P}_1$): $size(\mathcal{P}) = 1 + size(\mathcal{P}_1)$, and $paths(\mathcal{P}) = paths(\mathcal{P}_1)$ (since no path containing a loop can be simple).
Therefore, $|paths(\mathcal{P})| = |paths(\mathcal{P}_1)|$. By the inductive assumption, $|paths(\mathcal{P}_1)| \leq size(\mathcal{P}_1)$, so the claim holds.

Case 3 ($\mathcal{P} = \mathcal{P}_1 \cup \mathcal{P}_2$, for some regular expressions $\mathcal{P}_1$ and $\mathcal{P}_2$): $size(\mathcal{P}) = 1 + size(\mathcal{P}_1) + size(\mathcal{P}_2)$, and $paths(\mathcal{P}) = paths(\mathcal{P}_1) \cup paths(\mathcal{P}_2)$.
Therefore, $|paths(\mathcal{P})| \leq |paths(\mathcal{P}_1)| + |paths(\mathcal{P}_2)|$. By the inductive assumption, $|paths(\mathcal{P}_1)| \leq |size(\mathcal{P}_1)|$ and $paths(\mathcal{P}_2) \leq size(\mathcal{P}_2)$, so the claim holds.
Case 4 ($\mathcal{P} = \mathcal{P}_1 \cdot \mathcal{P}_2$, for some regular expressions $\mathcal{P}_1$ and $\mathcal{P}_2$): $size(\mathcal{P}) = 1 + size(\mathcal{P}_1) + size(\mathcal{P}_2)$, and $paths(\mathcal{P}) \subseteq \{p_1 p_2 : p_1 \in paths(\mathcal{P}_1), p_2 \in paths(\mathcal{P}_2)$, and the intersection of the domain of p_1 and the range of p_2 is not trivial$\}$.
By the definition of path expressions, all paths in $\mathcal{P}_1$ share the same starting node; they also share the same ending node. The same is true of the paths in $\mathcal{P}_2$, where the starting node of $\mathcal{P}_2$ is the ending node of $\mathcal{P}_1$. Therefore, by Lemma 3, $|paths(\mathcal{P})| \leq |paths(\mathcal{P}_1)| + |paths(\mathcal{P}_2)|$. By the inductive assumption, $|paths(\mathcal{P}_1)| \leq size(\mathcal{P}_1)$ and $|paths(\mathcal{P}_2)| \leq size(\mathcal{P}_2)$, so the claim holds. □

It follows from Lemma 7 that the number of simple non-trivial paths in $\mathcal{M}$ is polynomial in the size of $\mathcal{M}$ whenever the size of the path expression over $\mathcal{M}$ is polynomial. We are now ready to prove the following theorem:

Theorem 5 *The performance of Algorithm II is strongly polynomial in the size of M whenever the size of the path expression over the corresponding monotone network is polynomial in the size of M.*

Proof: Let the constant Q represent the number of simple non-trivial paths in $\mathcal{M}$. The number of edges incident on any one node at any stage of Algorithm II will never exceed Q. Therefore, from Lemma 6, the time complexity of each stage of Algorithm II is $O(Q \log Q)$.

Algorithm II consists of $O(k)$ such stages, where k is the number of variables in X. Therefore, its time complexity is $O(kQ \log Q)$, which is polynomial in the size of $\mathcal{M}$. □

5.3.3 Globally Consistent Constraint Sets

We have shown in the previous section that computing projections of monotone constraint sets has polynomial time complexity, 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 8) 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 7), there may be different ways to represent $\pi_S(E)$ (Definition 6). 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 8 *Let E be a set of two-variable constraints, and E' be constructed from E as follows:*

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

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 $C \in \pi_{x_1 x_2}(E)$. Let p be an arbitrary assignment satisfying E; by definition, the restriction of p to $\{x_1, x_2\}$ satisfies $\pi_{x_1 x_2}(E)$. 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 $\pi_{x_1 x_2}(E)$. 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 $\pi_{x_1 x_2}(E)$; 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 $\pi_S(E')$. □

Note that Lemma 8 is very general; it holds for *all* sets of 2-variable monotone constraints, not just monotone ones. The same is not true of the final theorem of this chapter:

Theorem 6 *For any monotone two-variable linear constraint set with a polynomial-size path expression, 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 8, the union of the resulting sets is globally consistent. □

Chapter 6

Variable Independence and Aggregation Closure

A query language resulting from adding aggregation to a constraint query calculus [KKR95] or algebra [GK96] should be well behaved. However, unrestricted use of aggregation may fail to produce a closed language, as shown in [Kup94].

In this section, we approach this problem by restricting the way aggregate operators are used in constraint queries. We show that under certain natural restrictions, captured by the notion of *variable independence* and reflecting the way aggregation is often used in real databases, we can add aggregate operators to relational algebra, and still get a closed language.

It is an open issue whether *variable independence* can also be defined for other query languages with aggregation (e.g., relational calculus, Datalog [MPR90, RS92, SSRB93, VG92, Rev95]). Also, it is possible that the *variable independence* can be applied to provide closure conditions for *CQA+fixpoint*.

6.1 Background

As shown in [Kup94], relational algebra over linear constraint databases is not closed under aggregation using **area**. The typical example where a query is not closed is where we have a region whose boundaries vary with time, and we want to know how the area of the region varies with time. While there are applications where one might conceivably need the full generality of such a language (and for which the *closure* problem would

Name	Time	Land owned
Bob	$t_1<t<t_2$	$C_1(x, y)$
Bob	$t_3<t<t_4$	$C_2(x, y)$
Joe	$t_5<t<t_6$	$C_3(x, y)$
Ann	$t_7<t<t_8$	$C_4(x, y)$

Figure 6.1: A geographical database with cadastral information.

be unavoidable), this is not the case for many applications.

Consider a geographical database with *cadastral* information, i.e., information on land ownership and land boundaries. Suppose we are interested in finding the area of the land owned by each person at all points of time. Land ownership does not vary continuously–pieces of land are acquired by individuals at single, discrete points of time. As a result, the attributes of the relation can be grouped into 'independent' subsets.

Example 7 *The constraint relation in Figure 6.1 can be represented by a set of generalized tuples of the form*

$$n = N \wedge t_1 \leq t \leq t_2 \wedge C(x, y)$$

where $C(x, y)$ are constraints that describe the region owned by N between times t_1 and t_2. If we want to find the area of the land owned by N as a function of t, we can represent the result as a set of generalized tuple of the form

$$t_1 < t < t_2 \wedge (n = N) \wedge (z = A)$$

which is clearly finitely representable. □

The important property of the tuples in the above example is that the constraints on x and y — those on which the area computation is performed — are separate from the constraints on n and t. Many typical uses of area computation in GIS systems have this property, which we will define as *variable independence.*

This chapter is organized as follows. In section 6.2, we review the basic concepts of *aggregation.* In section 6.3, we define the notion of *variable independence* that is central to the notion of restricted aggregation. In section 6.5, we define the relational algebra

with restricted aggregation and show that it is closed. In section 6.7, we present some results about inferring variable independence in relational algebra expressions. The extended abstract of the work presented here, written with Gabi Kuper and Jan Chomicki, has appeared in [CGK96].

6.2 Aggregation in Constraint Databases

We view a set of variables $\{x_1, \ldots, x_k\}$ as a *relation schema* and generalized relations over $\{x_1, \ldots, x_k\}$ as *instances* of this schema.

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

Definition 20 *Let Φ be a class of constraints interpreted over domain D, r a generalized relation of arity k with constraints in Φ, let ϕ_r be the formula corresponding to r with free variables $x_1, \ldots, x_k$. The generalized relation r represents the unrestricted k-ary relation which consists of all $(a_1, \ldots, a_k)$ in D^k such that $\phi_r(a_1, \ldots, a_k)$ is true. Two generalized relations over the same set of variables are* equivalent *if they represent the same unrestricted relation.* □

With some abuse of notation, we will use the same symbol for a generalized tuple (relation) and the unrestricted relation it represents.

A query ϕ on a constraint database is a first-order formula, whose predicates are constraints or generalized relation symbols. The semantics of a query, are, intuitively, the mapping from unrestricted relations to unrestricted relations defined by this formula. Such a query is well-defined if the result of substituting a generalized relation for each occurrence of the corresponding relation symbol, is equivalent to some generalized relation over the same constraint domain.

A query language L is *closed* for a class of constraint databases C if the result of any query belonging to L evaluated over a database from C can be finitely represented as a generalized relation from C.

We assume that the constraint language is closed under negation. Moreover, it should admit effective quantifier elimination. These assumptions are necessary for the closure of relational algebra operations. For the rest of this section, we consider only those constraint languages that satisfy the above conditions. In particular, we concentrate on *dense order* and *linear arithmetic* constraints; the latter are of the form $a_1x_1 + \cdots + a_mx_m$ *op* a_0 with *op* $\in \{\leq, <, =\}$ and with rational coefficients $a_0, \ldots, a_m$.

As shown in [Kup94], Klug's relational calculus and algebra with aggregation can be extended to constraint databases with minor modifications, at least as far as the underlying semantics on unrestricted relations is concerned. The definitions in [Kup94] are as follows:

Definition 21 *An* aggregate function f *maps (possibly infinite) relations with an appropriate schema to the domain D of the constraints. For every relation S over attributes X, if S' is a constant expansion of S over $X \cup Y$, i.e., if there is a projection such that $\pi_X(S') = S$ and $\pi_Y(S')$ contains exactly one tuple, then the function f' such that $f(S) = f'(S')$ is also an aggregate function.*

For our purposes here, it is sufficient to assume that the algebra is extended by the following operator for every aggregate function:

Definition 22 *Let r be an unrestricted relation over the set of attributes U, $X \subseteq U$ such that $|U - X| = n$, and f is an aggregate function of arity n which outputs a constraint over attributes Y. Then, the aggregate operator $\langle X, f \rangle$, when applied to r, produces a new relation r' with attributes $X \cup Y$:*

$$r\,\langle X, f\rangle = \{(t[X], y) | t \in r \wedge$$
$$\wedge y = f(\{t'[U - X] | t' \in r \wedge t'[X] = t[X]\})\}.$$

Intuitively, the above corresponds to grouping the tuples in r on X and applying the aggregation operator to the remaining attributes in each group. For more details, including the construction of an equivalent calculus, see [Kup94].

Generalized relational algebra with aggregation may fail to be closed even for dense order constraints [Kup94].

Example 8 *Consider the instance of R in Figure 2.1, consisting of a single generalized tuple:*

$$0 < x < y < z < 1.$$

The query $R\langle z, \mathbf{area}_{x,y}\rangle$ *evaluates to a binary relation* $S(z,a)$ *where* $S(z,a)$ *holds iff* $a = 0.5z^2$. *The latter constraint cannot be represented using order constraints (or linear arithmetic constraints).*

We propose to provide a restriction on applying the aggregate operator to a relation, and show closure for the resulting class of relational expressions. In particular, we will define the notion of *variable independence*, and stipulate that X and $U - X$ must be independent in r (see Definition 22). So, the expression $R\langle z, \mathbf{area}_{x,y}\rangle$ from the above example would not be permitted by our restriction.

It should be noted that the relation r in Definition 22 can be either extensional or intensional. We show how to maintain the restriction in both of those cases.

6.3 Independence of Variables

In this section, we define *variable independence* for generalized relations as well as for relational schemas. In the following definitions, we assume that R is a generalized relation over attribute set U.

Definition 23 *Let* $X, Y \subseteq U$ *and* t *be a generalized tuple in* R. *We say that* X *and* Y *are* independent in t *if:*

$$\pi_{XY}(t) = \pi_X(t) \bowtie \pi_Y(t);$$

they are related in t *otherwise.*

Clearly, variable independence in a tuple is decidable. Also, note that variable independence in a tuple t may be viewed as an *embedded join dependency* [Kan90] holding in the unrestricted relation corresponding to t.

Definition 24 *We say that* X *and* Y *are* independent in R *if there exists a relation* R' *equivalent to* R *where* X *and* Y *are independent in every (generalized) tuple of* R'. *Otherwise, we say that they are* related in R.

Example 9 *We return to the generalized relation in Figure 6.1, consisting of tuples of the following form:*

$$t_1 < t < t_2 \wedge (n = N) \wedge C(x,y)$$

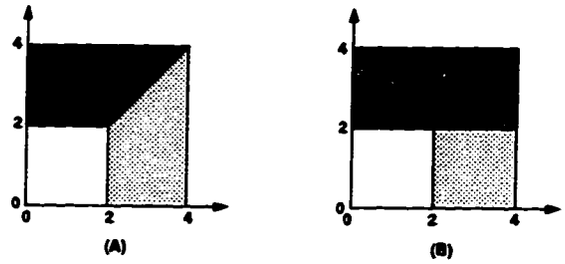


Figure 6.2: Instance (A) is equivalent to instance (B).

where $C(x, y)$ is a conjunction of linear arithmetic constraints describing a piece of land owned by person N over the time interval (t_1, t_2). In this relation, the attributes sets $\{N\}$, $\{t\}$, $\{x, y\}$ are all independent.

