Università degli Studi di Milano
Dipartimento di Scienze dell'Informazione

Ph.D in Computer Science

Ph.D. Thesis

Constraint Databases: Data Models and Architectural Aspects

Barbara Catania

Internal Advisor: Prof. Elisa Bertino

External Advisor: Dr. Gabriel Kuper

Abstract

Applications requiring the support of database technology have been increasing in the last few years. The need for sophisticated functionalities has also lead to the evolution of database theory, requiring the development of appropriate data models and optimization techniques.

The need of representing composite data has lead to the definition of *object-oriented* and *nested relational* data models. Such models extend the traditional relational model to represent not only flat data, modeled as sets of tuples, but also complex data, obtained by arbitrary combinations of set and tuple constructors.

More recently, other applications have required the use of database technology to manage not only finite data but also *infinite* ones. This is the case of spatial and temporal databases (i.e., databases representing spatial and temporal information).

Starting from the observation that often infinite information can be finitely represented by using mathematical constraints, constraint databases have been proposed with the aim of inserting mathematical constraints both at the data and at the language level. At the data level, constraints allow the finite representation of infinite objects. At the language level, constraints increase the expressive power of specific query languages, by allowing mathematical computations. Different mathematical theories can be chosen to represent and query different types of information in different database models.

At least two different issues should be addressed in order to make constraint databases a practical technology. First of all, advanced models have to be defined combining the constraint formalism with the support for sophisticated functionalities. The definition of constraint data models for complex objects and the integration of constraint query languages with external primitives are only some of the topics that should be addressed. As a second aspect, architectures supporting the efficient manipulation of constraints have to be designed. In particular, optimization techniques have to be defined in order to efficiently access constraint objects.

In this dissertation, we investigate the extension of the relational model and the nested relational model with constraints, both from the point of view of data models and optimization techniques. Results about data modeling are presented in the first part of the thesis, whereas results about optimization issues are presented in the second one.

With respect to data modeling, we propose a new semantics for relational constraint databases and we introduce data manipulation languages for this model. In particular, we propose an algebraic language and a calculus-based language, extended with external functions, and we prove their equivalence. As far as we know, this is the first approach to integrate external functions in constraint query languages. An update language is also proposed. As a second contribution, we formally present a nested constraint relational model. This model overcomes most of the limitations of the already proposed nested models by relying on a formal background based on structural recursion and monads.

With respect to optimization, we first discuss logical optimization for the proposed algebraic language. Then, we introduce two new indexing techniques. The first is based on a dual representation for multidimensional spatial objects and allows the efficient detection of all objects that intersect or are contained in a given half-plane. The second technique is based on a segment representation of constraint databases and efficiently detects all segments intersecting a given segment, with a fixed direction. This result is an improvement with respect to the classical stabbing query problem, determining all segments intersecting a given line. The proposed techniques are also theoretically and experimentally compared with respect to other existing techniques, such as R-trees.

Abstract

Sempre più frequentemente nuove applicazioni richiedono l'uso della tecnologia delle basi di dati. Allo stesso tempo, la necessità di funzionalità sempre più sofisticate ha determinato un'evoluzione della teoria delle basi di dati, richiedendo lo sviluppo di modelli dei dati e tecniche di ottimizzazione adeguati alle nuove esigenze.

Il bisogno di rappresentare dati complessi ha portato, ad esempio, alla definizione delle *basi di dati orientate ad oggetti* e delle *basi di dati relazionali annidate.* Questi modelli estendono il tradizionale modello relazionale alla rappresentazione di dati complessi, ottenuti dalla combinazione arbitraria di costruttori tupla e costruttori insieme. Più, recentemente, altre applicazioni hanno richiesto l'uso delle basi di dati per rappresentare e manipolare non solo informazione finita ma anche informazione infinita. Questo è il caso tipico delle basi di dati spaziali e temporali, che richiedono la manipolazione di aspetti spazio-temporali, tipicamente di natura infinita (si pensi all'insieme dei punti che compongono un oggetto spaziale o ad un evento che si ripete periodicamente nel tempo).

Partendo dall'osservazione che spesso l'informazione infinita può essere finitamente rappresentata utilizzando teorie logiche matematiche, le basi di dati con vincoli sono state proposte come un nuovo modello che estende i modelli esistenti alla modellazione e alla manipolazione di informazione infinita tramite l'utilizzo di adeguate teorie logiche. I vincoli, cioè le formule atomiche della teoria prescelta, possono essere utilizzati sotto due distinti punti di vista. Da un punto di vista della rappresentazione dei dati, essi permettono di rappresentare finitamente oggetti di natura infinita. Dal punto di vista del linguaggio di interrogazione, essi estendono i tipici linguaggi alla rappresentazione di computazioni matematiche. La scelta della teoria logica dipende ovviamente dal tipo di applicazione che si intende rappresentare.

Affinchè le basi di dati con vincoli diventino una tecnologia praticamente utilizzabile, almeno due aspetti devono essere presi in considerazione. In primo luogo, è necessario definire nuovi modelli dei dati in grado di sopperire ad esigenze applicative sempre più complesse. La definizione di modelli in grado di integrare vincoli ed oggetti complessi nonchè l'introduzione di primitive esterne nei linguaggi con vincoli

sono alcuni esempi di problematiche che devono essere risolte. In secondo luogo, è necessario definire nuove architetture che garantiscano una efficiente manipolazione dei vincoli. In particolare, la definizione di tecniche di ottimizzazione si pone come una esigenza primaria.

L'argomento centrale di questo lavoro di tesi è l'introduzione del concetto di vincolo nel contesto del modello relazionale e del modello relazionale annidato, sia dal punto di vista dei modelli dei dati sia dal punto di vista dell'ottimizzazione del sistema. I risultati relativi alla modellazione verranno presentati nella prima parte della tesi, mentre i risultati relativi all'ottimizzazione verranno presentati nella seconda parte.

Per quanto riguarda la parte relativa alla modellazione, la tesi propone una nuova semantica di riferimento per basi di dati relazionali con vincoli e introduce adeguati linguaggi di manipolazione. In particolare, in analogia con il modello relazionale, verranno introdotti rispettivamente un linguaggio algebrico ed un linguaggio logico. Tali linguaggi, a differenza di altri linguaggi definiti in letteratura, prevedono l'utilizzo di funzioni esterne per aumentare la capacità espressiva del linguaggio di base. L'equivalenza tra i linguaggi proposti, in presenza di funzioni esterne, verrà inoltre provata formalmente. Accanto alla definizione di opportuni linguaggi di interrogazione, la tesi propone inoltre un linguaggio per l'aggiornamento della base di dati, rappresentata secondo il modello introdotto. Come secondo contributo, la tesi presenta un modello formale per basi di dati relazionali annidate con vincoli. Questo modello estende i modelli di precedente definizione, utilizzando come base formale concetti relativi alla ricorsione strutturale.

Per quanto riguarda la parte relativa all'ottimizzazione, come primo contributo la tesi introduce un sistema di regole di riscrittura per il linguaggio algebrico proposto. Quindi, verranno introdotte due nuove tecniche di indice per basi di dati con vincoli. La prima tecnica si basa su una rappresentazione duale definita per oggetti spaziali multidimensionali e permette di determinare in modo efficiente tutti gli oggetti contenuti o intersecanti un dato semipiano. La seconda tecnica assume una rappresentazione basata su insiemi di segmenti e permette di determinare l'insieme dei segmenti che intersecano un certo segmento dato, avente una direzione prefissata, con complessità molto vicina a quella ottima. Questo problema estende una classica problematica analizzata in letteratura, relativa alla determinaazione di tutti i segmenti che intersecano una certa retta verticale. Le tecniche proposte saranno inoltre confrontate, teoricamente e sperimentamente, con altre tecniche esistenti.

Contents

Chapter 1

Introduction

Information is a crucial resource of any organization. Information must be acquired, processed, transmitted, and stored in order to be adequately used by the organization itself. All these tasks are performed by the *information system*, which usually consists of human resources and automatized tools and procedures to manage information. An information system represents information in form of data. Each datum represents the registration of the description of any characteristic of the reality to be modeled, on a persistent support (for example, a disk), by using a given set of symbols with an intended meaning for the users.

The component of the information system providing facilities for maintaining large amounts of data is called *database system*. A database system therefore consists of a, typically large, set of logically integrated data stored on a persistent support (the *database*) and a *database management system* (DBMS), providing several facilities for storing and efficiently manipulating data in the database.

One of the main goals of a DBMS is to make available a non-ambiguous formalism to represent and manipulate information. Such a formalism is called *data model*. Data manipulation is usually performed by providing a query language, to specify data to be accessed, and an update language, to specify data modifications.

Since databases are often very large and must be accessed and manipulated according to very strict time constraints, a basic requirement for data access and manipulation is *efficiency*. Efficiency is usually measured in terms of number of accesses of secondary storage. For these reasons, techniques have to be developed in order to reduce the number of accesses during data access and manipulation. The specific techniques to be used depend on the particular data representation, i.e., they depend on the chosen data model.

The aim of this dissertation is the investigation of data models and optimization techniques for *constraint databases*, a new approach to represent and manipulate data based on the use of mathematical constraints. In the following, after a brief introduction to constraint databases, we introduce in Section 1.2 research problems and objectives of this dissertation whereas Section 1.3 presents an overview of the dissertation.

1.1 A short introduction to constraint databases

Constraint databases represent a recent area of research that extends traditional database systems by enriching both the data model and the query primitives with constraints.

The idea of programming with constraints is not new. This topic has been investigated since the seventies in Artificial Intelligence [55, 99, 105, 134], Graphical Interfaces [23], and Logic Programming [48, 51, 77, 143]. The constraint programming paradigm is fully declarative, in that it specifies data and computations by stating how these data and computations are constrained.

Due to the declarativity of the constraint programming paradigm, the introduction of such a paradigm in database systems is very attractive. However, the first proposal to introduce constraints in database systems is relatively recent [83]. Constraints, intended as atomic formulas of a decidable logical theory, can be inserted in database systems at two different levels. At the data level, conjunctions of constraints allow infinite sets of objects (all tuples of values that make true the conjunction of constraints in the domain of the logical theory) to be finitely represented. At the language level, constraints increase the expressive power of specific query languages by allowing mathematical computations to be expressed. Different mathematical theories can be chosen to represent and query different types of information.

Due to their ability to finitely represent infinite information, constraint databases represent a good framework for applications requiring the use of database technology to manage not only finite but also *infinite* data. This is the case of spatial and temporal databases. Spatial data are generally infinite since spatial objects can be seen as composed of a possibly infinite set of points. From a temporal perspective, infinite information is very useful to represent situations that are repeated in time.

The concept of constraint is orthogonal to the chosen data model. Approaches have been proposed to insert constraints in the relational model, in the nested relational model, and in the object-oriented model. However, several issues, both related to the definition of advanced data models and to the design of efficient architectures, have still to be investigated in order to make constraint databases a practical technology.

In this dissertation, we focus on the introduction of constraints in relational and nested relational database systems, both from the point of view of data models and optimization techniques.

1.2 Research problems and objectives

The main contributions of this dissertation can be summarized as follows.

Data modeling. The integration of constraints in the relational model is possible by interpreting a conjunction of constraints as a *generalized tuple*, finitely representing a possibly infinite set of relational tuples. In this model, a generalized relation is defined as a set of generalized tuples and finitely represents a possibly infinite set of relational tuples. The *generalized relational calculus* and the *generalized relational algebra* have been introduced as a natural extension of the relational calculus and the relational algebra to deal with infinite, but finitely representable, relations.

Due to the semantics assigned to generalized relations, the user is forced to think in terms of single points (i.e., in terms of relational tuples) when specifying queries; as a consequence, the only way to manipulate generalized tuples as single objects is to assign to each generalized tuple an identifier. This approach may be unsuitable when the constraint database is used to model spatial and temporal information, and in general, in all applications where it may be useful to manipulate generalized tuples as single objects.

The first contribution of this dissertation is the introduction of a new semantics for generalized relations, that overcomes the previous limitation. In the proposed semantics (called *nested semantics*) each generalized relation is interpreted as a finite set of possibly infinite sets, each representing the extension (i.e., the set of solutions) of a generalized tuple in the domain of the chosen theory. Thus, the obtained model admits one level of nesting and represents a natural extension of the original relational semantics for constraint databases.

By using this new semantics, we provide new manipulation languages for relational constraint databases that allow the manipulation of generalized tuples as if they were single objects. In particular, we propose, and prove to be equivalent, an extended generalized algebra and an extended generalized calculus. One of the most interesting properties of these languages is that they contain operators dealing with external functions. The use of external functions is a very important topic in constraint databases since the logical theory, chosen with respect to specific complexity and expressivity requirements, may not always support all the functionalities needed by the considered application. As an example, consider the use of the linear polynomial constraint theory in modeling a spatial application. This theory is usually chosen since it guarantees a good trade-off between expressive power and computational complexity. However, the computation of the Euclidean distance cannot be represented by using this theory, even if this is a common operation in spatial databases [3]. This additional functionality can be provided by the constraint language in the form of external functions.

As we will see, the introduction of external functions complicates the proof of the equivalence of the proposed languages. In particular, the same technique used in [88] to prove the equivalence of the relational calculus and the relational algebra extended with aggregate functions (such as MAX, MIN, COUNT) has to be applied. As far as we know, this is the first approach to integrate external functions in constraint query languages.

Since a database system should provide languages to both query and modify the database, an update language based on the nested semantics is also proposed.

As we have already stated, the proposed model admits only one level of nesting. Some proposals exist in the literature to introduce constraints in the nested relational model, thus admitting any level of nesting. However, most of them are defined only for specific theories or model sets up to a given depth of nesting. Others do not have this restriction, but the definition of a formal basis, that supports the definition and the analysis of relevant language properties, has not been addressed.

The second contribution of this dissertation, with respect to data modeling, is the presentation of a formal nested constraint relational model, overcoming the limitations of other proposals. The definition of this model is based on structural recursion and monads.

Some results about data modeling have already been published. In particular, results about the extended generalized algebra can be found in [12, 13] whereas results about the update language can be found in [14].

Optimization issues. Each conjunction of constraints can be seen as the symbolic representation of the extension of a spatial object. For this reason, optimization techniques proposed in the context of spatial databases can also be applied to constraint databases. However, constraint databases differ from spatial databases in at least two aspects:

- typically, only 2- or 3-dimensional spatial objects are considered whereas constraint databases generally represent arbitrary d-dimensional objects (this is useful for example in Operations Research applications);
- spatial objects are typically closed whereas constraint objects may be unbound.

For the previous reasons, optimization techniques developed in the context of spatial databases cannot always be efficiently applied to constraint databases.

The first contribution of the dissertation with respect to optimization is the presentation of logical rewriting rules for the proposed extended generalized relational algebra. Logical rewriting rules allow rewriting algebraic expressions into equivalent expressions that guarantee a more efficient execution of the query they represent. Such rules are based on specific heuristics, such as performing selections as soon as possible, in order to reduce the size of the intermediate results. The heuristics of the algebraic approach are based on the assumption that selection conditions are readily available. As it has been pointed out in [27], extracting such conditions from the constraints of a query involves mathematical programming techniques which are in general expensive. Therefore, rules presented for the relational algebra have to be extended and modified in order to be applied to the proposed extended generalized algebra.

The second group of contributions is related to the definition of indexing techniques for constraint databases. In particular, two new approaches are presented. The first approach introduces indexing techniques for constraint databases, representing d-dimensional objects, based on the dual transformation for spatial objects presented in [67]. These techniques allow the efficient detection of all generalized tuples that intersect or are contained in a given half-plane. The proposed techniques are also experimentally compared with respect to R-trees, a well known spatial data structure [71, 130]. The obtained results show that the proposed techniques perform very well in almost all situations. The second approach is based on a segment representation for constraint databases. A technique is introduced to efficiently determine all segments intersecting a given segment, with a fixed direction. Preliminary experimental results are also reported.

Some results about the proposed indexing techniques have already been published. In particular, results about the indexing technique based on the dual representation can be found in [19] whereas results about the technique based on the segment representation can be found in [20].

The interested reader is referred to [18, 16] for some results, not discussed in this dissertation, about the application of constraint databases to model and manipulate shapes in multimedia databases.

1.3 Overview of this dissertation

Since the dissertation covers two distinct but complementary topics – data modeling and query optimization – the dissertation has been organized in two parts, the first presenting results about data modeling and the second presenting results about optimization techniques. The first chapter of each part surveys the basic concepts and the main results proposed in the context of constraint data modeling and constraint optimization, respectively. More precisely, the thesis is organized as follows.

Part I is dedicated to data modeling in constraint databases. Chapter 2 formally introduces relational constraint databases and surveys the existing proposals for introducing constraints in the nested relational model and in the object-oriented model. Examples of applications of constraints to model spatial and temporal data are also presented.

In Chapter 3, the new reference model for relational constraint databases is introduced. The general properties of languages based on such a model are formally stated. Then, an algebra and a calculus based on this model and extended with external functions are introduced and proved to be equivalent.

Chapter 4 presents an update language based on the same principles on which languages introduced in Chapter 3 are based.

In Chapter 5, a formal nested constraint relational model is introduced, overcoming the limitations of other proposals. The definition of this model is based on structural recursion and monads.

Part II deals with optimization techniques in constraint databases. Chapter 6 surveys the state of the art with respect to indexing techniques and query optimization.

In Chapter 7, optimization rules for the algebraic language, introduced in Chapter 3, are presented, pointing out the differences existing with respect to classical relational optimization rules.

In Chapter 8, an indexing technique for constraint databases representing objects in an arbitrary d-dimensional space is introduced. This technique is based on a dual representation for spatial objects, first proposed in [67].

In Chapter 9, another indexing technique, based on a segment representation for constraint databases is formally introduced.

In Chapter 10 the contributions of this dissertation are summarized and future work is outlined.

Finally, three appendices are also included. They present the proofs of the main results introduced in Chapters 3, 5, and 8, respectively.

Part I

Data modeling in constraint databases

Chapter 2

Introducing constraints in database systems

The constraint programming paradigm has been successfully used in several areas, such as Artificial Intelligence [55, 99, 105, 134], Graphical Interfaces [23], and Logic Programming [48, 51, 77, 143]. The main idea of constraint languages is to state a set of relations (*constraints*) among a set of objects in a given domain. It is a task of the *constraint satisfaction system* (or constraint solver) to find a solution satisfying these relations. An example of a constraint is $F = 1.8 * C + 32$, where C and F are respectively the Celsius and Fahrenheit temperature. The constraint defines the relation existing between F and C. Thus, the constraint programming paradigm is fully declarative, in that it specifies computations by specifying how these computations are constrained.

Constraints have been used in several contexts. For example they have been successfully integrated with Logic Programming, leading to the development of Constraint Logic Programming (CLP) as a general-purpose framework for computation [77]. One reason for this success is that the operation of first-order term unification is a form of efficient constraint solving (for equality constraints only). More expressive power can therefore be obtained by replacing unification with more general constraint solving techniques and allowing constraints in logic programs [48, 51, 77, 143]. This extension of logic programming has found applications in several contexts, including Operations Research and Scientific Problem Solving.

Even though constraints have been used in several fields, only recently this paradigm has been introduced in database systems. The delay in developing this integration has been due to the fact that for some time it was not clear how the bottom-up and set-at-a-time style of database query evaluation would be integrated with a top-down, depth-first evaluation typical of Constraint Logic Programming. The key intuition to fill this gap is due to Kuper, Kanellakis and Revesz [83]:

> *a tuple in traditional databases can be interpreted as a conjunction of equality constraints between attributes of the tuple and values on a given domain.*

The introduction of new logical theories to express relationships (i.e., constraints) between the attributes of a database item leads to the definition of *Constraint Databases* as a new research area, lying at the intersection of Database Management, Constraint Programming, Mathematical Logic, Computational Geometry, and Operations Research.

Note that this approach is different from the traditional use of constraints in database systems to express conditions on the semantic correctness of data. Those constraints are usually referred to as *semantic integrity constraints.* Integrity constraints have no computational implications. Indeed, they are not used to execute queries (even if they can be used to improve execution performance [69]) but only to check database validity.

The aim of this chapter is to introduce constraint databases from the point of view of data modeling. Issues related to optimization of constraint databases will be discussed in Chapter 6. The chapter is organized as follows. The basic aspects related to the introduction of constraints in database systems are discussed in Section 2.1. Section 2.2 deals with the use of constraints to model data whereas Section 2.3 introduces constraint query languages. Section 2.4 surveys some approaches to model complex objects in constraint databases. Finally, Section 2.5 discusses some relevant applications of constraint databases and Section 2.6 presents some conclusions.

2.1 Constraints and database systems

Formally, a *constraint* represents an atomic formula of a decidable logical theory [38]. Each first order formula ϕ with constraints (also called *constraint formula*), with free variables $X_1, ..., X_n$, is interpreted as a set of tuples $(a_1, ..., a_n)$ over the schema $X_1, ..., X_n$ that satisfy ϕ.

By taking this point of view, constraints can be added to database systems at different levels:

- At the data level, each constraint formula finitely represents a possibly infinite set of points (i.e., relational tuples). For example, the conjunction of constraints $X < 2 \wedge Y > 3$, where X and Y are real variables, represents the infinite set of tuples having the X attribute less than 2 and the Y attribute greater than 3.

Thus, constraints are a powerful mechanism that can finitely model infinite information, such as spatial and temporal data. Indeed, spatial objects can be seen as infinite sets of points, corresponding to the solutions of particular arithmetic constraints. For example, the constraint $X^2 + Y^2 \leq 9$ represents a circle with center in the point $(0, 0)$ and with radius equal to 3. From a temporal perspective, constraints are very useful to represent situations that are infinitely repeated in time. For example, we may think of a train leaving each day at the same time.

- At the query language level, constraints increase the expressive power of existing query languages by allowing mathematical computations to be specified. This integration raises several issues. Constraint query languages should preserve all the nice features of relational languages. For example, they should be closed (i.e., the result of each query execution should be representable by using the underlying data model) and bottom-up evaluable.

In the following, we assume that each theory Φ is associated with a specific structure of interpretation $\mathcal{D}$, having D has domain [38]. To simplify the notation, we use D to denote both the interpretation structure and its domain [38]. Among the theories that can be used to model and query data, we recall the following:

- *Real polynomial inequality constraints* (POLY): all the formulas of the form $p(X_1, ..., X_n)\ \theta\ 0$, where p is a polynomial with real coefficients in variables $X_1, ..., X_n$ and $\theta \in \{=, \neq, \leq, <, \geq, >\}$. The domain D is the set of real numbers and function symbols $+$, $*$, predicate symbols θ and constants are interpreted in the standard way over D.

- *Dense linear order inequality constraints* (DENSE): all the formulas of the form $X\theta Y$ and $X\theta c$, where X, Y are variables, c is a constant and $\theta \in \{=, \neq, \leq, <, \geq, >\}$. The domain D is a countably infinite set (e.g. the rational numbers) with a binary relation which is a dense linear order. Constants and predicate symbols θ are interpreted in the standard way over D.

- *Equality constraints over an infinite domain* (EQ): all the formulas of the form $X\theta Y$ and $X\theta c$, where X, Y are variables, c is a constant and $\theta \in \{=, \neq\}$. The domain D is a countably infinite set (e.g. the integer numbers) but without order. Constants and predicate symbols θ are interpreted in the standard way over D.

When polynomials are linear, the corresponding class of constraints (LPOLY) is of particular interest. Indeed, a wide range of applications (Geographic Information Systems, Operations Research applications) use linear polynomials. Linear polynomials have been studied in various fields (Linear Programming, Computational Geometry) and therefore several efficient techniques have been developed to deal with them [94].

In the logical theories listed above, variables range among elements of a certain numerical domain (for example reals or rationals). Other classes of constraints have been considered, in which variables range among *sets* of elements of a certain domain [35, 122]. Such constraints are useful to represent and query complex objects (see Section 2.4).

As stated by Kanellakis, Kuper and Revesz, the integration of constraints in traditional databases must not compromise the efficiency of the system [83]. This means that the resulting languages must have a reasonable complexity and that optimization techniques already developed for traditional databases have to be extended to deal with constraints. The second part of this dissertation deals with this topic.

2.2 Using constraints to represent data

The use of constraints to model data is based on the consideration that a relational tuple is a particular type of constraint [83]. For example, the tuple $(3, 4)$, contained in a relation r with two real attributes X and Y, can be interpreted as the conjunction of equality constraints $X = 3 \wedge Y = 4$.

By adopting more general theories to represent constraints, the concept of relational tuple can be generalized to be a conjunction of constraints on the chosen theory. For example, the formula $X < 2 \wedge Y > 5$, where X and Y are real variables, can be interpreted as a *generalized tuple*, representing the set of relational tuples $\{X = a \wedge Y = b \mid a < 2, b > 5, a \in \mathbb{R}, b \in \mathbb{R}\}$. From the previous example, it follows that the introduction of constraints at the data level allows us to finitely represent *infinite* sets of relational tuples, all those representing solutions of the generalized tuple in the domain associated with the chosen theory. In this framework, relational attributes can still be represented by simple equality constraints.[1]

Formally, the relational model can be extended to deal with constraints as follows.

Definition 2.1 (Generalized relational model) *Let Φ be a decidable logical theory.*

- *A generalized tuple t over variables $X_1, ..., X_k$ and on the logical theory Φ is a finite conjunction $\varphi_1 \wedge ... \wedge \varphi_N$, where each φ_i, $1 \leq i \leq N$, is a constraint over Φ. The variables in each φ_i are among $X_1, ..., X_k$. The schema of t, denoted by $\alpha(t)$, is the set $\{X_1, ..., X_k\}$.*

[1]This is a convenient simplification from a theoretical point of view. However, from a practical point of view, separately handling regular data and constraint data may have certain advantages.

- *A generalized relation r^2 of arity k on Φ is a finite set $r = \{t_1, ..., t_M\}$ where each t_i, $1 \leq i \leq M$, is a generalized tuple over variables $X_1, ..., X_k$ and on Φ. The schema of r, denoted by $\alpha(r)$, is the set $\{X_1, ..., X_k\}$. The degree of r, denoted by $deg(r)$, is k.*

- *A generalized database is a finite set of generalized relations. The schema of a generalized database is a set of relation names $R_1, ..., R_n$, each with the corresponding schema.* $\Box$

Generalized relations are interpreted as the finite representation of a possibly infinite set of relational tuples.

Definition 2.2 (Relational semantics) *Let Φ be a decidable logical theory. Let D be the domain of Φ. Let $r = \{t_1, ..., t_n\}$ be a generalized relation on Φ. Let $ext(t_i) = \{\sigma | \sigma : \alpha(t_i) \to D, D \models_\sigma t_i\}$.*[3] *A generalized tuple t_i is inconsistent if $ext(t_i) = \emptyset$, i.e., if $\not\exists \sigma$ such that $D \models_\sigma t_i$. The relational semantics of r, denoted by $rel(r)$, is $ext(t_1) \cup ... \cup ext(t_n)$. Two generalized tuples t_i and t_j, such that $\alpha(t_i) = \alpha(t_j)$, are equivalent (denoted by $t_i \equiv_r t_j$) iff $ext(t_i) = ext(t_j)$. Two generalized relations r_1 and r_2 are r-equivalent (denoted by $r_1 \equiv_r r_2$) iff $rel(r_1) = rel(r_2)$.* $\Box$

The set $ext(t)$ for a generalized tuple t, and the set $rel(r)$ for a generalized relation $r = \{t_1, ..., t_n\}$, are sets of assignments, making t or the formula $t_1 \vee ... \vee t_n$ true in the considered domain. However, each assignment can be seen as a relational tuple. Therefore, in the following the elements of $ext(t)$ or $rel(r)$ are called either assignments or relational tuples, depending from the context.

From the relational semantics of relational constraint databases it follows that there exists a strong connection between generalized tuples and spatial objects. In particular, a generalized tuple with d variables can always be interpreted as a d-dimensional spatial object.

Example 2.1 *Consider a spatial database consisting of a set of rectangles in the plane. A possible representation of this database in the relational model is with a relation containing a tuple of the form (n, a, b, c, d) for each rectangle. In such a tuple, n is the name of the rectangle with corners (a, b), (a, d), (c, b) and (c, d). In the generalized relational model, rectangles can be represented by generalized tuples of the form $(ID = n) \wedge (a \leq X \leq c) \wedge (b \leq Y \leq d)$. Figure 2.1 shows the rectangles corresponding to the generalized tuples contained in relation r_1 (white) and relation r_2 (shadow). r_1 contains the following generalized tuples:*

[2] In the following, we use lowercase letters to denote generalized relations and uppercase letters to denote generalized relation names.

[3] $\models$ denotes the logical consequence symbol. Thus, $D \models_\sigma t_i$ means that $t_i\sigma$ is true in D [38].

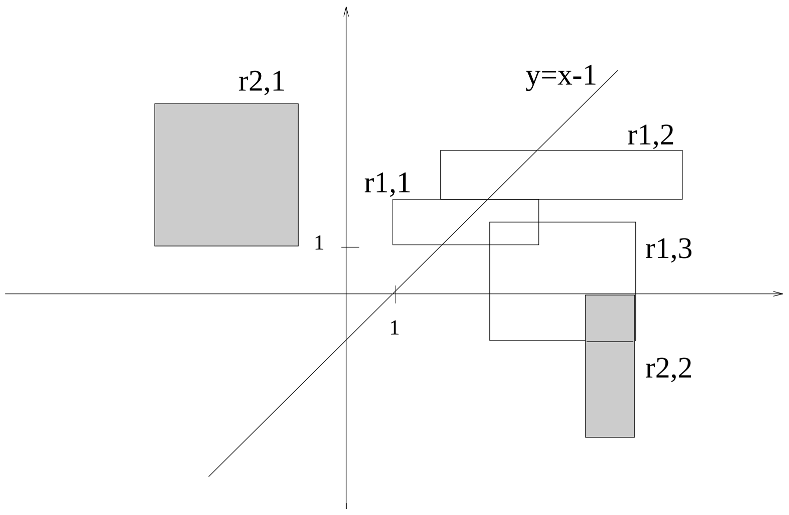


Figure 2.1: Relation r_1 (white) and r_2 (shadow).

$r_{1,1} : ID = 1 \wedge 1 \leq X \leq 4 \wedge 1 \leq Y \leq 2$
$r_{1,2} : ID = 2 \wedge 2 \leq X \leq 7 \wedge 2 \leq Y \leq 3$
$r_{1,3} : ID = 3 \wedge 3 \leq X \leq 6 \wedge -1 \leq Y \leq 1.5$

r_2 *contains the following generalized tuples:*

$r_{2,1} : ID = 1 \wedge -3 \leq X \leq -1 \wedge 1 \leq Y \leq 4$
$r_{2,2} : ID = 2 \wedge 5 \leq X \leq 6 \wedge -3 \leq Y \leq 0.$ $\diamond$

When dealing with the generalized relational model, a significant problem is how generalized tuples are represented inside the database. A specific representation for generalized tuples is often called *canonical form*. The use of canonical forms should reduce tuple storage space and query execution time. In particular, as pointed out in [58], canonical forms should satisfy the following properties:

- *Efficiency.* Canonical forms should be efficiently computed and efficiently stored.
- *Succinctness.* By using canonical forms, the detection of inconsistent generalized tuples should be efficiently performed, possibly in constant time. Indeed, inconsistent generalized tuples do not contribute to the definition of the generalized relation semantics but increase the storage occupied by the relation. Therefore, they should be removed from the database.

- *Redundancy.* Canonical forms are usually obtained by removing redundant constraints from the original generalized tuples, since they not add any new information. In general, this is an expensive operation. Therefore, often, only a partial removal of redundant constraints is performed.

  At the generalized relation level, redundancy corresponds to the presence of generalized tuples whose extension have a non empty intersection. Canonical forms for generalized relations should avoid this situation.

- *Updates.* In a database system, it is highly desirable to perform insertions and updates in time depending only on the size of the individual generalized tuple to be inserted or updated. Since generalized tuples have to be converted into their canonical form before being inserted or updated, the generation of the canonical form must not require a scan of the entire relation.

A canonical form for dense-order constraints, based on tableaux, has been proposed by Kanellakis and Goldin [82] whereas a canonical form for linear polynomial constraints has been proposed in [61].

2.2.1 Additional topics

As we have seen, each generalized tuple represents a possibly infinite set of relational tuples, one for each assignment that makes the generalized tuple true in the domain of the chosen theory. Thus, an universal quantification is assumed when assigning the semantics to a generalized tuple.

A different approach has been taken in [91] and [133], where each generalized tuple is interpreted in an existential way. This means that *only one* assignment that makes the generalized tuple true is taken as semantics of the generalized tuple. Thus, a generalized tuple is interpreted as a set of *possible* values for schema variables. In [133], several semantics with respect to different understandings of incomplete information are also proposed.

2.3 Using constraints to query data

From the language point of view, it is important to develop declarative, powerful, closed, and, at the same time, efficient and bottom-up evaluable constraint query languages.

A constraint language is closed if each query that can be expressed in the language, when applied to any input generalized database on a certain theory Φ, returns a new generalized relation that can be represented by using Φ. A constraint language is bottom-up evaluable if it is possible to assign a bottom-up semantics to any expression of the language, i.e., the semantics is assigned by induction on the structure of the expressions.

The basic idea in defining a constraint query language is to extend an already existing query language to deal with constraints. Such integration should preserve the "philosophy" of the host language and add only a few new concepts. In defining such languages, a careful balance between expressive power, computational complexity, and efficient representation should be achieved. This is in general possible by using Mathematical Programming (e.g. [129]), Computational Geometry (e.g. [118]) and Operations Research (e.g. [147]) techniques.

One of the first approaches in this direction has been proposed by Hansen, Hansen, Lucas and van Emde Boas [72]. In their proposal, constraints modeling infinite relations are used to increase the expressive power of relational languages. However, they do not introduce constraints at the data level. The first general principle underlying the design of constraint database languages has been proposed by Kanellakis, Kuper and Revesz [83]. In their paper, the syntax of a constraint query language is defined as the union of an existing database query language and a decidable logical theory.

Another fundamental property of data manipulation languages is *declarativity*. By using a declarative language, a user can specify what he/she wants to retrieve without specifying how the items of interest have to be retrieved. However, to make possible query optimization and efficient evaluation, declarative user queries have to be translated into equivalent procedural expressions before they are optimized and evaluated. In the relational model, the relational algebra represents the procedural language corresponding to the declarative relational calculus. Similarly to the relational model, two languages, the *generalized relational calculus* and the *generalized relational algebra*, for the generalized relational model are proposed and proved to be equivalent.

In the following we survey both the generalized relational calculus and the generalized relational algebra. Then, we briefly introduce complexity of constraint languages and survey some further topics related to constraint query languages, such as the introduction of aggregate functions and recursion.

2.3.1 The generalized relational calculus

The syntax of a calculus-based constraint query language can be defined as the union of an existing calculus-based query language and a decidable logical theory. In the relational model, the relational calculus is defined as first order logic extended with an additional predicate symbol for each relation name [47]. By extending the relational model with a logical theory, each constraint of the theory becomes a new atomic formula for the considered query language.

Example 2.2 *Consider a generalized relation name R, having as schema* $\{ID, X, Y\}$*, with the meaning introduced in Example 2.1. In order to express the query retrieving all the pairs of intersecting rectangles, using the relational approach, we have to write:*[4]

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

In the generalized relational calculus, the above query is expressed as follows:

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

Note that the use of constraint query languages supports more compact and clearer query representation.

A query to retrieve all rectangles intersecting the half-plane $Y \geq X+2$ *is expressed as follows:*

$$\{(n_1) \mid (\exists x, y)\ (R(n_1, x, y) \wedge y \geq x + 2)\}.$$

In the previous expression, constraint $y \geq x + 2$ *is used to express a condition on data.* ◇

In order to formally define the semantics of generalized relational calculus expressions, predicate symbols and constraints have to be interpreted. The meaning of predicate symbols depends on the input generalized relations, whereas the meaning of the constraint symbols depends on the particular constraint theory. Thus, the semantics of the language is based on the semantics of the chosen decidable logical theory, by interpreting database atoms as shorthand for formulas of the theory. Formally, let $\phi = \phi(x_1, ..., x_n)$ be a calculus expression using free variables $x_1, ..., x_n$. Let predicate symbols $R_1, ..., R_m$ in ϕ name the input generalized relations and let $r_1, ..., r_m$ be the corresponding input generalized relations. Let $\phi[r_1/R_1, ..., r_m/R_m]$ be the formula of the theory obtained by replacing each database atom $R_i(z_1, ..., z_k)$ in ϕ with the formula corresponding to the input generalized relation r_i, with its variables appropriately renamed to $z_1, ..., z_k$. The output is the possibly infinite set of points in the n-dimensional space D^n, such that the instantiation of the free variables $x_1, ..., x_m$ of formula $\phi[r_1/R_1, ..., r_m/R_m]$ to any one of these points makes the formula true.

[4]We use uppercase letters to denote variables belonging to the relation schema and lowercase letters to denote variables inside calculus expressions.

Example 2.3 *Consider a generalized relation r containing the generalized tuples $ID = 1 \wedge X \geq 2 \wedge Y \leq 1$ and $ID = 2 \wedge X = 3 \wedge Y \geq 4$. Let $\phi(n_1, n_2)$ be the first query presented in Example 2.2. The result of the query, when applied to r, can be represented as follows:*

$$\{(a_1, a_2) \mid \phi[R/r][a_1/n_1, a_2/n_2] \textit{ is true}\}$$

where $\phi[R/r]$ denotes the formula obtained by replacing $R(n_1, x, y)$ in ϕ with the formula $(n_1 = 1 \wedge x \geq 2 \wedge y \leq 1) \vee (n_1 = 2 \wedge x = 3 \wedge y \geq 4)$ and $R(n_2, x, y)$ in ϕ with the formula $(n_2 = 1 \wedge x \geq 2 \wedge y \leq 1) \vee (n_2 = 2 \wedge x = 3 \wedge y \geq 4)$. The result of this query is the set of all the pairs (a_1, a_2) such that, when n_1 is replaced with a_1 and n_2 is replaced with a_2, $\phi[R/r]$ is evaluated to true. $\diamond$

In the following, the set of generalized calculus expressions is denoted by GCAL. From a formal point of view, each query language $\mathcal{L}$ is associated with a *semantic function* μ. Such function takes an expression e of the language and returns a new function, called *query*, representing the semantics of e. Thus, $\mu(e)$ is a function that takes a database, represented by using the considered data model, and returns a new database.[5] In the following, the set of queries, represented by the language obtained by extending the relational calculus with a logical theory Φ, is denoted by GCAL(Φ).

Not all combinations of the relational calculus with decidable logical theories lead to a closed constraint query language, as the following example shows.

Example 2.4 *Consider the theory of real polynomial equalities. These are constraints of the form $p(x_1, ..., x_n)\ \theta\ 0$, where θ is $=$ or $\neq$. Let $R(X, Y)$ be a binary predicate symbol for the input generalized relation $\{Y = X^2\}$. The result of $\exists x.R(x, y)$ is the set $\{Y \mid Y \geq 0\}$, which cannot be represented by polynomial equality constraints.* $\diamond$

A fundamental property of the relational calculus is *safety* [140]. Safety guarantees that the result of any calculus expression is a finite relation. This assumption is superfluous for the generalized relational calculus; indeed, if the language is closed, the result of any query can be finitely represented by using the chosen theory Φ.

2.3.2 The generalized relational algebra

The algebraic approach represents the correct formalism to obtain both a formal specification of the language and a suitable basis for implementation.

[5]Actually, $\mu(e)$ is a query if it is a partial mapping between database instances, invariant with respect to permutations of the domain [36].

The class of algebras (one for each decidable theory) we present in the following is a direct extension of the relational algebra and is derived from the algebra presented in [59, 82, 115]. Since we do not consider any particular theory, no assumption about the constraint representation is used in defining such algebra.

Table 2.1 presents the operators of the algebra.[6] Following the approach proposed in [81], each operator of Table 2.1 is described by using two kinds of clauses: those presenting the *schema restrictions* required by the argument relations and by the result relation, and those introducing the operator semantics. $R_1, ..., R_n$ are relation names and e represents the syntactic expression under analysis. The semantics of expressions is described by using an interpretation function μ that takes an expression e and returns the corresponding query. The query takes a set of generalized relations on a theory Φ and computes a new generalized relation as result.

Finally, note that Table 2.1, together with the resulting relation, also presents the relational semantics of such relation.[7] Table 2.1 thus defines a class of algebras, one for each logical theory Φ.

From the definition of operators, it follows that the algebra is closed if projection and complement operators guarantee closure. Since projecting out some variables logically corresponds to existentially quantifying a formula and then removing the quantifier, closure of the projection operator is guaranteed if the chosen logical theory admits *variable elimination* [38]. A theory admits variable elimination if each formula $\exists X F(X)$ of the theory is equivalent to a formula G, where X does not appear. On the other hand, the complement operator is closed if the logical theory is *closed under complementation*, i.e., if, when c is a constraint of Φ, then $\neg c$ is equivalent to another constraint c' of Φ.

Given a constraint theory Φ, admitting variable elimination and closed under complementation, we denote by GRA(Φ) the set of all the queries that can be expressed in the algebra on theory Φ and with GRA the set of corresponding expressions.

GRA(Φ) satisfies an important property: the result of the application of a GRA(Φ) query to a generalized database corresponds to the application of the corresponding relational algebra query to the relational database, represented by the relational semantics of the input generalized relations. This property is stated by the following

[6] Complement has been included to prove the equivalence of the generalized relational algebra with the generalized relational calculus. Actually, the algebra proposed in [82] does not include the complement operator. This operator can be simulated if a relation representing all possible relational tuples on the given domain is provided. In our setting, we assume that algebraic operators can only be applied to relations belonging to the database schema. Therefore, we need to explicitly include this operator.

[7] Other interpretations could have been defined, maintaining the same semantics for resulting relations.

Op. name	**Syntax** e	**Semantics** $r = \mu(e)(r_1, ..., r_n)$[a] $n \in \{1, 2\}$
		Restrictions
atomic relation	R_1	$rel(r) = rel(r_1)$ $r = r_1$
		$\alpha(e)$[b] $= \alpha(R)$
selection	$\sigma_P(R_1)$	$rel(r) = \{t \mid t \in rel(r_1), t \in ext(P)\}$ $r = \{t \wedge P \mid t \in r_1, ext(t \wedge P) \neq \emptyset\}$
		$\alpha(P) \subseteq \alpha(R_1)$ $\alpha(e) = \alpha(R_1)$
renaming	$\varrho_{[A\mid B]}(R_1)$	$rel(r) = \{t[A \mid B]$[c]$: t \in rel(r_1)\}$ $r = \{t[A \mid B] : t \in r_1\}$
		$A \in \alpha(R_1), B \not\in \alpha(R_1)$ $\alpha(e) = (\alpha(R_1) \setminus \{A\}) \cup \{B\}$
union	$R_1 \cup R_2$	$rel(r) = rel(r_1) \cup rel(r_2)$ $r = \{t \mid t \in r_1$ or $t \in r_2\}$
		$\alpha(e) = \alpha(R_1) = \alpha(R_2)$
projection	$\Pi_{[X_{i_1},...,X_{i_p}]}(R_1)$	$rel(r) = \{\pi_{[X_{i_1},...,X_{i_p}]}(t)$[d]$: t \in rel(r_1)\}$ $r = \bigcup_{t \in r_1} \pi_{[X_{i_1},...,X_{i_p}]}(t)$[e]
		$\alpha(R_1) = \{X_1, ..., X_m\}$ $\alpha(e) = \{X_{i_1}, ..., X_{i_p}\}$ $\alpha(e) \subseteq \alpha(R)$
natural join	$R_1 \bowtie R_2$	$rel(r) = \{t_1 \bowtie t_2 : t_1 \in rel(r_1), t_2 \in rel(r_2)\}$ $r = \{t_1 \wedge t_2 \mid t_1 \in r_1, t_2 \in r_2, ext(t_1 \wedge t_2) \neq \emptyset\}$
		$\alpha(e) = \alpha(R_1) \cup \alpha(R_2)$
complement	$\neg R_1$	$rel(r) = \{t \mid t \not\in rel(r_1)\}$ $r = \{\overline{t}_1, ..., \overline{t}_m \mid \overline{t}_1 \vee ... \vee \overline{t}_m$ is the disjunctive normal form of $\neg t_1 \wedge ... \wedge \neg t_n$, $r_1 = \{t_1, ..., t_n\}$, $ext(\overline{t}_i) \neq \emptyset, i = 1, ..., m\}$
		$\alpha(e) = \alpha(R_1)$

[a]We assume that r_i does not contain inconsistent generalized tuples, $i = 1, ..., n$.

[b]We denote by $\alpha(e)$ the schema of the relation obtained by evaluating the query corresponding to expression e.

[c]Given an expression F, $F[A \mid B]$ replace variable A in F with variable B.

[d]This is the relational projection operator.

[e]Given a generalized tuple t, the expression $\pi_{[X_{i_1},...,X_{i_p}]}(t)$ represents the set of generalized tuples obtained by applying a quantifier elimination algorithm to the formula $\exists \alpha(R_1) \setminus \{X_{i_1}, ..., X_{i_p}\}\ t$.

Table 2.1: GRA operators.

Op. name	Syntax e	Restrictions	Derived expression
difference	$R_1 \setminus R_2$	$\alpha(e) = \alpha(R_1) = \alpha(R_2)$	$R_1 \bowtie \neg R_2$
Cartesian product	$R_1 \times R_2$	$\alpha(r_1) \cap \alpha(r_2) = \emptyset$ $\alpha(e) = \alpha(R_1) \cup \alpha(R_2)$	$R_1 \bowtie R_2$
intersection	$R_1 \cap R_2$	$\alpha(e) = \alpha(R_1) = \alpha(R_2)$	$R_1 \bowtie R_2$

Table 2.2: GRA derived operators.

proposition.

Proposition 2.1 (Closure property) *[82] Let OP be a GRA operator and let OP^{rel} be the corresponding relational algebra operator. Let r_i, $i = 1, ..., n$, be generalized relations on theory Φ. Then*
$rel(\mu(OP)(r_1, ..., r_n)) = \mu(OP^{rel})\ (rel(r_1), ..., rel(r_n))$. □

By using the operators of Table 2.1, some useful derived operators can be defined, whose semantics is described in Table 2.2.

It has been proved that, given a theory Φ admitting variable elimination and closed under complementation, GRA(Φ) and GCAL(Φ) are equivalent [82].

2.3.2.1 Operation efficiency

From a formal point of view, generalized algebraic operators are a direct extension of relational algebraic operators dealing with infinite relations. The same does not hold for their implementation. Indeed, algorithms for implementing generalized relational algebra operators are significantly different from those for the relational algebra, since they rely on approaches developed in mathematical programming, computational geometry, and operations research.

A minimal requirement for practical constraint databases is that each algebraic operation must be "efficiently implementable". This means that the additional operations that must be performed to evaluate algebraic operators, with respect to the corresponding relational ones, must have a "reasonable cost". In general, for generalized tuples containing m constraints and k variables, algorithms should be polynomial in m and k (also called *strongly polynomial algorithms*). The main algorithms that have to be applied in evaluating algebraic operators can be summarized as follows:

- *Projection.* One of the most critical issue in designing a "good" algebra is to make projection simple and cheap. Indeed, projection is a very trivial operation in relational databases; however, in constraint databases, this operation conceptually corresponds to the application of an existential quantifier elimination algorithm (for example, the Fourier-Motzkin algorithm [129]) to a generalized tuple. In general, for a set of m linear constraints with k variables, elimination of some variables has a worst case complexity bound exponential in m and k. This complexity is too high when these algorithms are applied in query execution. Strongly polynomial algorithms for dense-order constraints and for a specific class of linear constraints have been proposed [58, 59, 26].

In order to reduce the overhead deriving from the application of the projection operator, it may be useful to delay projection until the relation has to be returned to the user [58]. Under this approach, existentially quantified variables are maintained inside generalized tuples contained in a generalized relation, leading to a *lazy representation* of the relation itself.

- *Satisfiability check.* As we have already stated, inconsistent generalized tuples should be removed, in order to reduce the occupancy of the relation and the query execution time. In order to detect inconsistent generalized tuples, a *satisfiability* check must be performed. The satisfiability check is usually performed by eliminating all variables and establishing if the obtained formula is always true in the considered domain. Such algorithms have worst-case polynomial time complexity in k and m [74].

- *Redundancy elimination.* Algebraic operators often introduce redundant constraints inside the generalized tuples. For example, projection of a generalized tuple often contains more constraints than the original tuple, most of them being redundant. A similar situation arises for selection, join, and complement that, by conjuncting different generalized tuples, may introduce redundant constraints in the generalized tuples. Finally, when performing the union of two relations, redundant tuples, corresponding to the presence of duplicates in the relation, should be removed.

  Redundancy elimination is a very expensive operation. For example, by assuming we deal with LPOLY, removing redundant generalized tuples from a generalized relation is a co-NP complete problem [131].

2.3.3 Complexity of relational constraint languages

Constraint query languages can be used in practice only if the *data complexity* of the queries they represent is low. The *data complexity* of a query Q is the time complexity, measured with respect to the size of the database, of a Turing machine that, given an input database d, produces a new database $Q(d)$ as output, assuming a standard encoding of the database [83, 144] (see [8] for an introduction to structural complexity).

The data complexity of a language is considered acceptable if it is in PTIME, i.e., if all queries that can be expressed in the language can be executed in polynomial time in the size of the input database.

Data complexity is a common tool for analyzing expressibility in finite model theory. The complexity parameter in data complexity is the number of items contained in the database. Therefore, in constraint databases, this corresponds to the number of generalized tuples contained in the input generalized relations. Under this approach, the number of variables k contained in the generalized tuples is treated as a constant [37, 144]. This use of data complexity distinguishes the constraint database framework from arbitrary, and inherently exponential, theorem proving.

Under the hypothesis of data complexity, many combinations of database query languages and decidable theories have PTIME data complexity [3, 65, 92, 121]. For example, the relational calculus extended with POLY is in NC whereas the relational calculus extended with DENSE or EQ is in LOGSPACE. Note however that this does not necessarily means that the algorithms used are efficient. Indeed, data complexity often hides parameters in which algorithms are exponential (this is the case of parameter k for projection) in a large constant [113].

2.3.4 Additional topics

In order to be practically usable, constraint query languages should support all functionalities of typical relational languages, such as SQL.

In this respect, an interesting issue is related to the integration of aggregate functions inside constraint query languages. Aggregate operators in the relational context allow the expression of statistical operations such as AVG, MIN, COUNT [88]. When dealing with constraint databases, some aggregate operators, such as COUNT, are not applicable, since relations are infinite. However, other operators, such as *length, area* and *volume*, have to be considered [93].

The introduction of such aggregate functions in constraint query languages may result in new languages that are not closed. This is true for any of the interesting class of constraints, i.e., dense-order, linear, or polynomial constraints [93]. In [45], some aggregate operators under which constraint query languages are closed are presented. In [44] a restriction on the schema of constraint databases is proposed to guarantee closure of languages dealing with aggregate functions.

Another operation which is nowadays essential for expressing practical queries is recursion. It is well known that recursion cannot be expressed by using first order logic. Constraint query languages dealing with recursion can be obtained by introducing constraints in logic-based query languages, such as Datalog [83]. As for aggregate functions, the main problem is to define query languages having a tractable, i.e., polynomial data complexity and guaranteeing closure. We refer the reader to [60, 83] for some work on this topic.

Finally, as with any other database system, constraint database systems should include update languages. This is an essential aspect in order to define a complete data manipulation language for constraint databases. The only work on this topic we are aware of has been developed by Revesz [123]. Starting from the consideration that users should specify which kind on information has to be inserted in the database, without specifying how this can be achieved, Revesz introduces the concept of model-theoretical minimal change [85] for relational constraint databases, based on POLY.

2.4 Modeling complex objects in constraint databases

The generalized relational model extends the relational model to deal with possibly infinite, but finitely representable, relations. Thus, it inherits all its modeling limitations. For example, it cannot model complex entities nor support modular schema definitions. The need to represent composite data has lead to the definition of *object-oriented* data models [22] and *nested relational* data models [1]. Such models extend the traditional relational model to model not only flat data, represented by set of tuples, but also complex data, obtained by arbitrary combinations of set and tuple constructors.

An interesting topic is how the object-oriented data model and the nested relational data model can be extended to deal with constraints. In the following, we survey the approaches proposed in the literature for both kinds of extensions. Table 2.3 classifies these approaches according to four criteria: the underlying data model; the chosen theory and the underlying query language; the maximal allowed set depth; the data complexity.

2.4.1 Introducing constraints in the nested relational model

The simplest idea for introducing constraints in the nested relational model is to use generalized tuples to finitely represent infinite sets of tuples. As for the generalized relational model, it is necessary to provide a framework for extending the nested relational model with an arbitrary decidable logical theory. This allows the model to be used in different types of applications.

The first approach towards the definition of such a model has been proposed by Grumbach and Su [65]. The proposed language, called C-CALC, is obtained

Language	**Theory** / **Underlying query language**	**Max. nesting**	**Complexity**
$\mathcal{L}yri\mathcal{C}$	LPOLY / XSQL [87]	$n \geq 0$	$\subseteq$ PTIME
$Datalog^{\subseteq \mathbf{P}(\mathbf{Z})}$	set constraints on integer numbers / Datalog	2	$\subseteq$ EXPTIME
C-CALC	DENSE / relational calculus for complex objects [75]	$n \geq 0$	$\subseteq$ H-EXPTIME
$FO(Region, Region')$	- / FO	2	it depends on *Region* and *Region'*

Table 2.3: Language comparison.

by extending the nested relational calculus [75] to deal with infinite sets, finitely representable with DENSE.

The main limitation of C-CALC is that its semantics is well defined only for DENSE. Thus, the language cannot be used for real-life spatial applications, where at least linear polynomial constraints are needed. Moreover, the data complexity of C-CALC is hyper-exponential (denoted by H-EXPTIME in Table 2.3) in the size of the database. The high complexity is mainly due to the fact that variables may range over sets.

Another proposal to introduce complex objects into the relational model is due to Revesz, which proposed the $Datalog^{\subseteq \mathbf{P}(\mathbf{Z})}$ language. This language allows the representation of possibly infinite sets of integers by extending Datalog to deal with set constraints [35, 122]. The data complexity of this language is exponential. Moreover, it does not allow arbitrarily complex objects to be represented but only a specific type of sets.

Finally, we also recall the languages proposed in [112], where first-order logic (*FO*) is extended to deal with quantifiers ranging over specific regions (i.e., sets of points) and not over points, as usual.

2.4.2 Introducing constraints in the object-oriented model

Brodsky and Kornatsky [28] introduce constraints as first-class objects in an object-oriented framework. To guarantee a low data complexity, LPOLY is considered. However, the framework can be extended to deal with any other logical theory.

In this approach, constraint formulas (usually represented by existentially quantified disjunctions of conjunctions of constraints) are interpreted as objects, in the usual object-oriented terminology [22]. Object identifiers are represented by object canonical forms. Thus, equivalent constraints with different canonical forms are considered different objects. Constraint objects are organized in classes. The class $CST(k)$ identifies all constraints with k variables. Methods are in this case represented by the usual operations on constraints, such as union and intersection. An inheritance hierarchy exists between constraint classes. In particular, $CST(k+1)$ is considered a subclass of $CST(k)$.

To query the object-oriented database, a language, called $\mathcal{L}yri\mathcal{C}$, has been proposed, extending a typical object-oriented query language [87] to the new constraint framework. This language is equivalent to the usual generalized relational calculus, extended with linear polynomial constraints.

As a final consideration it is important to note the main difference between the introduction of constraints in the relational or the nested relational model, and in the object-oriented model. In the latter case, the model is extended to deal with a new type of object. In the former case, a specific type of constraint formulas, the generalized relations, represents the model itself [25].

2.5 Applications of constraint databases

From an application point of view, at least two main characteristics make constraint databases attractive:

- Constraints are characterized by a high modeling power and can serve as a *uniform data type* for conceptual representation of heterogeneous data, including spatial and temporal data and complex design requirements.
- Constraint query languages use the same formalism to represent typical data manipulation operators, instead of using a separate operator for each type of transformation, as it is typically done in spatial and temporal databases.

For these characteristics, constraint databases are suitable to model spatio-temporal applications, including multidimensional design [28], resource allocation, data fusion and sensor control, shape manipulation in multimedia databases [18, 16]. In the following we survey some of the specific topics arising when using constraints to model spatial and temporal applications.

2.5.1 Spatial applications

Spatial applications require both relational query features, arithmetic computation and extensibility to define new spatial data. Both the relational and the object-oriented model fail to model spatial data. Indeed, extensions of database systems with spatial operators typically are: (i) limited to low (typically 2- or 3-) dimensional spaces; (ii) have query languages restricted to predefined spatio-temporal operators; (iii) lack global filtering and optimization. Moreover, spatial and non-spatial data are often not homogeneously integrated.

The previous drawbacks can be overcome by constraint databases because:

- They support a homogeneous description of spatial data together with simple relational data.
- The geometry of point sets is implicit in the concept of constraint.
- Constraint theories often support a direct implementation of spatial operators, simplifying query optimization.

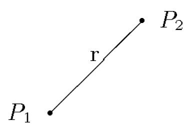
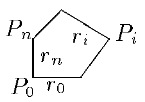
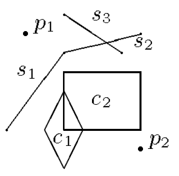
Kanellakis claims that by generalizing the relational formalism to the constraint formalism, it is in principle possible to generalize all the key features of the relational data model to spatial data models [80]. A first step in this direction is represented by the constraint relational algebra proposed for spatial databases by Paredaens et al. [114, 115].

According to the definition of spatial data given in [70], LPOLY has the sufficient expressive power to describe the geometric component of spatial data in geographical applications [94]. In order to represent geometry of geographical objects by using constraints, the approach is to use a generalized relation with n variables representing points of a n-dimensional space.

Note that concave spatial objects cannot be represented by a single generalized tuple. Rather, a set of generalized tuples is needed containing one generalized tuple respectively for each convex object belonging to the convex decomposition of the composite spatial object. Moreover, an identifier should be assigned to all generalized tuples representing the same object.

The types of point-sets of $\mathcal{E}^2$ which can be described by using generalized tuples on LPOLY are shown in Table 2.4. The first three types, POINT, SEGMENT and CONVEX, correspond to conjunctions of constraints of the LPOLY. The fourth type, COMPOSITE Spatial Object, corresponds to a set of generalized tuples. In the table, we restrict our attention to the Euclidean Plane ($\mathcal{E}^2$) and we assume that the generalized relation schema is $\{ID, X, Y\}$. Variable ID represents the object identifier whereas variables X and Y represent the object points.

Generalized relational languages introduced in Section 2.3 can be used to directly model typical spatial manipulations, as discussed in the following example.

Graphical representation	Analytical representation	Representation in LPOLY[a]
POINT (p) $\bullet p$	$p = (x, y)$	$C_p(p) \equiv (ID = c_{id})^b \wedge$ $(X - x = 0) \wedge$ $(Y - y = 0)$
SEGMENT (s) 	$s = \begin{cases} P_1 = (x_1, y_1) \\ P_2 = (x_2, y_1) \\ r : ax + by + c = 0 \end{cases}$	$C_s(s) \equiv$ $(ID = c_{id}) \wedge$ $(x_1 - X \leq 0) \wedge (X - x_2 \leq 0) \wedge$ $(aX + bY + c = 0)^c$
CONVEX (c) 	$c = \begin{cases} P_i = (x_i, y_i) \\ r_i : a_i x + b_i x + c_i = 0 \\ i = 0 \rightarrow n \end{cases}$	$C_{cx}(c) \equiv$ $(ID = c_{id}) \wedge$ $(sg(P_1, P_2)(a_1 X + b_1 Y + c_1) \geq 0)$ $\wedge \ldots\ldots \wedge$ $(sg(P_{n-1}, P_n)(a_n X + b_n Y + c_n) \geq 0)^d$
COMPOSITE (csp) 	$csp = (p_1 \cup \ldots \cup p_n) \cup$ $(s_1 \cup \ldots \cup s_m) \cup$ $(c_1 \cup \ldots \cup c_l)$	$C_{ct}(csp) \equiv$ $(ID = c_{id}) \wedge$ $\{C_p(p_1) \wedge ID = c_{id}, \ldots,$ $C_p(p_n) \wedge ID = c_{id}\} \cup$ $\{C_s(s_1) \wedge ID = c_{id}, \ldots,$ $C_s(s_m) \wedge ID = c_{id}\} \cup$ $\{C_{cx}(c_1) \wedge ID = c_{id}, \ldots,$ $C_{cx}(s_l) \wedge ID = c_{id}\}$

[a]In all the presented tables, symbol $\equiv$ is used to denote syntactic equivalence.

[b]c_{id} is a numeric constant.

[c]One or both of the first two disjuncts of this formula can be removed if a semi straight line or a complete straight line has to be represented.

[d]The introduction of the function $sg()$ is necessary in order to take into account that the polygonal region represented by a simple polygon is always on the left side of the polygon itself. Thus, function $sg(P_1, P_2)$ returns 1 or -1 according to the direction of the line defined by P_1 and P_2.

Table 2.4: Representation of point-sets of the Euclidean Plane in LPOLY.

Query	GRA expression	Conditions
RANGE ERASE QUERY: calculate all spatial objects obtained as difference of each object in R with a rectangle $rt \in \mathcal{E}^2$	$R \setminus \sigma_P(R \cup \neg R)$[a]	$P \equiv C_{cx}(rt)$[b] $\alpha(P) = \{X, Y\}$
RANGE QUERY ON PROJECTION: retrieve all spatial objects in R whose projection on the X axis intersects the interval $[x_0, x_1]$	$\Pi_{[ID]}(\sigma_P\ (\Pi_{[ID,X]}(R))) \bowtie R$	$P \equiv (x_0 \leq X) \wedge (X \leq x_1)$
SPATIAL INTERSECTION: generate all spatial objects that are intersection between one object in R and one object in S	$R \bowtie \varrho_{[ID_{\vert ID'}]}(S)$	

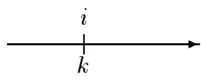
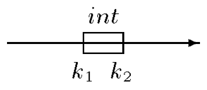
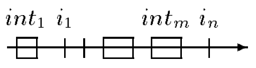
[a]The difference operator ($\setminus$) is defined as derived operator in Table 2.2.
[b]$C_{cx}()$ is defined in Table 2.4.

Table 2.5: Examples of spatial queries in GRA(LPOLY).

Example 2.5 (GRA(Φ) for spatial applications) *Table 2.5 shows some spatial queries referring to a geographical application (the reader can refer to [50, 57, 70, 128, 139] for some examples of spatial query languages and models). For each query, the table contains a textual description and the mapping to GRA. Queries refer to two sets of spatial objects, which are represented by two generalized relations R and S on* LPOLY, *where $\alpha(R) = \alpha(S) = \{ID, X, Y\}$. ID is the generalized tuple identifier whereas X and Y represent points of the spatial objects.* $\diamond$

2.5.2 Temporal applications

The management of temporal information is an important topic in current database research. Due to their ability to finitely model infinite sets of points, generalized databases have been successfully used to represent infinite temporal information [10, 79, 109], arising when describing situations that are repeated in time. *Linear repeating points* are a typical type of constraints used for these purposes. A linear repeating point has the form $c + kn$, where c and k are integer numbers and variable n takes values from the set of all integer numbers. Thus, each linear repeating point represents an infinite sequence of time points. A generalized tuple in most temporal models consists of a set of linear repeating points and a set of non-temporal data values.

Analytical and graphical representation	**Representation in** DENSE
INSTANT (i) 	$C_{ins}(i) \equiv (ID = c_{id})^{a} \wedge (X = k)$
INTERVAL (int) 	$C_{int}(int) \equiv (ID = c_{id}) \wedge$ $(X \geq k_1) \wedge (X \leq k_2)$
NON-CONTIGUOUS INTERVAL (int_D) $(i_1 \cup ... \cup i_n) \cup (int_1 \cup ... \cup int_m)$ 	$C_{nci}(int_D) \equiv \{ID = c_{id} \wedge C_{ins}(i_1), ...,$ $ID = c_{id} \wedge C_{ins}(i_n)\} \cup$ $\{ID = c_{id} \wedge C_{int}(int_1), ...,$ $ID = c_{id} \wedge C_{int}(int_m)\}$

[a] c_{id} is a numeric constant.

Table 2.6: Representation of subsets of the axis of time in DENSE.

Additional constraint on temporal attributes can be added to each tuple.

The representation of time intervals is another interesting application for constraint databases. An interval consists of a time duration which is bound by two endpoints. These endpoints are instants on the time axis. An interval degenerates to an instant when its endpoints coincide. Moreover, an interval is non-contiguous if it does not contain all instants of the axis of time which lie between its endpoints. The dense-order constraint theory is sufficient to represent the types INSTANT, INTERVAL and NON-CONTIGUOUS INTERVAL, as shown in Table 2.6.

Note that non-contiguous intervals, similarly to composite spatial objects, can only be represented by using sets of generalized tuples.

The generalized relational languages introduced in Section 2.3 can be used to directly model typical temporal manipulations, as discussed in the following example.

Example 2.6 (GRA(Φ) for temporal applications) *Table 2.7 shows some queries involving temporal data. The queries concern the trains arriving at a transit station S and leaving from the same station S. The entire set of information is represented by a generalized relation A on* DENSE *with four variables (ID, F, I, T).*

Query	GRA expression	Conditions
INSTANT QUERY: select all trains standing by at station S at time t	$\Pi_{[ID]}(\sigma_P\ (A))$	$P \equiv C_{ins}(t)^a$ $\alpha(P) = \{I\}$
INTERVAL QUERY: retrieve all trains that leave to station 3 after time t, together with their departure station	$\Pi_{[ID,F]}(\sigma_P(\sigma_Q\ (A)))$	$P \equiv (T = 3)$ $Q \equiv Q_{Post}(C_{ins}(t))^b$ $\alpha(Q) = \{I\}$
TEMPORAL JOIN: retrieve all trains with destination station 3, standing by at station S together with a train from station 4	$\Pi_{[ID,ID']}(\sigma_P\ (A) \bowtie A')$ $A' = \varrho_{[ID_{\mid ID'}, F_{\mid F'}, T_{\mid T'}]}(\sigma_Q\ (A))$	$P \equiv (T = 3)$ $Q \equiv (F = 4)$

[a] $C_{ins}()$ is defined in Table 2.6.
[b] $Q_{Post}(t)$ is a short form for the set of instants that follow the intervals represented by t.

Table 2.7: Examples of temporal queries GRA(DENSE).

Variable I represents the interval during which the train stops at station S, variable F represents the numeric code of the departure station of the train, and variable T represents the numeric code of the destination station of the train. Finally, variable ID uniquely identifies each group of information (thus, it is a generalized tuple identifier). The time is expressed in minutes from the beginning of the day. $\diamond$

2.6 Concluding remarks

Constraint databases are a young research area. In this chapter we have investigated the main issues arising in the extension of existing data models with constraints and we have surveyed several approaches.

Three main constraint database system prototypes have already been developed. CCUBE [30] is a constraint object-oriented database system. It has been implemented on top of a commercial object-oriented database system and supports standard database features. The query language provided is $\mathcal{L}yri\mathcal{C}$. The database supports extensible constraint families, aggregation, optimization and indexing.

Another implementation effort is the DISCO system [35]. DISCO (Datalog with Integer and Set Order COnstraints) has a high expressive power, due to the class of constraints considered and the underlying query language (see Section 2.4). However, its data complexity is exponential in the size of the database and it does not support important database features such as persistent storage.

Finally, DEDALE [62, 63] is a prototype of recent definition, introducing linear constraints inside the generalized relational calculus. Like CCUBE, DEDALE has also been implemented on the top of an object-oriented database system. The main issue in developing such a system has been the comparison of two different database technology: constraint databases and spatial databases. For this reason, several optimization issues have also been considered in the development of such a system.

As a final remark, it is useful to recall that an important direction of research is the analysis of the expressive power of constraint query languages. In this respect, the main questions are whether and how some typical spatial and temporal queries can be expressed in a constraint query language based on a given theory Φ. We refer the reader to [3, 15, 64, 66, 92, 142] for some results on this topic.

Besides the work on constraint databases reported in this chapter, constraints have also been considered in the database literature to optimize deductive queries, i.e., queries expressed using logical rules [140]. In particular, the problem of manipulating and repositioning constraints inside logical rules has been extensively investigated [86, 95, 100, 106, 132, 135].

Chapter 3

New languages for relational constraint databases

The languages presented in Section 2.3 handle a generalized relation as a (possibly infinite) set of relational tuples. This approach forces the user to think in term of single points when specifying queries; as a consequence, the only way to manipulate generalized tuples as single objects is to assign an identifier to each generalized tuple.

We believe that the relational semantics is not the only way to assign a meaning to generalized relations. In particular, a generalized relation can also be interpreted as a *nested* relation [1, 2], containing a finite number of possibly infinite sets, each corresponding to the extension of a generalized tuple.

By assigning a nested semantics to generalized relations, the user has to think in term of sets. Therefore, new languages should be introduced in order to be able to manipulate generalized relations under the new semantics. In particular, such languages should manipulate generalized relations in two different ways:

- *As a possibly infinite set of points in a d-dimensional space*: a typical example is the detection of the intersection of the extension of a set of generalized tuples with a specific region of space. This type of manipulation is called *point-based.*
- *As a finite set of objects, each represented by a possibly infinite set of points*: a typical example is the detection of all generalized tuples whose extension is contained in the extension of a given generalized tuple. In this case, the same computation is applied to *all* the points belonging to the extension of the generalized tuple. This type of manipulation is called *object-based.*

The aim of this chapter is the introduction of the nested semantics and of some languages based on it. As a second contribution we investigate the issue of introducing external functions in the proposed languages. This is a very important topic in constraint databases. Indeed, the chosen logical theory is often not adequate to support all the functionalities needed by the considered application. Such additional functionalities can be made available in the constraint language in the form of external functions.

The chapter is organized as follows. We first extend the definition and the semantics of generalized tuples, by introducing the *nested semantics* (Section 3.1). Then, we propose a classification of generalized relational languages for constraint databases (Section 3.2) with respect to the semantics on which they are based (either relational or nested). An algebra and a calculus based on the nested semantics are introduced in Section 3.3 and Section 3.4, respectively. The introduction of external functions in the proposed languages is investigated in Section 3.5 whereas in Section 3.6 we formally prove their equivalence. Finally, Section 3.7 presents some concluding remarks.

3.1 The extended generalized relational model

As we have seen, each generalized tuple represents a (possibly infinite) set of relational tuples. In Section 2.2, generalized tuples have been defined as conjunctions of constraints. Thus, an arbitrary set, which can be represented in FO extended with the theory Φ without quantifiers, cannot always be represented as the extension of a generalized tuple. In particular, only convex sets of points can be represented by a single generalized tuple. Non-convex sets of points can only be represented by using several generalized tuples (see Section 2.5).

In the following, we extend the definition of generalized tuple, in order to express more general sets in their extension. This is possible by using additional logical connectives. The basic requirement is that generalized tuples must be quantifier-free, to guarantee an efficient computation. As we will see in Section 3.2, the use of more expressive generalized tuples increases the expressive power of some classes of constraint languages.

Definition 3.1 (Extended generalized relational model) *Let Φ be a decidable logical theory and Σ a set of* FO *logical connectives without quantifiers (Σ is called a signature). A generalized tuple on Φ and Σ over variables $X_1, ..., X_k$ is a* FO *formula whose free variables belong to $X_1, ..., X_k$, atoms are atomic formulas on Φ, and logical connectives belong to Σ. A generalized relation on Φ and Σ is a set of generalized tuples over Φ and Σ; a generalized database on Φ and Σ is a set of generalized relations on Φ and Σ.* □

Notice that, under the new definition, generalized tuples introduced in Definition 2.1 are generalized tuples on Φ and $\{\wedge\}$.

Since we have defined Σ to be a set of FO logical connectives without quantifiers, the only possible signatures are: $\{\wedge\}, \{\vee\}, \{\wedge, \vee\}, \{\neg, \wedge\}, \{\neg, \vee\}, \{\neg, \wedge, \vee\}$. By considering only theories Φ closed under complementation, the sets of all generalized tuples on Φ and one of the signatures $\{\wedge, \vee\}, \{\neg, \wedge\}$, $\{\neg, \vee\}, \{\neg, \wedge, \vee\}$ coincide. Therefore, in the following, to simplify the notation, we only consider the signatures $\{\wedge\}, \{\vee\}$, and $\{\wedge, \vee\}$. Generalized tuples on Φ and $\{\wedge, \vee\}$ allow us to represent all the sets that can be characterized in FO without quantifiers and are called *disjunctive generalized tuples* or *d-generalized tuples.*

For what we will discuss in the following, it is useful to denote in some way the set of generalized relations on Φ and Σ, leading to the definition of *extended generalized relational support.*

Definition 3.2 (EGR support) *Let Φ be a decidable logical theory and Σ a signature. The set of all generalized relations over Φ and Σ (denoted by $S(\Phi, \Sigma)$) is called extended generalized relational support (EGR support for short) over Φ and Σ.* □

Note that the generalized relations introduced in Definition 2.1 belong to $S(\Phi, \{\wedge\})$.

Example 3.1 *Tables 3.1 and 3.2 show how composite spatial objects and non-contiguous intervals can be represented by using disjunctive generalized tuples. In such a representation, each disjunct represents, respectively, a convex polygon belonging to the convex decomposition of the original object, or either an instant or an interval belonging to the representation of the non-contiguous interval. No generalized tuple identifier is needed in this case.* ◇

3.1.1 Nested semantics for EGR supports

The relational semantics is not the only way to assign a meaning to generalized relations. In particular, generalized relations can be interpreted as *nested-relations* [1, 2]. A nested-relation is a relation in which attributes may contain sets as values. A generalized relation can be interpreted as a nested-relation containing a finite number of possibly infinite sets, each representing the extension of a generalized tuple. This interpretation leads to the definition of the following semantics.

Definition 3.3 (Nested semantics) *Let $r = \{t_1, ..., t_m\}$ be a generalized relation. The nested semantics of r, denoted by $nested(r)$, is the set $\{ext(t_1), ...,$*

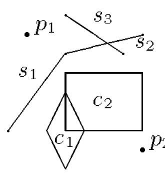
Graphical representation	**Analytical representation**	**Representation using d-gen. tuples in LPOLY**
COMPOSITE (csp) 	$csp = (p_1 \cup ... \cup p_n) \cup (s_1 \cup ... \cup s_m) \cup (c_1 \cup ... \cup c_l)$	$C_{ct}(csp) \equiv (C_p(p_1) \vee ... \vee C_p(p_n)) \vee (C_s(s_1) \vee ... \vee C_s(s_m)) \vee (C_{cx}(c_1) \vee ... \vee C_{cx}(s_l))$

Table 3.1: Representation of concave sets of points in the Euclidean Plane in LPOLY.

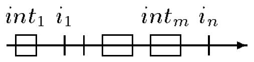
Analytical and graphical representation	**Representation using d-gen. tuples in DENSE**
NON-CONTIGUOUS INTERVAL (int_D) $(i_1 \cup ... \cup i_n) \cup (int_1 \cup ... \cup int_m)$ 	$C_{nci}(int_D) \equiv (C_{ins}(i_1) \vee ... \vee C_{ins}(i_n)) \vee (C_{int}(int_1) \vee ... \vee C_{int}(int_m))$

Table 3.2: Representation of non-contiguous subsets of the axis of time. in DENSE

$ext(t_m)\}$. Two generalized relations r_1 and r_2 are n-equivalent (denoted by $r_1 \equiv_n r_2$) iff $nested(r_1) = nested(r_2)$. □

Note that distinct generalized tuples with the same extension represent the same object. From Definition 3.3 it follows that, if two generalized relations are n-equivalent, they are also r-equivalent [13]. However, the converse does not hold, as shown by the following example.

Example 3.2 *Consider a generalized relation r_1 containing only the generalized tuple $1 \leq X \leq 2 \wedge 2 \leq Y \leq 4$ and the generalized relation r_2 containing the generalized tuples $1 \leq X \leq 2 \wedge 2 \leq Y \leq 3$ and $1 \leq X \leq 2 \wedge 3 \leq Y \leq 4$. It is simple to show that $r_1 \equiv_r r_2$. However, $r_1 \not\equiv_n r_2$, since the sets represented inside r_1, r_2 are different.* ◇

3.1.2 Equivalence between EGR supports

The aim of this subsection is to compare the expressive power of different EGR supports with particular attention to $S(\Phi, \{\wedge, \vee\})$ and $S(\Phi, \{\wedge\})$. For this purpose, we first introduce the concept of containment and equivalence for supports. We propose a general definition of these concepts, considering supports with arbitrary theories and signatures.

Definition 3.4 (Equivalence of EGR supports) *Let $S(\Phi, \Sigma_1)$ and $S(\Phi, \Sigma_2)$ be two EGR supports. Let $t \in \{r, n\}$. $S(\Phi, \Sigma_1)$ t-contains $S(\Phi, \Sigma_2)$ (denoted by $S(\Phi, \Sigma_1) \subseteq_t S(\Phi, \Sigma_2)$) iff for each generalized relation $r \in S(\Phi, \Sigma_2)$ there exists a generalized relation $r' \in S(\Phi, \Sigma_1)$ such that $r \equiv_t r'$. $S(\Phi, \Sigma_1)$ and $S(\Phi, \Sigma_2)$ are t-equivalent (denoted by $S(\Phi, \Sigma_1) \equiv_t S(\Phi, \Sigma_2)$) iff $S(\Phi, \Sigma_1)$ t-contains $S(\Phi, \Sigma_2)$ and $S(\Phi, \Sigma_2)$ t-contains $S(\Phi, \Sigma_1)$.* □

From the properties of first-order logical connectives [38], the following result holds.

Proposition 3.1 *Let $S(\Phi, \Sigma_1)$ and $S(\Phi, \Sigma_2)$ be two EGR supports. $S(\Phi, \Sigma_1) \equiv_r S(\Phi, \Sigma_2)$ iff the signature $\Sigma_1 \cup \{\vee\}$ is equivalent*[1] *to $\Sigma_2 \cup \{\vee\}$. $S(\Phi, \Sigma_1) \equiv_n S(\Phi, \Sigma_2)$ iff the signature Σ_1 is equivalent to Σ_2.* □

From the previous proposition, it follows that $S(\Phi, \{\wedge, \vee\})$ is r-equivalent to $S(\Phi, \{\wedge\})$, but $S(\Phi, \{\wedge, \vee\})$ is not n-equivalent to $S(\Phi, \{\wedge\})$.

3.2 Extended generalized relational languages

The classes of algebraic and calculus-based languages (one for each decidable logical theory admitting variable elimination and closed under complementation) presented in Subsection 2.3 are based on the relational semantics for generalized databases (see Table 2.1 and Proposition 2.1).

In general, when adopting the nested semantics for generalized relations, other operators can be defined, considering the extension of each generalized tuple as a single object. The following example better clarifies which operations can be useful.

Example 3.3 *Consider a relation R, representing spatial objects contained in the Euclidean plane and having schema N, X, Y, where N is a generalized tuple identifier and X and Y represent the object points. Consider the query* "Find all objects in R that are contained in the object o"*. Let P be the generalized tuple representing "o"*

[1]Two sets of first-order logic operators A and B are equivalent iff for each formula that can be expressed by using operators in A there exists an equivalent formula, expressed by using operators in B, and vice versa.

in the Euclidean space. Let $\alpha(P) = \{X, Y\}$. This query is expressed in GRA(Φ) as follows:

$$(\Pi_{[ID]}(R) \setminus (\Pi_{[ID]}(R \setminus \sigma_P(ID)))) \bowtie R.$$

The previous expression has the following meaning:

- *$\sigma_P(R)$ selects the points (X, Y) of R contained in P, together with the identifier of the object to which they belong.*
- *$\Pi_{[ID]}(R \setminus \sigma_P(R))$ selects the identifiers of the objects having at least one point not contained in P. Thus, all the retrieved identifiers correspond to objects not contained in P.*
- *$\Pi_{[ID]}(R) \setminus (\Pi_{[ID]}(R \setminus \sigma_P(R)))$ selects the identifiers of the objects contained in P.*
- *$(\Pi_{[ID]}(R) \setminus (\Pi_{[ID]}(R \setminus \sigma_P(R)))) \bowtie R$ selects the objects contained in P.*

The previous expression is not very simple to write and to understand, even if the query is one of the most common in spatial applications. The problem is that the query deals with the extension of generalized tuples taken as a single object, whereas, in general, GRA operators deal with single relational tuples, belonging to the extension of generalized tuples. $\diamond$

In a general setting, we believe that at least two classes of languages to manipulate generalized relations can be designed:

- *R-based languages.* R-based languages are such that the relational semantics of the result of any query they can express is equivalent to the result of an equivalent *relational language* query, when applied to a set of relations representing the relational semantics of the input generalized relations (as the algebras presented in Subsection 2.3.2).
- *N-based languages.* N-based algebras are such that the nested semantics of the result of any query they can express is equivalent to the result of an equivalent *nested-relational language* query [1, 2], when applied to a set of nested relations representing the nested semantics of the input generalized relations.

All relational algebra expressions can obviously be expressed in the nested relational algebra. The same holds for the calculus. It has been proved that also the opposite result holds [116], when input and output relations are not nested objects. When input/output relations are nested objects, the equivalence is guaranteed by the use of object identifiers to code nested objects into flat ones [141].

$$\begin{tikzcd}
r_1, ..., r_n \arrow[r, "q"] \arrow[d, "rel/nested"'] & r_{n+1} \arrow[d, "rel/nested"'] \\
r'_1, ..., r'_n \arrow[r, "q'"'] & r'_{n+1}
\end{tikzcd}$$

Figure 3.1: R-based and n-based languages.

In the remainder of this paper, we use the following notation. Let L be a constraint relational language.

- $L(\Phi)$ is the set of all the queries that can be expressed in L (also called semantic language). Thus, for each expression $e \in L$, there exists a function $\mu(e) \in L(\Phi)$ representing the semantics of e.

  Note that each query in $L(\Phi)$ is a function with polymorphic type, since it can be applied to arbitrary supports. Moreover, there exists a one-to-one correspondence between expressions contained in L and queries contained in $L(\Phi)$. For this reason, in the following, when it is clear from the context, we use indifferently L and $L(\Phi)$ to denote both the syntactic and the semantic language. Similarly, an expression e is also used to denote the semantic function $\mu(e)$.

- $L(\Phi, \Sigma)$ is the set of all the queries contained in $L(\Phi)$ and having $S(\Phi, \Sigma)$ as support.

  Note that, by using this notation, GRA(Φ), introduced in Subsection 2.3.2, corresponds to GRA($\Phi, \{\wedge\}$). Thus, from now on, we use this notation.

We can finally introduce *n-based* and *r-based* languages.

Definition 3.5 (R-based and n-based languages) *Let L be a constraint query language. Let Φ be a logical theory admitting variable elimination and closed under complementation. Let Rel be the set of all relational queries. Let N_Rel be the set of all nested-relational queries. Then:*

- *$L(\Phi)$ is r-based iff there exists a query mapping $h : L(\Phi) \rightarrow Rel$ such that $h(q) = q'$ and for all supports $S(\Phi, \Sigma)$, for all generalized relations $r_i \in S(\Phi, \Sigma)$, $i = 1, ..., n$, $rel(q(r_1, ..., r_n)) = q'(rel(r_1), ..., rel(r_n))$ (see Figure 3.1).*

- *$L(\Phi)$ is n-based iff there exists a query mapping $h : L(\Phi) \rightarrow N\_Rel$ such that $h(q) = q'$ and for all supports $S(\Phi, \Sigma)$, for all generalized relations $r_i \in S(\Phi, \Sigma)$, $i = 1, ..., n$, $nested(q(r_1, ..., r_n)) = q'(nested(r_1), ..., nested(r_n))$ (see Figure 3.1).* □

Note that Definition 3.5 implies that algebraic operators are independent of the chosen support, i.e., similar computations can be applied to different supports. Moreover, from Definition 3.5 and Proposition 2.1, it follows that GRA(Φ) and CAL(Φ) are r-based.

Since relational operators are part of any nested-relational algebra, r-based algebras are also n-based. The same holds for the calculus. We call *strict* n-based languages the languages that are n-based but are not r-based.

3.2.1 Relationship between languages and EGR supports

Given two semantic languages, the relationships existing between the supports on which they are based allow the detection of some relationships between the expressive power of such languages. In order to formalize these notions, the concept of equivalence between languages is introduced.

Definition 3.6 (Equivalence between languages) *Let L_1 and L_2 be two constraint query languages. Let Φ be a decidable theory, admitting variable elimination and closed under complementation. Let $S(\Phi, \Sigma_1)$ and $S(\Phi, \Sigma_2)$ be two EGR supports. Let $t \in \{r, n\}$. $L_1(\Phi, \Sigma_1)$ is t-contained in $L_2(\Phi, \Sigma_2)$ (denoted by $L_1(\Phi, \Sigma_1) \subseteq_t L_2(\Phi, \Sigma_2)$) iff for each query $q \in L_1(\Phi, \Sigma_1)$ there exists a query $q' \in L_2(\Phi, \Sigma_2)$ such that for each input generalized relation $r_i \in S(\Phi, \Sigma_1)$, $i = 1, ..., n$, a generalized relation $r'_i \in S(\Phi, \Sigma_2)$ exists such that $r_i \equiv_t r'_i$ and $q(r_1, ..., r_n) \equiv_t q'(r'_1, ..., r'_n)$. $L_1(\Phi, \Sigma_1)$ is t-equivalent to $L_2(\Phi, \Sigma_2)$ (denoted by $L_1(\Phi, \Sigma_1) \equiv_t L_2(\Phi, \Sigma_2)$) iff $L_2(\Phi, \Sigma_2) \subseteq_t L_1(\Phi, \Sigma_1)$ and $L_2(\Phi, \Sigma_2) \supseteq_t L_1(\Phi, \Sigma_1)$.* □

In the following, if the queries associated with two expressions e_1 and e_2 are t-equivalent we write $e_1 \equiv_t e_2$.

Note that in the previous definition of equivalence, equivalent expressions take equivalent input relations. We now analyze the expressive power of a constraint language $L(\Phi)$ with respect to different EGR supports (proofs of the following results are presented in Appendix A).

Proposition 3.2 *Let L_1 and $L_2 = L$ be two constraint query languages. Let Φ be a decidable theory admitting variable elimination and closed under complementation. Let $t \in \{r, n\}$. The following facts hold:*

1. *If $L_i(\Phi)$ is t-based, then for all $S(\Phi,\Sigma_1), S(\Phi,\Sigma_2)$, $S(\Phi,\Sigma_1) \subseteq_t S(\Phi,\Sigma_2)$ iff $L_i(\Phi,\Sigma_1) \subseteq_t L_i(\Phi,\Sigma_2)$.*

2. *If $L_1(\Phi,\Sigma_1) \equiv_t L_2(\Phi,\Sigma_2)$ then $S(\Phi,\Sigma_1) \equiv_t S(\Phi,\Sigma_2)$.* □

Since GRA(Φ) is r-based and $S(\Phi,\{\wedge\}) \equiv_r S(\Phi,\{\wedge,\vee\})$, Proposition 3.2 implies that queries that can be expressed in GRA($\Phi,\{\wedge\}$) can also be expressed by using GRA($\Phi,\{\wedge,\vee\}$).

Another interesting property is stated by the following proposition.

Proposition 3.3 *Let L be a constraint query language. Let Φ be a decidable theory admitting variable elimination and closed under complementation. Let $t \in \{r,n\}$. Let $S(\Phi,\Sigma_1)$ and $S(\Phi,\Sigma_2)$ be two EGR supports. If $L(\Phi)$ is t-based, for all $q \in L(\Phi)$, for all $r_1,...,r_n \in S(\Phi,\Sigma_1)$ and for all $r'_1,...,r'_n \in S(\Phi,\Sigma_2)$, such that $r'_i \equiv_t r_i$, $q(r_1,...,r_n) \equiv_t q(r'_1,...,r'_n)$ holds.* □

Proposition 3.3 specifies that queries expressed in a t-based language are independent of the particular representation given to t-equivalent generalized relations. Note that the previous propositions, as well as Definition 3.5, imply that the semantics of operators is independent of the chosen support.

3.3 EGRA: a n-based generalized relational algebra

In the following, we present a n-based algebra for constraint databases that we call *Extended Generalized Relational Algebra*, since it is obtained by extending the generalized relational algebra with new operators. In particular, it provides two sets of operators, representing two different types of data manipulation:

1. *Set operators.* They apply a certain object-based computation to groups of relational tuples, each represented by the extension of a generalized tuple.

   Consider a generalized relation $R(X,Y)$ where each generalized tuple represents a rectangle. Each tuple has the form: $X \geq a_1 \wedge X \leq a_2 \wedge Y \geq b_1 \wedge Y \leq b_2$. If we want to know which rectangles are contained in a given space, each generalized tuple must be interpreted as a single object and a subset of the input rectangles must be returned as query answer.

2. *Tuple operators.* They apply a certain point-based computation to generalized relations and assign a given nested representation to the result. As an example of an application, consider again a generalized relation $R(X,Y)$ where each generalized tuple represents a rectangle. The detection of the set of points

representing the intersection of each rectangle with a given spatial object is a typical tuple operation.

Note that, under the nested semantics, tuple operators apply computations to relational tuples, nested inside sets and represented by generalized tuples.

We believe that both types of operators are useful when dealing with constraint databases, since they correspond to two complementary types of generalized tuple manipulations.

The new syntactic language is denoted by EGRA. EGRA operators are the following:

- Tuple operators, except complement, are exactly the operators introduced in Table 2.1.

  The EGRA complement operator always returns a generalized relation which is relationally equivalent to the generalized relation returned by the GRA complement operator (when both operators are applied to the same generalized relation). However, such resulting relations are not nested equivalent.

- Set operators are the following:

  1. *Set difference.* Given two generalized relations r_1 and r_2, this operator returns all generalized tuples contained in r_1 for which there does not exist an equivalent generalized tuple contained in r_2. This is the usual difference operation in nested-relational databases [1, 2].
  2. *Set complement.* Given a generalized relation r, this operator returns a generalized relation containing a generalized tuple t' for each generalized tuple t contained in r; t' is the disjunctive normal form of the formula $\neg t$.
  3. *Set selection.* This operator selects from a generalized relation all the generalized tuples satisfying a certain *condition*. The condition is of the form (Q_1, Q_2, θ), where $\theta \in \{\subseteq, (\bowtie \neq \emptyset)\}$ and Q_1 and Q_2 are either:
     - A generalized tuple P on the chosen support.
     - Expressions generated by operators $\{t/0, \Pi_{[X_1,...,X_n]}/1\}$. t represents the input generalized tuple whereas the interpretation of $\Pi_{[X_1,...,X_n]}$ is a function taking a generalized tuple t' and returning the projection of t' on variables $X_1, ..., X_n$.

     In order to point out that Q_1 and Q_2 are applied on a single generalized tuple t, in the following they will be denoted by $Q_1(t)$ and $Q_2(t)$. Moreover, for the sake of simplicity, they will be used to represent both the syntactic expressions and their semantic function.