Note that for classical relations, where each tuple always represents exactly one point, it is *trivially true* that all subsets of variables are independent in every relation. In general, however, *decidability of variable independence* is far from obvious. As the next example shows, it is not sufficient to check the individual tuples in order to verify variable independence. In Section 6.6, we show that for linear constraints, there is an effective method to tell whether two variables are independent in an arbitrary generalized relation.

Example 10 *The instance of $R(x, y)$ in Figure 6.2 (A) contains two tuples:*

$$2 \leq y \leq 4 \wedge 0 \leq x \leq y;$$
$$2 \leq x \leq 4 \wedge 0 \leq y \leq x.$$

In each tuple, x and y are related. However, there is an an equivalent relation (Figure 6.2 (B)) where this is not the case:

$$2 \leq y \leq 4 \wedge 0 \leq x \leq 4;$$
$$2 \leq x \leq 4 \wedge 0 \leq y \leq 2.$$

To conclude the section, we define variable independence for the *schema* of a generalized relation. Here, variable independence is viewed as a restriction, to be satisfied by all relations that satisfy the schema. In this way, we *enrich* the relation schema beyond the standard attribute name and type information.

Definition 25 *We say X and Y are* independent in the relational schema $\mathcal{R}$ *if every relation satisfying $\mathcal{R}$ preserves the independence of X and Y; X and Y are* related in $\mathcal{R}$ *otherwise.*

In Section 6.6, we show how to test whether a generalized relation satisfies the independence restrictions on its schema. This test, performed upon relation updates, ensures that we *maintain* the independence restrictions on the schema of all relations in the constraint database.

6.4 Semantic vs. Syntactic Independence

The above definitions of variable independence are *semantic*: for each generalized tuple t, they consider the semantics of t. One can also define variable independence *syntactically*:

Definition 26 *Let $X, Y \subseteq U$ be disjoint, and t be a generalized tuple in R. We say that X and Y are* syntactically independent in t *if t is a conjunction of $C_1(X \cup Z)$ and $C_2(Y \cup Z)$, where $Z = U - X - Y$, and $C(X)$ denotes a conjunction of constraints over X.*

The above definition extends to relations, just as for semantic variable independence.

In Section 6.5, we will impose a variable independence condition on the schema of those relations to which aggregation may be applied, and show that for all reasonable constraint classes, relational algebra with *restricted aggregation* is closed. The proof will rely on the following lemma:

Lemma 9 *X and $U - X$ are semantically independent in R iff they are syntactically independent in R.*

Proof: The "if" direction is straightforward. We prove the "only if" direction. By Definition 24, there exists R' equivalent to R where for every tuple t in R', $t = \pi_X(t) \bowtie \pi_{U-X}(t)$. Since the constraint class admits quantifier elimination, both projections are representable as finite disjunctions of conjunctions of constraints; the first one is over X, the second over $U - X$. By pairing up the disjuncts from the two projections, we obtain a finite set of tuples whose disjunction is equivalent to t, where X and $U - X$ are syntactically independent in all tuples. By repeating this construction for every t in R', we obtain a relation equivalent to R where X and $U - X$ are syntactically independent in each tuple. □

Note that the above lemma does not generalize to arbitrary sets of attributes; in general, semantic independence does *not* imply syntactic and vice versa. For example, $\{x\}$ and $\{y\}$ are semantically (but not syntactically) independent in the singleton relation consisting of the tuple

$$0 < x < 1 \wedge 0 < y < 1 \wedge z = x + y.$$

On the other hand, $\{x\}$ and $\{y\}$ are syntactically (but not semantically) independent in the relation consisting of the tuple

$$x < y \wedge y < z.$$

In the rest of this chapter, *independence* will mean semantic independence, unless explicitly stated otherwise.

6.5 Restricted Aggregation

In this section, we impose a variable independence condition on the schema of those relations to which aggregation is applied. We show that for all reasonable constraint classes, relational algebra with *restricted aggregation* is *closed*; i.e., the result of any relational algebra expression can be represented as a generalized relation.

Definition 27 *The* relational algebra with restricted aggregation *consists of the standard relational algebra, together with expressions of the form* $e\,\langle X, f\rangle$, *provided that* X *and* $U - X$ *are independent in the schema of* e *(where* U *is the set of attributes of* e*).*

Example 11 *Let* L *be the cadastral relation in Figure 6.1. The relational algebra query*

$$L\,\langle\{1,2\}, \mathbf{area}_{3,4}\rangle$$

lists the area of land owned by each person at all times. Since the attributes sets $\{N\}$, $\{t\}$, $\{x, y\}$ *are all independent, this query satisfies our restriction on the use of aggregation.*

Theorem 7 *Relational algebra with restricted aggregation is closed for constraint databases whenever the constraint class admits quantifier elimination and is closed under negation.*

Proof: Let e be a relational expression on the given schema, with attributes U, and let X be a subset of U such that the expression $e\langle X, f\rangle$ is permitted in the language, and let e evaluate to the relation R_0. We must show that $R_0\langle X, f\rangle$ is representable as a generalized relation in our constraint language.

By our restrictions on the use of aggregation, combined with Lemma 9, there must be R equivalent to R_0, of the form

$$R \equiv \bigvee_{1 \le i \le l} (C^i \wedge D^i)$$

where each C^i (D^i) is a conjunction of constraints containing only variables from X ($U-X$). For each i, let $C^i = c^i_1 \wedge \cdots \wedge c^i_{j_i}$. Let $\mathcal{C} = \{c^i_k, \neg c^i_k : 1 \le i \le l, 1 \le k \le j_i\}$. Since the constraint language is closed under negation, $\mathcal{C}$ is a valid constraint set (containing all constraints over X in R, and their negations). We refer to a non-empty subset of $\mathcal{C}$ as a $\mathcal{C}$-set; for any $\mathcal{C}$-set S, we denote the conjunction of its constraints by ϕ_S.

A $\mathcal{C}$-set S is *minimal* if it is satisfiable and there does not exist a satisfiable S' such that $S \not\equiv S'$ and $\phi_{S'} \rightarrow \phi_S$ (clearly, every satisfiable $\mathcal{C}$-set is either minimal or is implied by some minimal one). Note that the models of minimal $\mathcal{C}$-sets are disjoint: suppose that some assignment x satisfies two $\mathcal{C}$-sets S_1 and S_2 (where $\phi_{S_1} \not\equiv \phi_{S_2}$), and let $S_3 = S_1 \cup S_2$. We know that S_3 is satisfiable, $\phi_{S_3} \rightarrow \phi_{S_1}$, and $\phi_{S_3} \rightarrow \phi_{S_2}$. Since ϕ_{S_3} cannot be equivalent to both ϕ_{S_1} and ϕ_{S_2}, at least one of $\{S_1, S_2\}$ is not minimal. Further, note that any assignment $\overline{x}$ satisfying some C^i also satisfies some minimal $\mathcal{C}$-set. If we suppose otherwise, then each minimal $\mathcal{C}$-set contains a constraint falsified by $\overline{x}$; let S_0 be the set of the negations of all such constraints (S_0 is a $\mathcal{C}$-set satisfied by $\overline{x}$). S_0 is not equivalent to any minimal $\mathcal{C}$-set, and is not implied by any minimal $\mathcal{C}$-set; this is impossible.

A $\mathcal{C}$-set S is *relevant* if it is minimal and there exists C^i such that $\phi_S \rightarrow C^i$. Given a relevant $\mathcal{C}$-set S, we denote by A_S the set of assignments to $U-X$ that satisfies some D^i such that $\phi_S \rightarrow C^i$. We now define the following relation R', over variables $X \cup Y$:

$$R' = \{(\overline{x}, \overline{y}) : \overline{x} \text{ satisfies some relevant } S, \overline{y} = f(A_S)\}.$$

Clearly, R' is finitely representable:

$$R' \equiv \bigvee_{S \text{ relevant}} \phi_S \wedge (\overline{y} = f(A_S)).$$

We complete the proof by showing that R' is equivalent to $R\langle X, f\rangle$:

$$R' \equiv \{(\overline{x}, \overline{y}) : \overline{x} = t[X] \text{ for some } t \in R, \overline{y} = f(G_{\overline{x}})\},$$

where $G_{\overline{x}} = \{t[U - X] : t \in R, \overline{x} = t[X]\}$. This equivalence follows from the following facts:

(a) $\overline{x}$ satisfies some relevant S iff $\overline{x} = t[X]$ for some $t \in R$;

(b) given $\overline{x}$ satisfying some relevant S, $f(A_S) = f(G_{\overline{x}})$.

□

Note: The above proof is different from the original proof in [CGK96], which was due to G. Kuper.

To make restricted aggregation practical, we must be able to determine effectively, for a generalized database $\{R_1, \ldots, R_n\}$ and a relational expression e $\langle X, f \rangle$, whether X and $U - X$ are independent in e. If e is some database relation R_i, this is accomplished by stipulating that X and $U - X$ are independent in R_i's schema (see Section 6.3). Otherwise, if e is an arbitrary relational expression, variable independence can be inferred at query compile-time (see Section 6.7), under the assumption that all schema restriction for $\{R_1, \ldots, R_n\}$ are properly maintained.

Note that the above is accomplished without affecting the run-time query performance; as a result, we believe that restricted aggregation is a very promising approach to assuring the closure of queries with aggregation.

6.6 Testing variable independence

As mentioned at the end of Section 6.3, given a relation R whose schema restricts X and Y to be independent, we must be able to test whether R satisfies the restriction. Clearly, if X and Y are independent in each tuple of R, the answer is positive. Therefore, one way of enforcing independence restrictions on R's schema is by stipulating that each tuple satisfies these restrictions.

To allow the user maximum flexibility, it is desirable to have an algorithm for testing, given an *arbitrary* R with attribute set U, and arbitrary $X, Y \subseteq U$, whether X and Y are independent in R. In this section, we provide such an *independence test* for linear constraint databases. As a side effect, this test generates on success a relation $\overline{R}$ equivalent to R where X and Y are independent for all tuples.

Unlike the work of [Las90], where all tests are performed on constraint sets (i.e., tuples), this test is for disjunctions of constraint sets (i.e., relations). The independence test consists of three steps:

Step 1. The first step of the test is to create the *boundary representation* $B(R)$ for the polyhedral object consisting of the *feasible points* of R, i.e., the points whose coordinates have values satisfying R's formula ϕ_R. This is done using standard techniques from CAD, summarized below.

Each generalized tuple t corresponds to a convex set P_t of n-dimensional points, bounded by n-dimensional half-planes. The boundary representation of P_t is computed by a convex hull algorithm, such as the "gift wrapping" method in [CK70].

$B(R)$ is obtained by *unioning* together the boundary representations of the individual tuples, a standard Solid Modeling operation. See [FvDFH] for an introduction to polyhedral Solid Modeling, or [MM] for details of the algorithms. These algorithms extend to an arbitrary number of dimensions, as in [PS86].

Step 2. The next step of the algorithm is to create a *vertex grid partitioning* $\overline{R}$ of R.

We project the set of $B(R)$'s vertices onto each of the n axes, and sort the k_j obtained values, for $1 \leq j \leq n$. This partitions each axis into $(k_j + 1)$ intervals:

$$\{(-\infty, v_1), (v_1, v_2), \ldots, (v_{k_j}, +\infty)\}.$$

In turn, this induces a partitioning of the n-space into $((k_1 + 1) \times \cdots \times (k_n + 1))$ n-dimensional rectangles, where each rectangle is the cross-product of the corresponding intervals. We denote this set of rectangles by V_R, the *vertex grid* of R.

The intersection of each rectangle in V_R with $B(R)$ consists of 0 or more disjoint point sets. These point sets are convex, since no rectangle can contain a vertex of $B(R)$ as its interior point; therefore, each one corresponds to some tuple t_j. The set of these tuples, denoted by $\overline{R}$, is the *vertex grid partitioning* of R. It is equivalent to R.

Step 3. The last step of the algorithm is to check, for each t in $\overline{R}$, whether X and Y are independent in t.

The following theorem provides the motivation for the *independence test*:

Theorem 8 *X and Y are independent in R if and only if, for all tuples t in $\overline{R}$, X and Y are independent in t.*

Proof: (Sketch) The "if" direction of the theorem follows from definitions. For the other direction, we assume that X and Y are independent in R. Let R' be equivalent to R where X and Y are independent for each tuple $t \in R'$. Let $\overline{R}'$ be the intersection

of the tuples of R' with the vertex grid of R':

$$\overline{R'} = \{t_j \cap t_k : t_j \in R', t_k \in V_{R'}\}.$$

It can be shown that X and Y are independent for each tuple in $\overline{R'}$. It can also be shown that for each tuple t in the vertex grid partitioning of R, there exists a subset of $\overline{R'}$ whose union is equivalent to t. This implies that X and Y are independent in t, completing the proof. □

We conclude the chapter with a short discussion of *complexity* issues. It can be shown that the data complexity of the independence test is polynomial. However, the combined complexity is exponential in n. This is due to the fact that the size of $B(R)$ can be exponential in n:

Example 12 *Consider a relation R over n variables consisting of one tuple with the following constraints:*

$$0 \leq x_1 \leq 1, \ldots, 0 \leq x_n \leq 1.$$

Its feasible points form a hypercube with 2^n vertices and $O(2^n)$ edges.