Op. name	Syntax e	Semantics $r = \mu(e)(r_1, \ldots, r_n), n \in \{1, 2\}$[a] Restrictions
Tuple operators		
atomic relation	R_1	$rel(r) = rel(r_1)$ $r = r_1$
selection	$\sigma_P(R_1)$	$r = \{t \wedge P : t \in r_1, ext(t \wedge P) \neq \emptyset\}$
		$\alpha(P) \subseteq \alpha(R_1)$ $\alpha(e) = \alpha(R_1)$
renaming	$\varrho_{[A\mid B]}(R_1)$	$r = \{t[A \mid B] : t \in r_1\}$
		$A \in \alpha(e), B \notin \alpha(e)$ $\alpha(e) = (\alpha(R_1) \setminus \{A\}) \cup \{B\}$
projection	$\Pi_{[X_{i_1},\ldots,X_{i_p}]}(R_1)$	$r = \{\pi_{[X_{i_1},\ldots,X_{i_p}]}(t) \mid t \in r_1\}$
		$\alpha(R_1) = \{X_1, \ldots, X_m\}$ $\alpha(e) = \{X_{i_1}, \ldots, X_{i_p}\}$ $\alpha(e) \subseteq \alpha(R)$
natural join	$R_1 \bowtie R_2$	$r = \{t_1 \wedge t_2 : t_1 \in r_1, t_2 \in r_2, ext(t_1 \wedge t_2) \neq \emptyset\}$
		$\alpha(e) = \alpha(R_1) \cup \alpha(R_2)$
complement	$\neg R$	$r = \{\bar{t}_1 \vee \ldots \vee \bar{t}_m \mid \bar{t}_1 \vee \ldots \vee \bar{t}_m$ is the disjunctive normal form of $\neg t_1 \wedge \ldots \wedge \neg t_n, r_1 = \{t_1, \ldots, t_n\}$, $ext(\bar{t}_i) \neq \emptyset, i = 1, \ldots, m\}$
		$\alpha(e) = \alpha(R)$
Set operators		
union	$R_1 \cup R_2$	$r = \{t : t \in r_1$ or $t \in r_2\}$
		$\alpha(R_1) = \alpha(R_2) = \alpha(e)$
set difference	$R_1 \setminus^s R_2$	$r = \{t : t \in r_1, \nexists t' \in r_2 : ext(t) = ext(t')\}$
		$\alpha(R_1) = \alpha(R_2) = \alpha(e)$
set complement	$\neg^s R_1$	$r = \{not\ t$[b]$: t \in r_1, ext(not\ t) \neq \emptyset\}$
		$\alpha(e) = \alpha(R_1)$
set selection	$\sigma^s_{(Q_1, Q_2, \subseteq))}(R_1)$	$r = \{t : t \in r_1,$ $ext(Q_1(t)) \subseteq ext(\Pi_{[\alpha(Q_1)]}(Q_2(t)))\}$
		$\alpha(Q_1) \subseteq \alpha(Q_2)$ $\alpha(e) = \alpha(R_1)$
	$\sigma^s_{(Q_1, Q_2, \bowtie \neq \emptyset))}(R_1)$	$r = \{t : t \in r_1, ext(Q_1(t)) \cap ext(Q_2(t)) \neq \emptyset\}$
		$\alpha(Q_1) = \alpha(Q_2)$ $\alpha(e) = \alpha(R_1)$

[a] We assume that r_i does not contain inconsistent generalized tuples, $i = 1, \ldots, n$.

[b] The expression *not t* represents the disjunctive normal form of the formula $\neg t$.

Table 3.3: EGRA operators.

The set selection operator with condition (Q_1, Q_2, θ), applied on a generalized relation r, selects from r only the generalized tuples t for which there exists a relation θ between $ext(Q_1(t))$ and $ext(Q_2(t))$. When a condition C is satisfied by a generalized tuple, we denote this fact by $C(t)$.

The possible meanings of θ operators are the following:

- $\theta = \subseteq$: in this case, we require that $\alpha(Q_1) \subseteq \alpha(Q_2)$. It selects all generalized tuples t in r such that $ext(Q_1(t)) \subseteq ext(\Pi_{[\alpha(Q_1)]}(Q_2(t)))$.
- $\theta = \bowtie \neq \emptyset$: in this case, we require that $\alpha(Q_1) = \alpha(Q_2)$. It selects all generalized tuples t in r such that $ext(Q_1(t)) \cap ext(Q_2(t)) \neq \emptyset$.

Note that, since the considered theory Φ is decidable, set selection conditions are also decidable.

Table 3.3 presents set and tuple operators, according to the notation introduced in Subsection 2.3.2. Note that, in order to guarantee operator closure, EGRA operators can only be applied to generalized relations belonging to the EGR support $S(\Phi, \{\wedge, \vee\})$, where Φ is a logical theory admitting variable elimination and closed under complementation. Thus, from now on, EGRA(Φ) should be interpreted as a shorthand for EGRA($\Phi, \{\wedge, \vee\}$).

It can be easily shown that EGRA(Φ) operators are independent, i.e., the semantic function of no operator can be expressed as the composition of the semantic functions associated with other operators [13].

Example 3.4 *Tables 3.4 and 3.5 show examples of spatial and temporal queries in EGRA(*LPOLY*) and EGRA(*DENSE*) respectively. Generalized relations are interpreted as in Examples 2.5 and 2.6.* $\diamond$

Clearly, all GRA derived operators can also be seen as EGRA derived operators. However, by using set operators, other EGRA derived operators can be defined, whose semantics is described in Table 3.6. For a more detailed description, see [13].

3.3.1 Properties of EGRA($\Phi, \{\wedge, \vee\}$)

In the following we prove that:

1. EGRA(Φ) is a n-based algebra.
2. GRA(Φ, Σ_1) $\not\equiv_r$ EGRA($\Phi, \{\wedge, \vee\}$), and therefore GRA(Φ, Σ_1) $\not\equiv_n$ EGRA($\Phi, \{\wedge, \vee\}$), for all Σ_1.

   However, we introduce a weaker notion of equivalence and we show that GRA(Φ, Σ_1) and EGRA($\Phi, \{\wedge, \vee\}$), under specific conditions for Σ_1, are equivalent under this new definition.

Query	**EGRA expression**	**Conditions**[a]
RANGE INTERSECTION QUERY: select all spatial objects in R that intersect the region of space identified by a given rectangle $rt \in \mathcal{E}^2$	$\sigma^s_{(t,P,(\bowtie \neq \emptyset))}(R)$	$P \equiv C_{cx}(rt)$ $\alpha(P) = \{X, Y\}$
RANGE CONTAINMENT QUERY: select all spatial objects in R that are contained in the region of the space identified by a given rectangle $rt \in \mathcal{E}^2$	$\sigma^s_{(t,P,\subseteq)}\ (R)$	$P \equiv C_{cx}(rt)$ $\alpha(P) = \{X, Y\}$
ADJACENT QUERY: select all spatial objects in R that are adjacent to a spatial object $sp \in \mathcal{E}^2$	$\sigma^s_{c_1}(\sigma^s_{c_2}(R))$	$P \equiv C_{cp}(sp)$ $\alpha(P) = \{X, Y\}$ $c_1 = (Q_{Int}(t), Q_{Int}(P), (\bowtie = \emptyset))$ $c_2 = (Q_{Bnd}(t), Q_{Bnd}(P), (\bowtie \neq \emptyset))$
SPATIAL JOIN (intersection based): generate all pairs of spatial objects (r, s) $r \in R$, $s \in S$, such that r intersects s	$\sigma^s_c(R \bowtie \varrho_{[X\|_{X'}, Y\|_{Y'}]}(S))$	$c = (Q_1(t), Q_2(t), (\bowtie \neq \emptyset))$ $Q_1(t) = \Pi_{[X,Y]}(t)$ $Q_2(t) = \varrho_{[X'\|_X, Y'\|_Y]}(\Pi_{[X',Y']}(t))$
SPATIAL JOIN (adjacency based): generate all pairs of spatial objects (r, s), $r \in R$, $s \in S$, such that r is adjacent to s	$\sigma^s_{c_1}(\sigma^s_{c_2}(R'))$ $R' = R \bowtie \varrho_{[X\|_{X'}, Y\|_{Y'}]}(S)$	$c_1 = (Q_{1,1}(t), Q_{1,2}(t), (\bowtie = \emptyset)))$ $Q_{1,1}(t) = Q_{Int}(\Pi_{[X,Y]}(t))$ $Q_{1,2}(t) = Q_{Int}(g(t))$ $c_2 = (Q_{2,1}(t), Q_{2,2}(t), (\bowtie \neq \emptyset))$ $Q_{2,1}(t) = Q_{Bnd}(\Pi_{[X,Y]}(t))$ $Q_{2,2}(t) = Q_{Bnd}(g(t))$ $g(t) = \varrho_{[X'\|_X, Y'\|_Y]}(\Pi_{[X',Y']}(t))$
DIFFERENCE QUERY: select all spatial objects in R for which there are no spatial objects in S with the same projection on X	$\Pi_{[X',Y']}(\sigma^s_c(R' \bowtie R''))$ $R' = \Pi_{[X]}(R) \setminus^s \Pi_{[X]}(S)$ $R'' = \varrho_{[X\|_{X'}, Y\|_{Y'}]}(R)$	$c = (\Pi_{[X]}(t), \varrho_{[X'\|_X]}(\Pi_{[X']}(t)), =)$
COMPLEMENT QUERY: compute the portions of $\mathcal{E}^2$ that are the complement of a spatial object of R	$\neg^s(R)$	

[a]In this column the following symbols are used:

- $C_{cx}()$ and $C_{ct}()$: see Table 2.4;
- Q_{Bnd} and Q_{Int} represent a short form for queries retrieving the boundary and the interior of a spatial object respectively [142].

Table 3.4: Examples of spatial queries in EGRA(LPOLY).

Query	EGRA expression	Conditions[a]
INSTANT SELECTION: select all trains that leave after time t (expressed in minutes from time $00:00$) to station a	$\sigma^s_{(c_1 \wedge c_2)}\ (A)$	$P \equiv Q_{Post}(C_{ins}(t))$ $P' \equiv (T = a)$ $Q(t) = Q_{StP}(\Pi_{[I]}(t))$ $\alpha(P) = \{I\}$ $c_1 = (Q(t), P, \subset)$ $c_2 = (t, P', (\bowtie \neq \emptyset))$
RANGE SELECTION: select all trains that arrive in the interval i	$\sigma^s_c\ (A)$	$P \equiv C_{int}(i)$ $\alpha(P) = \{I\}$ $c = (Q_{StP}(\Pi_{[I]}(t)), P, (\bowtie \neq \emptyset))$
TEMPORAL JOIN: select all trains that arrive at the station S when another train to destination d is standing by at the same station	$\sigma^s_{c_2}(A \bowtie A')$ $A' = \varrho_C(\sigma^s_{c_1}(A))$ $C \equiv [ID_{\|ID'}, F\ \|_{F'}, I\ \|_{I'}, T\ \|_{T'}]$	$P \equiv (T = d)$ $Q(t) = \varrho_{[I\|_{I'}]}(\Pi_{[I']}(t))$ $c_1 = (t, P, (\bowtie \neq \emptyset))$ $c_2 = (Q_{StP}(\Pi_{[I]}(t)), Q(t), \bowtie \neq \emptyset)$

[a]In this column the following symbols are used:

- $C_{ins}()$ and $C_{int}()$: see Table 2.6;
- $Q_{StP}(t)$: it is a short form for the query retrieving the set of instants that represent the starting points of all contiguous intervals contained in t;
- $Q_{Post}(t)$: it is a short form for the query retrieving the set of instants that follow the interval represented by t.

Table 3.5: Examples of temporal queries in EGRA(DENSE).

3. Under specific assumptions, the data complexity of EGRA($\Phi, \{\wedge, \vee\}$) is equal to the data complexity of GRA($\Phi, \{\wedge, \vee\}$).

3.3.1.1 EGRA(Φ) is a n-based language

In order to show that EGRA(Φ) is n-based, following Definition 3.5, we present a mapping from EGRA expressions to nested-relational algebra expressions, satisfying Definition 3.5.

Let D be a domain of values. The nested-relational model deals with objects of type:

$$\tau ::= D \mid \langle A_1 : \tau, ..., A_n : \tau \rangle \mid \{\tau\}$$

where $A_1, ..., A_n$ are attribute names. In the literature, several nested-relational algebras have been proposed, most of which are equivalent (see [1] and [2] for some

Op. name	Syntax e	Derived expression	Restrictions
set intersection	$R_1 \cap^s R_2$	$R_1 \setminus^s (R_1 \setminus^s R_2)$	$\alpha(e) = \alpha(R_1) = \alpha(R_2)$
derived set selection	$\sigma^s_{(Q_1,Q_2,\supseteq)}(R)$	$\sigma^s_{(Q_2,Q_1,\subseteq))}(R)$	$\alpha(Q_1) \supseteq \alpha(Q_2)$ $\alpha(e) = \alpha(R)$
	$\sigma^s_{(Q_1,Q_2,\not\subseteq)}(R)$	$R \setminus^s \sigma^s_{(Q_1,Q_2,\subset)}(R)$	$\alpha(Q_1) \subset \alpha(Q_2)$ $\alpha(e) = \alpha(R)$
	$\sigma^s_{(Q_1,Q_2,\not\supseteq)}(R)$	$R \setminus^s \sigma^s_{(Q_2,Q_1,\subset)}(R)$	$\alpha(Q_1) \supseteq \alpha(Q_2)$ $\alpha(e) = \alpha(R)$
	$\sigma^s_{(Q_1,Q_2,\bowtie=\emptyset)}(R)$	$R \setminus^s \sigma^s_{(Q_1,Q_2,\bowtie\neq\emptyset)}(R)$	$\alpha(Q_1) = \alpha(Q_2)$ $\alpha(e) = \alpha(R)$
	$\sigma^s_{C_1 \wedge C_2}(R)$	$\sigma^s_{C_1}(R) \cap \sigma^s_{C_2}(R)$	$\alpha(e) = \alpha(R)$
	$\sigma^s_{C_1 \vee C_2}(R)$	$\sigma^s_{C_1}(R) \cup \sigma^s_{C_2}(R)$	$\alpha(e) = \alpha(R)$
	$\sigma^s_{\neg C_1}(R)$	$R \setminus^s \sigma^s_{C_1}(R)$	$\alpha(e) = \alpha(R)$
	$\sigma^s_{(Q_1,Q_2,=)}(R)$	$\sigma^s_{(Q_1,Q_2,\subseteq)\wedge(Q_2,Q_1,\subseteq)}(R)$	$\alpha(Q_1) = \alpha(Q_2)$ $\alpha(e) = \alpha(R)$
	$\sigma^s_{(Q_1,Q_2,\subset)}(R)$	$\sigma^s_{(Q_1,Q_2,\subseteq)\wedge\neg(Q_1,Q_2,=)}(R)$	$\alpha(Q_1) \subseteq \alpha(Q_2)$ $\alpha(e) = \alpha(R)$
	$\sigma^s_{(Q_1,Q_2,\supset)}(R)$	$\sigma^s_{(Q_2,Q_1,\subseteq)\wedge\neg(Q_1,Q_2,=)}(R)$	$\alpha(Q_1) \subseteq \alpha(Q_2)$ $\alpha(e) = \alpha(R)$
	$\sigma^s_{(Q_1,Q_2,\theta)}(R)$	$\varrho_{[X_j^1/X_j]}(\Pi_{[X_j^1]}(\sigma^s_{C_2}(\sigma^s_{C_1}(Q_1' \bowtie Q_2'))))$ $Q_i'(r) = \{\varrho_{[X_j\|_{X_j^i}]}(t) \wedge \varrho_{[X_j\|_{\overline{X}_j^i}]}(Q_i(t))\|t \in r\}, i = 1, 2$ $C_1 = (\Pi_{[X_j^1]}(t), \varrho_{[X_j^2\|_{X_j^1}]}(\Pi_{[X_j^2]}(t)), =)$ $C_2 = (\Pi_{[\overline{X}_j^1]}(t), \varrho_{[\overline{X}_j^2\|_{\overline{X}_j^1}]}(\Pi_{[\overline{X}_j^2]}(t)), \theta)$	it depends on θ

Table 3.6: EGRA derived operators.

examples).

Theorem 3.1 *$EGRA(\Phi)$ is a n-based algebra.*

Proof: (Sketch) It is possible to show that for each EGRA(Φ) query there exists an equivalent nested-relational algebra query. Let D be the domain of Φ. The proof, presented in Appendix A, is based on the following translation of generalized relations and generalized tuples into nested relations:

- Each generalized relation R with schema $\{X_1, ..., X_n\}$ can be seen as a nested-relation of type $\{\langle A : \{\langle X_1 : D, ..., X_n : D\rangle\}\rangle\}$, where D is the domain of the chosen theory.

- Given a generalized tuple P with schema $\{X_1, ..., X_n\}$, P can be interpreted as the nested-relation $r(P)$ defined as $\{\langle A_P : \langle X_1 : a_1, ..., X_n : a_n\rangle\rangle \mid X_1 = a_1 \wedge ... \wedge X_n = a_n \in ext(P)\}$. Note that the type of $r(P)$ is $\{\langle A_P : \langle X_1 : D, ..., X_n : D\rangle\rangle\}$.

- Given a generalized tuple P with schema $\{X_1, ..., X_n\}$, P can also be interpreted as the nested-relation $n(P)$ containing only one element, represented by the set $ext(P)$. Thus, $n(P)$ coincides with the set $\{\langle A_P : \{\langle X_1 : a_1 \wedge ... \wedge X_n : a_n\rangle\}\rangle \mid X_1 = a_1 \wedge ... \wedge X_n = a_n \in ext(P)\}$. The type of $n(P)$ is $\{\langle A_P : \{\langle X_1 : D, ..., X_n : D\rangle\}\rangle\}$.

Using this representation, for each EGRA(Φ) query it is possible to construct an equivalent nested-relational algebra query. The complete proof is presented in Appendix A. □

3.3.1.2 Equivalence results

It is immediate to prove the following proposition.

Proposition 3.4 *$GRA(\Phi, \Sigma_1) \subseteq_r EGRA(\Phi, \{\wedge, \vee\})$.*

Proof: It is simple to show that, given some generalized relations $r_1, ..., r_n$, EGRA(Φ) tuple operators, when applied to $r_1, ..., r_n$, return a generalized relation that is r-equivalent to the generalized relation that is obtained by applying the corresponding GRA(Φ) operator to $r_1, ..., r_n$. Thus, GRA($\Phi, \{\wedge, \vee\}$) $\subseteq_r$ EGRA($\Phi, \{\wedge, \vee\}$).

Moreover, it can be shown that $S(\Phi, \Sigma_1) \subseteq_r S(\Phi, \{\wedge, \vee\})$, for all Σ_1.[2] From Proposition 3.2, it follows that GRA$(\Phi, \Sigma_1) \subseteq_r$ GRA$(\Phi, \{\wedge, \vee\})$. The proposition follows from the previous results by transitivity. □

Since the semantic function associated with the complement in GRA(Φ) always returns a generalized relation which is not n-equivalent to the generalized relation returned by the semantic function associated with the complement in EGRA(Φ) (when both semantic functions are applied to the same input generalized relation), it follows that GRA$(\Phi, \Sigma_1) \not\subseteq_n$ EGRA$(\Phi, \{\wedge, \vee\})$.

Now we analyze the opposite containment. A necessary condition for expressing an EGRA$(\Phi, \{\wedge, \vee\})$ query in GRA(Φ, Σ_2) is to modify the input database, coding in some way each generalized tuple as a set. The aim of this section is to prove that, due to this transformation, EGRA$(\Phi, \{\wedge, \vee\})$ and GRA(Φ, Σ_1) are not r-equivalent, whatever Σ_1 is.

To prove this result, a weaker notion of equivalence is first introduced. This new equivalence relation is called *weak*, since it relaxes the conditions under which the usual equivalence is defined (see Definition 3.6). The basic idea of weak equivalence is that of *coding* in some way the input of an EGRA$(\Phi, \{\wedge, \vee\})$ query, before applying the corresponding GRA(Φ, Σ_2) query. After that, a *decoding function* should be applied to the result, to remove the action of the encoding function. A similar approach has been taken in [137] and in [149] to prove results about the nested-relational algebra and the relational algebra. Encoding and decoding functions can be formalized as follows.

Definition 3.7 (Encoding and decoding functions) *An encoding function of type* $(\Phi, \Sigma_1, \Sigma_2)$ *is a total computable function* f *from* $S(\Phi, \Sigma_1)$ *to* $S(\Phi, \Sigma_2)$. *A decoding function of type* $(\Phi, \Sigma_1, \Sigma_2)$ *is a partial computable function* g *from* $S(\Phi, \Sigma_2)$ *to* $S(\Phi, \Sigma_1)$. □

Weak equivalence can be defined as follows.

Definition 3.8 (Weak equivalence) *Let* L_1 *and* L_2 *be two constraint query language. Let* $S(\Phi, \Sigma_1)$ *and* $S(\Phi, \Sigma_2)$ *be two EGR supports. Let* $t \in \{r, n\}$. $L_1(\Phi, \Sigma_1)$ *is weakly t-contained in* $L_2(\Phi, \Sigma_2)$ *(denoted by* $L_1(\Phi, \Sigma_1) \subseteq_t^w L_2(\Phi, \Sigma_2)$*) iff there exist an encoding function* f *of type* $(\Phi, \Sigma_1, \Sigma_2)$ *and a decoding function* g *of type* $(\Phi, \Sigma_1, \Sigma_2)$ *such that for each query* $q \in L_1(\Phi, \Sigma_1)$ *there exists a query* $q' \in L_2(\Phi, \Sigma_2)$ *with the following property:*

for all relations $r_i \in S(\Phi, \Sigma_1)$, $i = 1, ..., n$, $q(r_1, ..., r_n) \equiv_t g(q'(f(r_1), ..., f(r_n)))$.

[2]We recall that $\Sigma_1 \in \{\{\wedge\}, \{\vee\}, \{\wedge, \vee\}\}$.

$$\begin{tikzcd} r_1,\ldots,r_n \arrow[r, "q"] \arrow[d, "f"'] & r_{n+1} \\ r'_1,\ldots,r'_n \arrow[r, "q'"] & r'_{n+1} \arrow[u, "g"'] \end{tikzcd}$$

Figure 3.2: Graphical representation of weak containment.

$L_1(\Phi,\Sigma_1)$ *is weakly t-equivalent to* $L_2(\Phi,\Sigma_2)$ *(denoted by* $L_1(\Phi,\Sigma_1) \equiv^w_t L_2(\Phi,\Sigma_2)$*) iff* $L_1(\Phi,\Sigma_1) \subseteq^w_t L_2(\Phi,\Sigma_2)$ *and* $L_1(\Phi,\Sigma_1) \supseteq^w_t L_2(\Phi,\Sigma_2)$*.* $\square$

Figure 3.2 graphically represents weak containment. It is simple to show that if $L_1(\Phi,\Sigma_1) \subseteq_t L_2(\Phi,\Sigma_2)$, then $L_1(\Phi,\Sigma_1) \subseteq^w_t L_2(\Phi,\Sigma_2)$. Moreover, if two languages $L_1(\Phi,\Sigma_1)$ and $L_2(\Phi,\Sigma_2)$ are weak equivalent, they are also equivalent iff functions f and g can be represented in $L_2(\Phi,\Sigma_2)$.

In the following we prove that EGRA$(\Phi,\{\wedge,\vee\}) \subseteq^w_n$ GRA(Φ,Σ_1), assuming that $\wedge \in \Sigma_1$ (thus, either $\Sigma_1 = \{\wedge\}$ or $\Sigma_1 = \{\wedge,\vee\}$). However, to simplify the presentation, we suppose that $\Sigma_1 = \{\wedge\}$. The other case derives from that.

The chosen encoding and decoding functions of type (Φ,Σ_1,Σ_2) are presented in Table 3.7. Assuming we deal with a countable set of variables, without compromising the generality of the discussion, the definitions are given with respect to a countable set of variables $\tilde{N}$, only used to assign identifiers to generalized tuples.

The encoding function transforms a generalized relation $r \in S(\Phi,\{\wedge,\vee\})$ into a generalized relation $r' \in S(\Phi,\{\wedge\})$, such that each generalized tuple of r is contained in r' together with a new variable identifier, represented by a constraint admitting only one solution. Each generalized tuple of r containing disjunctions is divided in r' into several generalized tuples, all having the same identifier.

The decoding function projects the input relation on all variables, except those contained in $\tilde{N}$, if any. If more than one tuple in the input relation has the same values for variables in $\tilde{N}$, the disjunction of such tuples is taken.

Table 3.8 shows, for each EGRA operation, the corresponding weak equivalent GRA expression. In the following, the query function associated with an EGRA (GRA) operator is called *operator semantic function*. The two lemmas, presented below, are used in the proof of Theorem 3.2. See [13] for their complete proofs.

Lemma 3.1 *Let* $r_i \in S(\Phi,\Sigma)$ *such that* $\alpha(r_i) \cap \tilde{N} \neq \emptyset$, $i = 1,...,n$, $n \in \{1,2\}$*. Let* q *be the query associated with one of the GRA expressions listed in the second column*

Func. of type $(\Phi, \{\wedge, \vee\}, \{\wedge\})$	**Definition**
Encoding f	$f(r) = \bigcup_{t \in r'} f'(t)$ $r' = \{t_1^{m_1}, ..., t_n^{m_n} \mid r = \{t_1, ..., t_n\},$ for all $i, j,\ 1 \leq i \leq n,\ 1 \leq j \leq n,\ i \neq j,\ ext(t_i) \neq ext(t_j) \rightarrow m_i \neq m_j,$ $m_i \in D,\ i = 1, ..., n\}$ $t^m \equiv N = m \wedge t,\ N \in \tilde{N},\ N \notin \alpha(r),\ t \in r,\ m \in D$ $f'(t) = \{N = m \wedge t_1, ..., N = m \wedge t_n \mid t \equiv N = m \wedge (t_1 \vee ... \vee t_n)\}$ D is the domain of the considered theory Φ
Decoding g	$g(r) = \{\Pi_{[\alpha(r) \setminus \tilde{N}]}(t_1) \vee ... \vee \Pi_{[\alpha(r) \setminus \tilde{N}]}(t_n) \mid$ $t_1, ..., t_n \in r,\ \Pi_{[\tilde{N}]}(t_1) = ... = \Pi_{[\tilde{N}]}(t_n),$ $\nexists t_{n+1} \in r, \Pi_{[\tilde{N}]}(t_{n+1}) = \Pi_{[\tilde{N}]}(t_1)$, such that $ext(t_{n+1}) \neq ext(t_i),\ i = 1, ..., n\ \}$

Table 3.7: Encoding and decoding functions.

of Table 3.8. Let f and g as defined in Table 3.7. Then

$$g(q(f(g(r_1)), ..., f(g(r_n)))) \equiv_n g(q(r_1, ..., r_n)).$$

$\Box$

Lemma 3.2 *Let f and g as defined in Table 3.7. Let Σ_1 and Σ_2 be two signatures such that $\Sigma_1 = \{\wedge, \vee\}$ and $\wedge \in \Sigma_2$. For each EGRA(Φ, Σ_1) operator semantic function f_{OP} of arity n, $n \in \{1, 2\}$, there exists a GRA(Φ, Σ_2) query q such that for all $r_1, ..., r_n, n \in \{1, 2\}, r_i \in S(\Phi, \Sigma_1)$, $f_{OP}(r_1, ..., r_n) \equiv_n g(q(f(r_1), ..., f(r_n)))$.*

Proof: (Sketch) Let $R_1, ..., R_n$ be the names of the generalized relations belonging to the schema we consider. Let $\alpha(R_i') = \alpha(R_i) \cup \{N\}$, $i = 1, ..., n$, $N \in \tilde{N}$. Let D be the domain of Φ. Let Q_i' be the query obtained from query Q_i by inserting variable N in all projection operators. Table 3.8 shows for each basic EGRA$(\Phi, \{\wedge, \vee\})$ query the weakly equivalent GRA(Φ, Σ_2) query. See [13] for the complete proof. $\Box$

Theorem 3.2 *Let Σ_1 and Σ_2 be two signatures such that $\Sigma_1 = \{\wedge, \vee\}$ and $\wedge \in \Sigma_2$. Then, EGRA$(\Phi, \Sigma_1) \subseteq_n^w$ GRA(Φ, Σ_2).*

Proof: We prove the theorem by induction on the structure of an EGRA$(\Phi, \{\wedge, \vee\})$ query q.

Base case: q is an operator semantic function. The theorem follows from Lemma 3.2.

EGRA	GRA
R_i	R'_i
$\sigma_{P_1 \vee \ldots \vee P_n}(R)$	$\sigma_{P_1}(R')$ iff $n = 1$ $\sigma_{P_1}(R') \cup \ldots \cup \sigma_{P_n}(R')$ otherwise
$R_1 \bowtie R_2$	$R'_1 \bowtie \varrho_{[N/N']}(R'_2),\ N' \in \tilde{N}$
$R_1 \cup R_2$	$(R'_1 \bowtie \Pi_{[N']}(\varrho_{[N/N']}(\sigma_{N=n_1}(R'_1 \cup \neg R'_1)))) \cup A$ where $A = (R'_2 \bowtie \Pi_{[N']}(\varrho_{[N/N']}(\sigma_{N=n_2}(R'_2 \cup \neg R'_2))))$ $N' \in \tilde{N},\ n_1 \neq n_2,\ n_1, n_2 \in D$
$\Pi_{[\tilde{Y}]}(R)$	$\Pi_{[\tilde{Y} \cup \{N\}]}(R')$
$\neg R$	$\Pi_{[\alpha(R') \setminus \{N\}]}(\neg R') \bowtie \Pi_{[N]}(\sigma_{N=n_1}(R' \cup \neg R')),\ n_1 \in D$
$R_1 \setminus^s R_2$	$(\Pi_{[N]}(R'_1) \setminus \Pi_{[N]}(Y \bowtie Z)) \bowtie R'_1$ where $X = R'_1 \times \varrho^{R'_1}_{[R'_2]}(R'_2),\ N' \in \tilde{N}$ $Y = \Pi_{[N,N']}(X) \setminus \Pi_{[N,N']}(W)$ $W = \Pi_{[(\alpha(R'_1) \cup \{N'\})]}(X) \setminus \varrho^{-1}{}^{[R'_1]}_{R'_2}$[a] $(\Pi_{[(\alpha(X) \setminus \alpha(R'_1)) \cup \{N\}]}(X))$ $Z = \Pi_{[N,N']}(X) \setminus (\Pi_{[N,N']}(T)$ $T = \varrho^{-1}{}^{[R'_1]}_{R'_2}(\Pi_{[(\alpha(X) \setminus \alpha(R'_1)) \cup \{N\}]}(X))) \setminus \Pi_{[(\alpha(R'_1) \cup \{N'\})]}(X)$
$\neg^s(R)$	$\Pi_{[N]}(R') \bowtie \neg R'$
$\sigma^s_{(Q_1,Q_2,\subset)}(R)$	$(\Pi_{[N]}(R') \setminus \Pi_{[N]}(Q'_1(R')$[b] $\setminus Q'_2(R')) \bowtie R'$
$\sigma^s_{(Q_1,Q_2,\bowtie \neq \emptyset)}(R)$	$\Pi_{[N]}(Q'_1(R') \bowtie Q'_2(R')) \bowtie R'$

[a] $\varrho^{R'_1}_{[R'_2]}$ denotes the operation replacing each variable X of R'_2 also contained in the schema of R'_1 by a new variable X' such that $X' \in \tilde{N}$ iff $X \in \tilde{N}$. Moreover, $\varrho^{-1}{}^{[R'_1]}_{R'_2}$ denotes the operation replacing each variable $X' \notin \tilde{N}$ previously changed by $\varrho^{R'_1}_{[R'_2]}$ by its original symbol X.

[b] If Q_i is a generalized tuple P, $Q'_i(R')$ is the query that, for each generalized relation r, returns a new generalized relation containing n generalized tuples, where n is the cardinality of r. Each generalized tuple is equivalent to $N = m \wedge P$, where m is the generalized tuple identifier of a tuple in r.

Table 3.8: Translation of EGRA expressions into GRA expressions.

Inductive step: Let $q \equiv f_{OP}(q_1, q_2)$ where f_{OP} is an operator semantic function and q_1 and q_2 are queries (the proof assumes OP to be a binary operator; a similar proof holds also for unary operators). By inductive hypothesis we know that $q'_1, q'_2 \in GRA(\Phi, \Sigma_2)$ exist such that:

$$\forall r_1, \ldots, r_n \in S(\Phi, \Sigma_1)\ q_i(r_1, \ldots, r_n) \equiv_n g(q'_i(f(r_1), \ldots, f(r_n))), i = 1, 2$$

$$g(q'_i(f(r_1), \ldots, f(r_n))) \in S(\Phi, \Sigma_1)$$

From Theorem 3.1, we know that EGRA(Φ) is n-based. From Proposition 3.3 and the inductive hypothesis, we obtain that

$q(r_1, ..., r_n) = f_{OP}(q_1(r_1, ..., r_n), q_2(r_1, ..., r_n))$

is nested equivalent to

$S \equiv f_{OP}(g(q'_1(f(r_1), ..., f(r_n))), g(q'_2(f(r_1), ..., f(r_n))))$.

Let q' be the GRA(Φ) query corresponding to f_{OP} in Table 3.8. By Lemma 3.2, S is nested equivalent to $g(q'(f(r'_1), f(r'_2)))$, where $r'_i = g(q'_i(f(r_1), ..., f(r_n)))$, $i \in \{1, 2\}$.

From Lemma 3.1, we can replace $f(r'_i)$ with $q'_i(f(r_1), ..., f(r_n))$, $i = 1, 2$, obtaining that $g(q'(f(r'_1), f(r'_2)))$ is nested equivalent to

$g(q'(q'_1(f(r_1), ..., f(r_n)), q'_2(f(r_1), ..., f(r_n))))$.

Note that Lemma 3.1 can be applied since $q'_i(f(r_1), ..., f(r_n))$ satisfies the hypothesis of the lemma.

The query $\overline{q} \equiv q'(q'_1, q'_2)$ satisfies the theorem. □

Note that if $\wedge \notin \Sigma_2$, the equivalence does not hold. Indeed, in this case, there does not exist an encoding function of type $(\Phi, \{\wedge, \vee\}, \Sigma_2)$.

The following corollary presents final equivalence results about EGRA(Φ, Σ_1) and GRA(Φ, Σ_2).

Corollary 3.1 *Let Σ_1 and Σ_2 be two signatures such that $\Sigma_1 = \{\wedge, \vee\}$ and $\wedge \in \Sigma_2$. The following facts hold:*

1. *EGRA$(\Phi, \Sigma_1) \equiv_r^w$ GRA(Φ, Σ_2).*
2. *EGRA$(\Phi, \Sigma_1) \not\equiv_r$ GRA(Φ, Σ_2).*

Proof:

1. It follows from Proposition 3.4 and Theorem 3.2.
2. This result derives from the fact that the proposed encoding and decoding functions cannot be represented in GRA(Φ, Σ_2). □

The presented equivalence results are similar to equivalence results that have been presented for relational and nested relational languages. Indeed, nested computations can be embedded into FO, modulo the encoding of complex objects into flat ones and the corresponding decoding of output [117, 141].

Finally, note that even if EGRA($\Phi, \{\wedge, \vee\}$) $\equiv_r^w$ GRA($\Phi, \{\wedge\}$), GRA expressions are often very complex when compared with the equivalent EGRA expressions (see Table 3.8), even those implementing simple user requests.

Example 3.5 *The query of Example 3.3, which in GRA is represented as*

$$(\Pi_{[N]}(R) \setminus (\Pi_{[N]}(R \setminus \sigma_P(R)))) \bowtie R.$$

can be simply expressed in EGRA as $\sigma^s_{(t,P,\subseteq)}(R)$. $\diamond$

From Theorem 3.1 and Theorem 3.2, it is simple to prove that EGRA($\Phi, \{\wedge, \vee\}$) is a strict n-based language.

Corollary 3.2 *EGRA*(Φ) *is a strict n-based algebra.*

Proof:

- *EGRA*(Φ) *is a n-based algebra*: it follows from Theorem 3.1.
- *EGRA*(Φ) *is not r-based*: suppose EGRA(Φ) be r-based. Since GRA(Φ) is r-based, this means that for each signature Σ_2 and for $\Sigma_1 = \{\wedge, \vee\}$ EGRA(Φ, Σ_1) is r-equivalent to GRA(Φ, Σ_2). But this is not true. Indeed:
  - If $\wedge \in \Sigma_2$, from item (2) of Corollary 3.1 it follows that EGRA(Φ, Σ_1) is not r-equivalent to GRA(Φ, Σ_2).
  - If $\wedge \not\in \Sigma_2$, then $S(\Phi, \Sigma_1)$ is not r-equivalent to $S(\Phi, \Sigma_2)$. Therefore, by Proposition 3.2, EGRA(Φ, Σ_1) is not r-equivalent to GRA(Φ, Σ_2).

  Since in both cases we obtain a contradiction, EGRA(Φ) is not r-based. $\Box$

3.3.1.3 Data complexity

The analysis of data complexity of EGRA($\Phi, \{\wedge, \vee\}$) queries follows from the fact that EGRA($\Phi, \{\wedge, \vee\}$) $\equiv^w_r$ GRA(Φ, Σ_2), assuming that $\wedge \in \Sigma_2$, and results about data complexity of GRA(Φ, Σ_2).

It is simple to show that the data complexity of the chosen encoding and decoding functions f and g is in class NC.[3] Therefore, by considering Figure 3.2, we can deduce that, if the complexity of GRA(Φ, Σ_2) is in a complexity class C containing or equal to NC, the data complexity of EGRA($\Phi, \{\wedge, \vee\}$) is equal to the data complexity of GRA(Φ, Σ_2). Otherwise, it is at most in NC.

For example, from [82, 83] it follows that GRA(POLY, $\{\wedge\}$) has NC data complexity. Therefore, EGRA(POLY, $\{\wedge, \vee\}$) has NC data complexity.

[3]Thus, they can be executed in log-time, using a polynomial number of processors [8].

3.4 ECAL: a n-based generalized relational calculus

The relational algebra is a typical procedural language. This procedural language has a very natural declarative counterpart, represented by the relational calculus [47].

A similar situation arises in the generalized relational model where, as we have seen in Chapter 2, the generalized relational algebra and the generalized relational calculus have been defined as a natural extension of the corresponding relational languages to deal with infinite, but finitely representable, relations. As a consequence, the definition of an extended relational calculus which is equivalent to the algebra presented in Section 3.3, becomes an important issue.

There are at least two approaches in the literature to define a calculus and prove its equivalence with an algebra:

- *Codd's relational calculus* [47]. The calculus is based on first-order formulas. The calculus may generate unsafe relations, i.e., relations containing an infinite number of tuples. Safety is guaranteed by defining specific safety rules, that syntactically restrict the calculus expressions.

  In order to translate a calculus expression into the equivalent algebraic expression, a Cartesian product is generated from the set of all symbols existing in the database and then the answer is extracted from this set.

  This calculus has been extended in [125] to deal with relations having sets of atomic values as tuple components.

- *Klug's relational calculus* [88]. This calculus eliminates unsafe expressions by introducing explicit range expressions for variables. This approach results in a much cleaner way of expressing a calculus query and removes the burden of checking for safe expressions from users. The calculus also deals with aggregate function. Thus, the calculus must also be able to quantify over relations that are the result of an aggregate operation, and the only way to do this is to actually compute the result. For this reason, Klug's calculus is defined via mutual recursion on three types of expressions: terms, formulas, and alphas. Alphas are used to construct such intermediate relations.

  With aggregate functions, new aggregate values are created. Thus, in order to translate a calculus expression into the equivalent algebra expression, the approach based on the Cartesian product does not work. Therefore, a different approach is used. In particular, each object is translated into an algebraic expression; these algebraic expressions are then combined to generate the final expression.

Such a calculus has been extended in [111] to deal with relations containing sets of numbers as tuple components and in [93] to deal with constraints.

In order to define the extended generalized relational calculus, we take Klug's approach. The reason for this choice is motivated by the fact that, in Section 3.5, we will extend the generalized relational algebra and the generalized relational calculus to deal with external functions. External functions have some similarities with aggregate functions, in that they generate new values. Thus, the use of Klug's calculus simplifies the proof of the equivalence between the algebra and the calculus.

3.4.1 Syntax of the extended generalized relational calculus

The *extended generalized relational calculus* ECAL is defined via mutual recursion on three types of expressions: terms, formulas, and alphas. Terms represent the objects on which computations are performed (in our case, atomic values and generalized tuples). Formulas express properties about terms, and alphas are used to create new relations, composed either of relational tuples (thus defining a new generalized tuple) or of generalized tuples (thus defining a new generalized relation).

In defining the calculus, it is more convenient to use a positional notation. Thus, in the following, an attribute of a relational tuple is not identified by its name but by its position inside the tuple.

In defining the above objects, we assume we deal with two sets of variables:

- a set $V = \{v, v_1, v_2, ...\}$ of variables representing relational tuples;
- a set $G = \{g, g_1, g_2, ...\}$ of variables representing generalized tuples.

By considering a logical theory Φ, having D as domain, calculus objects are formally defined as follows.

Terms. Terms are used to represent the objects on which computations are performed. They can be either:

- *simple*, if they represent values from a given domain, such as real numbers;
- *set*, if they represent sets of relational tuples, whose attribute values are taken from the considered domain. Each set variable is a set term. Moreover, for each natural value n, we introduce a particular set term, representing the set of all possible relational tuples on domain D having n attributes. The introduction of these terms allows us to prove the equivalence of the algebra and the calculus even when queries generate new values with respect to those contained in the database.

No term is introduced to represent a single relational tuple since, due to the nested semantics, queries always manipulate (the extension of) generalized tuples.

Definition 3.9 (Terms) *A term has one of the following forms:*

- *c, such that $c \in D$;*
- *$v[A]$, where $v \in V$ and A is a column number;*
- *D^n, representing all relational tuples with degree n, with values from D;*
- *g, such that $g \in G$.*

The last two types of terms are set terms*, whereas the first two are* simple *terms.* □

Note that the considered set terms are different from those presented in [111]. Indeed, in that case, the available sets contain only atomic values whereas in our case, sets contain relational tuples.

Formulas. Formulas are used to express properties about terms. Atomic formulas are used to specify on which relation a generalized tuple or a relational tuple ranges, and to specify the relationship existing between two generalized tuples or some simple terms. Complex formulas are obtained by logically combining or quantifying other formulas. Both atomic and complex formulas can be either simple or set formulas. In the first case, they specify conditions on simple terms; in the second case, they specify conditions on set terms.

Definition 3.10 (Formulas) *A formula has one of the following forms:*

- *Atomic formula:*
  - *Simple formulas:*
    - *$t(v)$, where $v \in V$ and t is a closed target alpha (see below) or a set term;*
    - *$\mu(t_1, ..., t_n)$, where μ is a constraint and $t_1, ..., t_n$ are simple terms.*
  - *Set formulas:*
    - *$\alpha(g)$, where α is a closed general alpha (see below) and $g \in G$;*
    - *$t_1 \theta t_2$, where t_1, t_2 are set terms and $\theta \in \{\subseteq, \supseteq, =, \neq, \bowtie = \emptyset, \bowtie \neq \emptyset\}$.*
- *Complex formulas:*

- *$\psi_1 \wedge \psi_2$, where ψ_1 and ψ_2 are either simple formulas or set formulas; in the first case, $\psi_1 \wedge \psi_2$ is a simple formula, in the second case, is a set formula;*
- *$\psi_1 \vee \psi_2$, where ψ_1 and ψ_2 are either simple formulas or set formulas; in the first case, $\psi_1 \vee \psi_2$ is a simple formula, in the second case, is a set formula;*
- *$\neg\psi$ is a simple (set) formula if ψ is a simple (set) formula;*
- *$(\exists r_x)\psi$ is a simple (set) formula if ψ is a simple (set) formula and r_x is a range formula for x. The scope of $(\exists r_x)$ is ψ.* □

Since $\psi_1 \wedge \psi_2$ is equivalent to $\neg(\neg\psi_1 \vee \neg\psi_2)$, in the following we do not further consider symbol $\wedge$ [38].

Range formulas. Range formulas specify a range for either a simple variable or a set variable. Ranges for simple variables are closed target alphas (see below) or set terms and are called simple range formulas. Ranges for set variables are closed general alphas or atomic alphas (see below) and are called set range formulas.

Definition 3.11 (Range formulas) *Let $v \in V$ and let $\alpha_1, ..., \alpha_k$ be either closed target alphas or set terms. Then,*

$$\alpha_1(v) \vee ... \vee \alpha_k(v)$$

is a simple range formula.

Let $g \in G$ and let $\alpha_1, ..., \alpha_k$ be closed general alphas or atomic alphas. Then,

$$\alpha_1(g) \vee ... \vee \alpha_k(g)$$

is a set range formula. □

Alphas. An alpha represents either a set of relational tuples, i.e., a new generalized tuple, or a set of generalized tuples, i.e., a new generalized relation. Atomic alphas are a particular type of alphas, represented by generalized relation symbols.

Definition 3.12 (Alphas) *An alpha has one of the following forms:*

- *Atomic alpha: for each generalized relation symbol R, R is an alpha.*
- *Target alpha: if $t_1, ..., t_n$ are simple terms, $r_1, ..., r_m$ are simple range formulas for the free variables in $t_1, ..., t_n$, and ψ is a simple formula, then*

  $$((t_1, ..., t_n) : r_1, ..., r_m : \psi)$$

  is a target alpha.

- *General alpha: if t is a target alpha or a set term, $r_1, ..., r_m$ are set range formulas for the free variables in t, and ψ is a formula, then*

$$((t) : r_1, ..., r_m : \psi)$$

is a general alpha.

In the last two cases, ψ is called the qualifier *and $(t_1, ..., t_n)$ and t are called the* target. *Moreover, we denote n by $deg(\alpha)$.* □

When the target of a target alpha has the form $(v[1], ..., v[n])$, $v \in V$, and n is the arity of v, for the sake of simplicity we write v instead of $(v[1], ..., v[n])$.

The scope of a range formula in an alpha expression is the associated target and the qualifier of the alpha. Occurrence of a variable x is *free* if it is not bound by quantifiers or range formulas. A calculus object (term, formula, alpha) is *closed* if it has no free occurrences of any variable.

In the following, we denote with ECAL the language composed of all the closed set alphas generated by combining terms, formulas, and alphas, as explained before. Given a decidable logical theory Φ, admitting variable elimination and closed under complementation, we denote with ECAL(Φ) the set of queries represented by ECAL expressions. Note that, due to the fact that disjunction is allowed in formulas, the underlying signature is $\{\wedge, \vee\}$.

ECAL(Φ) allows the representation computations on generalized relations in two steps: first, conditions on generalized tuples are checked in the more external closed set alpha; then the more internal target alpha allows checking conditions on the extension of the selected generalized tuples.

The declarative nature of the calculus simplifies the interaction between the user and the system. The following examples present ECAL(Φ) queries corresponding to some of the EGRA(Φ) queries presented in Tables 2.5 and 3.4.

Example 3.6 *The following ECAL(Φ) expression represents the generalized tuple $X + Y \leq 2 \wedge Y \geq 7$.*

$$(v : D^2(v) : v[1] + v[2] \leq 2 \wedge v[2] \geq 7).$$

The range formula of the previous alpha specifies that we are interested in all relational tuples containing two attributes. The qualifier specifies the relation that must hold between the attributes of v. We assume that X corresponds to the first attribute and Y to the second one. Finally, the target specifies that we want to return all relational tuples v satisfying the qualifier. ◇

In the following, to simplify notation, the target alpha representing a generalized tuple P is denoted by t_P.

Example 3.7 *Consider the temporal join query presented in Table 2.7. In order to select all trains with destination station 3, standing by at a station S together with a train from station 4, the algebraic expression first selects all trains from station 4 (relation A') and all trains with destination station 3 and then performs a join of the two constructed generalized relations. Finally, it retrieves the identifiers of the selected pairs of trains. By using the calculus, assuming that the column number of ID is 1, we can declaratively express the previous query as follows:*

$$(((v_1[1], v_2[1]) : g_1(v_1), g_2(v_2), t_P(v_1), t_Q(v_2) :) : \alpha(g_1), \alpha(g_2) : g_1 \bowtie \neq \emptyset\ g_2)$$

where $\alpha = ((v[3] : g(v) :) : A(g) :)$ (position 3 identifies variable I). The previous alpha first checks for intersection all pairs of generalized tuples in A in order to determine all trains standing by at a station S together with another train. After performing this selection, some conditions are checked on the extension of the retrieved generalized tuples. In particular, the identifiers of the pairs of trains, the first having destination station 3 (specified by the range $t_P(v_1)$, where t_P represents the target alpha corresponding to the generalized tuple $P \equiv (T = 3)$) and the second being from station 4 (specified by the range $t_Q(v_2)$, where t_Q represents the target alpha corresponding to the generalized tuple $P \equiv (F = 4)$) are returned to the user.

As another example consider the spatial join, intersection based, presented in Table 3.4. The corresponding calculus expression is

$$(((v_1, v_2) : g_1(v_1), g_2(v_2) :) : R(g_1), R(g_2) : g_1 \bowtie \neq \emptyset\ g_2).$$

In the previous expression, first the intersection between spatial objects (i.e., generalized tuples) is checked and then the result is constructed starting from the extensions of each pair of intersecting tuples. This second step is required since the resulting tuple has to be a new generalized tuple, obtained by joining the extensions of the intersecting tuples. ◇

3.4.2 Interpretation of ECAL objects

In order to assign an interpretation to calculus objects introduced in the previous section, we follow the approach presented in [88], extended to deal with set terms. The proposed translation differs from that presented in [111], since the set terms we consider represent sets of relational tuples and not sets of atomic values, as in [111].

The result of the interpretation varies according to the type of the object under consideration:

- the interpretation of a formula produces values *true* (1) or *false* (0);
- the interpretation of a term is an atomic value or a set of relational tuples;
- the interpretation of an alpha is a relation.

In order to establish the association between variables in a calculus object and tuples in the current instances of the corresponding relations, the notion of *model* is introduced. Formally, a model M for a calculus object q is a triple $\langle I, S, X\rangle$, where:

- I is a database instance.
- S (the *free list* for object q) is a list of ordered pairs $\langle u_i, S_i\rangle$, where $u_i \in V \cup G$ is a free variable occurring in q and S_i is the domain (the relation) over which v_i ranges.
- X (the *valuation list* for q and D) is a list of pairs $\langle u_i, x_i\rangle$, where $u_i \in V \cup G$ is a free variable in q and $x_i \in S_i$ such that $\langle u_i, S_i\rangle \in S$.

Interpretations are assigned as follows.

Terms interpretation

- $c(M) = c$
- $v_i[A](M) = x_i[A]$
- $D^n(M) = D^n$
- $g_i(M) = x_i$

Formula interpretation

- $\alpha(g_i)(M) = \begin{cases} 1 & \text{if } x_i \in \alpha(M) \\ 0 & \text{otherwise} \end{cases}$
- $g_i(v_j)(M) = \begin{cases} 1 & \text{if } x_j \in x_i \\ 0 & \text{otherwise} \end{cases}$
- $(t_1 \theta t_2)(M) = \begin{cases} 1 & \text{if } t_1(M)\theta t_2(M) = 1 \\ 0 & \text{otherwise} \end{cases}$
- $(\mu(t_1, ..., t_n))(M) = \begin{cases} 1 & \text{if } \mu(t_1(M), ..., t_n(M)) = 1 \\ 0 & \text{otherwise} \end{cases}$

- $(\psi_1 \vee \psi_2)(M) = \begin{cases} 1 & \text{if } \psi_1(M) = 1 \textit{ or } \psi_2(M) = 1 \\ 0 & \text{otherwise} \end{cases}$

- $(\neg\psi)(M) = \begin{cases} 1 & \text{if } \psi(M) = 0 \\ 0 & \text{otherwise} \end{cases}$

- $((\exists r_{v_i})\psi)(M) = \begin{cases} 0 & \text{if } r_{v_i}(M) \text{ is empty} \\ MAX\{\psi(I, S', X') \mid u \in r_{v_i}(M)\} & \text{otherwise} \end{cases}$

  S' is similar to S except that the pair $\langle v_i, S_i\rangle$ is replaced in S' by $\langle v_i, r_{v_i}(M)\rangle$. X' is similar to X except that the pair $\langle v_i, u\rangle$ replaces $\langle v_i, x_i\rangle$.

Alpha interpretation

- $R_i(M) = r_i$ and r_i is the generalized relation named R_i in I.

- $((t_1, ..., t_n) : r_1, ..., r_m : \psi)(M) = \{(t_1(M'), ..., t_n(M')) \mid \psi(M') = 1\}$

  where $M' = \langle I', S', X'\rangle$. S' is the same as S except that for those variables v_j ranging over r_k, $1 \leq k \leq m$, S' contains $\langle v_j, r_k(M)\rangle$. X' is the same as X except that for those variables v_j ranging over r_k, $1 \leq k \leq m$, S' contains $\langle v_j, u\rangle$, $u \in r_k(M)$.

- $((t) : r_1, ..., r_m : \psi)(M) = \{s(M') \mid \psi(M') = 1\}$

  where $M' = \langle I', S', X'\rangle$. S' is the same as S except that for those variables u_j ranging over r_k, $1 \leq k \leq m$, S' contains $\langle u_j, r_k(M)\rangle$ (note that $u_j \in V \cup G$). X' is the same as X except that for those variables u_j ranging over r_k, $1 \leq k \leq m$, S' contains $\langle u_j, u\rangle$, $u \in r_k(M)$.

3.5 External functions

The introduction of external functions in database languages is an important topic. Functions increase the expressive power of database languages, relying on user defined procedures. External functions can be considered as library functions, completing the knowledge about a certain application domain.

In the context of constraint databases, external functions can be modeled as functions manipulating generalized tuples. Such manipulations must preserve the closure of the language. Thus, an external function f takes a generalized tuple t defined on a given theory Φ and a signature Σ and returns a new generalized tuple t' on Φ and Σ (to guarantee closure), obtained by applying function f to t. We assume that each function is total on the set of generalized tuples defined on Φ and Σ.

Given a generalized tuple t, it is often useful to characterize an external function f with respect to the following features:

- The set of variables belonging to $\alpha(t)$ to which the manipulation is applied. Indeed, it may happen that function f only transforms a part of a generalized tuple. Formally, this means that function f projects the generalized tuple on such variables before applying the transformation.

  This set is called *input set* of function f and it is denoted by $is(f)$. Thus, $is(f) \subseteq \alpha(t)$.

  In order to make the function independent of $\alpha(t)$, we consider an ordering of $\alpha(t)$. Such ordering is a total function, from $\{1, ..., card(\alpha(t))\}$[4] to $\alpha(t)$. Using such an ordering, $is(f)$ can be characterized as a set of natural numbers. We assume that each number $i \in is(f)$ identifies a variable X_i and it is denoted by $order_{\alpha(t)}(i)$.

- The set of variables, belonging to $\alpha(t)$, that are contained in $\alpha(f(t))$. This set of variables is called *local output set* and it is denoted by $los(f)$. Thus, $los(f) \subseteq is(t) \cap \alpha(f(t))$.

  Also $los(f)$ can be represented as a set of natural numbers.

  If $i \in is(f)$ but $i \notin los(f)$ this means that f uses variable X_i during its computation but it does not return any new value for X_i.

- The cardinality of set $\alpha(f(t)) \setminus \alpha(t)$, denoted by $n(f)$. For simplicity, we assume that, if $card(\alpha(f(t)) \setminus \alpha(t)) = n$, new variables are denoted by $New_1, ..., New_n$.

In summary, each function is associated with two sets of natural numbers $is(f)$ and $los(f)$ and an integer number $n(f)$.

By using the previous notation, an external function for constraint databases which guarantees closure is called an *admissible function* and can be formalized as follows (in the following, $DOM(\Phi, \Sigma, m)$ is the set of all the possible generalized tuples t on Φ and Σ, such that $card(\alpha(t)) = m$).

Definition 3.13 (Admissible functions) *Let Φ be a decidable logical theory and Σ be a signature. An admissible function f for Φ and Σ is a function from $DOM(\Phi, \Sigma, n_1)$ to $DOM(\Phi, \Sigma, n_2)$, such that $n_1 \geq max\{x | x \in is(f)\}$ and $n_2 = card(los(f)) + n(f)$. For any generalized tuple $t \in DOM(\Phi, \Sigma, n_1)$, associated with a given ordering $order_{\alpha(t)}$, function f returns a new generalized tuple $t' \in DOM(\Phi, \Sigma, n_2)$ such that $\alpha(t') = \{order_{\alpha(t)}(i) | i \in los(f)\} \cup \{New_1, ..., New_{n(f)}\}$.* □

[4]Given a set S, $card(S)$ represents the cardinality of S.

Given a generalized tuple t and an external function f, function $order_{\alpha(f(t))}$ is derived as follows:

$$order_{\alpha(f(t))}(i) = \begin{cases} order_{\alpha(t)}(j) & \text{if } j \text{ is the } i\text{-th element of the increasing} \\ & \text{ordering of } los(f) \\ New_k & \text{if } card(los(f)) < i \text{ and } k = i - card(los(f)). \end{cases}$$

The ordering induced by function f first lists variables in $los(f)$ and then new variables.

Example 3.8 *To show some examples of external functions, we consider metric relationships in spatial applications. Metric relationships are based on the concept of Euclidean distance referred to the reference space $\mathcal{E}^2$. Since a quadratic expression is needed to compute this type of distance, metric relationships can be represented in EGRA only if proper external functions are introduced. For example the following two functions can be considered:*

- Distance*: given a constraint c with four variables (X, Y, X', Y'), representing two spatial objects, it generates a constraint $Dis(c)$ obtained from c by adding a variable New_1 which represents the minimum Euclidean distance between the two spatial objects. Thus, assuming $order_{\alpha(c)}(1) = X$, $order_{\alpha(c)}(2) = Y$, $order_{\alpha(c)}(3) = X'$, and $order_{\alpha(c)}(4) = Y'$, we have $is(Dis) = \{1,2,3,4\}$, $los(Dis) = \{1,2,3,4\}$, and $n(Dis) = 1$.*

  A similar function, Dis', can be defined such that given a constraint c with four variables (X, Y, X', Y'), representing two spatial objects, it generates a constraint $Dis'(c)$ representing the minimum Euclidean distance between the two spatial objects. In this case, $is(Dis') = \{1,2,3,4\}$, $los(Dis') = \emptyset$, and $n(Dis') = 1$.

- Buffer*: given a constraint c with two variables (X, Y), it generates the constraint $Buf_\delta(c)$ which represents all points that have a distance from c less than or equal to δ. Thus, assuming $order_{\alpha(c)}(1) = X$ and $order_{\alpha(c)}(2) = Y$, we have $is(Buf_\delta) = \{1,2\}$, $los(Buf_\delta) = \{1,2\}$, and $n(Buf_\delta) = 0$. This means that the returned points are represented by using variables X and Y.*

In temporal applications, we believe that a "duration" function should also be included in the language. Note that the measure of the duration of an interval cannot be represented by DENSE*, since none of the mathematical operations are admitted in this theory. Therefore, in order to take into account the duration of an interval, the following external function has to be introduced:*

- Duration*: given an interval t, $\alpha(t) = \{X\}$, it produces the distance $Dur(t)$ on the axis of time between its starting point and its ending point (for example, the constraint $(X \geq 6) \wedge (X \leq 10)$ is transformed into $(X = 4)$). If t is a non-contiguous interval, the sum of the duration of all its intervals is produced. Thus, $is(Dur) = \{1\}$, $los(Dur) = \{1\}$, and $n(Dur) = 0$. This means that the returned value is represented by variable X.* $\diamond$

3.5.1 Introducing external functions in EGRA

When using external functions, new algebraic operators, called *application dependent operators*[5] can be added to EGRA(Φ):

- The family of *Apply Transformation* operators allows the application of an admissible function to all generalized tuples contained in a generalized relation. Two different types of apply transformations can be defined:
  - *Unconditioned apply transformation.*
    $AT_f(r) = \{f(t) \mid t \in r\}$.
    By using this operator, only the result of the function is maintained in the new relation.
  - *Conditioned apply transformation.*
    $AT_f^{\tilde{X}}(r) = \{\Pi_{[\tilde{X}]}(t) \bowtie f(t) \mid t \in r\}$
    where $\tilde{X} \subseteq \alpha(r)$. This transformation is called conditioned since the result of the application of function f to a generalized tuple t is combined with some information already contained in t. By changing $\tilde{X}$, we obtain different types of transformations.
    Note that for each conditioned apply transformation $AT_f^{\tilde{X}}$ there exists an external functions f' such that, for any generalized relation r, $AT_f^{\tilde{X}}(r) = AT_{f'}(r)$. The main difference between the two approaches is that the conditioned approach is more flexible and reasonable from a practical point of view

- The second operator (*Application dependent set selection*) is similar to the set selection of Table 3.3; the only difference is that now queries specified in the selection condition C_f may contain apply transformation operators.

[5]The term *application dependent operators* comes from the fact that functions reflect the application requirements.

Op. name	**Syntax** e	**Restrictions**	**Semantics** $r = \mu(e)(r_1)$
non-conditioned apply transformation	$AT_f(R)$		$r = \{f(t) \mid t \in r_1\}$
conditioned	$AT_f^{\tilde{X}}(R)$	$\tilde{X} \subseteq \alpha(R)$	$r = \{\Pi_{[\tilde{X}]}(t) \bowtie f(t) \mid t \in r_1\}$
set selection	$\sigma^s_{C_f}(R_1)$	$\alpha(e) = \alpha(R_1)$	$r = \{t : t \in r_1, C_f(t) \}$

Table 3.9: EGRA$(\Phi, \mathcal{F})$ application dependent operators.

By using the previous operators, we can now define the constraint algebra EGRA$(\Phi, \mathcal{F})$

Definition 3.14 *Let Φ be a decidable logical theory, admitting quantifier elimination and closed under complementation, and Σ be a signature. Let $\mathcal{F}$ be a set of admissible functions for Φ and Σ. We denote by EGRA$(\Phi, \mathcal{F})$ the set of queries that can be expressed by using EGRA operators and operators introduced in Table 3.9. We denote with EGRA$(\mathcal{F})$ the corresponding syntactic language.* □

Example 3.9 *Consider the external functions introduced in Example 3.8. As a first consideration, note that, given a generalized tuple t with four variables, expressions $AT_{Dis}(R)$ and $AT^{\alpha(t)}_{Dis'}(R)$ are equivalent. Indeed, in the first case each generalized tuple contained in the input generalized relation r is replaced by a new generalized tuple representing the old generalized tuple and, by using a new variable, the distance between the objects represented in the considered generalized tuple. In the second case, the function, only returns a new variable representing the distance between the two objects. The old objects are maintained due to the join performed by the $AT^{\alpha(t)}_{Dis'}(R)$ operator.*

Some relevant spatial queries using external functions are shown in Table 3.10(A). An example of temporal query using the function Duration is reported in Table 3.10(B). ◇

3.5.2 Introducing external functions in ECAL

In order to introduce external functions in ECAL, a new set term must be introduced in the language, representing the application of an external function to a generalized tuple. Given a set of admissible functions $\mathcal{F}$, the set term is

$f(g_i)$, where $f \in \mathcal{F}$ and $g_i \in G$.

Given a model M, the new set term is interpreted as follows:

$f(g_i)(M) = f(g_i(M))$.

Query	EGRA expression	Conditions
(A) Spatial Queries		
DISTANCE QUERY: select all spatial objects in R that are within 50 Km from the object in S identified by the point $pt \in \mathcal{E}^2$	$\sigma^s_c(R \bowtie \varrho_{[X\|_{X'},Y\|_{Y'}]}(S'))$ $S' = AT_{Buf_{50Km}}(\sigma_P(S))$	$P \equiv C_{point}(pt)$ $\alpha(P) = \{X, Y\}$ $c = (\Pi_{[X,Y]}(t), \Pi_{[X',Y']}(t), \bowtie \neq \emptyset)$
SPATIAL JOIN: generate all pairs $(r, s) \in R \times S$ such that the distance between r and s is less than 40 Km, together with the real distance between r and s	$\sigma_{New_1 \leq 40}(AT_{Dis}(R \bowtie S'))$ $S' = \varrho_{[X\|_{X'},Y\|_{Y'}]}(S)$	
(B) Temporal Queries		
DURATION SELECTION: select all trains standing at station S for more than two minutes	$\sigma^s_c\ (A)$	$P \equiv (I \geq 2)$ $Q(t) = AT_{Dur}(\Pi_{[I]}(t))$ $c = (Q(t), P, \bowtie \neq \emptyset)$

Table 3.10: Spatial and temporal queries in EGRA(LPOLY, $\mathcal{F}$) and EGRA(DENSE, $\mathcal{F}$).

This means that the interpretation of the application of a function to a generalized tuple variable is equivalent to applying function f to the interpretation of the generalized tuple variable.

Example 3.10 *Consider the spatial join introduced in Table 3.10. All pairs of spatial objects, the first contained in a generalized relation R and the second contained in a generalized relation S, have to be retrieved together with the distance among the objects, if it is less than 40 Km. In order to express this query in the calculus, first the alpha representing all pairs of spatial objects is generated; then, the distance is computed and, if it is lower than 40 Km, the pair is returned to the user. The expression is the following:*

$$(g : \alpha_1(g) : \exists\ g(v)\ v[5] \leq 40)$$

where $\alpha_1 = (Dis(g) : \alpha_2(g) :)$ and $\alpha_2 = (((v_1, v_2) : g_1(v_1), g_2(v_2) :) : R(g_1), R(g_2) :)$. In the previous expression, α_2 represents all pairs of spatial objects (corresponding to the algebraic Cartesian product), α_1 applies function Dis to the pairs of objects and the outer alpha checks the condition about the distance, represented by the fifth *column of the generalized tuples contained in α_1. As we can see, the previous expression allows us to represent the result in a "bottom-up" way, layering the different computations on different, but nested, alphas.* $\diamond$

Definition 3.15 *Let Φ be a decidable logical theory, admitting quantifier elimination and closed under complementation and Σ be a signature. Let $\mathcal{F}$ be a set of admissible functions for Φ and Σ. We denote by $ECAL(\Phi,\mathcal{F})$ the set of queries obtained by including term $f(t)$ in the calculus presented in Section 3.4. We denote with $ECAL(\mathcal{F})$ the corresponding syntactic language.* $\Box$

3.6 Equivalence between EGRA($\Phi,\mathcal{F}$) and ECAL($\Phi,\mathcal{F}$)

For the generalized relational calculus and algebra to be equivalent, the set $\mathcal{F}$ cannot be completely arbitrary, as the set of aggregate functions considered in [88] was not arbitrary. As in [88], we require that, if there is a function in $\mathcal{F}$ which operates on a given set of attributes, there must be similar functions which operate on *all* other possible sets of columns. This property, known as *uniformness property*, allows us to prove that each ECAL expression can be translated into an equivalent EGRA expression. It can be formally stated as follows.

Definition 3.16 (Uniformness property) *Let $\mathcal{F}$ be a set of admissible functions. Let $f \in F$, $f : DOM(\Phi,\Sigma,n_1) \rightarrow DOM(\Phi,\Sigma,n_2)$. Let $r \in S(\Phi,\Sigma)$ be a generalized relation such that $deg(r) \geq n_1$. Let $Z \subseteq \alpha(r)$. Suppose that $card(Z) \geq max\{x|x \in is(f)\}$. We define $AT_{f,Z}(r) = \{f(\Pi_{[Z]}(t)) \mid t \in r\}$.*

The uniformness property holds in $\mathcal{F}$ iff for all $f \in \mathcal{F}$, for all $r \in S(\Phi,\Sigma)$, for all $Z \subseteq \alpha(r)$ such that $card(Z) \geq max\{x|x \in is(f)\}$, there exists $\overline{f}$ such that $AT_{f,Z}(r) = AT_{\overline{f}}(r)$. $\Box$

Note that, due to the defined ordering on $\Pi_{[Z]}(t)$, the application of function f to $\Pi_{[Z]}(t)$ is well defined.

The uniformness property prevents situations as the following one. Suppose that a function f is defined for LPOLY. For example, f may count the number of edges belonging to the spatial object representing the extension of a given generalized tuple. Now consider a generalized relation r on POLY. In general f cannot be applied to generalized tuples of r. However, suppose that we know that the projection of each generalized tuple in r on X and Y is linear. In this case, f can be applied to r, ensuring that only attributes X and Y are considered. This assumption violates the uniformness property, saying that f should be applied to all pairs of attributes.

In the following, we formally prove that EGRA($\Phi,\mathcal{F}$) and ECAL($\Phi,\mathcal{F}$) are equivalent if $\mathcal{F}$ satisfies the uniformness property.

3.6.1 Translating EGRA($\Phi, \mathcal{F}$) to ECAL($\Phi, \mathcal{F}$)

In the following, for each algebraic expression $e \in$ EGRA($\mathcal{F}$), an equivalent closed alpha $\alpha \in$ ECAL($\mathcal{F}$) is presented such that for all generalized relational database instances I, $e(I) = \alpha(I)$. The notation $T_{ac}(e) = \alpha$ denotes this transformation.

To simplify the presentation, we use the following notation:

$T_{ac}(e) = \alpha = (t) : r_1, ..., r_m : \psi$
$T_{ac}(e_1) = \alpha_1 = (t_1) : r_1^1, ..., r_m^1 : \psi_1$
$T_{ac}(e) = \alpha_2 = (t_2) : r_1^2, ..., r_m^2 : \psi_2$
$deg(e) = n.$[6]

Translation T_{ac} is defined as follows. The proof of the equivalence between algebra and calculus expressions is presented in Appendix A.

1. $T_{ac}(R_i) = R_i$.

2. $T_{ac}(\sigma_P(e)) = ((x : g(x), t_P(x) :) : \alpha(g) :)$

   where t_P is the target alpha representing the generalized tuple P.

3. $T_{ac}(\Pi_{[X]}(e)) = (t[X] : r_1, ..., r_h : (\exists r_{h+1})...(\exists r_m)\psi)$

   $t[X]$ contains free variables $v_1, ..., v_h$ ranging over $r_1, ..., r_h$. Variables of α that are not included in the projection list range over $r_{h+1}, ..., r_m$.

   Since α is a general alpha, t is either a target alpha or a set term. In the first case, if $t = (t_1 : r'_1, ..., r'_n : \psi)$, $t[X]$ is defined as $(t_1[X] : r'_1, ..., r'_n : \psi)$, in the second case $t[X]$ is defined as $(v[X] : t(v) :)$.

4. $T_{ac}(e_1 \bowtie e_2) = (\alpha_3 : \alpha_1(g_1), \alpha_2(g_2) :)$

   where $\alpha_3 = ((v_1, v_3) : g_1(v_1), g_2(v_2) : \wedge_{k=1,n} v_1[X_{i_k}] = v_2[X_{j_k}])$ and each pair (X_{i_k}, X_{j_k}) represents a pair of variables on which natural join is performed and v_3 is the tuple formed by all columns of v_2 except $X_{i_1}, ..., X_{i_n}$.

5. $T_{ac}(\neg e) = ((v : D^n(v) : (\nexists \alpha(g))\ g(v)) : :)$

6. $T_{ac}(\neg^s(e)) = ((v : D^n(v) : \neg g(v)) : \alpha(g) :)$

7. $T_{ac}(e_1 \cup e_2) = (t : \alpha_1(g) \vee \alpha_2(g) :)$

8. $T_{ac}(e_1 \setminus^s e_2) = (t : \alpha_1(g) : \neg \alpha_2(g))$

[6]Given an expression e, we denote with $deg(e)$ the arity of any relation r' obtained as result of $\mu(e)$.

9. $T_{ac}(e_1 \setminus e_2) = ((v : g_1(v) : \neg\exists\alpha_2(g_2)g_2(v)) : \alpha_1(g_1) :)$

10. $T_{ac}(\sigma^s_{(Q_1,Q_2,\theta)}(e)) =$
$(g : \alpha(g) : (\exists T_{ac}(Q'_1)(g_1))((\exists T_{ac}(\Pi_{[\alpha(Q_1)]}(Q'_2))(g_2))\ g_1\theta g_2))$
where Q'_i is obtained from Q_i by replacing constant t with relation $\{t\}$. Note that $T_{ac}(Q'_i)(I)$ is always a generalized relation containing a single generalized tuple, that can be represented by using a target alpha $\overline{t}_i$. For the sake of simplicity, expression $(\exists T_{ac}(Q'_1)(g_1))((\exists T_{ac}(\Pi_{[\alpha(Q_1)]}(Q'_2))(g_2))\ g_1\theta g_2)$ is denoted by $\overline{t}_1\theta\overline{t}_2$.

11. $T_{ac}(AT_f(e)) = (f(g) : \alpha(g) :)$.

12. $T_{ac}(AT^{\tilde{X}}_f(e)) = (((v_2[\tilde{X} \setminus los(f)], v_1) : (f(g))(v_1), g(v_2) : v_2[\tilde{X} \cap los(f)] = v_1[\tilde{X} \cap los(f)]) : \alpha(g) :)$.[7]

3.6.2 Translating ECAL($\Phi, \mathcal{F}$) to EGRA($\Phi, \mathcal{F}$)

Similarly to what has been done in [88, 111], a calculus object q is translated into an algebraic expression by translating each individual component of q recursively and then combining these translations. The translation is based on the following principles:

- the translation of a simple term produces a relation containing only one (unary) relational tuple;
- the translation of a set term produces a generalized relation containing only one generalized tuple;
- a set formula is translated into a generalized relation containing those generalized tuples for which the interpretation of the formula is 1;
- a simple formula is translated into a relation containing those relational tuples for which the interpretation of the formula is 1;
- a set alpha is translated into a set of generalized tuples that satisfy the range formulas as well as the qualifier;
- a simple alpha is translated into a set of tuples that satisfy the range formulas as well as the qualifier.

[7]In the previous expression, $v_i[\tilde{X} \cap los(f)]$ is a shorthand for the tuple $(v_2[i_1], ..., v_2[i_s])$, $i_j \in \tilde{X} \cap los(f)$, $j = 1, ..., s$.

The problem of a recursive translation is the presence of free variables. Indeed, during the recursive translation, free variables must be represented in some way, since a calculus object cannot have a well-defined value until values for the free variables are given. To consider this aspect, the translation does not map a calculus object into just an algebraic expression. Rather, a list of pairs is associated with each calculus object and called *free-attribute list*. Each pair maintains information about the relationship between variables (or variable attributes) and column numbers of the algebraic expression associated with the calculus object.

More formally, as in [88, 111], the input of the translation consists of a calculus object q and a model M. The output contains an algebraic expression e, a *free attribute list* L and either a *projection list* Z (for terms and alphas) or another expression E (for formulas), whose meaning is explained below:

- Z contains the sequence of those columns numbers that are projected from e.
- E is an algebraic expression that provides the domain where a formula is evaluated.
- L represents either a *simple free attribute list* or a *set free attribute list*.

  A simple free attribute list is a set of pairs $\langle v_i[A], c\rangle$, where $v_i \in V$ ranges over a relation e such that $deg(e) \geq A$ and c is a column number of relation e', associated with q. Such a pair says that column c in e' "represents" column A of the free variable v_i.

  A set free attribute list is a set of pairs $\langle g_i, sc\rangle$, where $g_i \in G$ ranges over a generalized relation e, and sc is a set of column numbers of relation e' associated with q. Such a pair says that the projection of the generalized tuples contained in the resulting expression on sc "represents" variable g_i.

We denote with $T_{ca}(q)$ the translation function. To simplify the presentation, we assume that the qualifier of a set alpha is a set formula. This assumption does not reduce the expressivity of the calculus. Indeed, as we can observe from the translation presented in Subsection 3.6.1, all calculus expressions equivalent to the algebraic ones satisfy this condition.

In the following, if L_1 is a simple free attribute list and L_2 is a set attribute list, we use the following notation:

1. $L_1 = X$.

   If $\langle v_i[A], c\rangle \in L_1$, then $c = x_i[A]$[8] is a component of $L_1 = X$.

[8] Recall that we assume that $\langle v_i, x_i\rangle \in X$.

2. $L_2 = X$.

   If $\langle g_i, sc\rangle \in L_2$, then $(x_i, \Pi_{[sc]}(t), =)$ belongs to $L_2 = X$.

Given a model $M = \langle I, S, X\rangle$ and a calculus object q, the translation is defined as follows:

- For every simple term t, we define an expression e, a simple free attribute list L, and a projection list Z such that for all I, S, and x: $\Pi_{[Z]}(\sigma_{L=x}(e))(I) = \{t(I, S, x)\}$.
- For every set term t, we define an expression e, a set free attribute list L, and a projection list Z such that for all I, S, and x: $\Pi_{[Z]}(\sigma^s_{L=x}(e))(I) = \{t(I, S, x)\}$.
- For every simple formula ψ we define an expression e, a simple free attribute list L, and another expression E such that for all I, S, and x:
  $\sigma_{L=x}(E)(I) \neq \emptyset$
  $\sigma_{L=x}(e)(I) = \sigma_{L=x}(E)(I)$ if $\psi(I, S, x) = 1$
  $\sigma_{L=x}(e)(I) = \emptyset$ if $\psi(I, S, x) = 0$
- For every set formula ψ, we define an expression e, a set free attribute list L, and a another expression E such that for all I, S, and x:
  $\sigma^s_{L=x}(E)(I) \neq \emptyset$
  $\sigma^s_{L=x}(e)(I) = \sigma^s_{L=x}(E)(I)$ if $\psi(I, S, x) = 1$
  $\sigma^s_{L=x}(e)(I) = \emptyset$ if $\psi(I, S, x) = 0$
- For every simple alpha α, we define an expression e, a simple free attribute list L, and a projection list Z such that for all I, S, and x: $\Pi_{[Z]}(\sigma_{L=x}(e))(I) = \alpha(I, S, x)$.
- For every set alpha α define an expression e, a set free attribute list L, and a projection list Z such that for all I, S, and x: $\Pi_{[Z]}(\sigma^s_{L=x}(e))(I) = \alpha(I, S, x)$.

Without assuming that the qualifier of a set alpha is a set formula, the translation of set alphas becomes more complicated since the set selection has to be replaced by a sequence of set and tuple selections. The formal proof of the equivalence is very technical, as the proofs presented in [88] and [111], and it is presented in Appendix A. As a final remark, it is important to note that, in order to guarantee that in the given translation selection operators are always well defined, it is necessary to prove the following result. The proof can be easily derived by induction on the structure of calculus objects.

Lemma 3.3 *Let q be a calculus object. Let $T_{ca}(q) = \langle e, F, L\rangle$, where F may be an expression or a projection list. If q is a set object, then L is a set free attribute list; if q is a simple object, then L is a simple free attribute list.* $\square$

3.7 Concluding remarks

In this chapter we have introduced a new nested semantics for generalized relations. The new semantics is obtained by slightly modifying the relational semantics introduced in Chapter 2 and interpreting each generalized relation as a finite set of possibly infinite sets, each representing the extension of a generalized tuple. We have also characterized the properties of languages based on this semantics with respect to languages for generalized databases based on the relational semantics. An algebra and a calculus based on the nested semantics have then been proposed and extended with external functions. Finally, these two languages have been formally proved to be equivalent. As far as we know, this is the first approach introducing external functions in constraint query languages.

Chapter 4

An update language for relational constraint databases

A data manipulation language must provide constructs to both retrieve and update data. The definition of an update language is a much harder issue in constraint databases, especially when used to represent spatial data. Indeed, spatial objects are often subject to transformation with respect to either their shape (for example, in rescaling or adding a new object component) or their position in the space (for example, translation or rotation).

The aim of this chapter is to introduce an update language based on the same principles on which the query languages presented in Chapter 3 are based. Following the relational approach, at least three update operators must be defined for constraint databases: insertion, deletion, and modification of generalized tuples.

The distinction between point-based and object-based manipulation, introduced in Chapter 3, can also be taken into account in the definition of update operators. At this level, they have the following meaning:

- A point-based update modifies a set of generalized tuples, seen as a possibly infinite set of points. Thus, it may add, delete, or modify some points, possibly changing the extension of the already existing generalized tuples.
- An object-based update modifies a generalized relation by inserting, deleting, or modifying a generalized tuple, seen as a single value.

The remainder of this chapter is organized as follows. In Section 4.1, insert operators are presented, together with examples motivating their introduction. Delete operators are presented in Section 4.2 whereas update operators are introduced in Section 4.3. We show under which hypothesis the proposed update operator collapses

Operator Name	Syntax e	Restrictions	Semantics $r_1 := \mu(e)(r_1)$
tuple insert	$Ins^t(R_1, C, u)$	$\alpha(u) \subseteq \alpha(R_1)$	$r_1 :=^1 \{t \vee u \mid t \in r_1 \wedge C(t)\} \cup$ $\{t \mid t \in r_1 \wedge \neg C(t)\}$
set insert	$Ins^s(R_1, u)$	$\alpha(u) \subseteq \alpha(R_1)$	$r_1 := r_1 \cup \{u\}$
tuple delete	$Del^t(R_1, C, u)$	$\alpha(u) \subseteq \alpha(R_1)$	$r_1 := \{t \wedge \neg u \mid t \in r_1 \wedge C(t)\} \cup$ $\{t \mid t \in r_1 \wedge \neg C(t)\}$
set delete	$Del^s(R_1, C)$	$\alpha(C) \subseteq \alpha(R_1)$	$r_1 := \{t \mid t \in r_1 \wedge \neg C(t)\}$
set update	$Upd^s(R_1, C, Q)$	$\alpha(Q) \subseteq \alpha(R_1)$ $\alpha(C) \subseteq \alpha(R_1)$	$r_1 := \{t \mid t \in r_1 \wedge \neg C(t)\} \cup$ $\{Q(t) \mid t \in r_1 \wedge C(t)\}$

Table 4.1: Update operators.

to delete and insert operators and under which other hypothesis it corresponds to some useful spatial operations, such as translation, rotation, etc.

All the queries and conditions we consider in the definition of the update language are expressed by using EGRA($\mathcal{F}$), for some set of admissible functions $\mathcal{F}$. A similar definition can be proposed by using ECAL($\mathcal{F}$).

4.1 Insert operators

Specific application requirements lead to the definition of tuple and set insert operators. In particular:

- The definition of a *set insert* operator is motivated by the fact that a typical requirement is the insertion of a new generalized tuple in a generalized relation.

  The set insert operator satisfies this requirement by taking a generalized relation r and a generalized tuple t as input, and adding t to r, thus increasing the cardinality of r. Since r is a set, the set insert operation is a no-operation if t is already contained in r. This operation can be reduced to an equivalence test between generalized tuples. As generalized tuples are usually represented by using canonical forms [82], this test usually reduces to check whether two canonical forms are identical.

- Because a generalized relation contains sets of relational tuples, the user may be interested in inserting a relational tuple or a set of relational tuples into some of the existing sets of relational tuples. Note that this requirement is different from the previous one, since in this case we extend the extension of the already existing generalized tuples but we do not insert any new one.

Given a generalized relation r, a boolean condition C (see the definition of set selection in Chapter 3) and a generalized tuple t, the *tuple insert* operator selects all generalized tuples of r that satisfy C and adds to them the relational tuples contained in $ext(t)$. Notice that the tuple insert does not change the cardinality of the target generalized relation.

In Table 4.1, the syntax and the semantics of insert operations are presented following the style used in Table 2.1 in Chapter 2 and Table 3.3 in Chapter 3. For update operations, μ is a function that takes an update expression and returns a function, representing the update semantics.

Example 4.1 *Figure 4.1 shows a possible geographical domain. The space is decomposed in four districts. Districts may contain towns, railway sections, and stations. Districts, towns, and railway sections are concave objects, whereas stations are convex. A possible representation of this domain in the extended generalized relational model is the following:*

- *Districts can be stored in a generalized relation D. Each d-generalized tuple represents a single district.*
- *Towns can be stored in a generalized relation T. Each d-generalized tuple represents a single town.*
- *Railway sections and stations can be stored in several ways. For example, each railway section, together with the stations located along the section, can be represented by using a single d-generalized tuple. We assume that railway sections and stations are represented in this way inside a generalized relation R.*

We assume that the schema of generalized relations D, T, and R contains two variables X and Y, representing points belonging to the object extension, and a variable ID, representing the generalized tuple identifier. This identifier is not inserted to "glue" together the extension of different generalized tuples, as in the generalized relational model. Rather, it has been introduced to better identify the considered spatial objects in query expressions.

Figure 4.1 also shows a dashed line and an empty square, representing a new railway section and a new station to be inserted in the database, respectively. In particular, the new railway section identified by $ID = 5$ (in the figure, represented by V) is added to the generalized relation R and the new station is added to the railway sections identified by $ID \in \{1, 2, 4\}$, since it is an interchange node of the railway network. The first insertion is performed using the set insert operator, because a new spatial object has to be created. In order to perform the second insertion, the

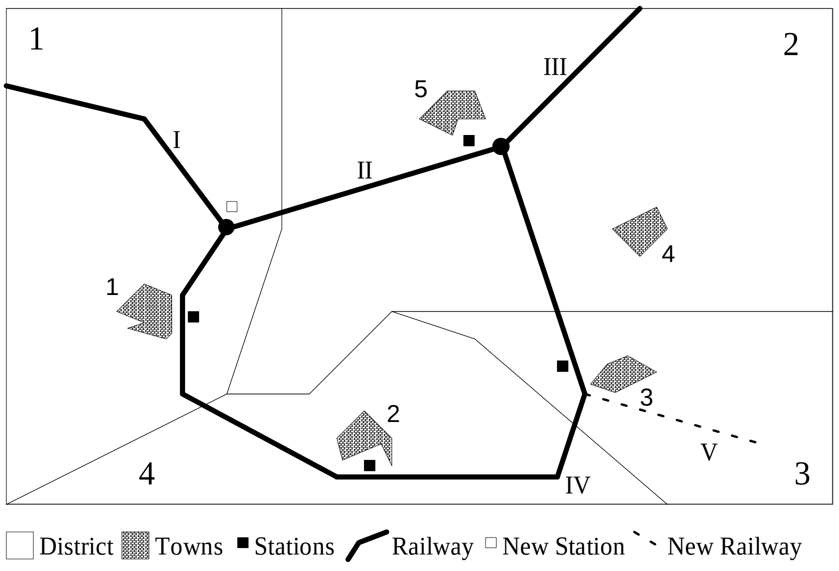


Figure 4.1: The map shows the content of the generalized relations D, R and T, together with the spatial objects to be inserted.

tuple insert operator can be used, since, due to the chosen representation, only a modification of the extent of existing spatial objects has to be performed. Table 4.2 shows the expressions corresponding to the two insertions. ◇

4.2 Delete operation

For the delete operations the discussion is similar to the one presented for the insert operations. We therefore introduce two operators:

- The *set delete* operator, given a generalized relation r and a boolean condition C, deletes from r all generalized tuples that satisfy C.
- The *tuple delete* operator, given a generalized relation r, a boolean condition C, and a generalized tuple t, selects all the generalized tuples of r that satisfy C and removes from their extension the relational tuples contained in $ext(t)$.

In Table 4.1, the syntax and semantics of delete operations are presented. As an example, referred to Example 4.1, Table 4.2 shows the expressions to delete respectively the town with $ID = 1$ and the station of the railway section with $ID = 2$.

4.3 Modify operators

Traditional database systems provide a modify operation to deal with updates that are function of the old values of the tuples. In constraint database systems this case is very common since operations of this kind, like rescaling, translation or rotation, are often applied to spatial objects, represented by generalized tuples. Therefore the introduction of a modify operator (also called update operator) in a spatial oriented data model is necessary.

Note that, due to the nested semantics assigned to a generalized relation, point-based and object-based modify operations coincide. However, we choose to classify the modify operator as a set operator, since it always modifies a generalized tuple.

In a traditional data manipulation language (for example, SQL), the modify operation allows computing the new value, to be assigned to the updated tuple, by a database query. Following the same approach we propose a *set update* operator with the following semantics. Given a generalized relation r, a boolean condition C, and a query $Q(t)$, the set update operator selects all tuples t of r that satisfy C and substitutes each t with $Q(t)$. The query $Q(t)$ acts on a single generalized tuple, denoted by t, at a time, as in the definition of the set selection operator. The generalized tuple t is considered as a generalized relation containing only one generalized tuple. This implies that all set operators of EGRA(Φ) are useless, since eventually they can only delete t. Note that, also the union operator cannot be used inside $Q(t)$, because it will necessary generate a relation with at least two generalized tuples. However, since an operator that generates the disjunction of two generalized tuples could be

useful to express some spatial transformations, we introduce a *tuple union* operator defined as

$$R_1 \cup^t R_2 = t_{1,1} \vee \ldots \vee t_{1,n} \vee t_{2,1} \vee \ldots \vee t_{2,m}$$

assuming that $R_1 = \{t_{1,1}, \ldots, t_{1,n}\}$ and $R_2 = \{t_{2,1}, \ldots, t_{2,n}\}$. Therefore, we restrict $Q(t)$ to be an expression of $(\text{EGRA}(\mathcal{F}) \setminus \{\cup, \sigma^s_C, \setminus^s, \neg^s\}) \cup \{\cup^t\}$[2]. Table 4.1 presents the definition of the set update operation.

Notice that, depending on the operators used in $Q(t)$, a different modification of tuple t is obtained. In particular, the following proposition holds.

[2]With $(\text{EGRA}(\mathcal{F}) \setminus S) \cup S'$ we denote the expressions of the language obtained from EGRA($\mathcal{F}$) by not using operators in S but possibly using new operators contained in S'.

Description	EGRA update expression
insertion of a new railway section in R, with $ID = 5$	$Ins^s(R, \langle ID = 5 \wedge 105 \leq X \leq 137^a \wedge Y = -\frac{9}{32}X + 50\rangle)$
insertion of a new station belonging to the railway sections of R with $ID \in \{1,2,4\}$	$Ins^t(R, (t, (ID = 1 \vee ID = 2 \vee ID = 4), \bowtie \neq \emptyset),$ $\langle 53 \leq X \leq 55 \wedge 40 \leq Y \leq 42\rangle)$
deletion of the town with $ID = 1$ contained in T	$Del^s(T, (t, (ID = 1), \bowtie \neq \emptyset))$
deletion of a specific station from the railway section in R with $ID = 2$	$Del^t(R, (t, (ID = 2), \bowtie \neq \emptyset),$ $\langle 65 \leq X \leq 67 \wedge 83 \leq Y \leq 85\rangle)$

[a] $a \leq X \leq b$ is an abbreviation for $X \geq a \wedge X \leq b$.

Table 4.2: Examples of EGRA(LPOLY) insertions and deletions.