It is our opinion that the exponential combined complexity is unavoidable, since any independence test will have to somehow consider the complete geometry of the feasible points of R.

6.7 Inference of variable independence

In this section we show how to infer variable independence in relational algebra expressions. In contrast to the previous section, the results in this section are applicable to any constraint language closed under negation and admitting quantifier elimination, not just linear arithmetic constraints. The notation and the inference rules presented below are from [CGK96], due to Jan Chomicki.

Notation:

- U denotes the schema of the generalized relation R,
- $R : \text{Indep}(X, Y)$ denotes the fact that X and Y are independent in R,
- $\beta(A_1, \ldots, A_k)$ denotes a constraint with free variables $A_1, \ldots, A_k$.

Theorem 9 *The following inference rules are valid:*

1. *if* $X \subseteq U$, *then* $\sigma_{A=a}(R) : \text{Indep}(\{A\}, X)$ *and* $\sigma_{A=a}(R) : \text{Indep}(X, \{A\})$;
2. *if* $R : \text{Indep}(X, Y)$ *and* ($\{A_1, \ldots, A_k\} \subseteq X$ *or* $\{A_1, \ldots, A_k\} \subseteq Y$), *then* $\sigma_{\beta(A_1,\ldots,A_k)}(R) : \text{Indep}(X, Y)$;
3. *if* $R : \text{Indep}(X, Y)$ *and* $X, Y \subseteq Z$, *then* $\pi_Z(R) : \text{Indep}(X, Y)$;
4. *if* $R : \text{Indep}(X, Y)$ *and* $S : \text{Indep}(X, Y)$, *then* $R \cup S : \text{Indep}(X, Y)$;
5. *if* $R : \text{Indep}(X, Y)$, *then* $R \times S : \text{Indep}(X, Y)$ *(similarly for* S*)*;
6. *if* S *has schema* U' *disjoint from the schema of* R *and* $X \subseteq U$ *and* $Y \subseteq U'$, *then* $R \times S : \text{Indep}(X, Y)$ *and* $R \times S : \text{Indep}(Y, X)$;
7. *if* $R : \text{Indep}(X, Y)$ *and* $X, Y \subseteq Z$ *then* $R\langle Z, f\rangle : \text{Indep}(X, Y)$;
8. *if* $Y \subseteq X$ *and* m *is the new column corresponding to the result of the aggregation, then* $R\langle X, f\rangle : \text{Indep}(Y, \{m\})$ *and* $R\langle X, f\rangle : \text{Indep}(\{m\}, Y)$.

Proof: We prove the validity the second inference rule. The validity of the remaining rules can be established in a similar way.

The basic idea is as follows: given a representation for R in which X and Y are independent, we construct from it a representation for $\sigma_{\beta(A_1,\ldots,A_k)}(R)$ in which X and Y are independent. In this case, we conjoin every (generalized) tuple t of R with $\sigma_{\beta(A_1,\ldots,A_k)}(R)$ to obtain another generalized tuple t'. We have to show that

$$\pi_{XY}(t') = \pi_X(t') \bowtie \pi_Y(t').$$

The fact that the left-hand side is contained in the right-hand side follows directly from the definition of projection and join. To obtain the containment in the other direction, let p be a (ground) tuple in $\pi_X(t') \bowtie \pi_Y(t')$. Thus, there are tuples $p' \in t'$ and $p'' \in t'$ such that:

- $p'[X] = p[X]$ and $\beta(p'[A_1], \ldots, p'[A_k])$ holds,
- $p''[Y] = p[Y]$ and $\beta(p''[A_1], \ldots, p''[A_k])$ holds,
- $p'[X \cap Y] = p''[X \cap Y]$.

Now clearly $p \in \pi_X(t) \bowtie \pi_Y(t)$. Because

$$\pi_{XY}(t) = \pi_X(t) \bowtie \pi_Y(t)$$

$p \in \pi_{XY}(t)$. But also $\beta(p[A_1], \ldots, p[A_k])$ holds (notice the importance of $A_1, \ldots, A_k$ being entirely in X or Y) and thus $p \in \pi_{XY}(t')$. □

Note that in general, we cannot infer variable independence in a *difference* of two relations.

Example 13 *Consider the instance I_0 with schema $\{x, y, z\}$:*

$$y > z$$
$$x < y.$$

Its complement consists of the tuple

$$z \leq y \wedge y \leq x.$$

Thus while x and z are independent in I_0, they are not independent in the complement of I_0.

However, for the special case of $Y = U - X$, we have:

Theorem 10 *The following inference rule is valid:*

- *if* $R : \text{Indep}(X, U - X)$ *and* $S : \text{Indep}(X, U - X)$, *then* $R - S : \text{Indep}(X, U - X)$.

Proof: (sketch) Assume I_1 is an instance of R satisfying $\text{Indep}(X, U - X)$ and I_2 is an instance of S satisfying $\text{Indep}(X, U - X)$. Then by definition there exist instances I_1' equivalent to I_1 and I_2' equivalent to I_2 consisting only of tuples in which X and $U - X$ are independent. Let $I_1' = \{t_1, \ldots, t_n\}$ and $I_2' = \{s_1, \ldots, s_m\}$. The difference of I_1' and I_2' can thus be represented as the set of tuples

$$w_i = (t_i \wedge \neg s_1 \wedge \neg s_m)$$

for $i = 1 = 1, \ldots, n$. Let $X = \{A_1, \ldots, A_k\}$ and $U - X = \{B_1, \ldots, B_m\}$. Now every s_i, $i = 1, \ldots, m$, can in turn be represented as a tuple of the form

$$\beta(A_1, \ldots, A_k) \wedge \gamma(B_1, \ldots, B_m)$$

by Lemma 9. Thus its negation is of the form

$$\neg\beta(A_1, \ldots, A_k) \vee \neg\gamma(B_1, \ldots, B_m).$$

Because of this and the fact that the constraint language is closed under negation, each tuple w_i can be represented using a finite number of tuples of the form

$$\beta(A_1, \ldots, A_k) \wedge \gamma(B_1, \ldots, B_m).$$

In every such a tuple X and $U - X$ are independent. □

The above special case is important because the application $e\,\langle X, f\rangle$ of a restricted aggregation operator in the version of relational algebra discussed in section 6.5 is allowed only if $e : \mathrm{Indep}(X, U - X)$ where U is the schema of e.

Remark: Though we have shown the above inference rules to be sound, no claim is being made about their completeness. A full treatment of variable independence should consider it as a dependency class, addressing the issues of axiomatization and implication. This is being left for future work.

Chapter 7

Similarity Queries for Time-Series Data: an Application of Multidimenstional Indexing

Constraints are a natural mechanism for the specification of similarity queries on time-series data. However, to realize the expressive power of constraint programming in this context, one must provide the matching implementation technology for efficient indexing of very large data sets. In this paper, we formalize the intuitive notions of exact and approximate similarity between time-series patterns and data. Our definition of similarity extends the distance metric used in [AFS93, FRM94] with invariance under a group of transformations. Our main observation is that the resulting, more expressive, set of constraint queries can be supported by a new indexing technique, which preserves all the desirable properties of the indexing scheme proposed in [AFS93, FRM94].

7.1 Problem Definition

7.1.1 Approximate Matching of Time-Series Data

Time-series are the principal format of data in many applications, from financial to scientific. Time-series data are sequences of real numbers representing measurements at uniformly-spaced temporal instances. The next generation of database technology, with its emphasis on multimedia, is expected to provide clean interfaces (i.e., declarative specification languages) to facilitate data mining of time-series. However, any proposal

of such linguistic facilities must be supported by indexing (i.e., be implementable with reasonable I/O efficiency) for very large data sets. Examples of recent database research towards this goal include [AFS93, FRM94, SLR94].

A most basic problem in this area is *First-Occurrence Subsequence Matching*, defined as follows: *given a query sequence Q of length n and a much longer data sequence $\overline{S}$ of length N, find the first occurrence of a contiguous subsequence within $\overline{S}$ that matches Q exactly.*

A wide range of algorithms has been developed for *internal* (i.e., in-core) versions of this question [AHO90] for strings over an alphabet or for values over bounded discrete domains. There are particularly elegant linear-time $O(n + N)$ algorithms (by Knuth-Morris-Pratt and Boyer-Moore) and practical searching utilities for more general patterns instead of query strings Q (e.g., regular patterns in grep). The part of this technology that is most related to our paper is the Rabin-Karp randomized linear-time algorithm [KR87], which provides an efficient in-core solution based on *fingerprint* functions. Fingerprints are a form of sequence hashing that allow constant-time comparisons between hash values and are incrementally computable.

A variant of the above problem involves finding all occurrences; this is called the *All-Occurrences Subsequence Matching* problem.

The above two problems have variants when the data consists of many sequences of the same length as the query. The *First(All)-Occurrence(s) Whole-Sequence Matching* problem is: *given a query sequence Q of length n and a set of N/n data sequences, all of the same length n, find the first (all) of the data sequences that match Q exactly.*

Since the size of the data in commercial applications of time-series data mining is usually too large to be stored internally, the research in various time-series matching problems has concentrated on the case when storage is *external* (i.e., secondary as opposed to in-core). Here we are interested in:

> External solutions to the All-Occurrences Matching problems (either Subsequence or Whole-Sequence), with two additional characteristics:

- the match can be *approximate*;
- the match is up to *similarity*.

We define "approximate" in this subsection and "similar" in the next.

Time-series data in continuous (e.g., real-valued) domains is inherently inexact, due to the unavoidable imprecision of measuring devices and clocking strategies. This forces us to work with the approximate version of the various matching problems.

Given a tolerance $\epsilon \geq 0$ and a distance metric D between sequences, sequences S_1 and S_2 match approximately within tolerance ϵ when $D(S_1, S_2) \leq \epsilon$.

The *All-Occurrences Approximate Matching* problems (either *Subsequence* or *Whole-Sequence*) are defined as before, but with "match approximately within tolerance ϵ" instead of "match exactly".

For external solutions to the All-Occurrence Whole-Sequence Approximate version, we refer to [AFS93, SLR94]; for the All-Occurrence Subsequence Approximate version, we refer to [FRM94].

A further characteristic of time-series data that is used to advantage, is that they have a *skewed energy spectrum*, to use the terminology borrowed from Discrete Signal Processing [DS]. As a result, most of the technology of information retrieval in this area is influenced by signal processing methods.

7.1.2 Approximately Similar Time-Series Data

The database applications of interest involve queries expressing notions of "user-perceived similarity". Here are some examples of applications for approximate time-series matching that illustrate this notion of similarity:

- find months in the last five years with sales patterns of minivans like last month's;
- find other companies whose stock price fluctuations resembles Microsoft's;
- find months in which the temperature in Paris showed patterns similar to this month's pattern.

In many cases, it is more natural to allow the matching of sequences that are not close to each other in an Euclidean sense. For example, two companies may have identical stock price fluctuations, but one's stock is worth twice as much as the other at all times. For another example, two sales patterns might be similar, even if the sales volumes are different. We shall refer to the difference between these sequences as *scale*. In another example, the temperature on two different days may start at different values but then go up and down in exactly the same way. We shall refer to the difference

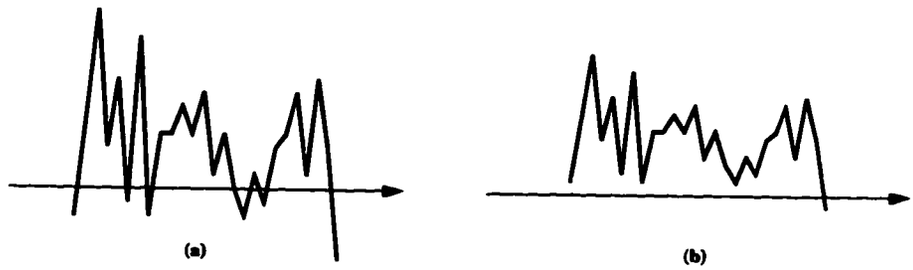


Figure 7.1: Sequence (b) is a similarity transformation of (a).

between these sequences as *shift*. A good time-series data-mining mechanism should be able to find similar sequences, as illustrated by these examples, up to scaling and shifting.

Combinations of scaling and shifting are shape-preserving transformations, known as *similarity transformations* in the mathematical field of Transformational Geometry [MP]. We will approach the definition of similarity from this well established geometrical perspective:

> Let G be a set of transformations then two sets of points are similar if there exists a transformation, in G, which maps one to the other.

In geometry, a transformation typically belongs to a group. Combinations of scale and shift are affine transformations. In practice, the user may restrict the allowable transformations to a set that may not be group, by imposing constraints on the scaling and shifting factors, or by fixing one or both factors.

Let D be a distance metric between sequences and $\epsilon \geq 0$ a tolerance. Query sequence Q is *approximately similar* within tolerance ϵ to data sequence S when there exists a similarity transformation T so that $D(Q, T(S)) \leq \epsilon$. When ϵ is set to 0, we have *exact similarity*.

Approximate and exact similarity queries can now be defined just like approximate and exact matching queries were defined in the last section.