Description	
ECAL	EGRA
Projection ($Q^{prj}_{X_{i_1},...,X_{i_m}}(u)$) it projects the n-dimensional generalized tuple u onto the $X_{i_1},...,X_{i_m}$ ($m < n$) coordinates	
$(((v[i_1],...,v[i_m]) : t(v) :) : R(t) :)$	$\Pi_{[X_{i_1},...,X_{i_m}]}(u)$
Minimum Bounding Rectangle ($Q^{mbr}(u)$) it generates the Minimum Bounding Box of the extension of u	
$(((v_1[1], v_2[2]) : t(v_1), t(v_2) :) : R(t) :)$	$\Pi_{[X]}(u) \bowtie \Pi_{[Y]}(u)$
Translation ($Q^{tra}_{a,b}(u)$) it translates the extension of u according to the vector $< a, b >$	
$(((v[1], v[2]) : t(v), t_c(v) :) : \alpha_1(t) :)$ where $\alpha_1 = (((v_1, v_2) : t_1(v_1), t_2(v_2) :) : R(t_1), D^n(t_2) :)$	$\Pi_{[X,Y]}(\sigma_c(\varrho_{[X\|_{X'}, Y\|_{Y'}]}(u) \bowtie (u \cup^t \neg u)))$ $c \equiv X = X' + a \wedge Y = Y' + b$
Rotation ($Q^{rot}_{\overline{X},\overline{Y},a_1,a_2,b_1,b_2}(u)$) it rotates the extension of u according to the rotation coefficients $< a_1, a_2, b_1, b_2 >$[a]	
$(((v[1], v[2]) : t(v), t_c(v) :) : \alpha_1(t) :)$ where $\alpha_1 = (((v_1, v_2) : t_1(v_1), t_2(v_2) :) : R(t_1), D^n(t_2) :)$	$\Pi_{[X,Y]}(\sigma_c(\varrho_{[X\|_{X'}, Y\|_{Y'}]}(u) \bowtie (u \cup^t \neg u)))$ $c \equiv (X = a_1(X' - \overline{X}) + a_2(Y' - \overline{Y})) \wedge$ $(Y = b_1(X' - \overline{X}) + b_2(Y' - \overline{Y}))$

[a] The coefficients a_1, a_2, b_1, b_2 define the rotation, however only 2 of them are independent, i.e., $a_{1,1} = a$, $a_{1,2} = b$, $a_{2,1} = -b$ and $a_{2,2} = a$, where $a = \cos\alpha$, $b = \sin\alpha$, and α is the rotation angle.

Table 4.3: Examples of queries to be specified in the modify operator. In the ECAL column, R represents the alpha (t_u ::), where t_u is the target alpha representing u (see Subsection 3.4.1).

Proposition 4.1 *Given a generalized relation identifier R, a boolean condition C and a query Q, expressed by using the operators of $EGRA \backslash \{\varrho, \Pi, \cup, \sigma^s_C, \backslash^s, \neg^s\}$, a generalized tuple P exists such that:*

$$Upd^s(R, C, Q(t)) = Upd^s(R, C, \sigma_P(t)).$$

Proof: Since t is a generalized tuple, $\sigma_P(t)$ is equal to $(t \wedge P)$. Thus, the proposition is proved if we show that a generalized tuple P always exists, such that $Q(t) = t \wedge P$. This is proved by induction on the structure of $Q(t)$:

- *Base step*: $Q(t) = t$. The generalized tuple P, representing the true formula on the schema of R, satisfies the proposition.
- *Inductive step*:
  - $Q(t) = \sigma_{\overline{P}}(f(t))$.
    By inductive hypothesis, $f(t) = t \wedge P'$. Thus, from the definition of the selection operator, $Q(t) = t \wedge P' \wedge \overline{P}$, and therefore $Q(t) = t \wedge P$, where $P = P' \wedge \overline{P}$.
  - $Q(t) = f'(t) \setminus f''(t)$.
    By inductive hypothesis, $f'(t) = t \wedge P'$, $f''(t) = t \wedge P''$. Thus, from the definition of the difference operator, $Q(t) = t \wedge P' \wedge \neg(t \wedge P'')$, and therefore $Q(t) = t \wedge P' \wedge \neg P'' = t \wedge P$, where $P = P' \wedge \neg P''$.
  - $Q(t) = f'(t) \bowtie f''(t)$.
    By inductive hypothesis $f'(t) = t \wedge P'$, $f''(t) = t \wedge P''$. Thus, from the definition of the natural join operator, $Q(t) = t \wedge P' \wedge P''$, and therefore $Q(t) = t \wedge P$, where $P = P' \wedge P''$. □

The previous result does not hold if the query $Q(t)$ contains a projection operator. Indeed, in this case, the query specified in the update operator may generate relational tuples that are not contained in $ext(t)$ (consider for example, the query $Q(t) = \Pi_{[X]}(t) \bowtie \Pi_{[Y]}(t)$ where $t \equiv (X = Y \wedge 1 < Y < 10)$). Thus, in general, a constraint P such that $Q(t) = \sigma_P(t)$ cannot be found. A similar consideration holds if $Q(t)$ contains the renaming operator.

Some examples of queries that can be used inside the set update operator in order to modify spatial data are shown in Table 4.3. For each query, together with the algebraic expression, an equivalent expression in ECAL($\mathcal{F}$) is also proposed. Notice that, since the translation and the rotation of a spatial object of $\mathcal{E}^2$ can be expressed in EGRA($\Phi, \mathcal{F}$), all the movements of a spatial object in $\mathcal{E}^2$ can be described in this language.

Some relationships exist between the proposed set update operator and the previously defined tuple operators, as stated by the following proposition. The proof trivially follows from the definition of the operators.

Proposition 4.2 *Given a generalized relation identifier R, a boolean condition C, and a generalized tuple P, the tuple insert and the tuple delete operators can be expressed, respectively, as follows:*

$$Del^t(r, C, P) = Upd^s(r, C, \sigma_{\neg P}(t)) \quad Ins^t(r, C, P) = Upd^s(r, C, t \cup^t \sigma_P(t \cup^t \neg t)).^3$$

$\square$

From the previous results it follows that the proposed set update operator is sufficient to model tuple insert and tuple delete operators, that therefore represent simpler syntactic forms to express data modifications.

4.4 Concluding remarks

In this chapter we have introduced an update language for constraint databases, based on the nested semantics. This language, together with the query languages introduced in Section 3, completes the definition of a data manipulation language for relational constraint databases based on a nested semantics.

As we have already remarked, in this chapter we have assumed that queries and conditions are expressed using EGRA. However, due to the equivalence between EGRA($\Phi, \mathcal{F}$) and ECAL($\Phi, \mathcal{F}$), they can also be expressed by using ECAL($\mathcal{F}$). In this case: (i) a query is translated into the equivalent alpha (see Section 3.6); (ii) a condition is translated into a calculus formula (see Section 3.6).

For some examples of alphas used in the definition of the update operator, see Table 4.3.

[3] Notice, that P is generated by the query $\sigma_P(t \cup^t \neg t)$.

Chapter 5

A formal model for nested relational constraint databases

In Chapter 3 we have proposed an algebra and a calculus for relational constraint databases, based on a simple nested model. Though this model overcomes some limitations of the generalized relational model, it still has some of the problems in supporting complex applications that standard relational database systems have. Indeed, in general, typical data modeled in constraint databases, such as spatial and temporal data, are not flat, as the relational model requires, but composite. On the other hand, neither the nested or the object-oriented models are suitable to model such types of data, since they are not able to represent infinite information. An integration of both paradigms, nested relations and constraint relations, is therefore needed to overcome the limitations of both.

As we have seen in Chapter 2, several approaches have been proposed to model complex data in constraint databases. Most of them model sets up to a given height of nesting [13, 122]. Thus, they do not allow the arbitrary combination of set and tuple constructors. Others do not have this restriction but are defined only for specific theories. This is the case of C-CALC [65]. For others, as $\mathcal{L}yri\mathcal{C}$ [28], the definition of a formal basis, supporting the definition and the analysis of relevant language properties, has been left to future work.

The aim of this chapter is the definition of a model and a query language for nested constraint relational databases overcoming some limitations of the previous proposals. The proposed language is obtained by extending $\mathcal{NRC}$ [148] to deal with possibly infinite relations, finitely representable by using POLY, and it is called g$\mathcal{NRC}$ (generalized $\mathcal{NRC}$).

$\mathcal{NRC}$ is similar to the well-known comprehension mechanism in functional programming and its formulation is based on structural recursion [34] and on monads [103, 145]. $\mathcal{NRC}$ has been proved equivalent to most nested relational languages presented before. The choice of this language is motivated by the fact that the formal semantics assigned to $\mathcal{NRC}$ and the structural recursion on which it is based allow us to prove several results about g$\mathcal{NRC}$ in a simple way. Moreover, even though g$\mathcal{NRC}$ has been defined for POLY, other theories can be easily modeled by the same formalism.

The formal framework on which g$\mathcal{NRC}$ is based allows us to easily prove that nested relational constraint languages have the *conservative extension property*. This means that, when input and output are restricted to a specific degree of nesting, any higher degree of nesting generated by the computation is useless. In particular, when input and output relations represent flat generalized relations, g$\mathcal{NRC}$ expressions can be mapped into FO extended with POLY. This property also holds for relational and nested relational languages [148]. Note that, even if this property may seem obvious, this is the first formal proof of its validity in the context of constraint databases.

Giving a constructive proof, we also prove that g$\mathcal{NRC}$ is effectively computable. The same proof shows that the language has NC data complexity. In summary, g$\mathcal{NRC}$ is a real nested constraint language but it does not have extra computational power compared to the usual constraint query languages. However, it allows a more natural representation of data, ensuring a low computational complexity and a high flexibility with respect to the chosen theory, thus overcoming most limitations of previous proposals.

The chapter is organized as follows. In Section 5.1, a slight modification of the nested relational model is proposed, to deal with finitely representable sets. The finitely representable nested relational calculus g$\mathcal{NRC}$ is then presented in Section 5.2. Section 5.3 proves that g$\mathcal{NRC}$ has the conservative extension property. Results about effective computability are then presented in Section 5.4, whereas Section 5.5 deals with complexity results. Finally, Section 5.6 presents some conclusions.

5.1 The generalized nested relational model

In the traditional generalized relational model, a relation can be an infinite set of tuples taking values from a given domain, as long as the set is finitely representable. We extend this paradigm to sets that can be nested to an arbitrary height. To this purpose, we choose as an example the polynomial inequality constraint theory POLY.[1] In particular, we allow such infinite sets of tuples of reals to appear at any depth in

[1]The proposed approach can be easily extended to deal with any other theory admitting variables elimination and closed under complementation.

a nested relation. However, we do not allow a nested set to have an infinite number of such infinite sets as its elements, to guarantee effective computability and low data complexity.

To be precise, the types that we want to consider are:

$$s ::= \mathbb{R} \mid s_1 \times \cdots \times s_n \mid \{s\} \mid \{_{fr}\mathbb{R} \times \cdots \times \mathbb{R}\}$$

The type $\mathbb{R}$ contains all the real numbers. The type $s_1 \times \cdots \times s_n$ contains n-ary tuples whose components have types s_1, ..., s_n respectively. The type $\{s\}$ are sets of finite cardinality whose elements are objects of type s. The type $\{_{fr}s\}$ are sets of (possibly) infinite cardinality whose elements are objects of type s, where s is a type of the form $\mathbb{R} \times \cdots \times \mathbb{R}$. We also require each set in $\{_{fr}s\}$ to be finitely representable in the sense of [64, 83, 115].

For convenience, we also introduce a 'type' $\mathbb{B}$ to stand for Booleans. However, for economy, we use the number 0 to stand for *false* and the number 1 to stand for *true*.

Example 5.1 *Consider a spatial database, representing regions, cities, and rivers. Each region is characterized by a name, its geographical extension, the geographical extension of its mountains, and the geographical extension of its flat countries. Cities and rivers are characterized by their name and their geographical extension. Regions, cities, and rivers can be represented using the proposed finitely representable nested types as follows:*

- Regions*: Regions can be represented using a finite set. Each element of the set represents a single region and is represented by a tuple. The tuple is composed of four elements: a real, representing the region identifier; a finitely representable set, representing the geographical extension of the region; a finite set, containing a finitely representable set for each group of mountains; a finite set, containing a finitely representable set for each flat country inside the region. Geographical extensions can be approximated by polygons and therefore each finitely representable set contains pairs of reals. Thus, regions can be represented by a complex object of type* $REGIONS : \{\mathbb{R} \times \{_{fr}\mathbb{R} \times \mathbb{R}\} \times \{\{_{fr}\mathbb{R} \times \mathbb{R}\}\} \times \{\{_{fr}\mathbb{R} \times \mathbb{R}\}\}\}$.

- Cities *and* Rivers*: Both cities and rivers can be represented by a finite set. Each element of the set represents a single city (a single river) and it is represented by a tuple. The tuple is composed of two elements: a real, representing the city identifier (the river identifier) and a finitely representable set, representing the geographical extension of the city (of the river). A city (a river) can be either represented as a point or as a polygon. Therefore, also in this case, each finitely representable set contains pairs of reals. Thus, cities and rivers can be* *represented by complex objects respectively of types* $CITIES : \{\mathbb{R} \times \{_{fr}\mathbb{R} \times \mathbb{R}\}\}$ *and* $RIVERS : \{\mathbb{R} \times \{_{fr}\mathbb{R} \times \mathbb{R}\}\}$. □

5.2 The generalized nested relational calculus

To express queries over our finitely representable nested relations, we extend the nested relational calculus $\mathcal{NRC}$ defined in [34, 148]. We call the extended calculus g$\mathcal{NRC}$, standing for *generalized* $\mathcal{NRC}$.

We present the language incrementally. We start from $\mathcal{NRC}$, which is equivalent to the usual nested relational algebra [2, 34]. The syntax and typing rules of $\mathcal{NRC}$ are given below.

$$\frac{}{x^s : s} \qquad \frac{}{c : \mathbb{R}}$$

$$\frac{e : s_1 \times \cdots \times s_n}{\pi_i\ e : s_i} \qquad \frac{e_1 : s_1 \quad \cdots \quad e_n : s_n}{(e_1, \ldots, e_n) : s_1 \times \cdots \times s_n}$$

$$\frac{}{\{\}^s : \{s\}} \qquad \frac{e : s}{\{e\} : \{s\}} \qquad \frac{e_1 : \{s\} \quad e_2 : \{s\}}{e_1 \cup e_2 : \{s\}} \qquad \frac{e_1 : \{t\} \quad e_2 : \{s\}}{\bigcup\{e_1 \mid x^s \in e_2\} : \{t\}}$$

$$\frac{e_1 : \mathbb{R} \quad e_2 : \mathbb{R}}{e_1 = e_2 : \mathbb{B}} \qquad \frac{e_1 : \mathbb{B} \quad e_2 : s \quad e_3 : s}{\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 : s} \qquad \frac{e : \{\mathbb{R}\}}{\mathit{empty}\ e : \mathbb{B}}$$

We often omit the type superscripts as they can be inferred. An expression e having free variables $\vec{x}$ is interpreted as a function $f(\vec{x}) = e$, which given input $\vec{O}$ produces $e[\vec{O}/\vec{x}]$ as its output. An expression e with no free variable can be regarded as a constant function $f(\vec{x}) = e$ that returns e on all input $\vec{x}$.

Let us briefly recall the semantics (see also [34]). Variables x^s are available for each type s. Every real number c is available. The operations for tuples are standard. Namely, $(e_1, \ldots, e_n)$ forms an n-tuple whose i component is e_i and $\pi_i\ e$ returns the i component of the n-tuple e.

$\{\}$ forms the empty set. $\{e\}$ forms the singleton set containing e. $e_1 \cup e_2$ unions the two sets e_1 and e_2. $\bigcup\{e_1 \mid x \in e_2\}$ maps the function $f(x) = e_1$ over all elements in e_2 and then returns their union; thus if e_2 is the set $\{o_1, \ldots, o_n\}$, the result of this operation would be $f(o_1) \cup \cdots \cup f(o_n)$. For example, $\bigcup\{\{(x, x)\} \mid x \in \{1, 2\}\}$ evaluates to $\{(1, 1), (2, 2)\}$.

The operations for Booleans are also typical, with the understanding that *true* is represented by 1 and *false* is represented by 0. $e_1 = e_2$ returns *true* if e_1 and e_2 have the same value and returns *false* otherwise. *empty* e returns *true* if e is an empty set and returns *false* otherwise. Finally, *if* e_1 *then* e_2 *else* e_3 evaluates to e_2 if e_1 is *true* and evaluates to e_3 if e_1 is *false*; it is undefined otherwise.

Now we deal with finitely representable relations and constraints. We add constructs analogous to the finite set constructs of $\mathcal{NRC}$ to manipulate finitely representable sets and constructs for arithmetic to express real polynomial constraints.[2]

$$\frac{}{\{_{fr}\}^s : \{_{fr}s\}} \qquad \frac{e : s}{\{_{fr}e\} : \{_{fr}s\}} \qquad \frac{e_1 : \{_{fr}s\} \quad e_2 : \{_{fr}s\}}{e_1 \cup_{fr} e_2 : \{_{fr}s\}}$$

$$\frac{e_1 : \{_{fr}s_1\} \quad e_2 : \{_{fr}s_2\}}{\bigcup\{_{fr}e_1 \mid x^{s_2} \in_{fr} e_2\} : \{_{fr}s_1\}}$$

$$\frac{e_1 : \mathbb{R} \quad e_2 : \mathbb{R}}{e_1 + e_2 : \mathbb{R}} \qquad \frac{e_1 : \mathbb{R} \quad e_2 : \mathbb{R}}{e_1 - e_2 : \mathbb{R}} \qquad \frac{e_1 : \mathbb{R} \quad e_2 : \mathbb{R}}{e_1 \cdot e_2 : \mathbb{R}} \qquad \frac{e_1 : \mathbb{R} \quad e_2 : \mathbb{R}}{e_1 \div e_2 : \mathbb{R}}$$

$$\frac{}{R : \{_{fr}\mathbb{R}\}} \qquad \frac{e : \{_{fr}\mathbb{R}\}}{\mathit{empty}_{fr}\ e : \mathbb{B}}$$

The semantics of these constructs is analogous to those of finite sets. $\{_{fr}\}$ forms the empty finitely representable set. $\{_{fr}e\}$ forms a singleton finitely representable set. $e_1 \cup_{fr} e_2$ produces a finitely representable set that is the union of the two finitely representable sets e_1 and e_2. $\bigcup\{_{fr}e_1 \mid x^{s_2} \in_{fr} e_2\}$ applies the function $f(x) = e_1$ to each element of e_2 and returns their union as a finitely representable set. For example, if the elements of e_2 are o_1, o_2, ..., then the result is $f(o_1) \cup_{fr} f(o_2) \cup_{fr} \cdots$.[3]

The four arithmetic operations have the usual interpretation. $\mathit{empty}_{fr}\ e$ tests if the finitely representable set e of reals is empty. Finally, the symbol R denotes the infinite (but finitely representable) set of all real numbers. It is the presence of this symbol R that allows to express unbound quantification. For example, given a polynomial $f(x)$, we can express its set of roots easily: $\bigcup\{_{fr}\ \mathit{if}\ f(x) = 0\ \mathit{then}\ \{_{fr}x\}\ \mathit{else}\ \{_{fr}\} \mid x \in_{fr} R\}$. Similarly, we can express the usual linear order on the reals, because the formula $\exists z.(z \neq 0) \wedge (y - x = z^2)$, which holds iff $x < y$, is expressible as $\mathit{not}(\mathit{empty}_{fr}(\bigcup\{_{fr}\ \mathit{if}\ \mathit{not}(z = 0)\ \mathit{then}\ \mathit{if}\ y - x = z \cdot z\ \mathit{then}\ \{_{fr}\ z\}\ \mathit{else}\ \{_{fr}\}\ \mathit{else}\ \{_{fr}\} \mid z \in_{fr} R\}))$, with *not* implemented in the obvious way.

The above constructs let us manipulate finite sets and finitely representable sets independently. In order for these two kinds of sets to interact, we need one more construct:

[2] Note that different sets of rules can be inserted to represent different logical theories admitting variable elimination and closed under complementation.

[3] In Section 5.4 we prove that this operation is computable, even if at a first sight this is not obvious.

$$\frac{e_1 : \{_{fr} s_1\} \quad e_2 : \{s_2\}}{\bigcup\{_{fr} e_1 \mid x^{s_2} \in e_2\} : \{_{fr} s_1\}}$$

This construct let us convert a finite set of real tuples into a finitely representable one. The semantics of $\bigcup\{_{fr} e_1 \mid x \in e_2\}$ is to apply the function $f(x) = e_1$ to each element of e_2 and then returns their union as a finitely representable set. That is, if e_2 is the set $\{o_1, \ldots, o_n\}$, then it produces the finitely representable set $f(o_1) \cup_{fr} \cdots \cup_{fr} f(o_n)$. For example, the conversion of a finite set e of real tuples to a finitely representable one can be expressed as $\bigcup\{_{fr}\{_{fr} x\} \mid x \in e\}$.

The above constructs constitute the language g$\mathcal{NRC}$. Before we study g$\mathcal{NRC}$ properties, let us briefly introduce a nice shorthand, based on the comprehension notation [33, 145], for writing g$\mathcal{NRC}$ queries. Recall from [33, 34, 148] that the comprehension $\{e \mid A_1, \ldots, A_n\}$, where each A_i either has the form $x_i \in e_i$ or is an expression e_i of type $\mathbb{B}$, has a direct correspondent in $\mathcal{NRC}$ that is given by recursively applying the following equations:

- $\{e \mid x_i \in e_i, \ldots\} = \bigcup\{\{e \mid \ldots\} \mid x_i \in e_i\}$
- $\{e \mid e_i, \ldots\} = \mathit{if}\ e_i\ \mathit{then}\ \{e \mid \ldots\}\ \mathit{else}\ \{\}$

The comprehension notation is very user-friendly. For example, it allows us to write $\{(x, y) \mid x \in e_1,\ y \in e_2\}$ for the Cartesian product of e_1 and e_2 instead of the clumsier $\bigcup\{\bigcup\{\{(x, y)\} \mid y \in e_2\} \mid x \in e_1\}$.

The comprehension notation can be extended naturally to all g$\mathcal{NRC}$ expressions. We can interpret the comprehension $\{_{fr} e \mid A_1, \ldots, A_n\}$, where each A_i either has the form $x_i \in e_i$ or has the form $x_i \in_{fr} e_i$ or is an expression e_i of type B, as an expression of g$\mathcal{NRC}$ by recursively applying the following equations:

- $\{_{fr} e \mid x_i \in e_i, \ldots\} = \bigcup\{_{fr}\{_{fr} e \mid \ldots\} \mid x_i \in e_i\}$
- $\{_{fr} e \mid x_i \in_{fr} e_i, \ldots\} = \bigcup\{_{fr}\{_{fr} e \mid \ldots\} \mid x_i \in_{fr} e_i\}$
- $\{_{fr} e \mid e_i, \ldots\} = \mathit{if}\ e_i\ \mathit{then}\ \{_{fr} e \mid \ldots\}\ \mathit{else}\ \{_{fr}\}$

For example, the query to find the roots of $f(x)$ becomes $\{_{fr} x \mid x \in_{fr} R,\ f(x) = 0\}$. Similarly, the query to test if $x < y$ becomes

$\mathit{not}(\mathit{empty}_{fr}(\{_{fr} z \mid z \in_{fr} R,\ \mathit{not}(z = 0),\ y - x = z \cdot z\}))$.

In addition to comprehension, we also find it convenient to use some pattern matching, which can be eliminated in a straightforward manner. For example, we write $\{(x, z) \mid (x, y) \in e_1,\ (y', z) \in e_2,\ y = y'\}$ for relational composition instead of the more formal $\{(\pi_1\ xy, \pi_2\ yz) \mid xy \in e_1,\ yz \in e_2,\ \pi_2\ xy = \pi_1\ yz\}$.

We should also remark that while g$\mathcal{NRC}$ provides only equality test on $\mathbb{R}$ and emptiness tests on $\{\mathbb{R}\}$ and $\{_{fr}\mathbb{R}\}$, these operations can be lifted to every type s using g$\mathcal{NRC}$ as the ambient language; see [148]. Similarly, commonly used operations such as set membership, set subset tests, set difference, and set intersection are expressible at all types in g$\mathcal{NRC}$. Thus, under the proposed framework, generalized relations on POLY, as defined in [83], are easily defined.

Example 5.2 *Consider the types introduced in Example 5.1. Suppose we want to represent a city, identified by number* 10*, whose extension, in some reference space, is approximated by the constraint* $50 \leq X \leq 60 \wedge 20 \leq Y \leq 25$.[4] *Using the comprehension syntax, this city is represented in g$\mathcal{NRC}$ as* $(10, \{_{fr}(x, y) \mid x \in_{fr} R, y \in_{fr} R, x \geq 50, x \leq 60, y \geq 20, y \leq 25\})$*, where* $x \leq y$ *is defined as before: if* $not(empty_{fr}(\{_{fr}z \mid z \in_{fr} R,\ not(z = 0),\ y - x = z \cdot z\}))$ *then* 1 *else* $x = y$.

Now consider three expressions `regions`*:* $REGIONS$*,* `cities`*:* $CITIES$*,* `rivers`*:* $RIVERS$*, respectively representing a set of regions, a set of cities, and a set of rivers, and a further expression* `reg`*:* $\mathbb{R}$*, representing a region identifier. Using the comprehension notation and pattern matching, g$\mathcal{NRC}$ can be used to formulate several interesting queries:*

- *"Find all rivers flowing in region* `reg`*". This query can be expressed in g$\mathcal{NRC}$ as follows:*

  $\{n_{riv} \mid (\texttt{reg}, e_{reg}, m_{reg}, f_{reg}) \in \texttt{regions}, (n_{riv}, e_{riv}) \in \texttt{rivers}, not(e_{reg} \cap e_{riv} = \emptyset)\}$.

- *"Find all cities whose extension contains some mountains". This query can be expressed in g$\mathcal{NRC}$ as follows:*

  $\{n_{city} \mid (n_{reg}, e_{reg}, M_{reg}, f_{reg}) \in \texttt{regions}, m_{reg} \in M_{reg}, (n_{city}, e_{city}) \in \texttt{cities}, not(e_{city} \cap m_{reg} = \emptyset)\}$.

- *"Find all rivers flowing in at least two different regions". This query can be expressed in g$\mathcal{NRC}$ as follows:*

  $\{n_{riv} \mid (n^1_{reg}, e^1_{reg}, m^1_{reg}, f^1_{reg}) \in \texttt{regions}, (n^2_{reg}, e^2_{reg}, m^2_{reg}, f^2_{reg}) \in \texttt{regions}, (n_{riv}, e_{riv}) \in \texttt{rivers}, not(e^1_{reg} \cap e_{riv} = \emptyset), not(e^2_{reg} \cap e_{riv} = \emptyset), not(n^1_{reg} = n^2_{reg})\}$. $\square$

[4]We use uppercase letters to denote variables belonging to the relation schema and lowercase letters to denote variables inside calculus expressions.

5.3 Conservative extension property

The conservative extension property basically says that the expressive power of a query language is independent of the height of set nesting in the intermediate data produced during the evaluation of a query. In the following, we give a precise definition and then prove that g$\mathcal{NRC}$ possesses it.

Given a type s, the height of s is defined as the depth of nesting of set brackets $\{\cdot\}$ and $\{_{fr}\cdot\}$ in s. Given an expression e of g$\mathcal{NRC}$, the height of e is defined as the maximum height of all the types that appear in e's typing derivation. For example, $\{(x, y) \mid x \in e_1,\ y \in e_2\}$ has height 1 if both e_1 and e_2 have height 1. On the other hand, $\{(x, \{_{fr} z \mid z \in_{fr} R, z < x\}) \mid x \in e\}$ have height 2 if e has height 1.

Definition 5.1 (Conservative extension property) *A language $\mathcal{L}$ is said to have the conservative extension property if every function $f : s_1 \to s_2$ that is expressible in $\mathcal{L}$ can be expressed using an expression of height no more than the maximum of the heights of s_1 and s_2.* □

We now prove that g$\mathcal{NRC}$ has the conservative extension property, just like $\mathcal{NRC}$ [148]. As in [148], a set of strongly normalizing rewriting rules that reduces set height is given. Then we show that the induced normal forms have height no more than that of their free variables (i.e., their input variables).

Table 5.1 shows the rewriting rules that we want to use. Those for $\mathcal{NRC}$ are taken from [148]. As usual, we assume that bound variables are renamed to avoid capture and that $e_1[e_2/x]$ denotes the expression obtained by replacing all free occurrences of x in e_1 by e_2.

It is readily verified that the proposed rewriting rules are sound. That is, expressions obtained from e_1 by rewriting are semantically equivalent to e_1. Furthermore, using a straightforward adaptation of the termination measure given in [148], we can prove the following result.

Proposition 5.1 *If $e_1 \rightsquigarrow e_2$, then $e_1 = e_2$.*[5] *Moreover, the rewriting system presented in Table 5.1 is guaranteed to stop no matter in what order these rules are applied (it is strongly normalizing).* □

The following result follows from the application of a simple induction on the structure of expressions.

Proposition 5.2 *Let $e : s$ be an expression of g$\mathcal{NRC}$ having free variables $x_1 : s_1$, ..., $x_n : s_n$ such that e is a normal form with respect to the above rewriting system. Then the height of e is at most the maximum of the heights of s, s_1, ..., s_n.* □

[5]The symbol = denotes semantic equivalence.

$\pi_i(e_1, \ldots, e_n) \leadsto e_i$
$\mathit{if}\ \mathit{true}\ \mathit{then}\ e_1\ \mathit{else}\ e_2\ \leadsto e_1$
$\mathit{if}\ \mathit{false}\ \mathit{then}\ e_1\ \mathit{else}\ e_2\ \leadsto e_2$
$\{\} \cup e \leadsto e$
$e \cup \{\} \leadsto e$
$\mathit{empty}(e_1 \cup \cdots \cup e_n) \leadsto \mathit{false}$, if some e_i has the form $\{e\}$
$\mathit{empty}(e_1 \cup \cdots \cup e_n) \leadsto \mathit{true}$, if every e_i has the form $\{\}$
$\mathit{empty}_{fr}(e_1 \cup_{fr} \cdots \cup_{fr} e_n) \leadsto \mathit{false}$, if some e_i has the form $\{_{fr} e\}$
$\mathit{empty}_{fr}(e_1 \cup_{fr} \cdots \cup_{fr} e_n) \leadsto \mathit{true}$, if every e_i has the form $\{_{fr}\}$
$\bigcup\{e \mid x \in \{\}\} \leadsto \{\}$
$\bigcup\{e_1 \mid x \in \{e_2\}\} \leadsto e_1[e_2/x]$
$\bigcup\{e_1 \mid x \in e_2 \cup e_3\} \leadsto \bigcup\{e_1 \mid x \in e_2\} \ \cup\ \bigcup\{e_1 \mid x \in e_3\}$
$\bigcup\{e_1 \mid x \in \bigcup\{e_2 \mid y \in e_3\} \leadsto \bigcup\{\bigcup\{e_1 \mid x \in e_2\} \mid y \in e_3\}$
$\bigcup\{e_1 \mid x \in \mathit{if}\ e_2\ \mathit{then}\ e_3\ \mathit{else}\ e_4\} \leadsto \mathit{if}\ e_2\ \mathit{then}\quad \bigcup\{e_1 \mid x \in e_3\}\ \mathit{else}$ $\bigcup\{e_1 \mid x \in e_4\}$
$\bigcup\{_{fr} e \mid x \in_{fr} \{_{fr}\}\} \leadsto \{_{fr}\}$
$\bigcup\{_{fr} e_1 \mid x \in_{fr} \{_{fr} e_2\}\} \leadsto e_1[e_2/x]$
$\bigcup\{_{fr} e_1 \mid x \in_{fr} e_2 \cup_{fr} e_3\} \leadsto \bigcup\{_{fr} e_1 \mid x \in_{fr} e_2\} \cup_{fr} \bigcup\{_{fr} e_1 \mid x \in_{fr} e_3\}$
$\bigcup\{_{fr} e_1 \mid x \in_{fr} \bigcup\{_{fr} e_2 \mid y \in_{fr} e_3\}\} \leadsto \bigcup\{_{fr} \bigcup\{_{fr} e_1 \mid x \in_{fr} e_2\} \mid y \in_{fr} e_3\}$
$\bigcup\{_{fr} e_1 \mid x \in_{fr} \mathit{if}\ e_2\ \mathit{then}\ e_3\ \mathit{else}\ e_4\ \} \ \leadsto \mathit{if}\ e_2\ \mathit{then}\quad \bigcup\{_{fr} e_1 \mid x \in_{fr} e_3\}\ \mathit{else}$ $\bigcup\{_{fr} e_1 \mid x \in_{fr} e_4\}$
$\bigcup\{_{fr} e \mid x \in \{\}\} \leadsto \{\}$
$\bigcup\{_{fr} e_1 \mid x \in \{e_2\}\} \leadsto e_1[e_2/x]$
$\bigcup\{_{fr} e_1 \mid x \in e_1 \cup e_2\} \leadsto \bigcup\{_{fr} e_1 \mid x \in e_2\} \cup \bigcup\{_{fr} e_1 \mid x \in e_3\}$
$\bigcup\{_{fr} e_1 \mid x \in \bigcup\{e_2 \mid y \in e_3\}\} \leadsto \bigcup\{_{fr} \bigcup\{_{fr} e_1 \mid x \in e_2\} \mid y \in e_3\}$
$\bigcup\{_{fr} e_1 \mid x \in_{fr} \bigcup\{_{fr} e_2 \mid y \in e_3\}\} \leadsto \bigcup\{_{fr} \bigcup\{_{fr} e_1 \mid x \in_{fr} e_2\} \mid y \in e_3\}$
$\bigcup\{_{fr}\ e_1 \mid x \in \mathit{if}\ e_2\ \mathit{then}\ e_3\ \mathit{else}\ e_4\} \leadsto \mathit{if}\ e_1\ \mathit{then}\quad \bigcup\{_{fr} e_1 \mid x \in e_3\}\ \mathit{else}$ $\bigcup\{_{fr} e_1 \mid x \in e_4\}$

Table 5.1: Rewriting rules.

Combining Propositions 5.1 and 5.2, we conclude the following.

Theorem 5.1 *g$\mathcal{NRC}$ has the conservative extension property.* $\square$

Note that the previous result implies that when the set height of a g$\mathcal{NRC}$ expression e is higher than the set height of input and output expressions, then e can be optimized, reducing the height of intermediate results, gaining in space and time overhead.

Paredaens and Van Gucht gave a translation for mapping nested relational algebra expressions having flat relations as input to an equivalent FO expression with bound quantification [117]. This translation can be easily adapted to provide a translation for mapping g$\mathcal{NRC}$ expressions of height 1 to FO extended with POLY. The next result follows from this and Theorem 5.1.

Corollary 5.1 *If $f : s_1 \rightarrow s_2$ is a function expressible in g$\mathcal{NRC}$ and s_1 and s_2 have height* 1*, then f is expressible in* FO *with* POLY*.* $\square$

Thus, all functions $f : s_1 \rightarrow s_2$ in g$\mathcal{NRC}$, with s_1 and s_2 of height 1, are effectively computable by compiling into constraint query languages such as those proposed in [64, 83, 115].

As a consequence, we can make use of well-known results [15, 66, etc.] on constraint query languages to analyze the expressiveness of g$\mathcal{NRC}$ with respect to such functions. It is therefore simple to prove the following result.

Corollary 5.2 *g$\mathcal{NRC}$ cannot express parity test, connectivity test, transitive closure, etc.* $\square$

We can also use the above "compilation procedure" to study the expressive power of g$\mathcal{NRC}$ on functions whose types have heights exceeding 1. We borrow an example from [97] for illustration. A set of sets $O = \{O_1, \ldots, O_n\} : \{\{\mathbb{R}\}\}$ is said to have a family of distinct representatives iff it is possible to pick an element x_i from each O_i such that $x_i \neq x_j$ whenever $i \neq j$. It is known from [97] that $\mathcal{NRC}$ cannot test if a set has distinct representatives. We now show that it cannot be expressed in g$\mathcal{NRC}$ either.

Corollary 5.3 *g$\mathcal{NRC}$ cannot test if a set of sets has distinct representatives.*

Proof: By Corollary 5.2, g$\mathcal{NRC}$ cannot express parity test. It follows that it cannot test if a chain has an even number of nodes. Let a set $X_m = \{(x_1, x_2), \ldots, (x_{m-1}, x_m)\}$ be given, where $m > 2$. Then we can construct in g$\mathcal{NRC}$ the set

$S_m = \{\{x_1\}, \{x_m\}, \{x_1, x_3\}, \{x_2, x_4\}, \ldots, \{x_{m-2}, x_m\}\}$.
According to [97], S_m has distinct representatives iff m is even. It follows that g$\mathcal{NRC}$ cannot test for distinct representatives. □

5.4 Effective computability

Recall that expressions in g$\mathcal{NRC}$ can iterate over infinite sets. An important question that arises is whether every function expressible in g$\mathcal{NRC}$ is computable. In the previous section, we saw that if a function in g$\mathcal{NRC}$ has input and output of height 1, then it is computable. In this section, we lift this result to functions of all heights.

Our strategy is as follows. We find a total computable function $p_s : s- > s'$ to encode nested finitely representable sets into flat finitely representable sets. We also find a partial computable decoding function $q_s : s' \to s$ so that $q_s \circ p_s = id$. Finally, we find a translation $(\cdot)'$ that maps $f : s_1 \to s_2$ in g$\mathcal{NRC}$ to $(f)' : s_1' \to s_2'$ in g$\mathcal{NRC}$ such that $q_{s_2} \circ (f)' \circ p_{s_1} = f$. Note that $(f)'$ has height 1 and is thus computable.

Before we define p and q, let us first define s', the type to which s is encoded. Notice that s' always has the form $\{_{fr}\mathbb{R} \times \cdots \times \mathbb{R}\}$.

- $\mathbb{R}' = \{_{fr}\mathbb{R}\}$
- $(s_1 \times \cdots \times s_n)' = \{_{fr}t_1 \times \cdots \times t_n\}$, where $s_i' = \{_{fr}t_i\}$.
- $\{_{fr}s\}' = \{_{fr}\mathbb{R} \times s\}$
- $\{s\}' = \{_{fr}\mathbb{R} \times \mathbb{R} \times t\}$, where $s' = \{_{fr}t\}$

The encoding function $p_s : s \to s'$ is defined by induction on s. In what follows, $\vec{0}$ stands for a tuple of zeros $(0, \ldots, 0)$ having the appropriate arity. A finitely representable set is coded by tagging each element by 1 if the set is nonempty and is coded by a tuple of zeros if it is empty. A finite set is coded by tagging each element by 1 and by a unique identifier if the set is nonempty and is coded by a tuple of zeros if it is empty. More precisely,

- $p_{\mathbb{R}}(o) = \{_{fr}o\}$
- $p_{s_1 \times \cdots \times s_n}((o_1, \ldots, o_n)) = \{_{fr}(x_1, \ldots, x_n) \mid x_1 \in_{fr} p_{s_1}(o_1), \ldots, x_n \in_{fr} p_{s_n}(o_n)\}$
- $p_{\{_{fr}s\}}(O) = \{_{fr}(0, \vec{0})\}$, if O is empty. Otherwise, $p_{\{_{fr}s\}}(O) = \{_{fr}(1, x) \mid x \in_{fr} O\}$.

- $p_{\{s\}}(O) = \{_{fr}(0, 0, \vec{0})\}$, if O is empty. Otherwise, $p_{\{s\}}(O) = O_1 \cup_{fr} \cdots \cup_{fr} O_n$, if $O = \{o_1, \ldots, o_n\}$ and $O_i = \{_{fr}(1, i, x) \mid x \in_{fr} p_s(o_i)\}$. Note that we allow the i's above to be any numbers, so long as they are distinct positive integers.

We use $\bigcup\{e_1 \mid x \in_{fr} e_2\}$ to stand for the application of $f(x) = e_1$ to each element of e_2, provided the finitely representable set e_2 has a finite number of elements, and then return the finite union of the results. Then the comprehension notation $\{e \mid A_1, \ldots, A_n\}$ is extended to allow A_i to be of the form $x_i \in_{fr} e_i$ and the translation equations are augmented to include the equation: $\{e \mid x_i \in_{fr} e_i, \ldots\} = \bigcup\{\{e \mid \ldots\} \mid x_i \in_{fr} e_i\}$.

The decoding function $q_s : s' \to s$, which strips tags and identifiers introduced by p_s, can be defined as follows:

- $q_{\mathbb{R}}(O) = o$, if $O = \{_{fr} o\}$.
- $q_{s_1 \times \cdots \times s_n}(O) = (o_1, \ldots, o_n)$, if $o_i = q_{s_i}(\{_{fr} x_i \mid (x_1, \ldots, x_n) \in_{fr} O\})$.
- $q_{\{_{fr} s\}}(O) = \{_{fr} x \mid (1, x) \in_{fr} O\}$.
- $q_{\{s\}}(O) = \{q_s(\{_{fr} y \mid (1, j, y) \in_{fr} O,\ i = j\}) \mid (1, i, x) \in_{fr} O\}$.

It is clear that p_s and q_s are both computable, even though they cannot be expressed in g$\mathcal{NRC}$. Moreover, using the fact that $p_s(O)$ is never empty, by induction on the structure of s we can show that q_s is inverse of p_s.

Proposition 5.3 $q_s \circ p_s = id$. □

Note that p_s is not deterministic. Let $O_1 : s'$ and $O_2 : s'$. Then we say $O_1 \sim O_2$ if $q_s(O_1) = q_s(O_2)$. That is, O_1 and O_2 are equivalent encodings of an object $O : s$. It is clear that whenever $O_1 \sim O_1'$, ..., and $O_n \sim O_n'$, then $\{_{fr}(x_1, \ldots, x_n) \mid x_1 \in_{fr} O_1, \ldots, x_n \in_{fr} O_n\} \sim \{_{fr}(x_1, \ldots, x_n) \mid x_1 \in_{fr} O_1', \ldots, x_n \in_{fr} O_n'\}$. It is also obvious that whenever $O \sim O'$, then $\{_{fr} x_i \mid (x_1, \ldots, x_n) \in_{fr} O\} \sim \{_{fr} x_i \mid (x_1, \ldots, x_n) \in_{fr} O'\}$. We can now state the following key proposition.

Proposition 5.4 *For every function $f : s_1 \to s_2$ in g$\mathcal{NRC}$, there is a function $(f)' : s_1' \to s_2'$ such that*

$$\begin{tikzcd}
s_1 \arrow[r, "id"] \arrow[d, "p_{s_1}"'] & s_1 \arrow[r, "f"] & s_2 \arrow[r, "id"] & s_2 \arrow[d, "p_{s_2}"] \\
s_1' \arrow[r, "\sim"'] & s_1' \arrow[r, "(f)'"'] \arrow[u, "q_{s_1}"] & s_2' \arrow[r, "\sim"'] \arrow[u, "q_{s_2}"'] & s_2'
\end{tikzcd}$$

Proof: *(Sketch)* Left and right squares commute by definitions of p_s, q_s, and $\sim$. It is then possible to construct $(f)'$ by induction on the structure of the g$\mathcal{NRC}$ expression that defines f such that the middle square and thus the entire diagram commutes. □

Now let $f : s_1 \to s_2$ be a function in g$\mathcal{NRC}$, where s_1 and s_2 have arbitrary nesting depths. Proposition 5.4 implies that there is a function $(f)' : s_1' \to s_2'$ in g$\mathcal{NRC}$ such that $q_{s_2} \circ (f)' \circ p_{s_1} = f$. Since s_1' and s_2' are both of height 1, by Theorem 5.1, we can assume that $(f)'$ has height 1. Then by Corollary 5.1, we conclude that $(f)'$ is effectively computable. Since q_s and p_s are also computable, we have the very desirable result below.

Theorem 5.2 *All functions expressible in g$\mathcal{NRC}$ are effectively computable.* □

The "compilation procedure" above essentially shows that the whole of g$\mathcal{NRC}$ can be embedded in FO extended with POLY, modulo the encodings p_s and q_s (thus, g$\mathcal{NRC}$ is closed). We should remark that the converse is also true. For example, a formula $\exists x.\Phi(x)$ can be expressed in g$\mathcal{NRC}$ as $not(empty_{fr}\{_{fr}1 \mid x \in_{fr} R,\ \Phi(x)\})$. So g$\mathcal{NRC}$ does not gain us extra expressive or computational power, compared to the usual constraint query languages. However, it gives a more natural data model and a more convenient query language, since it is no longer necessary to model our spatial databases as a set of flat tables.

5.5 Data complexity

A constraint query Q has data complexity in the complexity class C if there is a Turing machine that, given an input generalized database d, produces some generalized relation representing the output $Q(d)$ and uses time in class C, assuming some standard encoding of generalized relations [83].

Results about data complexity of g$\mathcal{NRC}$ can be obtained from results presented in Section 5.4 and from [83]. Consider the diagram introduced in Proposition 5.4. As f' is expressed in FO extended with POLY, it follows from [83] that its data complexity is in NC. Moreover, it is simple to show that encoding and decoding functions p_s and q_s are also in NC. The following result follows from these considerations.

Proposition 5.5 *g$\mathcal{NRC}$ has data complexity in NC.* □

From the previous considerations, it follows that g$\mathcal{NRC}$ overcomes some limitations of the previous proposals to model complex objects in constraint databases. Indeed, no maximum degree of nesting is assumed and different theories can be used to finitely represent relations, ensuring at the same time a low data complexity. Moreover, the formal semantics on which it is based allows us to easily analyze several interesting properties (as conservative extensions) of nested constraint relational languages.

5.6 Concluding remarks

We have proposed a formal model and a query language for constraint nested relations, overcoming some limitations of the previous approaches. The proposed language, g$\mathcal{NRC}$, has been obtained by extending $\mathcal{NRC}$ [148]. It is characterized by a clear formal foundation, a low data complexity, and the ability to model any degree of nesting. Moreover, even if the language has been defined for a specific theory, the framework can be easily extended to deal with different theories, closed under complementation and admitting variable elimination. The idea is to replace rules introduced to specify arithmetic with rules describing properties of the chosen theory.

Part II

Optimization issues in constraint databases

Chapter 6

Optimization techniques for constraint databases

In order to make constraint databases a practical technology, efficient optimization techniques must be developed. In traditional databases, at least two different approaches are adopted in order to achieve good performance:

- *Indexing.* In this case, specific data structures are used to more efficiently support retrieval and update operations of items stored in the database. Typical relational index structures are B-trees and their variant B^+-trees [11, 49].

- *Query optimization.* Different execution strategies can in general be applied to execute the same query. However, the cost of applying such strategies may be different. The aim of query optimization is to determine the execution plan with the optimal cost. In general, two different, but complementary, approaches can be used. First, the expression representing the query to be executed is rewritten as another expression, equivalent to the original one but more efficient to execute. This step, called *algebraic optimization*, is mainly based on the application of specific heuristics. After that, the available cost parameters and information about the available index data structures are used to determine the most efficient execution plan (*cost-based optimization*).

The aim of this chapter is to illustrate which indexing and query optimization approaches have been proposed for constraint databases, pointing out the difference with respect to traditional and spatial databases. The chapter is organized as follows. In Section 6.1, we survey indexing techniques for constraint databases whereas Section 6.2 surveys the few approaches that have been proposed to perform query optimization in constraint databases. Some conclusions are then pointed out in Section 6.3.

6.1 Indexing

Data structures for querying and updating constraint databases must be developed, with time and space complexities comparable to those of data structures for relational databases. Complexity of the various operations is usually expressed in terms of *input-output (I/O) operations*. An I/O operation is the operation of reading or writing one block of data from or to a disk. Thus, space complexity corresponds to the number of disk blocks used to store data structures; time complexity of query and update operations corresponds to the number of blocks that have to be read or written in order to execute the query or the update operation.

Typical parameters, used to compute complexity bounds, are:

- B, representing the number of items (generalized tuples) that can be stored in one page;
- n, representing the number of pages to store N generalized tuples (thus, $n = N/B$);
- t, representing the number of pages to store T generalized tuples, representing the result of a query evaluation (thus, $t = T/B$).

Complexity can be analyzed with respect to either the worst-case or the average case.[1] Efficient data structures are usually required to process queries in $O(\log_B n + t)$ I/O operations, perform insertions and deletions in $O(\log_B n)$ I/O operations (this is the case of B-trees and B$^+$-trees), and use $O(n)$ blocks of secondary storage. All complexities are worst-case.

In the following we say that a data structure has *optimal* complexity bounds if its space complexity is $O(n)$, its query complexity is $O(\log_B n + t)$, and its update complexity is $O(\log_B n)$.

At least two constraint language features should be supported by index structures:

- *ALL selection.* It retrieves all generalized tuples contained in a specified generalized relation whose extension is contained in the extension of a given generalized tuple, specified in the query (called *query generalized tuple*).

  If the extension of a generalized tuple t is contained in the extension of a query generalized tuple q, we denote this fact by $All(q, t)$. Given a generalized relation r and a query generalized tuple q, we denote by $ALL(q, r)$ the set $\{t | t \in r, All(q, t)\}$.

  An example of ALL selection is represented by the EGRA set operator $\sigma^s_{(P,t,\supseteq)}$.

[1]For the basic notions about complexity functions, see [8].

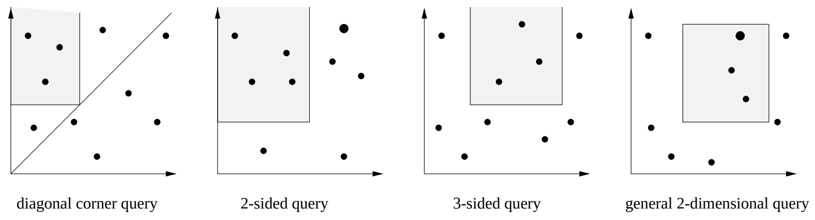


Figure 6.1: Various 2-dimensional range queries.

- *EXIST selection.* It retrieves all generalized tuples contained in a specified generalized relation whose extension has a non-empty intersection with the extension of a query generalized tuple.

  If the extensions of t and q have a non-empty intersection, we denote this fact by $Exist(q,t)$. Given a generalized relation r and a query generalized tuple q, we denote by $EXIST(q,r)$ the set $\{t|t \in r, Exist(q,t)\}$.

  Since $ALL(q,r) \subseteq EXIST(q,r)$, it is more convenient to define the query $EXIST_e(q,r) = EXIST(q,r) \setminus ALL(q,r)$.

  Therefore $EXIST_e(q,r) \cap ALL(q,r) = \emptyset$. In a similar way, we denote by $Exist_e(q,t)$ the fact $Exist(q,t) \wedge \neg All(q,t)$.

  Examples of EXIST selections are represented by the EGRA set operator $\sigma^s_{(P,t,\bowtie\neq\emptyset)}$ and the EGRA tuple operator σ_P.

Due to the analogies between constraint and spatial databases, efficient indexing techniques developed for spatial databases can often be applied to (linear) constraint databases.

For spatial problems, data structures with good worst-case complexity have been proposed only for some specific problems, in general for dealing with 1- or 2-dimensional spatial objects. In particular, several data structures, characterized by an I/O complexity for search and update operations comparable to the internal memory results, have been proposed for the so-called *point databases*, storing a set of (multi-dimensional) points [76, 84, 120, 136], and for segment databases, storing a set of 2-dimensional segments [7, 84, 119]. In point databases, the most intensively investigated problem is the 2-dimensional range searching for which several efficient algorithms have been proposed (see Figure 6.1) [76, 84, 120, 136].

Nevertheless, several data structures proposed for managing spatial data behave quite well in average. Examples of such data structures are grid files [107], various quad-trees [127], z-orders [110], hB-trees [98], cell-trees [68], and various R-trees [71, 130]. In general, these techniques are applied after objects are approximated in some way. A typical approximation is the one that replaces each object by its minimum bounding box (MBB). In 2-dimensional space, the MBB of a given object is the smallest rectangle that encloses the object and whose edges are parallel to the standard coordinate axes. The previous definition can be generalized to higher dimensions in a straightforward manner. When approximations are used, the evaluation of a query consists of two steps, *filtering* and *refinement*. In the filtering step, an index is used to retrieve only relevant objects, with respect to a certain query. To this purpose, the approximated objects are used instead of the objects themselves. During the refinement step, the objects retrieved by the filtering step are directly checked, to determine the exact result.

Similarly to the spatial case, in the context of constraint databases two different classes of techniques have been proposed, the first consisting of techniques with good worst-case complexity, and the second consisting of techniques with good average bounds. Techniques belonging to the first class apply to (linear) generalized tuples representing 1-dimensional spatial objects and mainly optimize EXIST selection. Techniques belonging to the second class allow indexing more general generalized tuples, by first applying some approximation.

In the following, both approaches will be surveyed.

6.1.1 Data structures with good worst-case complexities

In relational databases, the 1-dimensional searching problem on a relational attribute X is defined as follows:

> Find all tuples such that their X attribute satisfies the condition $a_1 \leq X \leq a_2$.

The problem of 1-dimensional searching on a relational attribute x can be reformulated in constraint databases, defining the problem of *1-dimensional searching on the generalized relational attribute* X, as follows:

> Find a generalized relation that represents all tuples of the input generalized relation such that their X attribute satisfies $a_1 \leq X \leq a_2$.

A simple initial, but inefficient, solution to the generalized 1-dimensional searching problem is to add the query range condition to each generalized tuple. In this case, the new generalized tuples represent all the points whose X attribute is between a_1 and a_2. This approach introduces a high level of redundancy in the constraint representation. Moreover, many inconsistent (with empty extension) generalized tuples can be generated.

A better solution can be defined for *convex* theories. A theory Φ is convex if the projection on X of any generalized tuple over Φ is one interval $b_1 \leq X \leq b_2$. This is true when the extension of the generalized tuple represents a convex set. Theories DENSE and POLY are examples of convex theories. The solution is based on the definition of a *generalized* 1-*dimensional index on* X as a set of intervals, where each interval is associated with a set of generalized tuples and represents the value of the search key for those tuples. Thus, each interval in the index is the projection on the attribute X of a generalized tuple. By using the above index, the detection of a generalized relation, representing all tuples from the input generalized relation such that their X attribute satisfies a given range condition $a_1 \leq X \leq a_2$, can be performed by adding the condition to only those generalized tuples whose associated interval has a non-empty intersection with $a_1 \leq X \leq a_2$ i.e., to only those tuples t satisfying $Exist(a_1 \leq X \leq a_2, t)$. Insertion (deletion) of a given generalized tuple is performed by computing its projection and inserting (deleting) the obtained interval into (from) a set of intervals.

From the previous discussion it follows that generalized 1-dimensional indexing reduces to *dynamic interval management* on secondary storage. Dynamic interval management is a well-known problem in computational geometry, with many optimal solutions in internal memory [43]. Secondary storage solutions for the same problem are, however, non-trivial, even for the static case. In the following, we survey some of the proposed solutions for secondary storage.

Reduction to stabbing queries. A first class of proposals is based on the reduction of the interval intersection problem to the *stabbing query problem* [43], and therefore is based on solutions that have been proposed for point and segment databases. Given a set of 1-dimensional intervals, to answer a stabbing query with respect to a point x, all intervals that contain x must be reported.

The main idea of the reduction is the following [84]. Intervals that intersect a query interval fall into four categories (see Figure 6.2). Categories (1) and (2) can be easily located by sorting all the intervals with respect to their left endpoint and using a B^+-tree to locate all intervals whose first endpoint lies in the query interval. Categories (3) and (4) can be located by finding all data intervals which contain the first endpoint of the query interval. This search represents a stabbing query.

By regarding an interval $[x_1, x_2]$ as the point (x_1, x_2) in the plane, a stabbing query reduces to a special case of 2-dimensional range searching. Indeed, all points (x_1, x_2), corresponding to intervals, lie above the line $X = Y$. An interval $[x_1, x_2]$ belongs to a stabbing query with respect to a point x if and only if the corresponding point (x_1, x_2) is contained in the region of space represented by the constraint $X \leq x \wedge Y \geq x$. Such 2-sided queries have their corner on line $X = Y$. For this reason, they are called

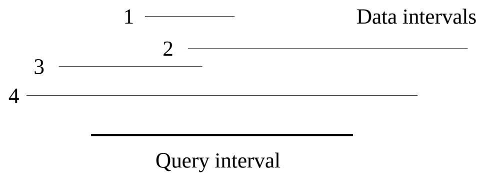


Figure 6.2: Categories of possible intersections of a query interval with a database of intervals.

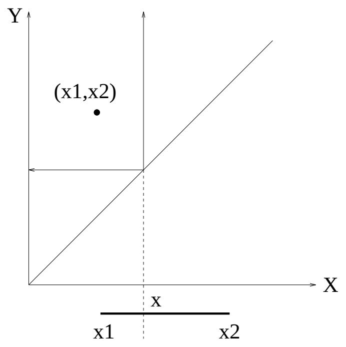


Figure 6.3: Reduction of the interval intersection problem to a diagonal-corner searching problem.

diagonal corner queries (see Figure 6.3).

The first data structure that has been proposed to solve diagonal-corner queries is the *meta-block tree* which does not support deletions (it is semi-dynamic) [84]. The meta-block tree is fairly complicated, has optimal worst-case space $O(n)$ and optimal I/O query time $O(\log_B n + t)$. Moreover, it has $O(\log_B n + (log_B^2 n)/B)$ amortized insert I/O time.

A dynamic (thus, also supporting deletions) optimal solution to the stabbing query problem [7] is based on the definition of an external memory version of the internal memory interval tree. The interval tree for internal memory is a data structure to answer stabbing queries and to store and update a set of intervals in optimal time [43]. It consists of a binary tree over the interval endpoints. Intervals are stored in secondary structures, associated with internal nodes of the binary tree. The extension of such a data structure to secondary storage entails two issues. First, the fan-out of nodes must be increased. The fan-out that has been chosen is $\sqrt{B}$ [7]. This fan-out allows the storage of all the needed information in internal nodes, increasing only of 2 the height of the tree. If interval endpoints belong to a fixed set E, the binary tree is replaced by a balanced tree, having $\sqrt{B}$ as branching factor, over the endpoints

E. Each leaf represents B consecutive points from E. Segments are associated with nodes generalizing the idea of the internal memory data structure. However, since now a node contains more endpoints, more than two secondary structures are required to store segments associated with a node. The main problem of the previous structure is that it requires the interval endpoints to belong to a fixed set. In order to remove such assumption, the *weight-balanced B-tree* has been introduced [7]. The main difference between a B-tree and a weight-balanced B-tree is that in the first case, for each internal node, the number of children is fixed; in the second case, only the *weight*, that is, the number of items stored under each node, is fixed. The weight-balanced B-tree allows the removal of the assumption on the interval endpoints, still retaining optimal worst-case bounds for stabbing queries.

Revisiting a Chazelle's algorithm. The solutions described above to solve stabbing queries in secondary storage are fairly complex and rely on reducing the interval intersection problem to special cases of the 2-dimensional range searching problem. A different and much simpler approach to solve the static (thus, not supporting insertions and deletions) generalized 1-dimensional searching problem [119] is based on an algorithm developed by Chazelle [39] for interval intersection in main memory and uses only B^+-trees, achieving optimal time and using linear space.

The proposed technique relies on the following consideration. A straightforward method to solve a stabbing query consists of identifying the set of unique endpoints of the set of input intervals. Each endpoint is associated with the set of intervals that contain such endpoint. These sets can then be indexed using a B^+-tree, taking endpoints as key values. To answer a stabbing query it is sufficient to look for the endpoint nearest to the query point, on the right, and examine the intervals associated with it, reporting those intervals that intersect the query point.

This method is able to answer stabbing queries in $O(\log_B n)$. However, it requires $O(n^2)$ space. It has been shown [119] that the space complexity can be reduced to $O(n)$ by appropriately choosing the considered endpoints. More precisely, let $e_1, e_2, ..., e_{2n}$ be the ordered lists of all endpoints. A set of *windows* $W_1, ..., W_p$ should be constructed over endpoints $w_1 = e_1, ..., w_{p+1} = e_{2n}$ such that $W_j = [w_j, w_{j+1}]$, $j = 1, ..., p$. Thus, windows represent a partition of the interval between e_1 and e_{2n} into p contiguous intervals. Each window W_j is associated with the list of intervals that intersect W_j.

Window-lists can be stored in a B^+-tree, using their starting points as key values. A stabbing query at point p can be answered by searching for the query point and retrieving the window-lists associated with the windows that it falls into. Each interval contained in such lists is then examined, reporting only those intervals intersecting the query point. Some algorithms have been proposed [119] to appropriately construct windows, in order to answer queries by applying the previous algorithm in $O(\log_B n)$, using only $O(n)$ pages.

6.1.2 Approximation-based query processing

To facilitate the definition of indexing structures for arbitrary objects in spatial databases, a *filtering-refinement* approach is often used. The same approach can be used in constraint databases to index generalized tuples with complex extensions. However, there are some differences between the use of the filtering approach in spatial databases and constraint databases. In the first case, only 2-dimensional and 3-dimensional objects are typically considered; in the second case, each generalized tuple may contain thousands of variables (for example, in Operations Research applications). Moreover, spatial objects are typically closed whereas constraint objects may be open. Therefore, filtering approaches can be applied to constraint databases if only if they scale well to large dimensions and can be applied to open objects.

The filtering approach based on MBBs, when applied to constraint databases, even if appealing, has some drawbacks. In particular, it may be ineffective if the set of objects returned by the filtering step is too large. This means that there are too many intersecting MBBs. Moreover, it does not scale well to large dimensions.

In order to improve the selectivity of filtering, an approach has been proposed, based on the notion of *minimum bounding polybox* [29]. A minimum bounding polybox is the minimum convex polyhedron that encloses the object and whose facets are normal to preselected axes. These axes are not necessarily the standard coordinate axes and, furthermore, their number is not determined by the dimension of the space. Algorithms for computing optimal axes (according to specific optimality criteria with respect to storage overhead or filtering rate) in a d-dimensional space have also been proposed [29].

If data structures for arbitrary dimensions are not available, a possible alternative is the following. Given a generalized relation of degree m, a data structure for 2-dimensional data has to be allocated for each pair of variables. Thus, m^2 data structures are needed, with a consequent increment of the space complexity. However, in this way, indexes can be used to answer queries with respect to any 2-dimensional space.

In any case, the constraint object is transformed in a simpler one, against which indexing is performed. For example, if MBB are used, R-trees and their variant can be used for indexing [71, 130]. If each object is approximated by a convex polyhedron, cell-trees may be used [67]. Sometimes it is more useful to decompose the spatial objects in several more simple objects, and then index these new ones. This topic has been considered in a very general way by Brodsky and Wang in [31], who have developed an infrastructure for approximation-based query processing based on monoid comprehension. Such infrastructure has then been tailored to constraint objects. The proposed approach is based on the concepts of *approximation grouping* and *inverse grouping*. *Approximation grouping* is the association of each approximating object with the set of decomposed objects it approximates. *Inverse grouping* is the association between each decomposed object with the objects it belongs to. Using these two concepts, approximation-based query processing strategies have been developed. Approximating objects can then be indexed by using techniques developed for spatial databases.

6.2 Query optimization

Besides the use of specific data structures, there have traditionally been two major approaches to query optimization. One is based on compile time algebraic simplification of a query by using heuristics [72, 140]. We call this approach *logical optimization*. Logical optimization is intended to improve the cost of answering the query independently of the actual data or the physical structure of data. A typical approach to logical optimization is to apply logical transformation to the query representation following three main steps:

1. standardize the query (*standardization*), i.e., an internal query representation is determined, leaving the system all necessary freedom to optimize query evaluation;

2. simplify the query to avoid duplication of effort, removing redundancy (*simplification*);

3. construct expressions that are more efficient with respect to query execution performance (*amelioration*). Amelioration is based on rewriting rules and heuristics, dictating when to apply these rules. A typical heuristic requires performing selection as soon as possible [140].

The second approach, here called *cost-based optimization*, is based on the cost estimation of different strategies [140]. This approach requires making assumptions about the distribution of data (like uniformity within attributes and independence of attributes). Based on these assumptions, the possible query execution strategies are typically represented as a tree and the cost of each of these strategies is estimated, starting from some basic parameters, assumed to be known.

As discussed in [27], both approaches fail when applied to constraint databases. Indeed, the heuristics of the algebraic approach are based on the assumption that selection conditions are readily available. By contrast, extracting such conditions from the constraints of a query involves linear programming techniques which are in general expensive. For the cost estimation approach, a similar problem exists, to extract explicit constraints which are needed for the estimation. Even if these constraints were readily available, there is a second problem; such approaches often rely on assumptions about the data distribution which do not always hold in constraint databases.

From all those considerations it follows that traditional query optimization methods are not adequate when applied to constraint databases. Therefore, new methods, tailored to constraint databases, have to be defined.

Up to now, very little work has been carried out in this context. With respect to logical optimization, since GRA is an extension of the relational algebra dealing with infinite relations, the rewriting rules that have been defined for the relational algebra expressions can still be applied, at least from a theoretical point of view, to GRA expressions. However, these rules not always are practically applicable since they often require the use of expensive algorithms (such as additional projections). The only work about logical optimization we are aware of has been proposed by Grumbach and Lacroix [61]. They investigated the problem of logical optimization for GRA when LPOLY is used. In particular, they proposed a canonical form for linear generalized relations and a list of primitive operations used in the evaluation of queries. Most of such operations are similar to classical techniques for constraint solving [51, 118]. Then, they analyze the complexity of applying relational rewriting rules to generalized relational algebra expressions, in terms of the proposed primitive operations. Finally, they propose various query evaluation schemes that refine the usual relational ones, by extending it with constraint solving techniques. This approach will be deeply discussed in Chapter 7, where it will be extended to deal with EGRA expressions.

With respect to cost-based optimization techniques, the only work in this direction we are aware of has been proposed by Brodsky, Jaffar, and Maher [27]. The idea is to use statistical sampling for the cost estimation of specific plans, which has the advantage of being independent of the data distribution. Since it is impracticable to consider all possible plans when searching for the best one, trials of evaluation plans are performed, one at the time, reducing the work required for the estimation. The amount of the avoided work is based on the best cost estimated so far. The algorithm is then used to optimize constraint queries, composed of selection, projection and join operators, by using statistical methods. This allows the detection of the optimal plan with reasonable costs.

The main limitation of such an approach is the use of sampling. Though the sampling method often gives more accurate estimation than other methods (see [41] for a short survey), it can be considered successful only in estimating the cost of statistical queries, usually not involving complex selectivity estimation. On the other hand, in the context of query optimization where selectivity estimation is much more frequent, the cost of the sampling method is prohibitive and not of practical use.

6.3 Concluding remarks

Optimizing constraint databases is a fundamental issue in order to make constraint databases a practical technology. Little work has been done in this context. In particular, from our point of view, the following issues require further investigation:

- Definition of other optimal worst-case complexity data structures for constraint databases, possibly scaling to arbitrary dimensions.
- Detection of the optimal execution plan, when filtering approaches are used.
- Since each generalized tuple with d variables can be seen as the symbolic representation of a spatial object in the d-dimensional space, and since each spatial object can be represented as a set of bounding segments, an interesting research direction is to analyze how indexing data structures defined for segment databases can be applied to constraint databases, retaining optimal worst-case complexity in the number of the generalized tuples.
- The use of constraints might sometimes simplify the execution of some spatial queries. For example, the intersection-based spatial join can be computed on constraints by applying a satisfiability check, without using a computational geometry algorithm. This new approach to process spatial queries has to be compared with the classical one, based on the use of computational geometry algorithms.

Chapter 7

Logical rewriting rules for EGRA expressions

The aim of this chapter is to investigate the logical optimization of EGRA and GRA expressions. The basic issues in defining a logical optimizer for GRA have been investigated in [61]. In this chapter, we review that already proposed approach and introduce some additional considerations. We then introduce rewriting rules and heuristics for EGRA set operators and show under which conditions they can be used to optimize GRA expressions. Set operators (especially set selection) allow the definition of new rewriting rules for GRA that cannot be derived from the relational ones. From this point of view, EGRA can be seen as a useful intermediate language to optimize GRA expressions.

The chapter is organized as follows. In Section 7.1, rewriting rules for GRA are introduced. Simplification and amelioration rules (also called optimization rules) for EGRA are presented in Section 7.2. A discussion about how such rules can be used as part of an optimizer is then presented in Section 7.3. The analysis is performed with respect to an arbitrary logical theory Φ, admitting variable elimination and closed under complementation.

7.1 Rewriting rules for GRA expressions

In the following we discuss how simplification and optimization relational rewriting rules can be applied to GRA expressions.

Simplification rules. In general, there may be several semantically equivalent expressions representing the same query. One source of differences between two equivalent expressions is their degree of redundancy. An operator is redundant if the result of the execution of the associated query against a given generalized relation r is equal to r. A straightforward execution of a query would lead to the execution of a set of operations, some of which are redundant. The aim of the *simplification* step is to rewrite an expression into a more efficient one, by reducing redundant subexpressions.

A typical simplification rule in relational databases removes *redundant* selection operators from a cascade of selections. Formally, if F is an EGRA expression, simplification rules can be expressed as follows:[1]

$$\sigma_{P_1}(\sigma_{P_2}(F)) \equiv_r \emptyset \text{ if } ext(P_1 \wedge P_2) \text{ is empty}$$
$$\sigma_{P_1}(\sigma_{P_2}(F)) \equiv_r \sigma_{P_2}(R) \text{ if } ext(P_1) \subseteq ext(P_2).$$

For other simplification rules, see [78].

Optimization rules. The application of rewriting rules does not necessarily produce a unique expression. Syntactically different but semantically equivalent expressions may greatly differ with respect to some performance parameters. The aim of the *optimization* is to rewrite an expression into an equivalent expression admitting a more efficient execution. Efficiency is measured in terms of the *dimension* of the input relations (i.e., the cardinality of their schema), the *size* of the relations (the number of generalized tuples) and the number of constraints per tuples. As it has been proved in [61] for LPOLY, the cost of any GRA expression can be expressed in terms of these parameters. In particular, all algebraic operations, except projection and complementation, are linear in the number of tuples. The cost of projection mainly depends on the arity of the relation and the number of constraints per tuple. The cost of complementation depends on the number of tuples and the number of constraints per tuple. Moreover, as we have seen in Chapter 2, all algebraic operations may generate redundant constraints and inconsistent tuples.

Rewriting is based on the application of a set of heuristics, aiming at reducing the parameters described above. In the relational context, the typical heuristics are the following:

1. *Perform selection and projection as early as possible.* This transformation allows the reduction of the dimension and the size of the intermediate relations, generated by the computation.

2. *Combine sequences of unary operations.* A cascade of unary operations – selections and projections – can be combined into a single operator. This allows us to access and analyze each tuple only once.

[1] In this chapter, $\emptyset$ denotes an empty expression.

When applying the relational rewriting rules to GRA expressions, two aspects have to be taken into account:

- A GRA operator may be costly for two different reasons: (i) the algorithm to perform the algebraic operation is expensive; (ii) the computation increases one of the parameters described above.
- The heuristics of the algebraic approach are based on the assumption that selection conditions are readily available. Thus, they can be easily checked, without additional costs. By contrast, extracting such conditions from the constraints of a query involves techniques which are in general expensive [27].

From the previous considerations it follows that not all heuristics successfully applied in the relational context can be applied to GRA expressions, since the trade-off between costs of the expressions involved in the rewriting may be different. An important aspect is the redundancy introduced by algebraic operators. For example, each time the selection operator is applied, a new constraint is inserted in each generalized tuple. No relational operator generates redundancy. Therefore, relational rules have to be modified in order to consider these additional aspects.

Relational rewriting rules have been carefully investigated by Grumbach and Lacroix in the context of GRA expressions, when applied to LPOLY [61]. Such rules are presented in Table 7.1. Besides the rules presented in [61], the table presents a further rule (rule (13)) that allows the rewriting of the union of two selections into a single selections with respect to a d-generalized tuple. For the sake of completeness, the table also presents the commutativity rule for tuple selections. The table, for each typical relational rewriting rule, introduces a specific heuristic (i.e., a rewriting direction, represented by an arrow) in order to efficiently apply such a rule to GRA expressions. Only heuristics for rules (8) and (11) are different with respect to the corresponding heuristics used in the relational context. The proof of the correctness of these heuristics is presented in [61].

7.2 Rewriting rules for EGRA expressions

In the following we analyze which further rewriting rules can be defined for EGRA expressions. Since GRA operators are a subset of EGRA operators, the additional rules deal with set operators, in particular with the set selection operator.

(1) $F_1 \bowtie F_2 \leftrightarrow F_2 \bowtie F_1$
(2) $F_1 \bowtie (F_2 \bowtie F_3) \leftrightarrow (F_1 \bowtie F_2) \bowtie F_3$
(3) $\sigma_{P_1}(\sigma_{P_2}(F)) \leftrightarrow \sigma_{P_2}(\sigma_{P_1}(F))$
(4) $\sigma_{P_1}(\sigma_{P_2}(F)) \rightarrow \sigma_{P_1 \wedge P_2}(F)$
(5) $\Pi_{[X_1,...,X_n]}(\sigma_P(F)) \rightarrow \sigma_P(\Pi_{[X_1,...,X_n]}(F))$
$\alpha(P) \subseteq \{X_1, ..., X_n\}$
(6) $\Pi_{[X_1,...,X_n]}(\sigma_P(F)) \rightarrow$
$\Pi_{[X_1,...,X_n]}(\sigma_P(\Pi_{[X_1,...,X_n,Y_1,...,Y_m]}(F)))$
$\alpha(F) = \{X_1, ..., X_n, Y_1, ..., Y_m\}$
(7) $\Pi_{[X_1,...,X_n]}(\Pi_{[Y_1,...,Y_m]}(F)) \rightarrow \Pi_{[X_1,...,X_n]}(F)$
$\{Y_1, ..., Y_m\} \supseteq \{X_1, ..., X_n\}$
(8) $\sigma_P(F_1 \bowtie F_2) \leftarrow \sigma_P(F_1) \bowtie \sigma_P(F_2)$
$\alpha(P) \subseteq \alpha(F_1) \cup \alpha(F_2)$
(9) $\sigma_P(F_1 \bowtie F_2) \rightarrow \sigma_P(F_1) \bowtie F_2$
$\alpha(P) \cap \alpha(F_2) = \emptyset$
(10) $\sigma_{P_1 \wedge P_2}(F_1 \bowtie F_2) \rightarrow \sigma_{P_1}(F_1) \bowtie \sigma_{P_2}(F_2)$
$\alpha(P_1) \cap \alpha(F_2) = \emptyset$
$\alpha(P_2) \cap \alpha(F_1) = \emptyset$
(11) $\sigma_{P_1 \wedge P_2}(F_1 \bowtie F_2) \leftarrow \sigma_{P_2}(\sigma_{P_1}(F_1) \bowtie F_2)$
$\alpha(P_1) \cap \alpha(F_2) = \emptyset$
(12) $\sigma_P(F_1 \cup F_2) \leftrightarrow \sigma_P(F_1) \cup \sigma_P(F_2)$
(13) $\sigma_{P_1}(F) \cup \sigma_{P_2}(F) \rightarrow \sigma_{P_1 \vee P_2}(F)$
(14) $\sigma_P(F_1 \setminus F_2) \rightarrow \sigma_P(F_1) \setminus F_2$
(15) $\Pi_{[X_1,...,X_n]}(F_1 \bowtie F_2) \rightarrow \Pi_{[Y_1,...,Y_h]}(F_1) \bowtie \Pi_{[Z_1,...,Z_k]}(F_2)$
$\{Y_1, ..., Y_h\} \subseteq \alpha(F_1)$, $\{Z_1, ..., Z_k\} \subseteq \alpha(F_1)$
$\{Y_1, ..., Y_h, Z_1, ..., Z_k\} = \{X_1, ..., X_n\}$
(16) $\Pi_{[X_1,...,X_n]}(F_1 \cup F_2) \leftrightarrow \Pi_{[Y_1,...,Y_h]}(F_1) \cup \Pi_{[Z_1,...,Z_k]}(F_2)$
$\{Y_1, ..., Y_h\} \subseteq \alpha(F_1)$, $\{Z_1, ..., Z_k\} \subseteq \alpha(F_1)$
$\{Y_1, ..., Y_h, Z_1, ..., Z_k\} = \{X_1, ..., X_n\}$

Table 7.1: Optimization rules for GRA operators.

7.2.1 Simplification rules

Simplification rules for EGRA tuple operators correspond to simplification rules presented in Section 7.1. However, new rules can be devised dealing with set operators, as shown by the following example.

Example 7.1 *Consider a generalized relation R with schema $\{X, Y\}$, on* LPOLY. *Suppose that each generalized tuple in R represents a given region of a map. Consider the following query:*

> *"Select the part of regions contained in R that intersect a river P and are contained in a given region G."*

If P and G are expressed by using LPOLY, *the previous query can be represented in EGRA as $\sigma^s_{(G,t,\supseteq)}(\sigma_P(R))$. Now suppose that the river is totally contained in G (the user may not know this information). In this case, the previous expression is equivalent to $\sigma_P(R)$.*

As another example consider the following query:

> *"Select all regions contained in R that intersect a river P and are contained in a given region G."*

The previous query can be represented in EGRA as $\sigma^s_{(G,t,\supseteq)}(\sigma^s_{(P,t,\bowtie\neq\emptyset)}(R))$. If P and G have empty intersection, for any input generalized database, the result of the previous expression is empty. $\diamond$

Simplification rules for set operators essentially involve cascades of selections. In particular, three types of expressions are manipulated by a simplification rule:

- *tuple selection followed by a set selection*

  (i.e., expressions like $\sigma_P(\sigma^s_{(Q_1,Q_2,\theta)}(F))$;
- *set selection followed by a tuple selection*

  (i.e., expressions like $\sigma^s_{(Q_1,Q_2,\theta)}(\sigma_P(F)))$;
- *combination of set selections*

  (i.e., expressions like $\sigma^s_{(Q_1,Q_2,\theta)}(\sigma^s_{(Q'_1,Q'_2,\theta')}(F)))$.

A simplification rule is composed of: an input expression e, a set of conditions c on e, and a resulting expression e' which is equivalent to e under the condition c and is structurally simpler than e (see Table 7.2). Based on the structure of e',

EGRA expression	**Simplified EGRA expression**
	Condition
Combination of set selections	
$\sigma^s_{(Q_1,Q_2,\theta)}(\sigma^s_{(Q'_1,Q'_2,\theta')}(F))$	$\emptyset$
	$\forall t \in F\ (\neg((Q_1(t)\theta Q_2(t)) \wedge (Q'_1(t)\theta' Q'_2(t)))$
$\sigma^s_{(Q_1,Q_2,\theta)}(\sigma^s_{(Q'_1,Q'_2,\theta')}(F))$	$\sigma^s_{(Q'_1,Q'_2,\theta')}(F)$
	$\forall t \in F\ ((Q_1(t)\theta Q_2(t)) \leftarrow (Q'_1(t)\theta' Q'_2(t)))$
A tuple selection followed by a set selection	
$\sigma_P(\sigma^s_{(Q_1,Q_2,\theta)}(F))$	$\emptyset$
	$\forall t \in F\ (Q_1(t)\theta Q_2(t) \wedge\ \nexists u\ u \in ext(t)\ P(u))$
$\sigma_P(\sigma^s_{(Q_1,Q_2,\theta)}(F))$	$\sigma^s_{(Q_1,Q_2,\theta)}(F)$
	$\forall t \in F\ (Q_1(t)\theta Q_2(t) \rightarrow \forall u \in ext(t)\ P(u))$
A set selection followed by a tuple selection	
$\sigma^s_{(Q_1,Q_2,\theta)}(\sigma_P(F))$	$\emptyset$
	$\forall t \in F\ \neg\ (Q_1(t \wedge P)\theta Q_2(t \wedge P))$
$\sigma^s_{(Q_1,Q_2,\theta)}(\sigma_P(F))$	$\sigma_P(F)$
	$\forall t \in F\ (Q_1(t \wedge P)\theta Q_2(t \wedge P))$

Table 7.2: Simplification rules for cascades of selections.

two different types of simplification rules can be devised. The first group of rules identifies cascade of selections leading to an empty result. the second group of rules identifies redundant selections.

In both cases, the simplified expression contains less operators than the original one. The proof of the correctness of these rules directly derives from the definition of the selection operators. In the table, $\theta \in \{\subseteq, \supseteq, \bowtie \neq \emptyset, \bowtie = \emptyset\}$. The condition is expressed by first-order logic. $Q_i(t)$, $i = 1, 2$, is a term representing the generalized tuple obtained by applying the query represented by Q_i to t.

The table presents rules that can be applied to two adjacent selection operators in cascade. However, the same rules can be applied to *all* pairs of (not necessarily adjacent) selection operators belonging to a cascade of selections. Rules dealing with pairs of set selection operators can be applied independently of the order in which they appears, since set selections commute. However, rules dealing with a set selection and a tuple selection must be applied to selection operators appearing in the order specified in the table, since set and tuple selections do not commute (see Subsection 7.2.2).

EGRA expression	**Simplified EGRA expression**
	Condition[a]
A tuple selection followed by a set selection	
$\sigma_{P_1}(\sigma^s_{(P_2,Q,\supseteq)}(F))$	$\emptyset$
	$P_1 \wedge \exists\alpha(Q)\ P_2$ is satisfiable
	$\sigma^s_{(P_2,Q,\supseteq)}(F)$
	$P_1 \leftarrow \exists\alpha(Q)\ P_2$
$\sigma_{P_1}(\sigma^s_{(P_2,Q,\bowtie=\emptyset)}(F))$	$\emptyset$
	$P_1 \rightarrow \exists\alpha(Q)\ P_2$

[a]See also the restrictions imposed by the selection operators (Table 3.3).

Table 7.3: Simplification rules with explicit conditions (A).

From Table 7.2, it follows that, in order to check the conditions for the application of the simplification rules, the state of the database must be taken into account. This check is obviously very inefficient. Therefore, simplification rules can be efficiently applied only if the conditions presented in Table 7.2 are replaced by equivalent conditions not requiring the analysis of the database state.

This property holds only for some combinations of selections (see Tables 7.3, 7.4, and 7.5, and 7.6). Selection conditions used in the tables have the form (P, Q, θ), where P is a generalized tuple and Q is a projection operator. Note that this is not a restriction in that:

- due to the results presented in Section 3.3, all other types of set selection can be reduced to an expression in which set selection contains conditions in which only generalized tuples and projections appear;
- condition $(Q, P, \subseteq)$ is equivalent to condition $(P, Q, \supseteq)$;
- condition $(Q, P, \supseteq)$ is equivalent to condition $(P, Q, \subseteq)$;
- condition $(Q, P, \bowtie\neq\emptyset)$ is equivalent to condition $(P, Q, \bowtie\neq\emptyset)$;
- condition $(Q, P, \bowtie=\emptyset)$ is equivalent to condition $(P, Q, \bowtie=\emptyset)$.

Only two combinations of set selections do not appear in Tables 7.3, 7.4, 7.5, and 7.6:

- $\sigma_{P_1}(\sigma^s_{(P_2,Q,\subseteq)}(F))$;

EGRA expression	**Simplified EGRA expression** / **Condition**[a]
A set selection followed by a tuple selection	
$\sigma^s_{(P_1,Q,\supseteq)}(\sigma_{P_2}(F))$	$\sigma_{P_2}(F)$
	$P_2 \rightarrow \exists\alpha(Q)\ P_1$
	$\emptyset$
	$P_2 \wedge \exists\alpha(Q)\ P_1$ is satisfiable
$\sigma^s_{(P_1,Q,\subseteq)}(\sigma_{P_2}(F))$	$\emptyset$
	$\neg\exists\alpha(Q)\ P_1 \rightarrow P_2$
$\sigma^s_{(P_1,Q,\bowtie\neq\emptyset)}(\sigma_{P_2}(F))$	$\emptyset$
	$P_2 \wedge \exists\alpha(Q)\ P_1$ is satisfiable
	$\sigma_{P_2}(F)$
	$P_2 \rightarrow \exists\alpha(Q)\ P_1$
$\sigma^s_{(P_1,Q,\bowtie=\emptyset)}(\sigma_{P_2}(F))$	$\emptyset$
	$P_2 \rightarrow \exists\alpha(Q)\ P_1$
	$\sigma_{P_2}(F)$
	$P_2 \wedge \exists\alpha(Q)\ P_1$ is satisfiable

[a]See also the restrictions imposed by the selection operators (Table 3.3).

Table 7.4: Simplification rules with explicit conditions (B).

- $\sigma_{P_1}(\sigma^s_{(P_2,Q,\bowtie\neq\emptyset)}(F))$.

It is simple to prove that, in both cases, no check between P_1 and P_2 helps in simplifying the expression.

As we can see from tables presenting simplification rules, the new conditions are based on containment (logically represented by implication) and intersection (logically represented by satisfiability) check between generalized tuples. As we have already remarked, in a general case, the cost of projection is high.[2] Therefore, the application of such rules is much more efficient when the schema of the generalized tuples and the queries appearing in set and tuple selection conditions coincide. When this is not true, the check of the condition is in general more expensive. However, we argue that also in this case the proposed optimization rules have to be applied. Indeed, even if some additional projection has to be executed at compile-time, rewriting rules may avoid the execution of some selection operator at run-time, thus avoiding the execution

[2]The cost of projection is not high when it is applied to non-constrained variables or to a limited set of attributes.

EGRA expression	**Simplified EGRA expression** / **Condition**[a]
Cascade of set selections	
$\sigma^s_{(P_1,Q_1,\supseteq)}(\sigma^s_{(P_2,Q_2,\supseteq)}(F))$	$\emptyset$
	$\exists\alpha(Q_1)\ P_1 \wedge \exists\alpha(Q_2)\ P_2$ is satisfiable
	$\sigma^s_{(P_2,Q_2,\supseteq)}(F)$
	$\exists\alpha(Q_1)\ P_1 \leftarrow \exists\alpha(Q_2)\ P_2$
$\sigma^s_{(P_1,Q_1,\subseteq)}(\sigma^s_{(P_2,Q_2,\supseteq)}(F))$	$\emptyset$
	$\neg\exists\alpha(Q_1)\ P_1 \rightarrow \exists\alpha(Q_2)\ P_2$
$\sigma^s_{(P_1,Q_1,\subseteq)}(\sigma^s_{(P_2,Q_2,\subseteq)}(F))$	$\sigma^s_{(P_2,Q_2,\subseteq)}(F)$
	$\exists\alpha(Q_1)\ P_1 \rightarrow \exists\alpha(Q_2)\ P_2$ is satisfiable
$\sigma^s_{(P_1,Q_1,\bowtie\neq\emptyset)}(\sigma^s_{(P_2,Q_2,\supseteq)}(F))$	$\emptyset$
	$\exists\alpha(Q_1)\ P_1 \wedge \exists\alpha(Q_2)\ P_2$ is satisfiable
	$\sigma^s_{(P_2,Q_2,\supseteq)}(F)$
	$\exists\alpha(Q_1)\ P_1 \leftarrow \exists\alpha(Q_2)\ P_2$

[a] See also the restrictions imposed by the selection operators (Table 3.3).

Table 7.5: Simplification rules with explicit conditions (C).

of much more expensive projections.

7.2.2 Optimization rules

Optimization rules for set EGRA operators can be derived from the optimization rules presented for GRA expressions (see Table 7.7). Such rules are obtained from the corresponding ones presented in Table 7.1 by observing that the set selection cannot increase the redundancy of generalized tuples. Indeed, it works exactly as a relational selection. This means that typical relational heuristics holds in this case (this is the case of rules (23) and (26)).

Besides the rules derived from those presented in Table 7.1, other rules involving set selections have been introduced. These rules combine two set conditions in order to generate a new, non boolean, condition. In particular, they replace two containment tests with a single containment test (rule (19)) and two empty-intersection tests with a single empty-intersection test (rule (20)).

For the sake of completeness, the table also presents the commutativity rule for set selections. Note that tuple and set operators do not commute, as pointed out by the following proposition.

EGRA expression	Simplified EGRA expression
	Condition[a]
Cascade of set selections	
$\sigma^s_{(P_1,Q_1,\bowtie=\emptyset)}(\sigma^s_{(P_2,Q_2,\supseteq)}(F))$	$\sigma^s_{(P_2,Q_2,\supseteq)}(F)$
	$\exists\alpha(Q_1)\ P_1 \wedge \exists\alpha(Q_2)\ P_2$ is satisfiable
	$\emptyset$
	$\exists\alpha(Q_1)\ P_1 \leftarrow \exists\alpha(Q_2)\ P_2$
$\sigma^s_{(P_1,Q_1,\bowtie\neq\emptyset)}(\sigma^s_{(P_2,Q_2,\subseteq)}(F))$	$\sigma^s_{(P_2,Q_2,\subseteq)}(F)$
	$\exists\alpha(Q_1)\ P_1 \wedge \exists\alpha(Q_2)\ P_2$ is satisfiable
$\sigma^s_{(P_1,Q_1,\bowtie=\emptyset)}(\sigma^s_{(P_2,Q_2,\subseteq)}(F))$	$\emptyset$
	$\exists\alpha(Q_1)\ P_1 \wedge \exists\alpha(Q_2)\ P_2$ is satisfiable
$\sigma^s_{(P_1,Q_1,\bowtie\neq\emptyset)}(\sigma^s_{(P_2,Q_2,\bowtie\neq\emptyset)}(F))$	$\sigma^s_{(P_2,Q_2,\bowtie\neq\emptyset)}(F)$
	$\exists\alpha(Q_1)\ P_1 \leftarrow \exists\alpha(Q_2)\ P_2$
$\sigma^s_{(P_1,Q_1,\bowtie=\emptyset)}(\sigma^s_{(P_2,Q_2,\bowtie\neq\emptyset)}(F))$	$\emptyset$
	$\exists\alpha(Q_1)\ P_1 \leftarrow \exists\alpha(Q_2)\ P_2$
$\sigma^s_{(P_1,Q_1,\bowtie=\emptyset)}(\sigma^s_{(P_2,Q_2,\bowtie=\emptyset)}(F))$	$\sigma^s_{(P_2,Q_2,\bowtie=\emptyset)}(F)$
	$\exists\alpha(Q_1)\ P_1 \rightarrow \exists\alpha(Q_2)\ P_2$

[a] See also the restrictions imposed by the selection operators (Table 3.3).

Table 7.6: Simplification rules with explicit conditions (D).

Proposition 7.1 $\sigma_P(\sigma^s_{(Q_1,Q_2,\theta)}(F)) \neq \sigma^s_{(Q_1,Q_2,\theta)}(\sigma_P(F))$.

Proof: Suppose that set and tuple selection operators commute. This would mean that if a generalized tuple t satisfies a condition (Q_1, Q_2, θ), the same condition has always to be satisfied by $t \wedge P$ and vice versa. But this is of course false. For example, consider the generalized relation r containing the generalized tuple $2 \leq X \leq 5 \wedge 4 \leq Y \leq 8$. Now consider the following queries:

- $\sigma_{X\leq 3}(\sigma^s_{(t,X\geq 4,\bowtie\neq\emptyset)}(r))$.

  It is simple to show that the resulting relation contains the generalized tuple $2 \leq X \leq 3 \wedge 4 \leq Y \leq 8$.

- $\sigma^s_{(t,X\geq 4,\bowtie\neq\emptyset)}(\sigma_{X\leq 3}(r))$.

  The result of the previous query is an empty relation.