The *All-Occurrences Approximate Similarity* problems (either *Subsequence* or *Whole-Sequence*) are defined as was Matching, but with "approximately similar within tolerance ϵ" instead of "match approximately within tolerance ϵ". Analogous definitions apply for *All-Occurrences Exact Similarity* problems (either *Subsequence* or *Whole-Sequence*).

When T is restricted to be the identity transformation, the various similarity problems become the matching problems of the last section. In this sense, our work is a generalization of the work of [FRM94]. This generalization is in the direction of [Jag91], which discusses translation and distortion transformations but does not provide the guarantees of [FRM94] and of our indexing scheme.

A general framework for similarity queries is described in [JMM95]. Our work happens to be (an efficiently solvable) special case. The [JMM95] framework for similarity-based queries has three components: a pattern language P, an approximation language (they refer to it as transformation rule language), and a query language. In our case, P is the set of allowable transformations on the query sequence Q. An expression in P specifies a set of data objects; in our case, it is the set of all sequences exactly similar to Q. Approximations have a cost, and the distance between objects is defined as the minimal cost of reaching one object from the other via approximations. In our case, the approximations are the distortions in the time-series data (i.e., the jiggling of individual points); the cost is the distance between the original sequence and the distorted one. Note that membership testing in the [JMM95] framework is at best exponential; thus this framework is too general for our purposes.

7.1.3 An Overview of Similarity Querying

Our main contribution is: *A syntax and semantics for similarity queries, that account for approximate matching, scaling and shifting, and that have efficient indexing support.* We show this using a new indexing technique, which preserves all the desirable properties of the indexing scheme proposed in [AFS93, FRM94].

In Section 2, we provide a *semantics* for similarity querying where we use the *similarity distance* between Q and S (defined in Section 7.2.2) as the distance metric. Similarity distance constitutes a *good distance metric* because it is non-negative, symmetric, and effectively computable; it also obeys the triangle inequality. This is not true of the "naive" distance metrics that correspond more closely to the formulation of the problem in the Section 1.2. The semantics of Section 7.2.2 serves as the basis for the *internal representation* of the query, i.e., our *normal form.*

In Section 3, we show that the semantics of Section 2.2 has several desirable properties, such as *updateability* and *well-behaved trails*, which allow us to provide efficient

implementations for similarity querying, both in the internal and external query setting. We first adapt the criteria put forth in [AFS93, FRM94] (Section 3.1) and satisfy them using fingerprints of the normal form (Section 3.2). We then argue that fingerprints are incrementally computable (Section 3.3), can be used ala Rabin-Karp [KR87] for internal searching (Section 3.4), and most importantly external indexing (Section 3.5).

This is the implementation technology that is needed to support the internal representation of Section 2. Our new indexing technique combines the MBR structure of [FRM94] with our internal representation. Many spatial data-structures can be used, for examples varieties of R-trees (see [Sam] for a comprehensive survey of the available external data-structures).

In Section 4, we provide a constraint *syntax* for similarity querying. We show how various query variations can be expressed and translated into the internal representation of Section 2. This translation also clarifies the relationship of the problem as defined in Section 1.2 with the semantics of Section 2.2.

The syntax could be embedded in most constraint logic programming languages [Col80, DVSAGB88, JL87] or constraint query languages [BNW91, BJM93, KSW90, KKR95]. This completes the connection between high level specification and implementation.

The importance of combining high-level specification with efficient implementation is the common theme of constraint databases (e.g., see [BJM93, KRVV93]) and the main motivation for this work. In Section 5, we close with some open problems and future work.

7.2 The Semantics of Similarity Queries

7.2.1 Similarity Transformations and Normal Forms

An *n-sequence* X is a sequence $\{x_1, \ldots, x_n\}$ of real numbers. Each n-sequence X has an *average* $\alpha(X)$ and a *deviation* $\sigma(X)$:

$$\alpha(X) = (1/n) \sum_{1 \le i \le n} x_i; \quad \sigma(X) = ((1/n) \sum_{1 \le i \le n} (x_i - \alpha(X))^2)^{1/2}.$$

We shall feel free to drop the arguments to α and σ, treating them as constants, when the context is not ambiguous.

A pair of reals (a, b) defines a *similarity transformation* $T_{a,b}$ over n-sequences by mapping each element x_i to $a*x_i+b$. We will assume that all similarity transformations are *non-degenerate*, i.e., that $a \neq 0$. In fact, we will further assume that $a > 0$; this restriction on a implies that a sequence symmetric to X w.r.t. the x-axis is not considered similar to it.

Definition 28 *We say that X is* similar *to Y if there exist some $(a, b) \in [R^+ \times R]$ such that $X = T_{a,b}(Y)$.*

This *similarity relation* is reflexive, symmetric, and transitive:

- **Reflexivity:** for any sequence X, $X = T_{1,0}(X)$ [the *identity* transformation];
- **Symmetry:** if $X = T_{a,b}(Y)$ then $Y = T_{1/a,-b/a}(X) = T^{-1}_{a,b}(X)$ [the *inverse* of $T_{a,b}$];
- **Transitivity:** if $X = T_{a,b}(Y)$ and $Y = T_{c,d}(Z)$, then $X = T_{ac,ad+b}(Z) = (T_{a,b} * T_{c,d})(Z)$ [the non-commutative *product* of $T_{a,b}$ and $T_{c,d}$].

Therefore, the set of all sequences similar to a given one constitutes an equivalence class, which we call a *similarity class*; we shall denote the similarity class of X by X^*. The similarity relation partitions all n-sequences into similarity classes.

To be able to refer to similarity classes, we need a way to compute a unique representative for each class, given any member in it. Towards that end, we now define normal forms of sequences.

Definition 29 *An n-sequence X is* normal *if $\alpha(X) = 0$ and $\sigma(X) = 1$.*

Let X be normal and Y be similar to X, i.e., $Y = T_{a,b}(X)$ for some $(a, b) \in [R^+ \times R]$. Then, $\alpha(Y) = b$ and $\sigma(Y) = a$:

$\alpha(Y) = (1/n) \sum_{1 \leq i \leq n} y_i = (1/n) \sum_{1 \leq i \leq n} (ax_i + b) = (a/n)\alpha(X) + b = b$;
$\sigma^2(Y) = (1/n) \sum_{1 \leq i \leq n} (y_i - \alpha(Y))^2 = (1/n) \sum_{1 \leq i \leq n} (ax_i + b - b)^2 = (a^2/n) \sum_{1 \leq i \leq n} (x_i - 0)^2 = a^2 * \sigma^2(X) = a^2$.

Y is normal only if $\sigma(Y) = a = 1$ and $\alpha(Y) = b = 0$; this is the identity transformation. This means that a similarity class has exactly one normal member; we will call it the *normal form* of all the members of the class.

Given any n-sequence X, $\overline{X}$ denotes the normal form of X^*. If α is the average of X, and σ is the deviation of X, we've shown that $X = \sigma * \overline{X} + \alpha$. Therefore, we can compute $\overline{X}$ from X by the inverse transformation:

$$\overline{X} = T^{-1}_{\sigma,\alpha}(X) = T_{1/\sigma,-\alpha/\sigma}(X).$$

In a transformation $T_{a,b}$, we call a the *scale factor* and b the *shift factor*. If a is 1, the transformation is a pure *shift*; if b is 0, it is a pure *scaling*. The identity transformation is a pure shift; the inverse of a shift is a shift; and the product of two shifts is also a shift. This allows us to conclude that the set of all shifts of a given sequence is an equivalence class. The same is true of the set of all scalings. The normal form for these classes is defined just as for the general case.

7.2.2 Similarity Distance and Semantics

Given two sequences X and Y, the *similarity distance* between X and Y is the distance between the normal forms of their respective similarity classes:

Definition 30

$$D_S(X,Y) = D_N(\overline{X},\overline{Y}),$$

where D_N is the *normalized Euclidean distance*, defined as follows:

Definition 31

$$D_N(X,Y) = \sqrt{(1/n)\sum_{1\le i\le n}(x_i - y_i)^2}$$

The intuition behind D_N is that it represents the average distance per point between two sequences. For example, the deviation of a sequence X can be thought of as the normalized Euclidean distance between X and a sequence all of whose points are the average of X. Any proper distance metric D for n-sequences can be used instead of D_N. Since we will be using techniques from Discrete Signal Processing (DSP) and the Euclidean distance is a standard distance metric in DSP, we have chosen to use a variant of it here.

Note that the similarity distance between any pair of sequences from X^* and Y^* is the same; this gives us a *distance metric for similarity classes*:

Definition 32

$$D_S(X^*,Y^*) = D_S(X,Y).$$

A distance metric should be non-negative and symmetric, and it should obey the triangle inequality. A good distance metric should also be effectively computable. It is easy to see that similarity distance satisfies all these criteria.

> **Remark:** Note that there are definitions of distance that correspond more closely to the naive formulation of the problem. For example, given Q and S, we could have used the minimum Euclidean distance between Q and all $S' \in S^*$(the equivalence class of S); let us denote this distance by $D_m(Q, S^*)$. However, this definition is not symmetric: $D_m(Q, S^*) \neq D_m(S, Q^*)$. Given Q and S, we could have also tried to choose the minimum Euclidean distance between Q' and S' for all $Q' \in Q^*$ (the equivalence class of S) and $S' \in S^*$ (the equivalence class of Q). However, by choosing the members of Q^* and S^* with arbitrarily small deviations, this distance will always approach 0. The normal forms provide a distance metric that does not suffer from any of these defects.

By using similarity distance, we are now ready to define a *similarity semantics* for the *All-Occurrences Subsequence Approximate Similarity* problem.

> Given a query sequence Q, a time-series $\overline{S}$, a tolerance $\epsilon \geq 0$, and a similarity relation [which partitions sequences into equivalence classes with normal forms], find all contiguous subsequences S in the time-series $\overline{S}$ such that $D_S(Q, S) \leq \epsilon$.

Note that these semantics are slightly different from the problem formulation in Section 1.2. The differences will be clarified (and bridged) in Section 4. To conclude this subsection, we want to consider the *All-Occurrences Subsequence Exact Similarity* problem (i.e., when $\epsilon = 0$).

> Given a query sequence Q, a time-series $\overline{S}$, and a similarity relation [which partitions sequences into equivalence classes with normal forms], find all contiguous subsequences S in the time-series $\overline{S}$ such that Q and S are similar [belong to the same equivalence class].

The exact case can be answered using the normal forms, because Q and S are in the same equivalence class if and only if $\overline{Q} = \overline{X}$. Finally, analogous definitions apply to Whole-Sequence problems.

7.3 Indexing of Similarity Queries

7.3.1 Sequence Fingerprints: Criteria and Definitions

Computing the similarity distance D_S between any two n-sequences requires $O(n)$ operations. An efficient implementation of similarity querying cannot afford to compute D_S every time for each sequence in the data set (for the Whole-Sequence case), or for each contiguous subsequence in the time-series (for the Subsequence case).

Following the approach of [KR87], which has gained wide acceptance, we introduce a *fingerprint* function F, together with a fingerprint distance metric D_F. This fingerprint mechanism provides *fast rejection*, filtering out most of the non-similar sequences.

A fingerprint mechanism needs to satisfy the following criteria:

- **Compactness:** The comparison of the fingerprints of two n-sequences can be done in constant time.
- **Validity:** If S is a valid query answer, then the comparison of $F(S)$ and $F(Q)$ should return *TRUE*:

$$D_S(X,Y) \leq \epsilon \Longrightarrow D_F(F(X),F(Y)) \leq \epsilon.$$

- **Accuracy:** If the comparison of $F(S)$ and $F(Q)$ returns *TRUE*, S is highly likely to be a valid query answer.
- **Updateability:** Computing the fingerprints of all subsequences of an N-sequence for N much larger than n can be done in $O(N)$ time, by *updating* the fingerprint value as we move along rather than recomputing it for every subsequence.

We now define the fingerprint function F as well as the fingerprint distance function D_F. These definitions are similar to the ones used for Approximate Matching in [AFS93] and [FRM94].

Definition 33 *A* fingerprint $F(X)$ *of an n-sequence* $X = \{x_1, \ldots, x_n\}$ *is the tuple*

$$[DFT_1(\overline{X}), \ldots, DFT_l(\overline{X})],$$

where l *is a small constant (such as 3), and* DFT_m *is the* m*'th coefficient of the* Discrete Fourier Transform *of* $\overline{X}$*:*

$$DFT_m(s_0, \ldots, s_{n-1}) = \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} (s_j e^{-j(2\pi i)m/n}),$$

where $i = \sqrt{-1}$.

Note that DFT_m is a complex number: $DFT_m = a_m + b_m i$ for some $a_m, b_m \in R$. Thus, F is specified by a sequence of $2l$ real values. Note that we are not including DFT_0 in the fingerprint, since its coefficients are both 0 for normal sequences.

Definition 34 *The* fingerprint distance D_F *between* $F(X)$ *and* $F(Y)$ *is the Euclidean distance between the real-valued sequences for* $F(X)$ *and* $F(Y)$ *divided by* $\sqrt{n}$, *where* $n = |X| = |Y|$.

By taking l to be a small constant our fingerprint mechanism is compact. In the subsections below, we establish its validity, accuracy, and updateability.