(17) $\sigma^s_{C_1}(\sigma^s_{C_2}(F)) \leftrightarrow \sigma^s_{C_2}(\sigma^s_{C_1}(F))$[a]
(18) $\sigma^s_{C_1}(\sigma^s_{C_2}(F)) \rightarrow \sigma^s_{C_1 \wedge C_2}(F)$
(19) $\sigma^s_{(P_1,\subseteq)}(\sigma^s_{(P_2,\subseteq)}(F)) \rightarrow \sigma^s_{(P_1 \wedge P_2,\subseteq)}(F)$[b]
(20) $\sigma^s_{(P_1,\bowtie=\emptyset)}(\sigma^s_{(P_2,\bowtie=\emptyset)}(F)) \rightarrow \sigma^s_{(P_1 \vee P_2,\bowtie=\emptyset)}(F)$
(21) $\Pi_{[X_1,...,X_n]}(\sigma^s_C(F)) \rightarrow \sigma^s_C(\Pi_{[X_1,...,X_n]}(F))$
$\alpha(C) \subseteq \{X_1, ..., X_n\}$
(22) $\Pi_{[X_1,...,X_n]}(\sigma^s_C(F)) \rightarrow$
$\Pi_{[X_1,...,X_n]}(\sigma^s_C(\Pi_{[X_1,...,X_n,Y_1,...,Y_m]}(F)))$
$\alpha(F) = \{X_1, ..., X_n, Y_1, ..., Y_m\}$
(23) $\sigma^s_{C_1}(F_1 \bowtie F_2) \rightarrow \sigma^s_{C_1}(F_1) \bowtie \sigma^s_{C_1}(F_2)$
$\alpha(C_1) \subseteq \alpha(F_1) \cap \alpha(F_2)$
(24) $\sigma^s_C(F_1 \bowtie F_2) \rightarrow \sigma^s_C(F_1) \bowtie F_2$
$\alpha(C) \cap \alpha(F_2) = \emptyset$
(25) $\sigma^s_{C_1 \wedge C_2}(F_1 \bowtie F_2) \rightarrow \sigma^s_{C_1}(F_1) \bowtie \sigma^s_{C_2}(F_2)$
$\alpha(C_1) \cap \alpha(F_2) = \emptyset$
$\alpha(C_2) \cap \alpha(F_1) = \emptyset$
(26) $\sigma^s_{C_1 \wedge C_2}(F_1 \bowtie F_2) \rightarrow \sigma^s_{C_2}((\sigma^s_{C_1}(F_1) \bowtie F_2)$
$\alpha(C_1) \cap \alpha(F_2) = \emptyset$
(27) $\sigma^s_{C_1}(F_1 \cup F_2) \leftrightarrow \sigma^s_{C_1}(F_1) \cup \sigma^s_{C_1}(F_2)$
(28) $\sigma^s_{C_1}(F) \cup \sigma^s_{C_2}(F) \rightarrow \sigma^s_{C_1 \vee C_2}(F)$
(29) $\sigma^s_{C_1}(F_1 \setminus^s F_2) \rightarrow \sigma^s_{C_1}(F_1) \setminus^s F_2$

[a] C_1 and C_2 represent boolean conditions.
[b] P_1 and P_2 represent (possibly disjunctive) generalized tuples.

Table 7.7: Optimization rules for EGRA set operators.

Since the results of the previous expressions do not coincide, it follows that set and tuple selections do not commute. □

7.3 Issues in designing GRA and EGRA optimizers

The rules presented in the previous section can be used to design a logical optimizer for EGRA and GRA expressions. The input of such an optimizer is an expression and the result is another expression, equivalent to the original one, but guaranteeing a more efficient execution. In the following we briefly discuss the issues arising when defining such an optimizer. In particular, we first introduce the basic issues in designing a GRA optimizer (see [61] for additional details). Then, we show how such an optimizer can be extended with rules for EGRA set operators.

7.3.1 Basic issues in designing a GRA optimizer

As it has been first recognized in [61], relational query evaluation schemes have to be refined when applied to GRA expressions. In particular, when dealing with such expressions, three different aspects have to be considered:

- *syntactic computation*: purely what the algebra does (see Table 2.1);
- *semantics computation*: this step corresponds to the removal of empty tuples and redundant constraints (see Section 2.3.2.1);
- *normalization*: this step corresponds to the generation of normal forms for generalized tuples.

The evaluation of a query in constraint databases implies several syntactic computation and at least one semantics computation step. Starting from these considerations, Grumbach and Lacroix proposed three different logical optimizers, whose properties can be summarized as follows:

- *Naive evaluation.* A typical relational logical optimization algorithm is applied [140]. A semantic computation is then applied to the result.
- *Semantic evaluation.* In this case, syntactic and semantic computations are mixed. In particular, semantic computations are applied in order to reduce the number of tuples and the number of constraint per tuples after costly operations, that is after intersection, projection, and selection.
- *Normalization strategy.* This strategy can be used when dealing with LPOLY. In that case, a specific normal form has been proposed in [61], in order to represent each generalized tuple by using a number of constraints which is bound by the arity of the relation. The normalization strategy consists in mixing syntactic steps with steps normalizing the output generalized tuples, assuming that the input generalized tuples are normalized.

The application of these algorithms produces different results. In particular, the semantic evaluation and the normalization strategy guarantees to better optimize the query. However, no experiment has been conducted in order to validate these approaches.

7.3.2 Extending GRA optimizers to deal with set operators

In the following, we discuss the issues arising in extending GRA optimizers with rewriting rules for set operators. In particular, we first discuss the basic concepts underlying the definition of a EGRA optimizer and then we show how it can be adapted to GRA expressions.

7.3.2.1 Basic issues in defining an EGRA optimizer

The first problem arising when designing an EGRA logical optimizer is the choice of the order by which EGRA and GRA simplification and optimization rules are applied. Since the proposed simplification rules mainly apply to cascades of selections, and since the generation of cascades of selections is one of the goals of optimization, we claim that simplification should be applied in two steps, before and after optimization. Indeed, the optimization may generate new subexpressions that can be further simplified. On the other hand, the application of a simplification step before optimization reduces the length of the expression to which optimization is applied. It is simple to show that no new simplification step can further optimize the expression. The following example motivates this choice.

Example 7.2 *Consider the following simplified expression:*

$$\sigma^s_{C_1}(\sigma_{C_2}(R) \bowtie \sigma^s_{C_3}(R)).$$

Now suppose that the condition for applying rule (24) of Table 7.7 is satisfied; the expression obtained is $\sigma^s_{C_1}(\sigma_{C_2}(R)) \bowtie \sigma^s_{C_3}(R)$. By applying the previous optimization rule, a new cascade of selections has been generated. Now suppose that $\sigma^s_{C_1}(\sigma_{C_2}(R))$ can be rewritten in the empty expression $\emptyset$. In this case, the original expression is rewritten in $\sigma^s_{C_3}(R)$. This reduction would have not be possible if simplification and optimization rules were applied in a different order. $\diamond$

Another consideration is related to boolean conditions. Simplification rules have to be applied to non-boolean conditions. Therefore, as a first step before simplification it can be useful to remove boolean conditions by applying rules introduced in Chapter 3. After the last simplification step, boolean conditions can be eventually reconstructed.

Based on the previous considerations, typical logical optimization algorithms, proposed for the relational algebra, such as the one presented in [61, 140], can be easily extended to deal with EGRA expressions.

7.3.2.2 Introducing set rewriting rules in the definition of a GRA optimizer

In Sections 7.2.1 and 7.2.2, we have proposed new simplification and optimization rules for EGRA expressions. From Chapter 3, we know that each set operator is equivalent to a tuple expression, when generalized tuple identifiers are inserted inside generalized relations. Under this assumption, set optimization rules can also be interpreted as optimization rules for GRA expressions.

The use of the new rules to optimize GRA expressions is useful since it may generate optimized GRA expressions that would have not necessarily been generated by applying GRA optimization rules. Indeed, by using GRA rules, we are not sure to generate the expression that is instead generated by applying EGRA optimization rules. This is due to the fact that the equivalence of two relational expressions is in general undecidable. [81]. The following example better clarifies this concept.

Example 7.3 *Consider the following expression:*

$$\sigma^s_{(P_1,\subseteq)}(\sigma^s_{(P_2,\subseteq)}(R)). \tag{7.1}$$

By applying rule (19) of Table 7.7, this expression is rewritten as follows:

$$\sigma^s_{(P_1 \wedge P_2,\theta)}(R). \tag{7.2}$$

The GRA expression equivalent to (7.1) is the following:

$$(\Pi_{[N]}(R_1) \setminus \Pi_{[N]}(R \setminus \sigma_{(P_1)}(R_1)) \bowtie R_1) \tag{7.3}$$

where

$$R_1 \equiv (\Pi_{[N]}(R) \setminus \Pi_{[N]}(R \setminus \sigma_{(P_2)}(R)) \bowtie R).$$

The GRA expression equivalent to (7.2) is the following:

$$(\Pi_{[N]}(R) \setminus \Pi_{[N]}(R \setminus \sigma_{(P_1 \wedge P_2)}(R)) \bowtie R) \tag{7.4}$$

It can be shown that, in order to rewrite expression (7.3) into expression (7.4), the rules presented in Table 7.1 are not sufficient. Indeed, information about containment among generalized relations generated as intermediate results of the evaluation are required in order to perform this rewriting. This is mainly due to the presence of the difference operator. ◇

Three main issues have to be considered when using EGRA rules to optimize GRA expressions:

1. Since GRA expressions do not contain set operators, the query processor must decide *when* patterns corresponding to set operators have to be detected inside GRA expressions, according to Table 3.8.

2. A decision must also be taken with respect to when simplification and optimization rules should be applied.

3. Since rewriting may generate new subexpressions corresponding to set operators, a decision should be taken with respect to the number of times the optimization is iterated.

In order to design a GRA optimizer, we assume that there exists a GRA optimizer, as one of those presented in Section 7.3.1 and we suggest the following guidelines:

1. Since in spatial and temporal contexts it is likely that users will insert set operators directly in their queries, GRA subexpressions should be rewritten into set operators as the first step of the optimization. Note that no boolean condition is generated by this step.

2. Simplification and optimization rules for set operators have to be first applied. In particular, as discussed in Subsection 7.3.2.1, first simplification rules and then optimization rules have to be applied. Then, a new simplification is applied in order to simplify new cascades of selections generated by the optimization step.

3. After this step, the obtained EGRA expression must be standardized and given as input to GRA optimizer.

4. The previous steps may be executed more than once, since each step may generate new possible optimizable sub-expressions. In this case, a loop termination condition is required.

Figure 7.1 illustrates the suggested heuristics to design a GRA optimizer.

7.4 Concluding remarks

In this chapter, we have presented GRA and EGRA rewriting rules. GRA rules have been taken from [61]. Additional EGRA rules dealing with set operators have then been proposed. Such rules can be used not only to optimize EGRA expressions but, due to the equivalence between GRA and EGRA, they can also be used to improve the efficiency of GRA optimizers. The basic issues in designing such an optimizer have also been discussed.

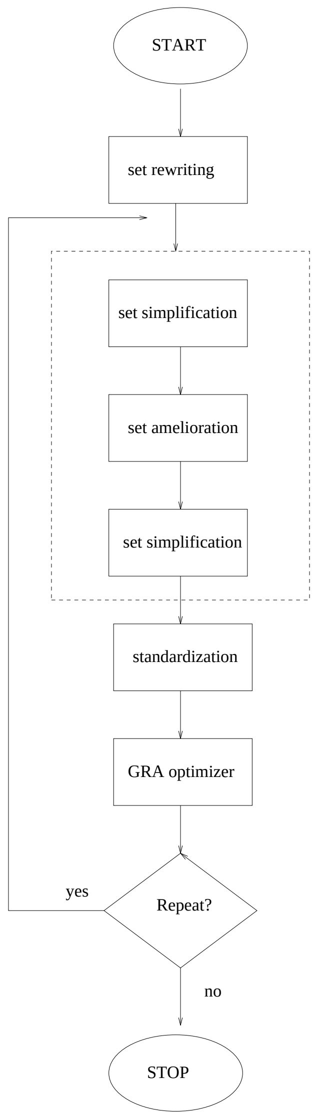


Figure 7.1: The suggested heuristics to design a GRA optimizer.

Chapter 8

A dual representation for indexing constraint databases

As we have seen in Chapter 6, good worst-case complexity data structures have been defined only for 1-dimensional constraints. In this chapter, we analyze optimal worst-case data structures for supporting a specific type of ALL and EXIST selections applied to 2-dimensional constraints. The proposed techniques rely on the use of a specific dual representation for polyhedra, first presented in [67]. The main advantage of the dual representation is that it is defined for arbitrary d-dimensional objects; this is particularly useful in constraint databases, where the dimension is usually not limited a priori.

The specific problem we consider concerns the detection of all generalized tuples whose extension intersects or is contained in a given d-dimensional half-plane. This problem is not relevant in spatial databases, where closed objects are usually considered; however in constraint databases such queries are more significant since open objects are often represented. When optimal solutions to this problem are not found, techniques based on a filtering-refinement approach (see Chapter 6) are introduced. The main characteristic of these techniques is that, differently from data structures proposed for spatial databases, the approximation is not applied to the polyhedra represented by generalized tuples but to the query half-plane. The proposed techniques are then compared both from a theoretical and experimental point of view. A comparison with R-trees, a typical spatial data structure [71, 130], is also presented.

The chapter is organized as follows. Section 8.1 motivates the investigation of indexing techniques supporting half-plane queries. In Section 8.2, the dual representation for generalized tuples is presented. The problem of detecting the intersection between a polyhedron and a half-plane is considered in Section 8.3 whereas in Section

8.4 we investigate the problem of detecting the intersection between two polyhedra. External memory solutions for half-plane queries are considered in Section 8.5; an optimal solution for a weaker problem is also presented. Sections 8.6, 8.7, and 8.8 present three different techniques approximating the solution of a half-plane query when the problem cannot be reduced to the weaker one. A theoretical comparison of the proposed approximated techniques is then presented in Section 8.9; experimental results are finally discussed in Section 8.10.

8.1 Motivations and basic notation

As we have seen in Chapter 6, the main difference between spatial and constraint databases is that spatial databases typically represent 2- or 3-dimensional closed spatial objects whereas constraint databases admit the representation of arbitrary d-dimensional, possibly open, objects. Spatial data structures have good performance when the object specified in the query is closed. Similar performance cannot be guaranteed when spatial objects are open[1]. The design of data structures supporting selections based on open objects is therefore an important issue for constraint databases.

Example 8.1 *Consider a generalized relation Prod_process containing, in each generalized tuple, information about a specific production process. Each production process relates two products and three resources. Thus, each generalized tuple is a conjunction of linear constraints with five variables: P_1 representing the quantity of the first product, P_2, representing the quantity of the second product, R_i representing the available quantity of the i-th resource, $i = 1, ..., 3$. The following is an example of a possible generalized tuple:*

$$3P_1 + 4P_2 \leq R_1 \wedge 100P_1 + P_2 \leq R_2 \wedge P_1 + 50P_2 \leq R_3.$$

This tuple specifies how resources and products are related in a given production process. Now suppose that we want to determine all processes that can be satisfied (thus, specific quantities of products P_1 and P_2 can be found) assuming that a global constraint $3R_1 + 5R_2 + 6R_3 \leq 400$ holds. In order to solve this query, all generalized tuples whose extension has a non-empty intersection with the extension of the global constraint must be determined. This corresponds to an EXIST selection with respect to a query half-plane.

[1]In the following, the term "open" has not the classical topological meaning but it means "unbound".

As another example, suppose to determine all processes that, whatever the quantities produced are, satisfy the global constraint. In this case, all generalized tuples whose extension is contained in the extension of the global constraint must be determined. Thus this is an ALL selection with respect to a query half-plane. $\diamond$

From the previous example it follows that the definition of data structures efficiently supporting queries with respect to open spatial objects is an important issue in constraint databases.

In the following, we analyze ALL and EXIST selection problems with respect to a half-plane (also called *query half-plane*). Generalized tuples are assumed to be represented by using LPOLY. Thus, each generalized tuple has the form: $\wedge_{i=1}^{n} a_1^i X_1 + ... + a_d^i X_d + c^i \ \theta \ 0$, where $\theta_i \in \{\geq, \leq, =\}$. Generalized tuples of this kind and generalized relations containing only such generalized tuples are called *regular*. In the following any generalized tuple is assumed to be regular. Given a half-plane q, a generalized relation r, and $E \in \{ALL, EXIST\}$, $E(q, r)$ is called a *query* whereas E is called the *type* of $E(q, r)$. When the generalized relation is not specified, a query is denoted by $E(q)$.

Without limiting the generality of the discussion, we assume that each equality constraint $a_1^i X_1 + ... + a_d^i X_d + c^i = 0$ is replaced by the equivalent conjunction of constraints $a_1^i X_1 + ... + a_d^i X_d + c^i \geq 0 \wedge a_1^i X_1 + ... + a_d^i X_d + c^i \leq 0$. Moreover, we use a notation very close that used in spatial contexts. In particular, we call *hyperplane* in a d-dimensional space (denoted by E^d) the spatial object having equation $a_1^i X_1 + ... + a_d^i X_d + c^i = 0$ and *half-plane* in a d-dimensional space the spatial object having equation $a_1^i X_1 + ... + a_d^i X_d + c^i \ \theta \ 0$, $\theta \in \{\geq, \leq\}$. A half-plane is called *1-half-plane* if it can be rewritten as $X_d \geq b_1 X_1 + ... + b_{d-1} X_{d-1} + b_d$, otherwise it is a *0-half-plane*. A *d-dimensional convex polyhedron* P in a d-dimensional plane is defined as the intersection of a finite number of half-planes in E^d. Moreover, we denote by $p(P)$ the boundary of P and with t_P the generalized tuple representing P (thus, $P = ext(t_P)$).

Given a polyhedron P and a hyperplane H, H is a *supporting hyperplane* with respect to P if H intersects P and P lies in one of the half-spaces defined by H. If H is a supporting hyperplane for P then $H \cap P$ is a *face* of P. The faces of dimension 1 are called *edges*; those of dimension 0 are called *vertices*. A supporting hyperplane is called *boundary hyperplane* if the face $H \cap P$ is of dimension $d - 1$. The faces of P that are a subset of some supporting hyperplane with $\theta = `\geq'$ and $a_d \leq 0$ form the *upper hull* of $P$; the faces of $P$ that are a subset of some supporting hyperplane with $\theta = `\geq'$ and $a_d \geq 0$ form the *lower hull* of P.

8.2 The dual transformation

In [67], Gunther proposed a dual transformation for polyhedra; using this transformation, he gave complexity bounds for the problem of finding the intersections between a polyhedron and a hyperplane or another polyhedron. In the following, we use this transformation to determine:

- all generalized tuples intersecting a given half-plane;
- all generalized tuples contained in a given half-plane.

Thus, we extend results presented in [67] to deal with a set of polyhedra and containment. The following concepts are taken from [67].

In order to present the dual transformation, we assume that none of the considered half-planes are vertical.[2] Under this hypothesis, a hyperplane $X_d = b_1x_1 + ... + b_{d-1}X_{d-1} + b_d$ intersects the d-th coordinate in a unique point represented by the equation:

$$X_d = b_1x_1 + ... + b_{d-1}X_{d-1} + b_d$$

where $b_i = -a_i/a_d, i = 1, ..., d-1$ and $b_d = c$. Given a hyperplane H, the following function is introduced:

$$F_H : E^{d-1} \rightarrow E^1$$
$$F_H(X_1, ..., X_{d-1}) = b_1X_1 + ... + b_{d-1}X_{d-1} + b_d.$$

A point $p = (p_1, ..., p_d)$ lies *above* (*on*, *below*) H if $p_d > (=, <)\ F_H(p_1, ..., p_{d-1})$. Using the dual transformation proposed by Brown [32], each hyperplane can be mapped into a point and vice versa. In particular, the dual representation $D(H)$ of a hyperplane $X_d = b_1x_1 + ... + b_{d-1}X_{d-1} + b_d$ is the point $(b_1, ..., b_d)$ in E^d. Conversely, the dual representation $D(p)$ of a point $p = (p_1, ..., p_d)$ is the hyperplane defined by the equation $X_d = -p_1X_1 ... - p_{d-1}X_{d-1} + p_d$. In the following, we call *primal space* the reference space of the original polyhedra and *dual space* the reference space of the dual representations. The following result holds.

Lemma 8.1 *[67] A point p lies above (on, below) a hyperplane H iff the dual $D(H)$ lies below (on, above) $D(p)$.* □

The dual representation can be extended to convex polyhedra by associating each polyhedron with a pair of functions. Let V_P be the set of vertices of a polyhedron P. Such functions are defined as follows:

[2]Actually, the proposed transformation can be extended to deal with vertical hyperplanes. We refer the reader to [67] for some further details.

$$TOP^P(X_1, .., X_{d-1}) = max_{v \in V_P}\{F_{D(v)}(X_1, ..., X_{d-1})\}$$
$$BOT^P(X_1, .., X_{d-1}) = min_{v \in V_P}\{F_{D(v)}(X_1, ..., X_{d-1})\}.$$

These functions are piecewise linear and continuous. TOP^P is convex whereas BOT^P is concave [124]. Moreover, they map any slope $(b_1, ..., b_{d-1})$ of a non-vertical hyperplane into the maximum (TOP^P) or the minimum (BOT^P) value b_d such that the hyperplane $X_d = b_1x_1 + ... + b_{d-1}X_{d-1} + b_d$ intersects P. It can be shown that this representation is non-ambiguous, i.e., each polyhedron is associated with exactly one pair of functions and vice versa. Such functions satisfy the following property.

Proposition 8.1 *For any point* $(p_1, ..., p_{d-1})$, $TOP^P(p_1, ..., p_{d-1}) \geq BOT^P(p_1, ..., p_{d-1})$. □

If the polyhedron is not bounded (this case is very common in constraint databases), the definition of functions TOP^P and BOT^P does not work and have to be extended in order to deal with some virtual vertex at infinity. In order to deal with such vertices, let C_P denote a d-dimensional cube with edge length $E(C_P)$ that contains all vertices of P. The bounded polyhedron $P \cap C_P$ has a set of vertices $V_{P \cap C_P} = V_P \cup \overline{V}$, where $\overline{V}$ contains those vertices that are formed by the intersections of C_P with edges of P. As $E(C_P)$ goes to infinity, so do the vertices in $\overline{V}$. The dual $D(\overline{v})$ of any vertex $\overline{v} \in \overline{V}$ goes towards a vertical hyperplane with a corresponding function $F_{D(\overline{v})} : E^{d-1} \rightarrow +/-\infty$. Functions $TOP^P, BOT^P : E^{d-1} \rightarrow E^1 \cup \{+\infty, -\infty\}$ can be defined as follows:

$$TOP^P(X_1, .., X_{d-1}) = lim_{E(C_P) \rightarrow \infty} max_{v \in V_P \cup \overline{V}}\{F_{D(v)}(X_1, ..., X_{d-1})\}$$
$$BOT^P(X_1, .., X_{d-1}) = lim_{E(C_P) \rightarrow \infty} min_{v \in V_P \cup \overline{V}}\{F_{D(v)}(X_1, ..., X_{d-1})\}.$$

From [32], it follows that there exists an isomorphism between the upper hull of a polyhedron P and the graph of TOP^P. Each k-dimensional face f of the upper hull of P corresponds to exactly one $(d-k-1)$-dimensional face $D(f)$ of TOP^P graph and vice versa. Moreover, if two faces f_1 and f_2 of the P upper hull are adjacent, then so are $D(f_1)$ and $D(f_2)$. The same isomorphism exists between the P lower hull and the graph of BOT^P. From this consideration it follows that, if the number of vertices of P is n_v, the graphs of TOP^P and of BOT^P are polyhedral surfaces in E^d consisting of no more than n_v convex $(d-1)$-dimensional faces and no more than $O(n_v^2)$ $(d-2)$-dimensional faces. Such surfaces can be constructed as follows (this algorithm has not been presented in [67]).

Dual Transformation Algorithm

1. Let a polyhedron P be represented by the intersection of half-planes $H_1, ..., H_s$. Let $H_1, ..., H_m$ be 1-half-planes and $H_{m+1}, ..., H_s$ be 0-half-planes. Let V_P be the set of vertices of P.[3] Let P_{up} be the polyhedron represented by the intersection of $H_1, ..., H_m$. Let P_{down} be the polyhedron represented by the intersection of $H_{m+1}, ..., H_s$. Note that, if both P_{up} and P_{down} exist, $V = (V_P \setminus V_{P_{up}}) \setminus V_{P_{down}} \neq \emptyset$.

2. Let $UP(P)$ be the polyhedron represented by the intersection of the 1-half-planes K_i such that $p(K_i) = D(v_i)$, $v_i \in V_{P_{up}} \cup V$.

3. Let $DOWN(P)$ be the polyhedron represented by the intersection of the 0-half-planes K_i such that $p(K_i) = D(v_i)$, $v_i \in V_{P_{down}} \cup V$.

4. Consider the unbound hyperplanes belonging to P_{up}. Let V_1 be the set of vertices defined by such hyperplanes. It can be proved that each such vertex is defined by the intersection of at least $(d-1)$ unbound hyperplanes. For each such set of hyperplanes, generate the vertical hyperplane H_v passing through the points corresponding to these hyperplanes in the dual plane.

   Generate the half-plane supported by H_v and containing the vertices of P_{up}, and add it to $UP(P)$.

5. Consider the unbound hyperplanes belonging to P_{down}. Let V_1 be the set of vertices defined by such hyperplanes. It can be proved that each such vertex is defined by the intersection of at least $(d-1)$ unbound hyperplanes. For each such set of hyperplanes, generate the vertical hyperplane H_v passing through the points corresponding to these hyperplanes in the dual plane.

   Generate the half-plane supported by H_v and containing the vertices of P_{down}, and add it to $DOWN(P)$.

The previous algorithm can be better understood in the 2-dimensional case. In those case, a hyperplane is a line.

1. Step 2 generates, for each vertex in $V_{P_{up}} \cup V$, the corresponding dual line. $UP(P)$ is the convex polygon obtained by the intersection of the 1-half-planes supported by such lines.

2. Step 3 performs a similar construction for vertices in $V_{P_{down}} \cup V$.

[3]Note that faces of dimensions greater than 0 can always be seen as an infinite number of vertices.

3. Step 4 and Step 5 can be better understood as follows. In the 2-dimensional case, some hyperplanes are unbound if P_{up} and P_{down} do not intersect either on the left or on the right. Consider an unbound line $Y = a_1X + b_1$ belonging to P_{up}. In Step 4, the half-plane $X \ \theta \ a_1$, $\theta \in \{\leq, \geq\}$, containing V_{up} is added to $UP(P)$.

4. A similar reasoning is done for $DOWN(P)$ in Step 5.

Note that the graph of TOP^P coincides with the boundary of $UP(P)$ from which the vertical hyperplanes, added in Steps 4 and 5 of the algorithm, have been removed. A similar condition holds for BOT^P and $DOWN(P)$.

The following results hold (see Appendix C for proofs).

Lemma 8.2 *Let P be a polyhedron. Let H be a hyperplane. Then, $D(H)$ belongs to the TOP^P graph or to the BOT^P graph iff H is a supporting hyperplane for P.*

Proof: It directly depends on the characterization of the graphs of TOP^P and BOT^P. Indeed, these two functions map any slope $(b_1, ..., b_{d-1})$ of a non-vertical hyperplane into the maximum (TOP^P) or the minimum (BOT^P) intercept b_d such that the hyperplane given by $X_d = b_1x_1 + ... + b_{d-1}X_{d-1} + b_d$ intersects the polyhedron. □

Lemma 8.3 *Let $UP^-(P) = \{(p_1, ..., p_d) | (p_1, ..., p_d) \in UP(P)$ and $p_d = min\{p'_d | (p_1, ..., p_{d-1}, p'_d) \in UP(P)\}$. Let $DOWN^-(P) = \{(p_1, ..., p_d) | (p_1, ..., p_d) \in UP(P)$ and $p_d = max\{p'_d | (p_1, ..., p_{d-1}, p'_d) \in DOWN(P)\}$. Then, $(p_1, ..., p_d) \in UP^-(P)$ iff $TOP^P(p_1, ..., p_{d-1}) = p_d$ and $(p_1, ..., p_d) \in DOWN^-(P)$ iff $BOT^P(p_1, ..., p_{d-1}) = p_d$.* □

Lemma 8.4 *Let P be a polyhedron. The following facts hold:*

1. *All points contained in $UP(P) \cup DOWN(P)$ represent in the primal plane hyperplanes that do not intersect P or are supporting with respect to P.*

2. *All points not contained in $UP(P) \cup DOWN(P)$ represent in the primal plane hyperplanes that intersect P but are not supporting with respect to P.* □

By using the previous results, it is possible to prove the following theorem.

Theorem 8.1 *For all points $(X_1, ..., X_{d-1}) \in E^d$:*

$$TOP^P(X_1, ..., X_{d-1}) = \begin{cases} X_d & \text{if } (X_1, ..., X_d) \in UP^-(P) \\ +\infty & \text{otherwise} \end{cases}$$

$$BOT^P(X_1, ..., X_{d-1}) = \begin{cases} X_d & \text{if } (X_1, ..., X_d) \in DOWN^-(P) \\ -\infty & \text{otherwise} \end{cases}$$ □

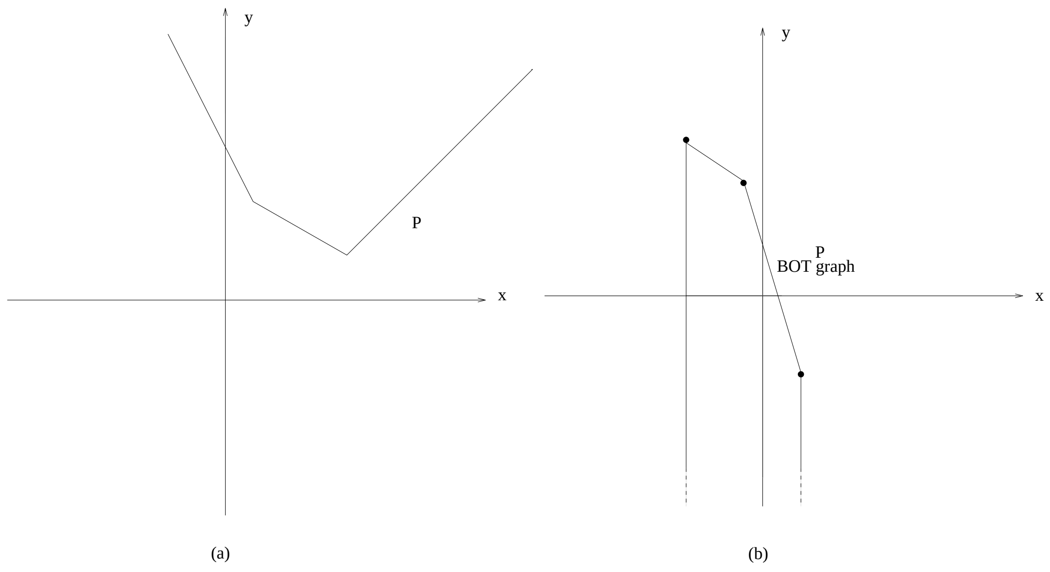


Figure 8.1: An upward open polyhedron: (a) in the primal plane; (b) in the dual plane.

The following examples show the dual construction for open and closed polyhedra in E^2.

Example 8.2 *Figures 8.1, 8.2, 8.3, and 8.4 present some examples of dual representations. In each figure, the polyhedron is represented in (a) and the corresponding dual representation is represented in (b). Note that:*

- *In Figure 8.1 only $DOWN(P)$ is generated, since no 0-half-plane is used in defining P.*
- *In Figure 8.2 only $UP(P)$ is generated, since no 1-half-planes is used in defining P.*
- *In Figure 8.3, both $UP(P)$ and $DOWN(P)$ are generated. Since the polyhedron is closed, no vertical hyperplanes have been added.*
- *In Figure 8.4, both $UP(P)$ and $DOWN(P)$ are generated. Since the polyhedron is open, two vertical hyperplanes have been added.* $\diamond$

In [67], it has been shown how $TOP^P(b_1, ..., b_{d-1})$ and $BOT^P(b_1, ..., b_{d-1})$ can be computed without constructing $UP(P)$ and $DOWN(P)$. Consider the computation

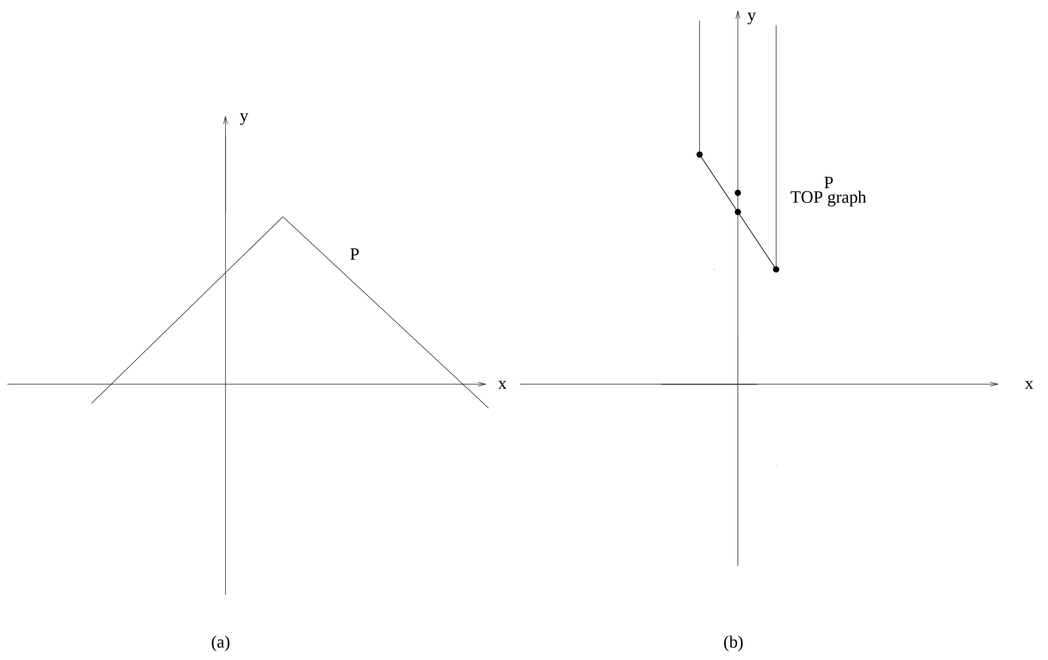


Figure 8.2: A downward open polyhedron: (a) in the primal plane; (b) in the dual plane.

of $TOP^P(b_1, ..., b_{d-1})$ ($BOT^P(b_1, ..., b_{d-1})$ can be similarly generated). The projection of the TOP^P graph on the $(d-1)$- dimensional hyperplane $J : b_d = 0$ subdivides J into no more than ${n_v}^4$ convex $(d-1)$-dimensional polyhedral partitions with no more than $O(n_v^2)$ $(d-2)$-dimensional boundary segments. Any given partition $E \subseteq J$ corresponds to a vertex $v(E)$ of P upper hull, such that for any point $(p_1, ..., p_{d-1}) \in E$ $TOP^P(p_1, ..., p_{d-1}) = F_{D(v(E))}(p_1, ..., p_{d-1})$. Hence, $TOP^P(b_1, ..., b_{d-1})$ can be obtained by a $(d-1)$-dimensional point location in J to find the partition E that contains the point $(b_1, ..., b_{d-1})$, followed by a computation of $F_{D(v(E))}(b_1, ..., b_{d-1})$. The complexity of this operation depends on the complexity of computing $F_{D(v(E))}(b_1, ..., b_{d-1})$ and the complexity to perform point location (see Section 8.3).

[4]We recall that n_v denotes the number of vertices of P.

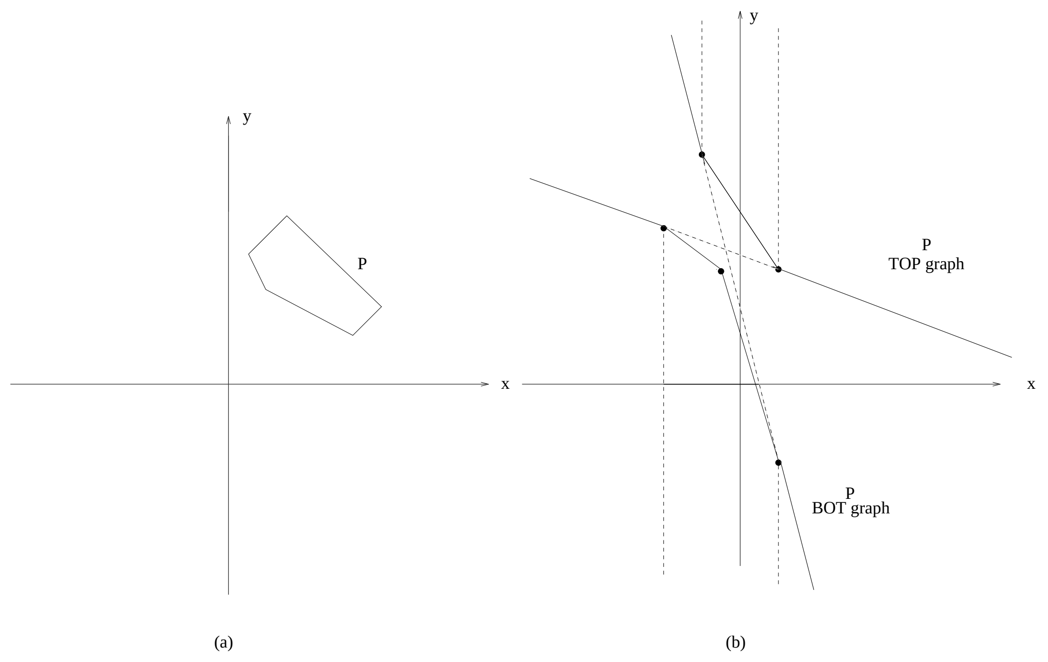


Figure 8.3: A closed polyhedron: (a) in the primal plane; (b) in the dual plane.

8.3 Intersection and containment with respect to a half-plane

As we have seen, functions TOP^P and BOT^P map any slope $(b_1, ..., b_{d-1})$ of a non-vertical hyperplane into the maximum (TOP^P) or the minimum (BOT^P) intercept b_d such that the hyperplane $X_d = b_1x_1 + ... + b_{d-1}X_{d-1} + b_d$ intersects the polyhedron. The following result is a direct consequence of this fact.

Theorem 8.2 *Let P be a polyhedron in E^d.*

- *A hyperplane $X_d = b_1x_1 + ... + b_{d-1}X_{d-1} + b_d$ intersects P iff $BOT^P(b_1, ..., b_{d-1}) \leq b_d \leq TOP^P(b_1, ..., b_{d-1})$.*
- *A half-plane $X_d \geq b_1X_1 + ... + b_{d-1}X_{d-1} + b_d$ intersects P iff $b_d \leq TOP^P(b_1, ..., b_{d-1})$.*
- *A half-plane $X_d \leq b_1X_1 + ... + b_{d-1}X_{d-1} + b_d$ intersects P iff $b_d \geq BOT^P(b_1, ..., b_{d-1})$.*

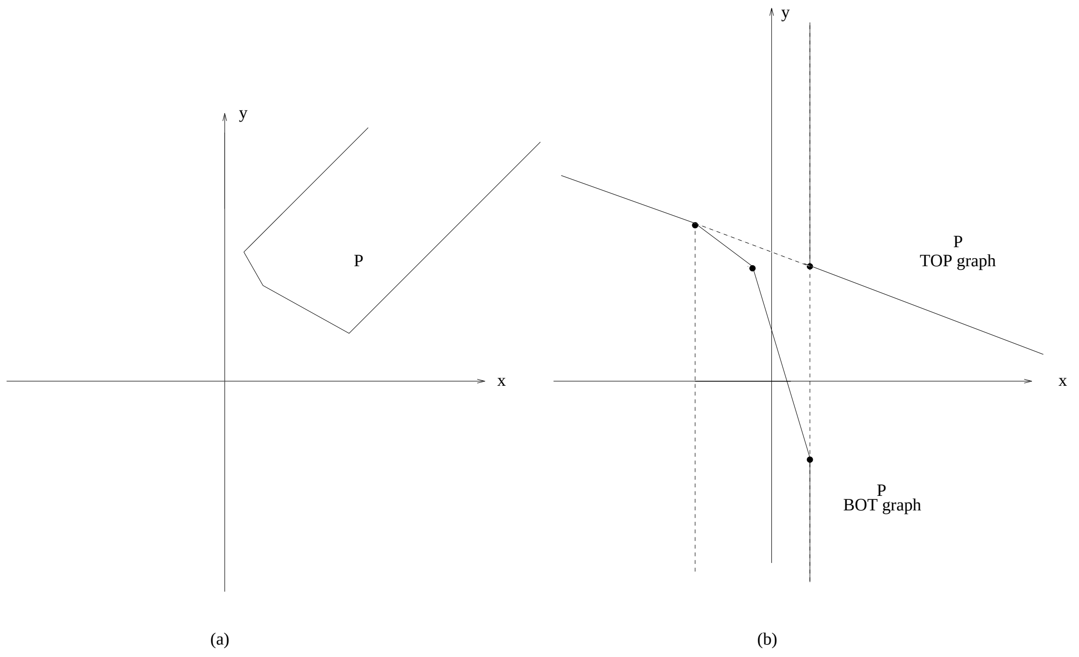


Figure 8.4: An open polyhedron: (a) in the primal plane; (b) in the dual plane.

- *A half-plane* $X_d \geq b_1X_1 + ... + b_{d-1}X_{d-1} + b_d$ *contains* P *iff* $b_d \leq BOT^P(b_1, ..., b_{d-1})$.
- *A half-plane* $X_d \leq b_1X_1 + ... + b_{d-1}X_{d-1} + b_d$ *contains* P *iff* $b_d \geq TOP^P(b_1, ..., b_{d-1})$. □

By considering generalized tuples instead of polyhedra, from Theorem 8.2, we obtain the following result.

Corollary 8.1 *Let* t_P *be a generalized tuple. Let* $q(\theta)$ *be the query generalized tuple* $X_d\ \theta\ b_1X_1 + ... + b_{d-1}X_{d-1} + b_d$, *where* $\theta \in \{\geq, \leq\}$. *Then:*

- $All(q(\geq), t_P)$ *iff* $b_d \leq BOT^P(b_1, ..., b_{d-1})$;
- $All(q(\leq), t_P)$ *iff* $b_d \geq TOP^P(b_1, ..., b_{d-1})$;
- $Exist(q(\geq), t_P)$ *iff* $b_d \leq TOP^P(b_1, ..., b_{d-1})$;
- $Exist(q(\leq), t_P)$ *iff* $b_d \geq BOT^P(b_1, ..., b_{d-1})$;

d	**Preprocessing**	**Space**	**Time**
$d = 2$	$O(\log n_v)$	$O(n_v)$	$O(n_v)$
$d = 3$	$O(\log n_v)$	$O(n_v)$	$O(n_v)$
$d > 3$	$O(2^d \log n_v)$	$O(n_v^{2^d})$	$O(n_v^{2^d})$

Table 8.1: Summary of the complexity results for the intersection and the containment problems between a polyhedron and a half-plane. In the table, n_v identifies the number of vertices of the considered polyhedron.

- $Exist_e(q(\geq), t_P)$ *iff* $BOT^P(b_1, ..., b_{d-1}) < b_d \leq TOP^P(b_1, ..., b_{d-1})$;
- $Exist_e(q(\leq), t_P)$ *iff* $BOT^P(b_1, ..., b_{d-1}) \leq b_d < TOP^P(b_1, ..., b_{d-1})$. □

The complexity of the previous problems is bound by the complexity of computing $TOP^P(b_1, ..., b_{d-1})$ and $BOT^P(b_1, ..., b_{d-1})$, thus of performing point location. The point location problem has good in-memory algorithms for the 2-dimensional space [53, 118]. However, solutions for higher d-dimensional space are not so efficient, especially for space complexity. Table 8.1, taken from [67], summarizes such complexity results (see [67] for more details).

Example 8.3 *Consider the polyhedron presented in Figure 8.1. Consider the 1-half-planes $q_1 \equiv Y \geq -X-1$ and $q_2 \equiv Y \geq 5$. Figure 8.1(b) shows that $-1 \leq BOT^P(-1)$ and $BOT^P(0) < 5 < TOP^P(0)$. According to Corollary 8.1, it means that $All(q_1, t)$ and $Exist_e(q_2, t)$ are satisfied. Figure 8.1(a) confirms the results. If we instead consider the 0-half-planes $q'_1 \equiv Y \leq -X - 1$ and $q'_2 \equiv Y \leq 5$, we obtain from Corollary 8.1 that only the selection $Exist_e(q'_2, t)$ is satisfied. The correctness of this result can be observed in Figure 8.1(a).*

Now consider the polyhedron presented in Figure 8.3. Given the 1-half-planes $q_1 \equiv Y \geq -X-1$, $q_2 \equiv Y \geq 5$, $q_3 \equiv Y \geq 4.5$, $q_4 \equiv Y \geq X$, we can see in Figure 8.3(b) that $-1 < BOT^P(-1)$, $5 > TOP^P(0)$, $4.5 = TOP^P(0)$ and $BOT^P(1) < 0 < TOP^P(1)$. It follows from Corollary 8.1 that $All(q_1, t)$, $Exist_e(q_3, t)$, and $Exist_e(q_4, t)$ are satisfied. Figure 8.3(a) shows that this result is correct. If we consider the up-queries $q'_1 \equiv Y \leq -X - 1$, $q'_2 \equiv Y \leq 5$, $q'_3 \equiv Y \leq 4.5$ and $q_4 \equiv Y \leq x$, from Corollary 8.1 it follows that $All(q'_2, t)$, $All(q'_3, t)$, and $Exist_e(q'_4, t)$ are satisfied. Figure 8.3(a) confirms the results. ◇

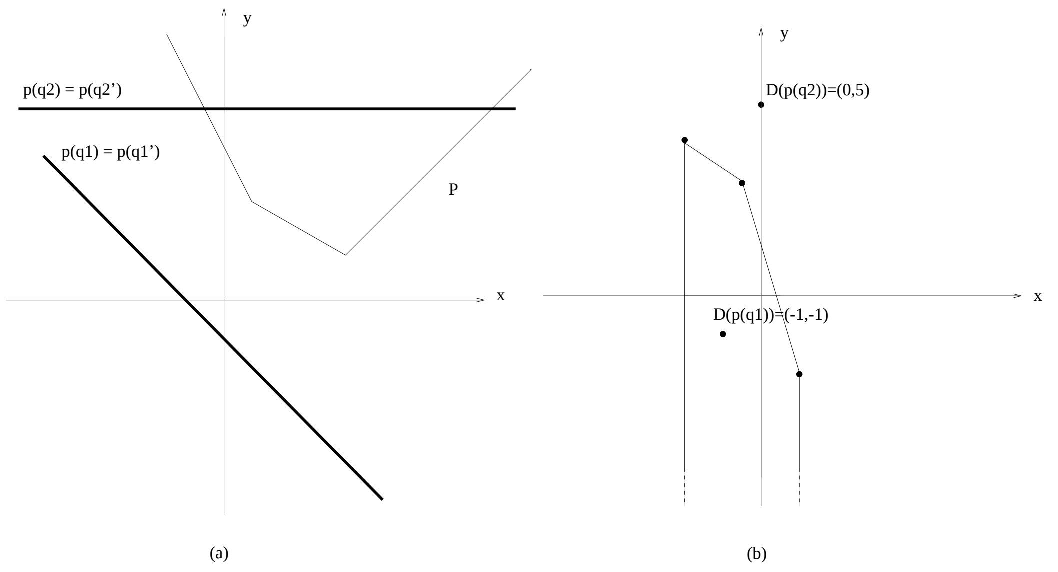


Figure 8.5: A downward open polyhedron and some query half-planes: (a) in the primal plane $\mathcal{P}$; (b) in the dual plane $\mathcal{D}$.