7.3.2 Validity and Accuracy of Fingerprinting

To establish the validity of fingerprinting, we need to show that

$$D_S(X,Y) \leq \epsilon \Longrightarrow D_F(F(X),F(Y)) \leq \epsilon.$$

We make use of the fact that the DFT is a *linear function*, i.e.,

$$DFT_m(aX + bY) = aDFT_m(X) + bDFT_m(Y)$$

for all scalars a and b. Also, we rely on *Parseval's theorem*, well-known in DSP:

$$\sum_{0 \leq m \leq n-1} |DFT_m(X)|^2 = \sum_{0 \leq i \leq n-1} |x_i|^2.$$

And we make use of the fact that the coefficients of DFT_0 for normal sequences are both 0. First, we show that $D_S(X,Y) \geq D_F(F(X),F(Y))$:

$$\begin{aligned}
&D_S(X,Y) = D_N(\overline{X},\overline{Y}) = (\textstyle\sum_{0 \leq i \leq n-1} |\overline{X}[i] - \overline{Y}[i]|^2)^{1/2}/\sqrt{n} = \\
&= (\textstyle\sum_{0 \leq m \leq n-1} |DFT_m(\overline{X} - \overline{Y})|^2)^{1/2}\sqrt{n} \geq \\
&\geq (\textstyle\sum_{0 \leq m \leq l} |DFT_m(\overline{X} - \overline{Y})|^2)^{1/2}\sqrt{n} = D_F(F(X),F(Y)).
\end{aligned}$$

It immediately follows that $D_F(F(X),F(Y)) \leq \epsilon$ whenever $D_S(X,Y) \leq \epsilon$.

To establish accuracy, we want to know how likely it is that $D_S(X,Y) \leq \epsilon$ provided that $D_F(F(X),F(Y)) \leq \epsilon$. The cases when $D_F(F(X),F(Y)) \leq \epsilon$ but $D_S(X,Y) \leq \epsilon$

represent *false alarms*, and we want to minimize their occurrence. Therefore, we would like the ratio $D_F(F(X), F(Y))/D_S(X, Y)$ to be close to 1.

The actual ratio strongly depends on the nature of the data sequences. It is worst in the case of *white noise*, when

$$D_F(F(X), F(Y))/D_S(X, Y) = (l+1)/n.$$

However, a large class of signals (the *colored noises*) have a *skewed energy spectrum*, and it is the case that:

> The amplitude of DFT_m decreases rapidly for increasing values of m.

In fact, the decrease is as $O(f^{-b})$ [Fal96], where:

- $b = 1$ for *pink noise*, such as musical scores;
- $b = 2$ for *brown noise*, such as stock movements and exchange rates;
- $b > 2$ for *black noise*, such as rainfall patterns.

As a result, 2-3 coefficients are sufficient to provide good accuracy for most applications, including the one used for our experiments. Therefore, we may assume that the length l of the fingerprint is ≤ 3. (Note that, the randomization ala Rabin-Karp [KR87] makes no assumptions about the spectrum.)

7.3.3 Updateability of Fingerprinting

When computing fingerprints of all subsequences of length n for a much longer sequence of length N, the efficiency of the algorithm hinges on a property of the fingerprint that we call *updateability*:

> Given the fingerprint of a subsequence $\{x_k, \ldots, x_{k+n-1}\}$, it is possible to compute the fingerprint for $\{x_{k+1}, \ldots, x_{k+n}\}$ in constant time.

Let X_k be the first subsequence, and X_{k+1} be the second subsequence. We show how to compute the fingerprint $F(X_{k+1})$ from the fingerprint $F(X_k)$ in constant time. As inputs to the update step, we assume that we have the values of the following expressions:

$$\sum_{k \leq j \leq k+n-1} x_j, \sum_{k \leq j \leq k+n-1} (x_j)^2, \alpha(X_k), \sigma(X_k), DFT_1(X_k), \ldots, DFT_l(X_k), F(X_k).$$

We also assume that all constants (such as $1/n$) are pre-computed. During the update step, we obtain the values for the above expressions with $k+1$ instead of k; the computation proceeds as follows:

1. Increment k to $k+1$;
2. Look up x_{k+1}, x_{k+n};
3. Compute $\sum_{k+1\leq j\leq k+n} x_j$. This involves one subtraction and one addition:

$$\sum_{k+1\leq j\leq k+n} x_j = (\sum_{k\leq j\leq k+n-1} x_j) - x_k + x_{k+n};$$

4. Compute $\sum_{k+1\leq j\leq k+n} (x_j)^2$, using two multiplications, one subtraction and one addition:

$$\sum_{k+1\leq j\leq k+n} x_j^2 = (\sum_{k\leq j\leq k+n-1} x_j^2) - x_k^2 + x_{k+n}^2;$$

5. Compute $\alpha(X_{k+1})$, using one multiplication:

$$\alpha(X_{k+1}) = \alpha_{k+1} = (1/n) \sum_{k+1\leq j\leq k+n} x_j;$$

6. Compute $\sigma(X_{k+1})$, using two multiplications, one subtraction and one square root:

$$\sigma^2(X_{k+1}) = \sigma_{k+1}^2 = (1/n)(\sum_{k+1\leq j\leq k+n} x_j^2) - \alpha_{k+1}^2;$$

7. Compute $DFT_1(X_{k+1}), \ldots, DFT_l(X_{k+1})$, using one subtraction, one addition and two multiplications for each index:

$$DFT_m(X_{k+1}) = \frac{1}{\sqrt{n}} \sum_{0\leq j\leq n-1} (x_{j+k+1} e^{-j(2\pi i)m/n}) =$$

$$= \frac{1}{\sqrt{n}} \sum_{1\leq j\leq n} (x_{j+k} e^{-(j-1)(2\pi i)m/n}) = e^{(2\pi i)m/n}(DFT_m(X_k) + \frac{x_{n+k} - x_k}{\sqrt{n}});$$

8. Compute the fingerprint of X_{k+1}, using one division for each index. Here, we rely on the linearity of DFT's, and on the fact that $DFT_m(\mathbf{1})$ is 0 when $m > 0$:

$$DFT_m(\overline{X_{k+1}}) = DFT_m(X_{k+1}/\sigma_{k+1} - \alpha_{k+1}/\sigma_{k+1}) =$$

$$= (1/\sigma_{k+1})DFT_m(X_{k+1}) - (\alpha_{k+1}/\sigma_{k+1})DFT_m(\mathbf{1}) = DFT_m(X_{k+1})/\sigma_{k+1}.$$

Note that the above algorithm is *on-line*, suitable in a situation when the data are streaming past and we can never back over it. In addition, we have shown that the fingerprint of [AFS93] for time-series approximate matching (without similarity) is also updateable.

7.3.4 Internal Query Representation

In this section, we sketch out the internal implementation of similarity queries. The implementation is based on an *internal representation* of a similarity query:

Definition 35 *The* internal representation *of a similarity query consists of:*

(query sequence Q, the values $\{\epsilon_i, l_\alpha, u_\alpha, l_\sigma, u_\sigma\}$).

The internal representation corresponds to an *internal query* with the following semantics:

Internal query: Find all S such that $D_S(Q,S) \leq \epsilon_i$, $l_\alpha \leq \alpha(S) \leq u_\alpha$, $l_\sigma \leq \sigma(S) \leq u_\sigma$.

In Section 7.4, we will show how to translate the constraint-based syntax of a user query to this internal query. Here, we just note that the translation is not "tight"; i.e., though it will not force any false dismissals, it may generate some false alarms.

The updateability results established in Section 7.3.3 allow us to answer the internal query for the *in-core case* with an algorithm much like that of [KR87]. We proceed through the sequences and the subsequences, comparing $D_F(F(Q),F(S))$ to ϵ_i and checking $\alpha(S)$ and $\sigma(S)$ against the bounds.

If we want to avoid run-time linear scanning of the data, we need to create an *index structure* for it. This is the case for any database application of similarity querying. A naive index structure is a list of tuples of the following form:

$[F(X), \alpha(X), \sigma(X)$, location of $X]$.

The algorithm is very similar to the in-core case; we scan the index and look for *potential matches*:

Potential match: Is $D_F(F(Q),F(S)) \leq \epsilon_i$, $l_\alpha \leq \alpha(S) \leq u_\alpha$, and $l_\sigma \leq \sigma(S) \leq u_\sigma$?

Whichever index structure we use for the search (if any), all potential matches are then checked against the internal query to determine which of them actually satisfy it.

7.3.5 External Indexing

To speed up index searching, we can instead build an indexing mechanism that allows spatial access methods for range queries of multidimensional points. The query point P_Q is computed from the internal query representation:

$$P_Q = (F(Q), m_\alpha, m_\sigma), \text{ where } m_\alpha = (u_\alpha + l_\alpha)/2,\ m_\sigma = (u_\sigma + l_\sigma)/2.$$

The answer to the similarity query is the set of all points $(F(X), \alpha(X), \sigma(X))$ such that:

$F(X)$ is within ϵ of $F(Q)$, $\alpha(X)$ is within $(u_\alpha - l_\alpha)/2$ of m_α, and
$\sigma(X)$ is within $(u_\sigma - l_\sigma)/2$ of m_σ.

Though the index structure described above performs well for the whole-sequence case, it is unsuitable in the subsequence setting. This is due to a very simple observation: for a real sequence of length m, the there would be $m - n + 1$ indices with $2l$ reals each. Such space overhead renders indexing less efficient than a direct sequential scan of the data [FRM94].

This problem is overcome with the Minimum Bounding Rectangle (MBR) technique, introduced in [FRM94]. This technique significantly reduces the size of the indexing structure, though introducing some false alarms in the process. Our final indexing method consists of combining the MBR technique with the spatial access approach described above.

The MBR technique relies of the *continuity* of subsequence indices:

Continuity: Given two adjacent subsequences, the difference between the corresponding coefficients of their indices is likely to be small.

We conclude the discussion of indexing by verifying that our indexing possesses the continuity property. This is a "heuristic" statistical argument, that also applies to [FRM94] (where continuity was assumed, but not shown).

Denote adjacent sequences by $X_0 = \{x_0, \ldots, x_{n-1}\}$ and $X_1 = \{x_1, \ldots, x_n\}$; denote the corresponding indices by $(\alpha_0, \sigma_0, F_0)$ and $(\alpha_1, \sigma_1, F_1)$; and Denote the order of the expected value of an expression by $\approx$. Assume that n is reasonably large, so that $1/n$ is considered to be a small constant. Proceed by considering each element of the index separately.

1. The expected value of $|\alpha_1 - \alpha_0|$ is on the order of σ_0/n:

$$|\alpha_1 - \alpha_0| = |x_n - x_0|/n \approx \sigma_0/n.$$

2. The expected value of $|\sigma_1 - \sigma_0|$ is on the order of σ_0/n:

$$|\sigma_1 - \sigma_0|(\sigma_1 + \sigma_0) = |\sigma_1^2 - \sigma_0^2| =$$
$$= \frac{1}{n}|(x_n - \alpha_1)^2 + (x_0 - \alpha_0)^2 + \sum_{1 \le j \le n-1} ((x_j - \alpha_1)^2 - (x_j - \alpha_0)^2)| =$$
$$= |(x_n - x_0)(x_n - \alpha_1 + x_0 - \alpha_0)|/n \approx (\sigma_0/n)(\sigma_1 + \sigma_0).$$

3. The expected value of $|DFT_m(\overline{X_1}) - DFT_m(\overline{X_0})|$ is on the order of $|DFT_m(\overline{X_0})|/n + 1/\sqrt{n}$. Here, we use the equations for DFT's derived in Section 7.3.3, as well as the ones derived above:

$$|DFT_m(\overline{X_1}) - DFT_m(\overline{X_0})| = |DFT_m(X_1)/\sigma_1 - DFT_m(\overline{X_0})| =$$
$$= |\frac{e^{(2\pi i)m/n}}{\sigma_1}(\sigma_0 DFT_m(\overline{X_0}) + \frac{x_n - x_0}{\sqrt{n}}) - DFT_m(\overline{X_0})| =$$
$$= |DFT_m(\overline{X_0})(\frac{\sigma_0}{\sigma_1}e^{(2\pi i)m/n} - 1) + \frac{x_n - x_0}{\sigma_1\sqrt{n}}e^{(2\pi i)m/n}| \approx$$
$$\approx |DFT_m(\overline{X_0})((1 + 1/n)(1 - 2\pi m/n) - 1) + (1 + 1/n)(1 - 2\pi m/n)/\sqrt{n}| \approx$$
$$\approx |DFT_m(\overline{X_0})|/n + O(1/\sqrt{n})$$

7.4 Constraint Specification of Similarity Queries

In the previous section, we have shown how to provide an efficient database indexing mechanism for queries with similarity semantics. This enables us to answer the following question:

Internal query: Given Q, ϵ_i, (l_α, u_α) and (l_σ, u_σ),
find all S such that $D_S(Q, S) \le \epsilon$, $l_\alpha \le \alpha(S) \le u_\alpha$, $l_\sigma \le \sigma(S) \le u_\sigma$.

The syntax of the user query may be very different from this. In particular, the user querying a database of sequences should probably not be expected to refer to normal forms, similarity distance, or even to averages and deviations.

In this section, we provide a *syntax* for similarity queries, based on constraints. Then, we show how to translate from this syntactic formulation to the internal representation.

7.4.1 Constraint-Based Syntax of Similarity Queries

For the general similarity query, we define the following constraint-based syntax, which expresses the queries described in Section 1.2:

General similarity query: Given Q, ϵ, (l_a, u_a), and (l_b, u_b),
find all $[S, a, b]$ such that $D(Q, aS + b) \leq \epsilon$, $l_a \leq a \leq u_a$ and $l_b \leq b \leq u_b$.

We assume that the sequence distance D is D_N (see Definition 31). Of course, regular Euclidean distance may be used, multiplying ϵ by a factor of $\sqrt{n}$. Of course, the user may choose to omit the bounds on a and/or b:

Unbounded Case: Given Q and ϵ,
find all $[S, a, b]$ such that $D(Q, aS + b) \leq \epsilon$.

In all the following queries, bounds either a or b are optional.

The user may want to query for scaling transformations only, or for shift transformations only:

Scaling: Find all $[S, a]$ such that $D(Q, aS) \leq \epsilon$.

Shifting: Find all $[S, b]$ such that $D(Q, S + b) \leq \epsilon$.

Finally, the user may ask approximate matching queries or exact similarity queries:

Approximate Match: Find all S such that $D(Q, S) \leq \epsilon$.

Exact Similarity: Find all $[S, a, b]$ such that $Q = aS + b$.

For all these variations on similarity queries, it is possible to efficiently find the corresponding values for the variables used in the internal query representation:

$$\{\epsilon_i, l_\alpha, u_\alpha, l_\sigma, u_\sigma\}.$$

We shall show how to do that in the next subsection.

7.4.2 From Constraint-Based Syntax to Internal Representation

We first reduce the cases of scaling, shifting, alternate matching, and exact similarity to the general case:

Scaling: force b to have the value 0 by specifying an equality constraint $b = 0$:

ϵ_i	$\sqrt{2-2\sqrt{1-\epsilon^2/\sigma^2}}$
l_σ	$(\alpha_Q - u_b - \epsilon)/u_a$
u_σ	$(\alpha_Q - l_b + \epsilon)/l_a$
l_α	$(\sigma_Q + \epsilon)/l_a$
u_α	$(\sigma_Q + \epsilon)/l_a$

Figure 7.2: Computing the values for the Internal Query

$$D(Q, aS) \leq \epsilon \iff D(Q, aS+0) \leq \epsilon \iff D(Q, aS+b) \leq \epsilon \wedge b = 0.$$

Shifting: force a to have the value 1 by specifying an equality constraint $a = 1$:

$$D(Q, S+b) \leq \epsilon \iff D(Q, aS+b) \leq \epsilon \wedge a = 1.$$

Approximate Matching: by similar reasoning, set a to 1 and b to 0.

Exact Similarity: set ϵ to 0:

$$Q = aS + b \iff D(Q, aS+b) = 0.$$

Then, we convert the general case to the internal representation. This is done via the conversion formulas summarized in Figure 7.2; these are derived in Section 7.5.

It is important to note that the translation is not always *bi-directional*. If a sequence satisfies the internal query, another *filtering* step needs to be performed before we can be sure that it also satisfies the user query. This step answers the following question:

Filter: Given a subsequence X, do there exist a and b within the bounds specified by the external query such that $D_N(Q, aX+b) \leq \epsilon$?

The additional filtering indicates a potential performance disadvantage of our approach and is a trade-off to achieve generality. We conclude this section with a sketch of the filtering method.

We know that $l_a \leq a \leq u_a$ and $l_b \leq b \leq u_b$. This defines a rectangle R in the (a, b)-coordinate system. Let t be the following linear transformation:

$$t(a, b) = (a\sigma_X - \sigma_Q\overline{XQ}, a\alpha_X + b - \alpha_Q).$$

Let (a_t, b_t) be the point in $t(R)$ closest to the origin. Then, it is the case that X passes the filter *only if* $D_N(Q, aX+b) \leq \epsilon$, where $(a, b) = t^{-1}(a_t, b_t)$ (see the corollary to Proposition 7 in Section 7.5).

7.5 From External to Internal Queries and Back

In this section, we derive the formulas for the translation between the general similarity query and the internal query (throughout this chapter, we denote the normal form of X by $\overline{X}$ and the average and standard deviation of X by α_X and σ_X, respectively):

General similarity query: Given Q, ϵ, (l_a, u_a), and (l_b, u_b),
find all $[S, a, b]$ such that $D_N(Q, aS + b) \leq \epsilon$, $l_a \leq a \leq u_a$ and $l_b \leq b \leq u_b$.

Internal query: Given Q, ϵ_i, (l_α, u_α) and (l_σ, u_σ),
find all S such that $D_N(\overline{Q}, \overline{X}) \leq \epsilon$, $l_\alpha \leq \alpha_S \leq u_\alpha$, $l_\sigma \leq \sigma_S \leq u_\sigma$.

The translation needs to be made in both directions:

Direction I (Computing internal query values): Given a sequence Q and the values for $\{\epsilon, l_a, u_a, l_b, u_b\}$, find the values for $\{\epsilon_i, l_\alpha, u_\alpha, l_\sigma, u_\sigma\}$ such that for any sequence S, there exist (a, b) with $[S, a, b]$ satisfying the general similarity query *only if* S satisfies the internal query.

Direction II (Filtering potential matches): Given a sequence S returned by the internal query, find the values of a and b (if any) such that $[S, a, b]$ satisfies the external query.

7.5.1 Normalized Products and Distances

First, we define the *normalized product* of two sequences and introduce some of its properties; we also introduce some properties of D_N, the *normalized Euclidean distance*. These properties are used for the proofs in the remainder of the section.

Definition 36 *Let $X = \{x_1, \ldots, x_n\}$ and $Y = \{y_1, \ldots, y_n\}$ be two sequences of length n. XY is the* normalized product *of X and Y:*

$$XY = (1/n) \sum_{1 \leq i \leq n} (x_i y_i)$$

It is easy to verify that normalized product operation is commutative, and that it is distributative with respect to sequence sums and differences (i.e., $X(Y + Z) = XY + XZ$). The normalized product has some other useful properties, well-known in the field of mathematical statistics (see [MSW] for details):

Property 1: $XY = \alpha_X \alpha_Y + \sigma_X \sigma_Y \overline{XY}$

Property 2: $-1 \leq \overline{XY} \leq 1$

Property 3: $X^2 = \alpha_X^2 + \sigma_X^2$

Property 4: $\overline{X}^2 = \alpha_{\overline{X}}^2 + \sigma_{\overline{X}}^2 = 0^2 + 1^2 = 1$

Here, X^2 is the normalized product of X with itself. Note that in our calculations, we will assume that $0 \leq \overline{XY} \leq 1$, because if $\overline{XY}$ is negative, this means that as the values in X increase, the values in Y tend to decrease, and vice versa [MSW]. This, then, corresponds to the need for a negative scale factor in our similarity relation, which we disallow. Therefore, we do not wish these two sequences to be considered similar under our definition of similarity.

Note that the normalized product has an intimate connection to the normalized distance D_N (see Definition 31), and in fact:

Proposition 4 $D_N^2(X, Y) = (X - Y)^2$

Proof: Follows from definitions. □

The following two propositions are also about D_N.

Proposition 5 $D_N^2(\overline{X}, \overline{Y}) \leq 2$

Proof: $D_N^2(\overline{X}, \overline{Y}) = (\overline{X} - \overline{Y})^2 = \overline{X}^2 + \overline{Y}^2 - 2\overline{XY} = 1 + 1 - 2\overline{XY} = 2 - 2\overline{XY} \leq 2$ □

Corollary 2 $\overline{XY} = 1 - D_N^2(\overline{X}, \overline{Y})/2$

7.5.2 Computing the Internal Epsilon

Let us denote the minimum normalized distance between X and Y', for all $Y' \in Y^*$, by $D_m(X, Y^*)$:

$$D_m(X, Y^*) = \min\{D_N(X, Y') : Y' = aY + b, a > 0\}.$$

Proposition 6 $D_m^2(X, Y^*) = \sigma_X^2 (D_N^2(\overline{X}, \overline{Y}) - D_N^4(\overline{X}, \overline{Y})/4)$

Proof:

1. Let us denote $D_N^2(X, aY+b)$ by $f(a,b)$; $f(a,b) = (1/n)\sum(x_i - (ay_i+b))^2$. We will obtain its minimum by finding the values of a, b where $df/da = df/db = 0$. Then, we will complete the proof by substituting these values back into $f(a,b)$.

2. The rest of the proof provides the details for these calculations. Let us denote $(y_i - \alpha_Y)$ by $\tilde{y}_i$; similarly, $(x_i - \alpha_X)$ is $\tilde{x}_i$. It is easy to verify that:

   (a) $\sum \tilde{x}_i = \sum \tilde{y}_i = 0$; (b) $\sum \tilde{x}_i\tilde{y}_i = n\sigma_X\sigma_Y\overline{XY}$.

   From (b), it also follows that

   (c) $\sum \tilde{x}_i^2 = n\sigma_X^2$, $\sum \tilde{y}_i^2 = n\sigma_Y^2$.

3. $df/db = (1/n)\sum 2(x_i - ay_i - b) = 2(\alpha_X - a\alpha_Y - b) = 0$. Therefore, the desired value for b is $(\alpha_X - a\alpha_Y)$.

4. $df/da = (1/n)\sum 2y_i(x_i - ay_i - b) = (2/n)\sum y_i(x_i - ay_i - (\alpha_X - a\alpha_Y)) = (2/n)\sum y_i(\tilde{x}_i - a\tilde{y}_i) = (2/n)\sum(\alpha_Y + \tilde{y}_i)(\tilde{x}_i - a\tilde{y}_i) = (2/n)(\alpha_Y\sum\tilde{x}_i - a\alpha_Y\sum\tilde{y}_i + \sum\tilde{y}_i(\tilde{x}_i - a\tilde{y}_i)) = (2/n)\sum\tilde{y}_i(\tilde{x}_i - a\tilde{y}_i) = 0$. Therefore, the desired value for a is $(\sum\tilde{x}_i\tilde{y}_i)/(\sum\tilde{y}_i^2)$.

5. Now, we do the substitution into $f(a,b)$.
   $f(a,b) = (1/n)\sum(x_i - ay_i - b)^2 = (1/n)\sum(\tilde{x}_i - a\tilde{y}_i)^2 = (1/n)\sum(\tilde{x}_i^2 + a^2\tilde{y}_i^2 - 2a\tilde{x}_i\tilde{y}_i) = (1/n)(\sum\tilde{x}_i^2 + (\sum\tilde{y}_i^2)(\sum\tilde{x}_i\tilde{y}_i/\sum\tilde{y}_i^2)^2 - 2(\sum\tilde{x}_i\tilde{y}_i)(\sum\tilde{x}_i\tilde{y}_i/\sum\tilde{y}_i^2)) = (1/n)(\sum\tilde{x}_i^2 - (\sum\tilde{x}_i\tilde{y}_i)^2/\sum\tilde{y}_i^2) = (1/n)(n\sigma_X^2 - (n\sigma_X\sigma_Y\overline{XY})^2/(n\sigma_Y^2)) = \sigma_X^2(1 - (\overline{XY})^2)$.

6. Let us denote $D_N^2(\overline{X}, \overline{Y})$ by $\mathcal{D}$. We make use of Corollary 2 to complete the proof: $D_m^2(X, Y^*) = \sigma_X^2(1 - (\overline{XY})^2) = \sigma_X^2(1 - (1 - \mathcal{D}^2/2)^2) = \sigma_X^2(\mathcal{D}^2 - \mathcal{D}^4/4)$.

□

Now, we are ready to compute the value for ϵ_i, used in the internal query representation.

$$D(Q, aX+b) \le \epsilon \Rightarrow D_m(Q, X^*) \le \epsilon \iff D_m^2(Q, X^*) \le \epsilon^2 \iff$$
$$\iff \sigma_Q^2(D_N^2(\overline{Q}, \overline{X}) - D_N^4(\overline{Q}, \overline{X})/4) \le \epsilon^2 \iff$$
$$\iff D_N^4(\overline{Q}, \overline{X}) - 4D_N^2(\overline{Q}, \overline{X}) + 4\epsilon^2/\sigma_Q^2 \ge 0 \iff$$
$$\iff (D_N^2(\overline{Q}, \overline{X}) \le 2 - 2\sqrt{1 - \epsilon^2/\sigma_Q^2}) \vee (D_N^2(\overline{Q}, \overline{X}) \ge 2 + 2\sqrt{1 - \epsilon^2/\sigma_Q^2})$$

By Proposition 5, we see that only the first inequality for $D_N^2(\overline{Q}, \overline{X})$ is valid. Therefore,

$$D_N(\overline{Q}, \overline{X}) \leq \sqrt{2 - 2\sqrt{1 - \epsilon^2/\sigma_Q^2}} = \epsilon_i.$$

Notice that if a and b are unbounded, then

$$D(Q, aX + b) \leq \epsilon \iff D_m(Q, X^*) \leq \epsilon,$$

and this value for ϵ_i is tight.

> **Remark:** The above value for ϵ_i only makes sense when $\epsilon^2 \geq \sigma_Q^2$. However, this does not reduce the expressibility, because $D_m^2(Q, X^*) \leq \sigma_Q^2$ for all sequences X. Therefore, any tolerance value greater than σ_Q^2 will produce no new matches.

7.5.3 Computing Internal Bounds

Now we compute bounds on the average and deviation. First, we prove the following proposition:

Proposition 7 *Given two sequences Q and X, let t be the following linear transformation:*

$$t(a, b) = (a\sigma_X - \sigma_Q\overline{XQ}, a\alpha_X + b - \alpha_Q).$$

Then, for all (a, b), if $(a', b') = t(a, b)$ and $c = \sigma_Q^2(1 - (\overline{XQ})^2)$, we have:

$$D_N^2(Q, aX + b) = (a')^2 + (b')^2 + c.$$

Proof: $D_N^2(Q, aX+b) = (Q-(aX+b))^2 = Q^2+a^2X^2+b^2+2ab\alpha_X-2b\alpha_Q-2aXQ =$
$= (\alpha_Q^2 + \sigma_Q^2) + a^2(\alpha_X^2 + \sigma_X^2) + b^2 + 2ab\alpha_X - 2b\alpha_Q - 2a(\alpha_X\alpha_Q + \sigma_X\sigma_Q\overline{XQ}) =$
$= (a\alpha_X + b - \alpha_Q)^2 + (a\sigma_X - \sigma_Q\overline{XQ})^2 + \sigma_Q^2(1 - (\overline{XQ})^2) = (b')^2 + (a')^2 + c.$ □

Next, we combine the fact that $D_N(Q, aX + b) \leq \epsilon$ with the equality derived in Proposition 7:

$D_N(Q, aX + b) \leq \epsilon \iff D_N^2(Q, aX + b) \leq \epsilon^2 \iff (a')^2 + (b')^2 + c \leq$
$\epsilon^2 \iff (a')^2 + (b')^2 \leq \epsilon^2 - c.$

Note that σ_X (α_X) appears only in a' (b'), and its range can be maximized by setting b' (a') to 0:

$$(a')^2 = (a\sigma_X - \sigma_Q\overline{XQ})^2 \leq \epsilon^2 - c$$

$$(b')^2 = (a\alpha_X + b - \alpha_Q)^2 \leq \epsilon^2 - c$$

Solving the resulting inequalities for α_X and σ_X, we obtain:

$$(\alpha_Q - b - \sqrt{\epsilon^2 - c})/a \leq \alpha_X \leq (\alpha_Q - b + \sqrt{\epsilon^2 - c})/a$$

$$(\sigma_Q\overline{XQ} - \sqrt{\epsilon^2 - c})/a \leq \sigma_X \leq (\sigma_Q\overline{XQ} + \sqrt{\epsilon^2 - c})/a$$

Unfortunately, the above formulas involve values that are not constant at the time of computation; i.e., a, b and $\overline{XQ}$. We need to substitute constants for a, b and $\overline{XQ}$; we do this conservatively so as to ensure no false dismissals, while trying to allow for the widest range of α_X and σ_X. Specifically, we find that taking $\overline{XQ} = 1$ always minimizes the lower bounds and maximizes the upper bounds. Therefore, the square root expression always collapses to $\sqrt{\epsilon^2} = \epsilon$. For a and b, we choose appropriately from the external query bounds on these values in order to obtain the widest bounds. Therefore, we obtain the following bounds:

$$(\alpha_Q - u_b - \epsilon)/u_a \leq \alpha_X \leq (\alpha_Q - l_b + \epsilon)/l_a$$

$$(\sigma_Q - \epsilon)/u_a \leq \sigma_X \leq (\sigma_Q + \epsilon)/l_a$$

This completes the specification of the conversion from the external to the internal query. The conversion results are summarized in Figure 7.2

7.5.4 Filtering Potential Matches

Proposition 8 *Given a query Q, a potential match X, the (open) bounds (l_a, u_a), (l_b, u_b) on a and b define an (open) rectangle R:*

$$(a, b) \in R \iff (l_a \leq a \leq u_a) \wedge (l_b \leq b \leq u_b).$$

The transformation t (defined in Proposition 7) maps R to an (open) quadrilateral $t(R)$. Let (a_t, b_t) be the point in $t(R)$ closest to the origin, and let (a_0, b_0) be the inverse transformation $t^{-1}(a_t, b_t)$:

$$a_0 = (a_t + \sigma_Q\overline{XQ})/\sigma_X,\ b_0 = b_t + \alpha_Q - a_0\alpha_X.$$

Then, $min\{D_N(Q, aX + b) : (l_a \leq a \leq u_a) \wedge (l_b \leq b \leq u_b)\} = D_N(Q, a_0X + b_0) = \sqrt{a_t^2 + b_t^2 + c}$.

Proof:

1. Let us consider some arbitrary $(a_1, b_1) \in R$, with $t(a_1, b_1) = (a_1', b_1')$. We know that $(a_t^2 + b_t^2) \leq (a_1'^2 + b_1'^2)$. This means that $0 \leq (a_1'^2 + b_1'^2) - (a_t^2 + b_t^2) = (a_1'^2 + b_1'^2 + c) - (a_t^2 + b_t^2 + c) = D_N^2(Q, a_1X + b_1) - D_N^2(Q, a_0X + b_0)$ (by Proposition 7).

2. We've just shown that for all (a_1, b_1) in R, $D_N(Q, a_1X + b_1) \geq D_N(Q, a_0X + b_0)$. Since t is linear, (a_0, b_0) is itself in R. Since we know that $D_N(Q, a_0X + b_0) = \sqrt{a_t^2 + b_t^2 + c}$, this completes the proof.

□

7.6 Current Work and Future Directions

We have implemented a framework for Time-Series Approximate Similarity Queries that allows the user to pose a wide variety of queries and that preserves desirable indexing properties of Time-Series Approximate Matching. Possible extensions involve more powerful similarity queries and different distance functions; also indexing of time-series data that is represented using constraints (see [BNW91, KSW90]).

In addition to implementing the general version of similarity querying, as described in this chapter, we have implemented other versions, by tailoring the internal representation and the corresponding indexing scheme for specialized subsets of similarity queries.

As with a generalized version of any problem, there is a trade-off between a gain in expressibility and a decrease in performance, though it is not significant. There is some slow-down due to the extra keys in the new indexing scheme (as opposed to [FRM94]), as well as due to the additional false alarms generated by our fingerprinting (as compared to the specialized cases without a similarity transformation). We are currently examining these trade-off through performance evaluations; this experimental work is in progress.

Some other questions, more theoretical in nature, remain. The existence of a fingerprint function for internal similarity querying that is a real hash function but is also distance-preserving and updateable is an interesting open question. *Is there a fingerprint method that gives a provably linear performance for the Rabin-Karp algorithm [KR87], either for approximate matching or for similarity querying? Can it be truly randomized for any adversary?*

Chapter 8

Summary

In this work, we have studied Constraint Query Algebras (CQAs). CQAs constitute the algebraic component of the Constraint Query Language (CQL) paradigm. CQLs were first introduced in [KKR95], with a focus on the semantics of the declarative constraint query paradigm (Constraint Query Calculi). This paper complements the original work by focusing specifically on Constraint Query Algebras, which we believe is the proper query language for considering the implementational issues of constraint query languages.

In Chapter 2, we provided the foundations for the rest of the work, by presenting the issues involved in designing Constraint Query Algebras. We considered the desirable properties for *data representation*, and for *operator implementation*, and examined the notion of duality of representation (syntax) and meaning (semantics), central to the theory of CQAs. Finally, we defined the principle of *semantic closure*, one which makes explicit the commutativity of the syntactic and semantic interpretations of a CQA.

In Chapter 3, we briefly considered two issues that are crucial to practical implementations of constraint database querying: indexing and optimization. For optimization, we suggested two promising approaches to optimizing CQA queries. The first is *lazy evaluation* of linear and nonlinear constraints (for *real polynomial constraints* see [Tar]; for recent developments and a symbolic computation survey see [Ren92], and for numerical computation see [VMK95]. The second approach is to optimize CQAs by using the indexing information, just as for relational algebras.

We then highlighted the importance of the syntax-semantics duality on two implementational issues in particular: representation of data and implementation of the

algebraic operations. In Chapter 4, we presented the syntax for data representation for dense-order [FG77, Kan95, Klu88] and temporal constraints, and define the algebraic operations over the data. We established the correctness of the dense-order algebra, by proving the commutativity of its syntactic and semantic interpretations. In Section 5, we considered how the implementation of the PROJECT operation can be made more efficient for a subclass of *linear inequality constraints* (see the comprehensive survey in [Sch]. We presented an algorithm for projection over *monotone constraints* which is strongly polynomial in the size of the constraint set whenever the *path expression* for the corresponding *monotone network* is of polynomial size.

In Section 6, we generalized the notion of relational schema to include specification of *variable dependence*, and apply it to produce a restriction on CQA expressions which guarantees closure under the aggregation operator. In the context of constraint databases, we have defined a restricted version of aggregation whose addition to relational algebra results in a closed language. The restriction requires that the set of variables grouped upon is independent of the remaining variables. We formalized this restriction using the notion of *variable independence*. We have shown that for constraint databases with linear arithmetic constraints variable independence is decidable with acceptable complexity. We have also provided a set of rules for inferring variable independence in relational expressions.

Finally, in section 7, we show how queries over spatiotemporal data can be defined in a constraint setting, and implemented with a multidimensional indexing structure. We provide both the description and the implementational details for a framework for similarity querying of time-series data. Similarity queries are strictly more expressive than approximate match queries, for which a framework had been supplied in [FRM94].

Sections 5-7 represent original work, most of which has already been published [KG94, GK95, CGK96, GK96, BG96]. Further work is in progress in the areas of aggregation, monotone constraints, and similarity querying.

Appendix A

The Closure Theorem

In Appendix A, 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 **OP**s instead of *OP*s has the property that:

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

The first three lemmas involve projection.

Lemma 10 *If $\overline{t}$ is canonical and $\overline{t}' = (\overline{t} \downarrow Z)$, then $\overline{t}'$ is canonical.*

Proof:

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

□

Lemma 11 *If $\overline{t}$ is a canonical n-tuple and $\overline{t}' = (\overline{t} \downarrow Z)$, then $P(\overline{t}') = \pi_Z(P(\overline{t}))$*

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 $X = \{x_1, \ldots, x_n\}$, $Z = X - \{x_n\}$. Let p be an assignment satisfying $\bar{t}'$. We define a sequence of constraint sets $\{C_1, \ldots, C_n\}$ as follows:

   $C_0 = \bar{t}' \cup (\bigcup_{1 \leq i \leq n-1}(x_i = p(x_i)))$, where $p(x_i)$ is the constant assigned to x_i in p;
   $C_1 = C_0 \cup \theta$, where $\theta \equiv (l_n = x_n)$ if $(l_n = u_n)$, and $\theta \equiv (l_n \leq x_n \leq u_n)$ otherwise;
   $C_2 = C_1 \cup \xi_1$, where ξ_1 is the two-variable constraint over (x_n, x_1) in $\bar{t}$;
   ...
   $C_n = C_{n-1} \cup \xi_{n-1}$, where ξ_{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 θ 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 ξ_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 ξ_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 12 *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 10 and 11. □

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

Lemma 13 *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 14 *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 15 *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 16 *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 17 $\sigma(tupdif(\bar{t}_1, \bar{t}_2)) = P(\bar{t}_1) - P(\bar{t}_2)$.

Proof:

1. 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, $tupdif(\bar{t}_1, \bar{t}_2) = P(\bar{t}_1) - P(\bar{t}_2)$.

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, $tupdif(\bar{t}_1, \bar{t}_2) = \emptyset = P(\bar{t}_1) - P(\bar{t}_2)$.

3. *Inductive Step*:

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

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

   (b) $\bar{t}_1^0$ and $\bar{t}_2^0$ have one more constraint in common than did $\bar{t}_1$ and $\bar{t}_2$ (ϕ_0 is that constraint), so by inductive assumption, $\sigma(tupdif(\bar{t}_1^0, \bar{t}_0^2)) = P(\bar{t}_1^0) - P(\bar{t}_2^0)$. Therefore,

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

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

□

Lemma 18 *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:

1. 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 $|\bar{r}_2| < m$, $\sigma(\bar{r}_1 - \bar{r}_2) = \sigma(\bar{r}_1) - \sigma(\bar{r}_2)$.

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, $\sigma(\bar{r}_1 - \bar{r}_2) = \sigma(\bar{r}_1) = \sigma(\bar{r}_1) - \sigma(\bar{r}_2)$.

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

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

   (b) Let $\bar{r}_3$ be $tupdif(\bar{t}_1, \bar{t}_2)$. By Lemma 17, $\sigma(\bar{r}_3) = P(\bar{t}_1) - P(\bar{t}_2)$.

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

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

□

Bibliography

[AHV] S. Abiteboul, R. Hull, V. Vianu. *Foundations of Databases.* Addison-Wesley, 1994.

[AV95] L. Agre, J. S. Vitter. Optimal Interval Management in External Memory. Manuscript, November 1995.

[AHO90] A. Aho. Algorithms for Finding Patterns in Strings. *Handbook of TCS.*, J. Van Leeuwen editor, volume A, chapter 5, Elsevier, 1990.

[AHU] A.V. Aho, J.E. Hopcroft, J.D. Ullman. *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Co., 1974.

[AFS93] R. Agrawal, C. Faloutsos, A. Swami. Efficient Similarity Search in Sequence Databases. *FODO Conf.*, Evanston, Ill., Oct. 1993

[AS91] W.G. Aref and H. Samet. Extending a DBMS with Spatial Operations. *International Symposium on Large Spatial Databases*, 299–318, 1991.

[AS80] 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.

[AGSS86] A.K. Aylamazyan, M.M. Gilula, A.P. Stolboushkin, G.F. Schwartz. Reduction of the Relational Model with Infinite Domain to the Case of Finite Domains. *Proc. USSR Acad. of Science (Doklady)*, 286:2:308–311, 1986.

[BNW91] M. Baudinet, M. Niezette, P. Wolper. On the Representation of Infinite Temporal Data and Queries. *Proc. 10th ACM PODS*, 280–290, 1991.

[BM72] R. Bayer, E. McCreight. Organization of Large Ordered Indexes. *Acta Informatica*, 1:173–189, 1972.

[Bor81] A.H. Borning. The Programming Language Aspects of ThingLab, A Constraint-Oriented Simulation Laboratory. *ACM TOPLAS* 3:4:353–387, 1981.

[BJM93] A. Brodsky, J. Jaffar, M.J. Maher. Towards Practical Constraint Databases. *Proc. 19th International Conference on Very Large Data Bases*, Dublin, Ireland, 322–331, 1993.

[BG96] A. Brodsky, D. Q. Goldin, V. Segal. On Strongly Polynomial Projections in d-monotone Constraint Databases. *Workshop on Constraints and Databases, 2st Int'l Conf. on the Principles and Practice of Constraint Programming*, Cambridge Massachusetts, August 1996.

[BK95] A. Brodsky and Y. Kornatzky. The LyriC Language: Constraining Objects. *ACM SIGMOD International Conference on Management of Data*, San Jose, California, 1995.

[BLLM95] A. Brodsky, C. Lassez, J-L. Lassez, and M.J. Maher. Separability of Polyhedra for Optimal Filtering of Spatial and Constraint Data. *ACM Symposium on Principles of Database Systems*, San Jose, California, 1995.

[CK70] D.R.Chand, S.S. Kapur. An Algorithm for Convex Polytopes. *JACM*, 17:1:78–86, 1970.

[CH82] A. K. Chandra and D. Harel. Structure and Complexity of Relational Queries. *J. Comp. System Sci.*, 25:99–128, 1982.

[Cho94] J. Chomicki. Temporal Query Languages: A Survey. *Temporal Logic, First International Conference*, editors D.M. Gabbay and H.J. Ohlbach, Springer-Verlag, LNAI 827, 506–534, 1994.

[CK95] J. Chomicki and G. Kuper. Measuring Infinite Relations. *ACM Symposium on Principles of Database Systems*, 78–85, 1995.

[CGK96] J. Chomicki, D. Goldin, G. Kuper. Variable Independence and Aggregation Closure. *ACM Symposium on Principles of Database Systems*, 40–48, 1996.

[CI89] J. Chomicki, T. Imielinski. Relational Specifications of Infinite Query Answers. *Proc. ACM SIGMOD*, 174–183, 1989.

[Cod70] E.F. Codd. A Relational Model for Large Shared Data Banks. *CACM*, 13:6:377–387, 1970.

[Col80] A. Colmerauer. An Introduction to Prolog III. *CACM*, 33:7:69–90, 1990.

[Dec92] R. Dechter. From Local to Global Consistency. *Artificial Intelligence*, 55:87–107, 1992.

[DMP91] R. Dechter, I. Meiri, J. Pearl. Temporal Constraint Networks. *Artificial Intelligence*, 49:61–95, 1991.

[DVSAGB88] 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.

[Fal96] C. Faloutsos. *Searching Multimedia Databases by Content* Kluwer Academic Publishers, 1996

[FRM94] C. Faloutsos, M. Ranganathan, Y. Manolopoulos. Fast Subsequence Matching in Time-Series Databases. *Proc. ACM SIGMOD Conf.*, 419–429, May 1994

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

[FvDFH] J.D.Foley, A. van Dam, S.K.Feiner, J.F. Hughes *Computer Graphics, Principles and Practice.* Addison-Wesley, 1990.

[Fre78] E. Freuder. Synthesizing Constraint Expressions. *CACM*, 21:11, 1978.

[Fre82] E. Freuder. A sufficient condition for backtrack-free search. *CACM*, 29:1, 1982.

[GK95] D.Q. Goldin and P.C. Kanellakis. On Similarity Queries for Time-Series Data: Constraint Specification and Implementation. *Proc. 1st Int'l Conf. on the Principles and Practice of Constraint Programming*, LNCS 976, 137–153, Cassis France, September 1995.

[GK96] D.Q. Goldin and P.C. Kanellakis. Constraint Query Algebras. *Constraints Journal*, 1st issue (E. Freuder editor), 1–41, 1996.

[GK94] P. Gritzmann and V. Klee. On the Complexity of Some Basic Problems in Computational Convexity: II. Volume and Mixed Volumes. Technical Report TR-94-31, DIMACS, Rutgers University, New Brunswick, NJ, 1994.

[GK97] S. Grumbach, G. Kuper. Tractable Query Languages for Geometric and Topological Data. Submitted to *ACM PODS*, 1997.

[GS95] S. Grumbach, J. Su. Dense Order Constraint Databases. *Proc. 14th ACM PODS*, May 1995, 66–77.

[GST94] S. Grumbach, J. Su, C. Tollu. Linear Constraint Query Languages: Expressive Power and Complexity. *Proc. Workshop on Finite Model Theory*, Indiana, Fall 1994.

[HN94] 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.

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

[Jag91] H.V. Jagadish. A Retrieval Technique for Similar Shapes. *Proc. ACM SIGMOD Conf.*, 208–217, May 1991

[JMM95] H. V. Jagadish, A. O. Mendelzon, T. Milo. Similarity-Based Queries. *Proc. 14th ACM PODS*, 1995

[KSW90] F. Kabanza, J-M. Stevenne, P. Wolper. Handling Infinite Temporal Data. *Proc. 9th ACM PODS*, 392–403, 1990.

[Kan90] P.C. Kanellakis. Elements of Relational Database Theory. *Handbook of Theoretical Computer Science*, (J. van Leeuwen editor), volume B, chapter 17, 1073–1158, North-Holland, 1990.

[Kan95] P.C. Kanellakis. Constraint Programming and Database Languages: A Tutorial. *ACM Symposium on Principles of Database Systems*, 1995.

[KG94] 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.

[KKR95] P. C. Kanellakis, G. M. Kuper, P. Z. Revesz. Constraint Query Languages. *JCSS*, 51:1:26–52, August 1995.

[KRVV93] 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.

[KR87] R. M. Karp and M. O. Rabin. Efficient Randomized Pattern-Matching Algorithms. *IBM J. Res. Develop.*, 31(2), 1987

[Klu82] A. Klug. Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions. *Journal of the ACM*, 29:3:699–717, 1982.

[Klu88] A. Klug. On Conjunctive Queries Containing Inequalities. *JACM*, 35:1:146–160, 1988.

[Kou93] M. Koubarakis. *Foundations of Temporal Constraint Databases.* PhD Thesis. Nat. Tech. Univ. of Athens and Imperial College. 1993.

[Kup94] G. Kuper. Aggregation in Constraint Databases. *PPCP'93, First International Workshop on Principles and Practice of Constraint Programming*, 161–172. MIT Press, 1994.

[Las90] J.L. Lassez. Querying Constraints. *9th ACM Symposium on Principles of Database Systems*, 288–298, 1990.

[Mac77] A.K. Mackworth. Consistency in Networks of Relations. *AI*, 8:1, 1977.

[MM] M. Mantyla. *Solid Modeling.* Computer Science Press, 1988.

[MP] Modenov and Pakhomenko. *Geometric Transformations*, Academic Press, 1965

[Mon74] U. Montanari. Networks of Constraints: Fundamental Properties and Application to Picture Processing. *Information Science*, 7, 1974.

[MFPR90] I. S. Mumick, S. J. Finkelstein, H. Pirahesh, R. Ramakrishnan. Magic Conditions. *Proc. 9th ACM PODS*, 314–330, 1990.

[MPR90] I.S. Mumick, H. Pirahesh, and R. Ramakrishnan. Duplicates and Aggregates in Deductive Databases. *International Conference on Very Large Data Bases*, August 1990.

[MSW] W. Mendenhall, R. Scheaffer, D. Wackerly. *Mathematical Statistics with Applications*. Duxbury Press, Boston MA, 1986.

[Nel78] G. Nelson. An $n^{\log n}$ algorithm for the two-variable-per-constraint linear programming satisfiability problem. Technical Report AIM-319, Stanford University, 1978.

[DS] A.V. Oppenheim and R.W. Schafer. *Digital Signal Processing*, Prentice Hall, 1975

[PBCF93] A.Paoluzzi, F.Bernandini, C.Cattani, V.Ferrucci. Dimension-Independent Modeling with Simplicial Complexes. *ACM Trans. Graphics*, 12:1:56–102, 1993.

[Pa94] C. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[PVV94] J. Paredaens, J. Van den Bussche, and D. Van Gucht. Towards a Theory of Spatial Database Queries. *ACM Symposium on Principles of Database Systems*, 279–288, Minneapolis, Minnesota, 1994.

[PVV95] J. Paredaens, J. Van den Bussche, D. Van Gucht. First-order Queries on Finite Structures over the Reals. *Proc. IEEE LICS*, 1995.

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

[PS86] L.K. Putnam, P.A. Subrahmanyam. Boolean Operations on n-dimensional objects. *IEEE Comput. Graph. Appl.*, 6:6:43–51, 1986.

[Ram88] R. Ramakrishnan. Magic Templates: A Spellbinding Approach to Logic Programs. *Proc. 5th International Conference on Logic Programming*, 141–159, 1988.

[RS94] S. Ramaswamy, S. Subramanian. Path Caching: A Technique for Optimal External Searching. *Proc. 13th ACM PODS*, 14–25, 1994.

[Ren92] 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.

[Rev95] P. Z. Revesz. Safe Stratified Datalog with Integer Order Programs. *International Conference on Constraint Programming*, Marseilles, France, September 1995. Springer-Verlag, LNCS 1000.

[RS92] K. A. Ross and Y. Sagiv. Monotonic Aggregation in Deductive Databases. *ACM Symposium on Principles of Database Systems*, 114–126, 1992.

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

[Sch] A. Schrijver. Theory of Linear and Integer Programming. John Wiley and Sons, 1986.

[SLR94] P. Sheshadri, M. Livny, R. Ramakrishnan. Sequence Query Processing *Proc. ACM SIGMOD Conf.*, 430–441, May 1994

[SR92] D. Srivastava, R. Ramakrishnan. Pushing Constraint Selections. *Proc. 11th ACM PODS*, 301–316, 1992.

[Ste80] G.L. Steele. The Definition and Implementation of a Computer Programming Language Based on Constraints. Ph.D. thesis, MIT, AI-TR 595, 1980.

[SS94] P.J. Stuckey, S. Sudarshan. Compiling Query Constraints. *Proc. 13th ACM PODS*, 56–68, 1994.

[SSRB93] S. Sudarshan, D. Srivastava, R. Ramakrishnan, and C. Beeri. Extending the Well-Founded and Valid Model Semantics for Aggregation. *International Logic Programming Symposium*, 1993.

[Sut] I.E. Sutherland. *SKETCHPAD: A Man-Machine Graphical Communication System.* Spartan Books, 1963.

[Tar81a] R.E. Tarjan. A Unified Approach to Path Problems. *JACM*, 28:3:577–593, 1981.

[Tar81b] R.E. Tarjan. Fast Algorithms for Solving Path Problems. *JACM*, 28:3:594–614

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

[Ull] J. D. Ullman. *Principles of Database Systems*, 2nd edition. Computer Science Press, 1982.

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

[VMK95] 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.

[VG92] A. Van Gelder. The Well-Founded Semantics of Aggregation. *ACM Symposium on Principles of Database Systems*, 127–138, San Diego, California, June 1992.

[VGVG95] L. Vandeurzen, M. Gyssens, and D. Van Gucht. On the Desirability and Limitations of Linear Spatial Database Models. *International Symposium on Large Spatial Databases*, 14–28, 1995.

[Var82] M. Y. Vardi. The Complexity of Relational Query Languages. *Proc. 14th ACM STOC*, 137–146, 1982.

[Yan88] M. Yannakakis. Expressing Combinatorial Optimization Problems by Linear Programs. *Proc. 20th ACM STOC*, 223–228, 1988.