8.4 Intersection and containment between two polyhedra

Two convex polyhedra P and Q do not intersect if and only if there is a separating non-vertical hyperplane between them. Any such hyperplane H does not intersect either P or Q but there are hyperplanes H' and H'', parallel to H, such that H' is above H, H'' is below H, H' intersects P, and H'' intersects Q (see Figure 8.7 for a 2-dimensional example). The formal definition of separating hyperplane is the following.

Definition 8.1 *[67] A non-vertical hyperplane $X_d = b_1x_1 + ... + b_{d-1}X_{d-1} + b_d$ separates two polyhedra P and Q if and only if*

$$TOP^P(b_1,...,b_{d-1}) < b_d < BOT^Q(b_1,...,b_{d-1}) \text{ or}$$
$$TOP^Q(b_1,...,b_{d-1}) < b_d < BOT^P(b_1,...,b_{d-1}). \qquad \square$$

On the other hand, a polyhedron P is contained in a polyhedron Q if any hyperplane intersecting P also intersects Q. From these considerations and Definition 8.1, the following theorem holds.

Theorem 8.3 *Let P and Q be two polyhedra in E^d.*

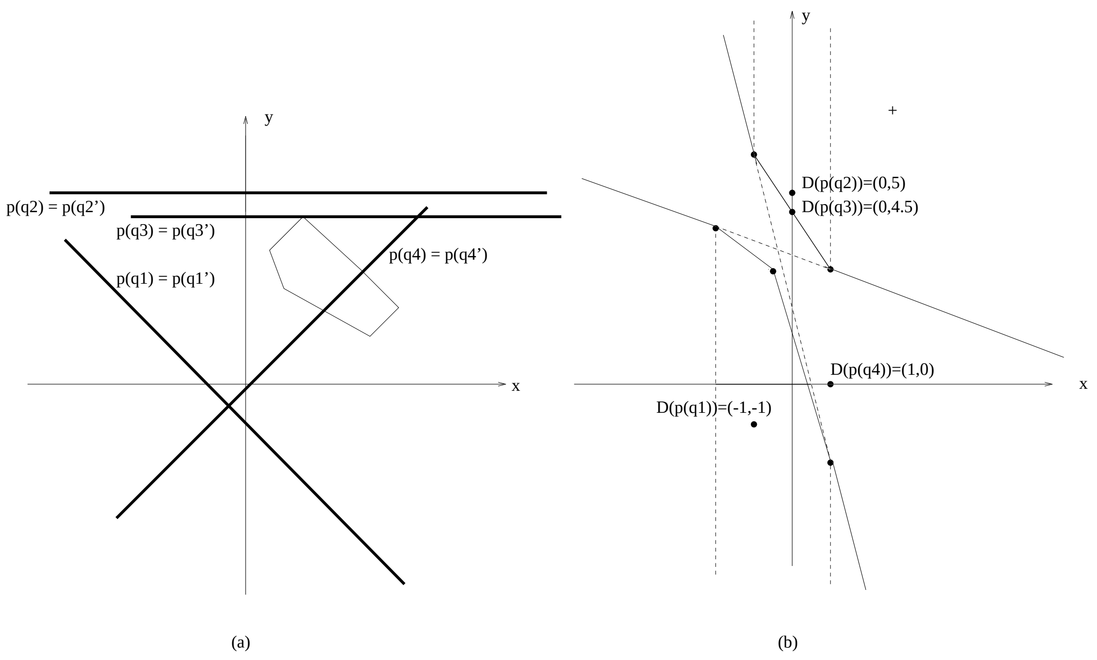


Figure 8.6: A closed polyhedron and some query half-planes: (a) in the primal plane $\mathcal{P}$; (b) in the dual plane $\mathcal{D}$.

- *P intersects Q iff* $min_{(X_1,...,X_{d-1})\in E^{d-1}}\{TOP^P(X_1,...,X_{d-1}) - BOT^Q(b_1,...,b_{d-1})\} \geq 0$ *and* $min_{(X_1,...,X_{d-1})\in E^{d-1}}\{TOP^Q(X_1,...,X_{d-1}) - BOT^P(b_1,...,b_{d-1})\} \geq 0$.
- *P is contained in Q iff for all* $(X_1,...,X_{d-1}) \in E^{d-1}$, $TOP^Q(X_1,...,X_{d-1}) \geq TOP^P(X_1,...,X_{d-1})$ *and* $BOT^Q(X_1,...,X_{d-1}) \leq BOT^P(X_1,...,X_{d-1})$. □

From the previous theorem it also follow that P is contained in Q iff $UP(Q) \subseteq UP(P)$ and $DOWN(Q) \subseteq DOWN(P)$ (see Figure 8.8). Moreover, P intersects Q iff $UP(P) \cap DOWN(Q) = \emptyset$ and $DOWN(P) \cap UP(Q) = \emptyset$.

Corollary 8.2 *Let P and Q be two polyhedra. Then:*

- *$All(t_P, t_Q)$ iff for all* $(X_1,...,X_{d-1})$, $TOP^Q(X_1,...,X_{d-1}) \geq TOP^P(X_1,...,X_{d-1})$ *and* $BOT^Q(X_1,...,X_{d-1}) \leq BOT^P(X_1,...,X_{d-1})$.

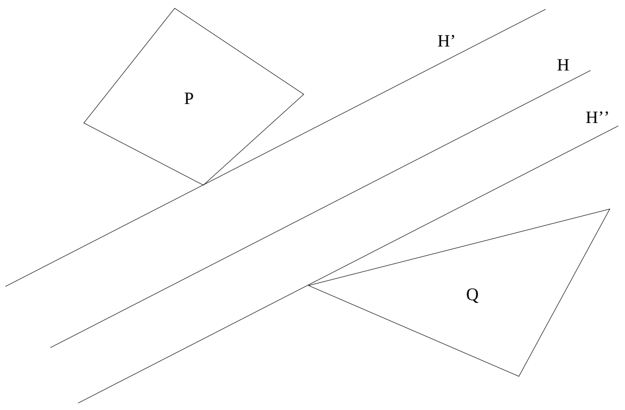


Figure 8.7: An example of separating hyperplane.

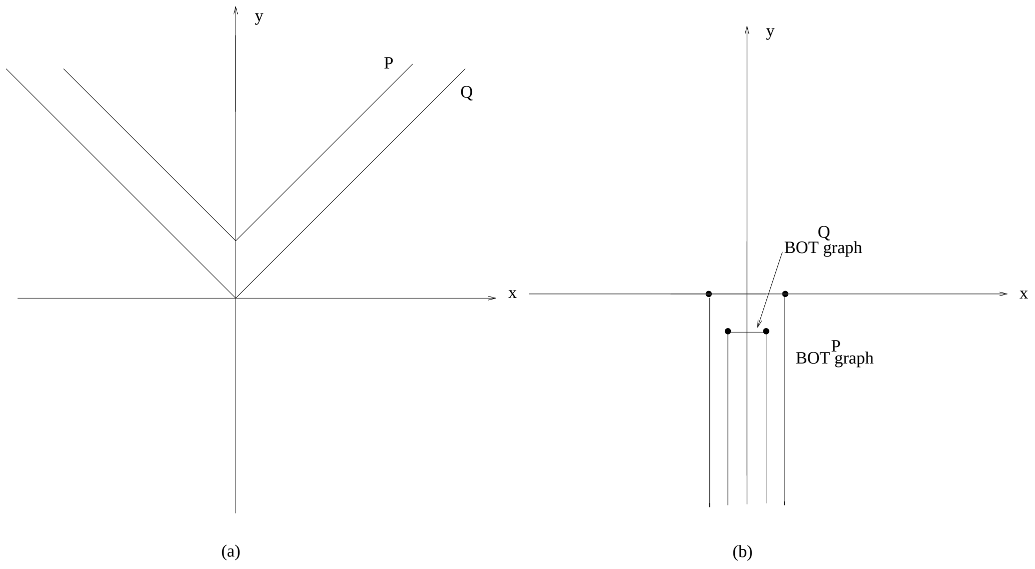


Figure 8.8: Containment between two polyhedra: (a) in the primal plane; (b) in the dual plane.

d	**Preprocessing**	**Space**	**Time**
$d = 2$	$O(\log n_v)$	$O(n_v)$	$O(n_v)$
$d = 3$	$O(\log^2 n_v)$	$O(n_v^2)$	$O(n_v^2)$
$d > 3$	$O((2d)^{d-1}\log^{d-1} n_v)$	$O(n_v^{2^d})$	$O(n_v^{2^d})$

Table 8.2: Summary of the complexity results for the intersection and the containment problems between two polyhedra. In the table, n_v identifies the number of vertices of the considered polyhedron.

- $Exist(t_P, t_Q)$ *iff*
  $min_{(X_1,...,X_{d-1}) \in E^{d-1}}\{TOP^P(X_1, ..., X_{d-1}) - BOT^Q(b_1, ..., b_{d-1})\} \geq 0$ *and*
  $min_{(X_1,...,X_{d-1}) \in E^{d-1}}\{TOP^Q(X_1, ..., X_{d-1}) - BOT^P(b_1, ..., b_{d-1})\} \geq 0.$ □

Table 8.2 summarizes the complexity bounds for the problems introduced above.

8.5 Secondary storage solutions for half-plane queries

In the previous section, we have introduced the basic dual transformation for polyhedra (and therefore for generalized tuples) and shown its main properties. In the following, we show how indexing techniques supporting ALL and EXIST selections with respect to a half-plane can be defined, based on this representation. In particular, we first consider a restriction of this problem, then we relax the considered hypothesis and we discuss the general case.

8.5.1 A secondary storage solution for a weaker problem

In the following, we consider a restricted type of ALL and EXIST selections. We assume that, given a query half-plane $X_d\ \theta\ b_1X_1+...+b_{d-1}X_{d-1}+b_d$, point $(b_1, ..., b_{d-1})$ belongs to a predefined set S (note that point $(b_1, ..., b_{d-1})$ is the normal vector of $X_d\ \theta\ b_1X_1 + ... + b_{d-1}X_{d-1} + b_d$ [118]). This assumption allows us to precompute TOP^P and BOT^P values for specific points.

Under the previous hypothesis, due to Corollary 8.1, in order to check intersection and containment between a set of polyhedra and $X_d\ \theta\ b_1X_1 + ... + b_{d-1}X_{d-1} + b_d$, it is sufficient to maintain two sets of values. Given a generalized relation r, for each generalized tuple $t_P \in r$, the first set contains value $TOP^P(b_1, ..., b_{d-1})$ whereas the second set contains value $BOT^P(b_1, ..., b_{d-1})$. Since both sets of points are totally ordered, they can be organized in two lists, denoted by L_{BOT} and L_{TOP}. We assume that the lists are ordered with respect to their increasing values. Given a query generalized tuple $q(\theta) \equiv X_d \ \theta \ b_1X_1 + ... + b_{d-1}X_{d-1} + b_d$, it is easy to see that the position of b_d in the total order determines the result of the query. Indeed:

- ALL$(q(\geq), r)$ is represented by all the generalized tuples associated with points following or equal to b_d in L_{BOT}.
- ALL$(q(\leq), r)$ is represented by all the generalized tuples associated with points preceding or equal to b_d in L_{TOP}.
- EXIST$(q(\geq), r)$ is represented by all the generalized tuples associated with points following or equal to b_d in L_{TOP}.
- EXIST$(q(\leq), r)$ is represented by all the generalized tuples associated with points preceding or equal to b_d in L_{BOT}.

Note that a similar solution cannot be applied to $EXIST_e$ queries.

From the previous discussion, it follows that, in order to perform selections against a set of generalized tuples in secondary storage, it is sufficient to maintain, for each point in S, two ordered sets of values. B$^+$-trees can be used to this purpose.

Given the query $E(X_d \ \theta \ b_1X_1 + ... + b_{d-1}X_{d-1} + b_d, r)$, where r is a generalized relation and $E \in \{$ALL,EXIST$\}$, the search algorithm first selects the B$^+$-tree associated with point $(b_1, ..., b_{d-1})$; then, value b_d is searched in such a B$^+$-tree.

Another solution to the same problem can be provided by reducing ALL and EXIST selection problems to the 1-dimensional interval management problem. This solution homogeneously supports ALL, EXIST, and EXIST$_e$ selections.

The reduction is based on the following considerations. Given a half-plane $q(\theta) \equiv X_d \ \theta \ b_1X_1 + ... + b_{d-1}X_{d-1} + b_d$, any tuple t_P can be associated with three intervals $]-\infty, BOT^t(b_1, ..., b_{d-1})[,]BOT^t(b_1, ..., b_{d-1}), TOP^t(b_1, ..., b_{d-1})[$, and $]TOP^t(b_1, ..., b_{d-1}), +\infty[$. By Corollary 8.1, if $(b_1, ..., b_{d-1}) \in S$, we have the following cases:

- if value b_d belongs to the interval $]-\infty, BOT^P(b_1, ..., b_{d-1})]$, predicate $All(q(\geq), t)$ is satisfied;
- if $b_d \in [TOP^P(b_1, ..., b_{d-1}), +\infty[$, predicate $All(q(\leq), t)$ is satisfied;
- if $b_d \in [BOT^P(b_1, ..., b_{d-1}), \ TOP^P(b_1, ..., b_{d-1})[$, predicate $Exist_e(q(\leq), t)$ is satisfied;
- if $b_d \ \in]BOT^P(b_1, ..., b_{d-1}), TOP^P(b_1, ..., b_{d-1})]$, predicate $Exist_e(q(\geq), t)$ is satisfied;

- if $b_d \in]-\infty, TOP^P(b_1,...,b_{d-1})]$, $Exist(q(\geq),t)$ is satisfied;
- if $b_d \in [BOT^P(b_1,...,b_{d-1}), +\infty[$, $Exist(q(\leq),t)$ is satisfied.

Thus, to perform selections against a set of generalized tuples, it is sufficient to maintain three 1-dimensional interval sets for each value in set S. Management of 1-dimensional intervals is a classic problem from computational geometry [53, 118]. An optimal solution to the problem in secondary storage has been recently proposed in [7]. It requires linear space and logarithmic time for query and update operations applied on a set of N intervals.

The next result follows from the previous discussion.

Theorem 8.4 *Let r be a generalized relation containing N generalized tuples. Let $q \equiv X_d\ \theta\ b_1X_1 + ... + b_{d-1}X_{d-1} + b_d$ be a query half-plane. Let T be the cardinality of the set $ALL(q,r)$ (respectively $EXIST_e(q,r)$ and $EXIST_e(q,r)$). If $(b_1,...,b_{d-1})$ is contained in a predefined set of cardinality k, there is an indexing structure for storing r in $O(k\ N/B)$ pages such that $ALL(q,r)$, $EXIST_e(q,r)$, and $EXIST(q,r)$ selections are performed in $O(\log_B N/B + T/B)$ time, and generalized tuple update operations are performed in $O(k\ \log_B N/B)$ time.* $\square$

8.5.2 Secondary storage solutions for the general problem

If $(b_1,...,b_{d-1}) \notin S$, as we have seen in Section 8.2, a d-dimensional point location must be performed in order to compute $TOP^P(b_1,...,b_{d-1})$ and $BOT^P(b_1,...,b_{d-1})$.

Consider for example the computation of $TOP^P(b_1,...,b_{d-1})$. A point location has to be performed with respect to the partition of the $(d-1)$-dimensional hyperplane $J: b_d = 0$, induced by the open polyhedra $UP(P)$'s. If the considered generalized relation contains N generalized tuples and if each generalized tuple is composed of at most k constraints (thus, the corresponding polyhedron has at most k vertices), $J : b_d = 0$ is decomposed in at most N^2k^2 partitions. Indeed, the projection of each polyhedron divides J into no more than k convex $(d-1)$-dimensional polyhedral partitions with no more than $O(k^2)$ $(d-2)$-dimensional boundary segments. Therefore, by combining together N different partitions, J is divided in at most N^2k^2 partitions. Any given partition E corresponds to a specific order of polyhedra. This means that, given a partition, any line parallel to the d-th axis and passing through E, intersects the N polyhedra in the same order. Moreover, any given partition E corresponds to a specific vertex of the upper hull of each polyhedron.

From the previous discussion it follows that one obvious way to define a data structure to answer a general half-plane query requires maintaining one B$^+$-tree for each

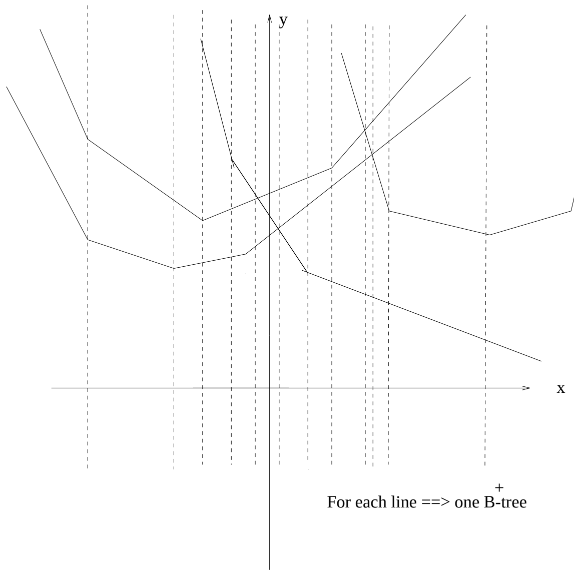


Figure 8.9: The basic idea of the indexing technique in the general indexing case.

given partition of J, induced by $UP(P)'s$, and one B$^+$-tree for each given partition of J, induced by $DOWN(P)'s$. Then, given a half-plane $X_d\ \theta\ b_1X_1 + ... + b_{d-1}X_{d-1} + b_d$, a point location is performed to determine the partition containing point $(b_1, ..., b_{d-1})$; the corresponding B$^+$-tree is then used to answer the query.

The previous solution, though very simple, cannot be considered satisfactory. Indeed, N^2k^2 data structures have to be maintained (see Figure 8.9 for a 2-dimensional example). This also means that the use of the dual representation does not allow the efficient indexing of generalized tuples to perform selections with respect to an arbitrary half-plane.

A possible solution to this problem is to apply a filtering/refinement approach when $(b_1, ..., b_{d-1}) \not\in S$. In the remainder of this chapter, we propose three different solutions:

- The first technique, denoted by T1, (see Section 8.6) replaces the original half-plane query with two new half-plane queries. The union of the results of the two new queries is a superset of the generalized tuples belonging to the result of the original query. This technique can always be applied but, besides the generation of false hits, due to the fact that the results of the two queries may not be disjoint, some generalized tuples may be returned twice.

  T1 is based on the B$^+$-tree data structure presented in Section 8.5.1.

- The second technique, denoted by T2, (see Section 8.7) replaces the original half-plane query with a new half-plane query. This technique can be applied only if the database satisfies some specific properties but, since only one new query is executed, no duplicates are generated.

  T2 is based on the B$^+$-tree data structure presented in Section 8.5.1.

- The third technique, denoted by T3, (see Section 8.8) reduces the original problem to a new problem. Specific solutions to the new problem are presented and used to answer the original query. No duplicates are generated, however new data structures have to be used.

In the following, these solutions will be described. To simplify the notation, each solution is presented for the 2-dimensional case and then extended to deal with arbitrary dimensions. Moreover, we denote with S a set of angular coefficients and we assume that, given a query half-plane $X_d\ \theta\ b_1X_1+...+b_{d-1}X_{d-1}+b_d$, $(b_1,...,b_{d-1}) \notin S$. We call *up-query* a query with respect to a 0-half-plane and *down-query* a query with respect to a 1-half-plane.

8.6 Approximating the query with two new queries

Consider the query $E(Y\ \theta\ aX + b, r)$, such that $E \in \{\text{ALL,EXIST}\}$. The simplest way to approximate an arbitrary half-plane query is to replace the query with two new queries such that the angular coefficients associated with the new query half-planes are contained in S. The evaluation of the new queries must retrieve at least all the generalized tuples that would be generated by the evaluation of $Y\ \theta\ aX + b$. This is achieved by replacing the original query half-plane with two new query half-planes such that the region of space totally covered by the two new half-planes contains the region of space covered by the original half-plane. This guarantees that each tuple belonging to the result of the original query also belongs to the result of at least one new query, ensuring the safety of the approximation (see Figure 8.10).

There are two main issues in applying such an approach:

- The results returned by the two new queries may not be disjoint. Thus, some generalized tuples may be returned twice (see Figure 8.10); this means that some *duplicates* can be generated.

- Not all the generalized tuples returned by the evaluation of the two queries satisfy the original query, thus some *false hits* can be generated (see Figure 8.10) and a refinement step should be applied in order to remove them from the result.

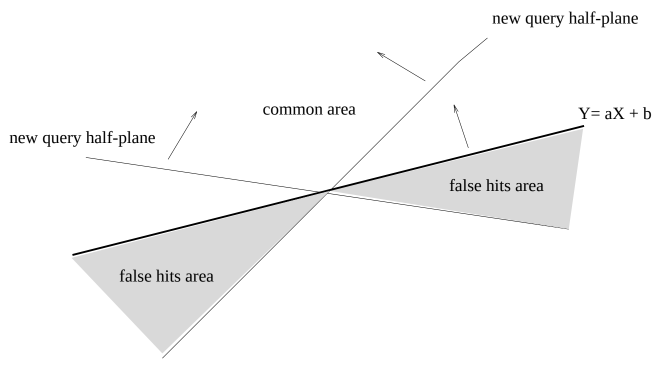


Figure 8.10: An example of safe approximation.

The number of duplicates and false hits depends on the choice of the new queries. Indeed, there exist infinite pairs of new queries (a finite number for each point of line $Y = aX + b$) retrieving a superset of the result. However different pairs may generate more or less duplicates and false hits.

The determination of the new queries depends on the following choices:

1. *Choice of the angular coefficients.* First, the angular coefficients a' and a'' of the lines associated with the new query half-planes must be determined in order to reduce duplicates or false hits. We choose to reduce the number of false hits.

2. *Choice of the line.* The equation of the line associated with the new query half-plane must be determined. This is possible by choosing a point P on the line associated with the original query half-plane and then determining the lines passing through P and having a' and a'' as angular coefficients.

3. *Choice of the half-plane.* Each line must be transformed in a half-plane, partially covering the original one.

4. *Choice of the type of the query.* A type (ALL or EXIST) for the new queries must be specified, ensuring the safety of the approximation.

In the following, we present solutions to all these problems.

Choice of the angular coefficients. We assume that S contains the angular coefficients of k lines, dividing the two-dimensional space in $2k$ sectors (no vertical line is permitted – see Section 8.2 –). Different choices may lead to the generation of different sets of false hits and duplicates.

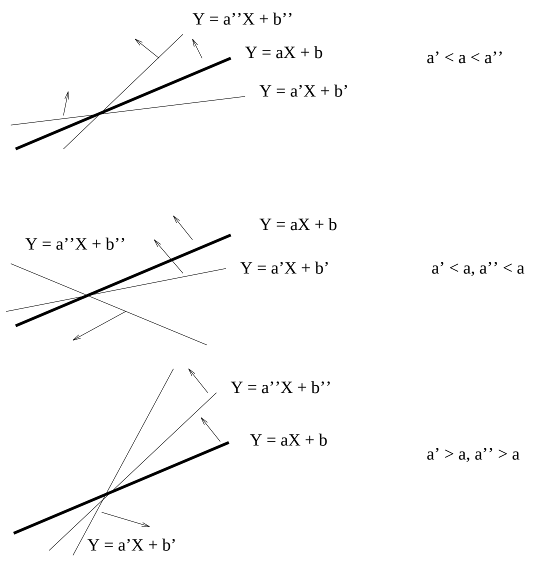


Figure 8.11: Choice of the new query half-planes, for an original down-query.

In order to reduce the number of generated false hits, the coefficients a' and a'' of the lines associated with the new query half-planes must be the angular coefficients of the lines representing the border of the sector in which the line $Y = aX + b$ is contained. Note that this heuristic allows minimizing the area corresponding to the difference between the space covered by the new queries and the space covered by the original one (also called *false hits area*). In most cases, this choice results in the generation of a higher number of duplicates and a lower number of false hits. This approach seems to be more reliable for large databases. Indeed, the generation of fewer duplicates results in a greater number of false hits, and all the database objects may be selected.

We denote by a' the angular coefficient of the first line which is encountered by performing a clockwise rotation of line $Y = aX + b$ and with a'' the angular coefficient of the first line which is encountered by performing an anti-clockwise rotation of line $Y = aX + b$.

Choice of the lines. Given the angular coefficients a' and a'', the lines can be determined by choosing a point P on line $Y = aX + b$. If $P = (\overline{x}, \overline{y})$, the lines have the following equations:

- $Y = a'X + (\overline{y} - a'\overline{x})$

Conditions on a, a', a''	Values for θ' and θ''
$a' < a < a''$	$\theta' \equiv \theta$, $\theta'' \equiv \theta$
$a' < a, a'' < a$	$\theta' \equiv \theta$, $\theta'' \equiv \neg\theta$
$a < a', a < a''$	$\theta' \equiv \neg\theta$, $\theta'' \equiv \theta$

Table 8.3: Choice of the query half-planes.

- $Y = a''X + (\overline{y} - a''\overline{x})$.

Different choices of P may lead to different distributions of false hits.

Choice of the half-planes. Given the lines constructed as above, we should decide which half-plane queries must be considered. The union of the points belonging to the two new half-planes must cover the space already covered by the original half-plane. The new query half-planes are given by:

- $Y\ \theta'\ a'X + (\overline{x} - a'\overline{y})$
- $Y\ \theta''\ a''X + (\overline{y} - a''\overline{x})$

where θ' and θ'' are presented in Table 8.3, for each combination of a, a' and a''. In the tables, $\neg\theta$ corresponds to '$\leq$' if θ is '$\geq$' and to '$\geq$' if θ is '$\leq$'. Figure 8.11 graphically explains these choices for an original down-query.

Choice of the type of the new queries. A type must be assigned to each half-plane query constructed as above. The choice is the following:

- *The original query is* EXIST. If the original query is approximated with two new EXIST queries, the approximation is safe, because each generalized tuple satisfying the original query is returned by at least one new query.
- *The original query is* ALL. If we substitute the original ALL query with two new ALL queries, some generalized tuples satisfying the original query may not be returned by the union of the results of the two queries; this happens, for example, if at least one generalized tuple exists such that its extension is contained in the original half-plane but it is not contained in any of the new half-planes (see Figure 8.12). A possible solution is to approximate the ALL query with an EXIST and an ALL query. In such a case, the approximation is safe. Indeed, if a generalized tuple does not satisfy an EXIST query with respect to the half-plane $Y \geq a''X + b''$ (see Figure 8.12), it must be contained in the opposite half-plane $Y \leq a''X + b''$. Due to the original query, we are

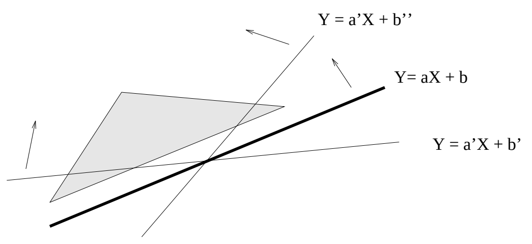


Figure 8.12: An example of unsafe approximation.

interested only in generalized tuples whose extension is contained in the sector on the right of the intersection point of lines $Y = aX + b$ and $Y = a''X + b''$. This set of generalized tuples can be approximated by evaluating an ALL query with respect to the half-plane $Y \geq a'X + b'$.

From the previous considerations, the following result holds.

Proposition 8.2 *Let $E(q)$ be a query, such that $E \in \{ALL,EXIST\}$. Let P be a point of line $p(q)$. Let $E_1(q_1)$, $E_2(q_2)$ be the two new queries constructed as above with respect to P, such that $E_1, E_2 \in \{ALL,EXIST\}$. Then, for each generalized relation r, $E(q,r) \subseteq E_1(q_1,r) \cup E_2(q_2,r)$.* □

The previous result ensures that the proposed approximation is safe. Since the angular coefficients of the new query half-planes belong to S, the technique presented in Subsection 8.5.1 can be used to execute the corresponding queries. Thus, we obtain the following result.

Theorem 8.5 *Let r be a generalized relation containing N generalized tuples. Let q be a query half-plane. Let T be the cardinality of the set $ALL(q,r)$ $(EXIST(q,r))$. If the angular coefficient of $p(q)$ is not contained in a predefined set of cardinality k, there is an indexing structure for storing r in $O(k\ N/B)$ pages such that $ALL(q,r)$ and $EXIST(q,r)$ selections are performed in $O(\log_B N/B + T_1/B + T_2/B)$ time, where T_1 and T_2 represent the number of generalized tuples returned by the two new queries generated as above, and generalized tuple update operations are performed in $O(k\ \log_B N/B)$ time.* □

In the following, the technique introduced in this section is denoted by T1.

8.6.1 Extension to an arbitrary d-dimensional space

The proposed technique can be extended to deal with an arbitrary d-dimensional space as follows:

1. *Choice of the angular coefficients.* In this case, the ordering between angular coefficients (which are real numbers) has to be replaced with the ordering between the angle formed by two hyperplanes [118]. In particular, for each point $(b_1, ..., b_{d-1}) \in S$, we maintain the angle formed by the normal vector represented by this point and the normal vector of the hyperplane $X_d = 0$. We denote this order by $\preceq$. Also in this case we assume that S contains k points corresponding to hyperplanes that divide the space in $2k$ sectors of equal dimension.

2. *Choice of the hyperplanes.* In order to determine the hyperplanes supporting the new query half-plane, given their normal vectors, a $(d-1)$-hyperplane lying on the original query half-plane have to be chosen and the hyperplanes having the chosen normal vectors and passing through that hyperplane must be determined.

   By assuming that points in S characterize hyperplanes dividing the d-dimensional space in k equal sectors, all such hyperplanes intersect a given $(d-1)$-hyperplane l. Therefore, the technique can be applied only when the original query hyperplane is parallel to l. If this condition is not satisfied, the approximation cannot be applied. This is not true for the 2-dimensional case, where, given a $(2-1)$-hyperplane (thus, a point) and an angular coefficient in S, it is always possible to find the 2-dimensional hyperplane (thus, a line) characterized by such a coefficient and passing through that point.

3. *Choice of the half-plane.* Rules proposed for the 2-dimensional cases are still valid by replacing $\leq$ with $\preceq$.

4. *Choice of the type of the query.* Rules proposed for the 2-dimensional cases are still valid.

8.7 Approximating the query with a single new query

The technique proposed in Section 8.6 may generate duplicates and false hits. Duplicates are generated since the original query is approximated by two new queries, whose results may not be disjoint. In the following we give sufficient conditions to safely approximate the original query with a single new query. This approach eliminates duplicates. The new query has the same type of the original query. For this reason, in the following, we do not further specify the type of the query. We also show how point P should be chosen in order to reduce the number of generated false hits.

From Figure 8.10 we can see that, for an arbitrary choice of point P, each new half-plane does not completely cover the original half-plane, even if it has a non-empty

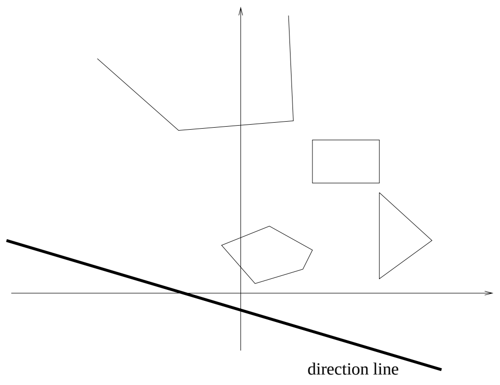


Figure 8.13: The extension of a generalized relation r and a direction line for r.

intersection with it. Moreover, each new half-plane covers the part of the original half-plane not covered by the other half-plane. From this consideration, it follows that the query can be approximated by using a single new query if it is possible to replace the original half-plane with a new half-plane such that the part of the original half-plane not covered by the new one does not contain the extension of any generalized tuple. This approximation can be applied if we know something about the distribution of the extensions of the generalized tuples in the plane. The notion of *direction half-plane*, introduced by the following definition, gives this kind of information, specifying that none is present in a given region of space (see Figure 8.13).

Definition 8.2 *Let r be a generalized relation. A line l is a direction line for r if $ext(r)$ is contained in a single half-plane with respect to l. Such half-plane is called direction half-plane for r.* □

Given a query $E(q)$, such that $E \in \{\text{ALL,EXIST}\}$, if at least one direction half-plane q_1 exists such that $p(q_1)$ and $p(q)$ are not parallel lines, at least one query $E(q')$ exists approximating the given one, such that $p(q)$ and $p(q')$ are not parallel lines. Note that if the original query line and the direction line are parallel, the direction line does not give enough information to find a new query approximating the original one, excluding queries whose query half-plane contains the direction half-plane (thus, retrieving all database objects). If $p(q)$ and $p(q_1)$ are not parallel lines, we say that q_1 is *approximating* for q.

Proposition 8.3 *Let r be a generalized relation. Let $E(q)$ be a query, such that $E \in \{ALL,EXIST\}$. Assume that there exists a direction half-plane q_1 for r, which*

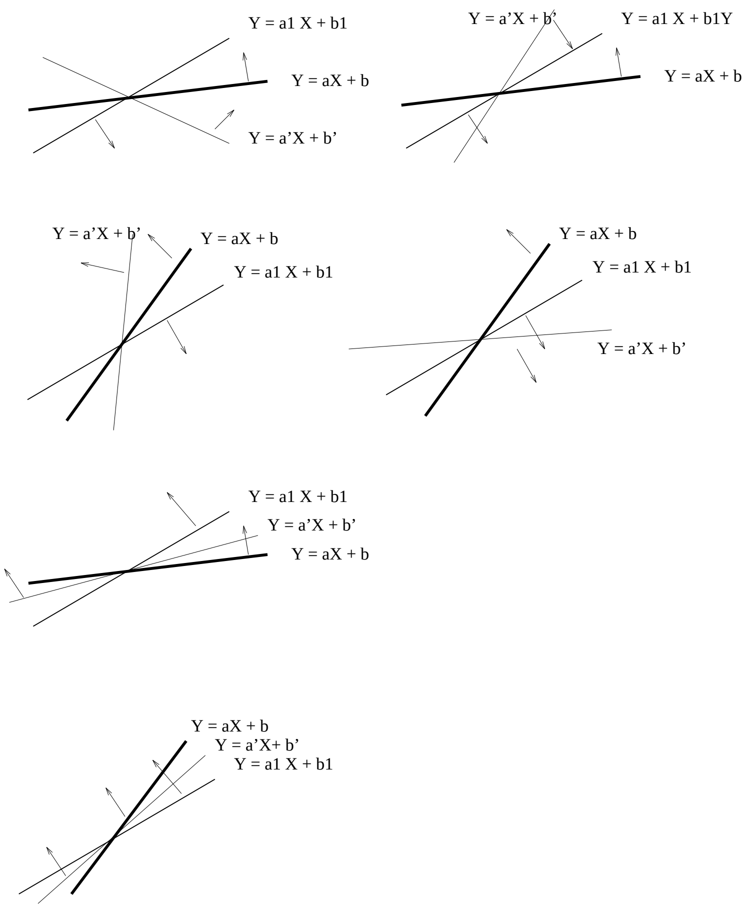


Figure 8.14: Choice of the new query half-plane, for an original down-query.

is approximating for q. Then, there exists at least one other query $E(q')$ such that $p(q')$ and $p(q)$ are not parallel lines and $E(q,r) \subseteq E(q',r)$. □

Given a query $E(Y\ \theta\ aX+b)$ and an approximating direction half-plane $Y\ \theta_1\ a_1X+b_1$, in order to construct the new query $Y\ \theta'\ a'X+b'$, as we have done in Subsection 8.6, the following choices must be taken:

1. *Choice of the point.* Since we assume that $a_1 \neq a$, the direction line and the query line $Y\ \theta\ aX+b$ intersect. We choose their intersection point to construct the new query. The reason for this choice is due to the fact that for each query half-plane, whose associated line passes through this point, it is immediate to

Conditions on a, a_1	θ_1	a'	θ'
$a_1 > a$	$\leq$	$max\{a' \mid a' \in S, a' < a\}$	$\geq$
		$min\{a' \mid a' \in S, a' \geq a_1\}$	$\leq$
$a_1 < a$	$\leq$	$min\{a' \mid a' \in S, a' > a\}$	$\geq$
		$max\{a' \mid a' \in S, a' \leq a_1\}$	$\leq$
$a_1 > a$	$\geq$	$max\{a' \mid a' \in S, a < a' \leq a_1\}$	$\geq$
$a_1 < a$	$\geq$	$min\{a' \mid a' \in S, a_1 \leq a' < a\}$	$\geq$

Table 8.4: Choice of the new queries, for an original down query (in the first two cases, there are two alternative solutions).

establish if the part of the original query half-plane not covered by the new one contains the extension of some generalized tuple.

2. *Choice of the half-plane query.* Conditions on the angular coefficient of the new query line and on the direction of the new half-plane must be given with respect to the position of line $Y = aX + b$ and the direction line. Table 8.4 summarizes the various cases for an original down-query. Similar conditions can be given for an up-query. Note that the proposed conditions also depend on set S. Figure 8.14 graphically represents the various cases for an original down-query.

   If, given a, a_1, θ, θ_1 and a set of angular coefficients S, at least one angular coefficient a' can be found satisfying the previous conditions, we say that $Y\ \theta_1\ a_1X + b_1$ is *acceptable* for S and $Y\ \theta\ aX + b$.

Given a relation r and a query $E(q)$, if no acceptable direction half-plane exists, the new query cannot be found. Thus, with respect to the technique presented in Subsection 8.6, this technique can be applied in fewer cases.

The following result holds.

Proposition 8.4 *Let S be a set of angular coefficients. Let r be a generalized relation. Let $E(q)$ be a query, such that $E \in \{ALL, EXIST\}$. Let q_1 be a direction half-plane for r, which is acceptable for q and S. Then, the previous algorithm ensures to find a new query $E(q')$ such that $E(q, r) \subseteq E(q', r)$.* □

Of course, acceptable direction lines with different angular coefficient may exist. However, not all direction half-planes have the same behavior with respect to the generation of false-hits. We distinguish three main cases:

1. If only one acceptable direction half-plane $Y\ \theta_1\ a_1X + b_1$ is known, a good measure to associate with the selected new query $Y\ \theta'\ a'X + b'$ is the angle formed by lines $Y = aX + b$ and $y = a'X + b'$, external to the half-plane $Y\ \theta\ aX + b$.

2. If two acceptable direction half-plane $Y\ \theta_1\ a_1X+b_1$ and $Y\ \theta_2\ a_2X+b_2$ exist, and the new query is constructed with respect to the direction half-plane $Y\ \theta_1\ a_1X+b_1$, the previously defined measure can be refined by considering the area of the triangle obtained by cutting the sector formed by lines $Y = aX + b$ and $Y = a'X + b'$ with line $Y = a_2X + b_2$. If no triangle is generated (either $a = a_2$ or $a' = a_2$), the area is associate with ∞. In both cases, the new measure is the pair (A_i, α_i), where α_i is the angle defined as before and A_i is the area constructed as before for the direction half-plane $Y\ \theta_i\ a_iX+b_i$ (see Figure 8.15).

3. The previous case can be generalized to the existence of n acceptable direction half-planes $Y\ \theta_i\ a_iX + b_i$, $i = 1, ..., n$. In this case, the (open) polygon defined by the direction lines must be constructed. A query line can intersect such a polygon in at most two points. The two points lay on (at least) two direction lines. A new query is constructed for each direction line. The query corresponding to the lowest measure, with respect to the lexicographic ordering, is then chosen. Note that if all areas are infinite, the new query is selected with respect to the generated angles. However, if the area of at least one measure is finite, the query generating the lowest false hits area is selected.

Note that the previous algorithm does not guarantee that the query generating the lowest number of false hits is always found. Rather, it only applies a good heuristic to select the new query, assuming that direction half-planes are the only known information on the extension of the considered generalized relation.

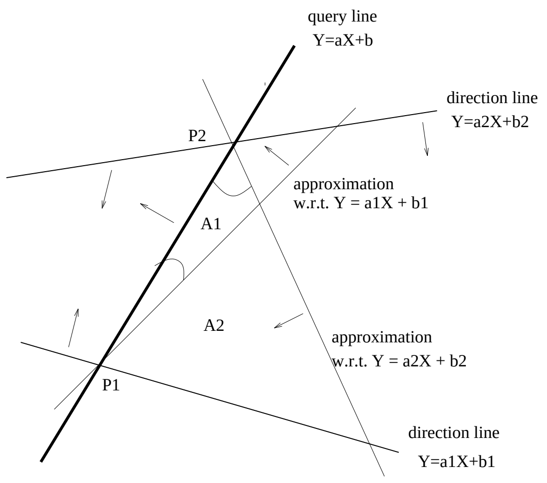


Figure 8.15: Choice of the direction line.

The final problem to solve is how direction lines are determined. The general approach is to maintain a (possibly open) minimum bounding polybox of the entire extension of the relation. If such a polybox does not exist, this means that the generalized relation does not admit any direction line. Otherwise, the lines on which the edges of the polybox lay represent direction lines for the relation.

Since the angular coefficient of the new query belongs to S, the following result holds.

Theorem 8.6 *Let S be a set of angular coefficients. Let r be a generalized relation containing N generalized tuples. Let q be a query half-plane. Let T be the cardinality of the set $ALL(q,r)$ ($EXIST(q,r)$). Assume that there exists at least one direction half-plane for r, which is acceptable for q and S. If the angular coefficient of $p(q)$ is not contained in S, there is an indexing structure for storing r in $O(k\ N/B)$ pages such that $ALL(q,r)$ and $EXIST(q,r)$ selections are performed in $O(\log_B N/B + T_1/B)$ time, where T_1 represents the number of generalized tuples returned by the new query constructed as before, and generalized tuple update operations are performed in $O(k\ \log_B N/B)$ time.* □

In the following, the technique presented in this section is denoted by T2.

When compared with the technique presented in Subsection 8.6, T2 does not generate duplicates, since only one new query is selected. However, no clear relationship exists between the number of false hits generated by the two techniques. It essentially depends on the choice of point P for the first technique and on the choice of direction lines for the second one.

8.7.1 Extension to an arbitrary d-dimensional space

In a d-dimensional space, a direction line becomes a direction hyperplane. The proposed technique can be extended to deal with such a space as follows:

1. *Choice of the new hyperplane.* As in the 2-dimensional case, we assume that the direction hyperplane and the query hyperplane intersect. The new hyperplane must pass through the $(d-1)$-hyperplane defined by this intersection.

   As for T1, the extension of this approximation technique to a d-dimensional space, with $d > 2$ does not allow the approximation of a generic half-plane query. The set of queries that can be approximated depends, in this case, by the direction hyperplane.

2. *Choice of the half-plane query.* The cases presented in Table 8.4 can still be used by replacing $\leq$ with $\preceq$ in order to determine the normal vector and the half-plane direction of the new query.

8.8 Using sector queries to approximate half-plane queries

The solutions proposed in Section 8.6 and Section 8.7 to answer half-plane queries are based on data structures proposed for the weaker problem (see Subsection 8.5.1). A different solution is based on the following consideration. From Figure 8.10 it follows that, in order to approximate an EXIST half-plane query, all generalized tuples which *are not* contained in a specific (depending on the chosen approximation lines) sector must be determined. In a similar way, to approximate an ALL selection all generalized tuples which do not intersect a given sector must be determined.

Starting from this consideration, the result of an EXIST or ALL selection with respect to a half-plane is equivalent to the difference between the input generalized relation and the generalized tuples satisfying respectively the ALL or the EXIST selection with respect to a specific sector.

In the following we first introduce sector queries and then we show how they can be used to approximated half-plane queries. The proposed technique, denoted by T3, can be applied when, given a query half-plane q, the new query half-planes q_1 and q_2 are both 0-half-planes or both 1-half-planes.

8.8.1 Sector queries

An ALL (EXIST) sector query is defined as an ALL (EXIST) query with respect to a space sector, defined by the intersection of two d-dimensional half-planes. It is simple to show that the dual representation of a sector is a subset of a 2-dimensional half-plane H. In particular, let H_1 and H_2 be the half-planes defining the sector. Let $P_1 = D(p(H_1))$ and $P_2 = D(p(H_2))$. Let H be the vertical 2-dimensional half-plane intersecting P_1 and P_2. The points of H lying over the segment connecting P_1 and P_2 belong to the dual representation of the sector.

In the following we propose some external-memory solutions for ALL and EXIST sector queries. The technique proposed for ALL sector queries allows the exact detection of all and only those generalized tuples which are contained in the sector whereas the solution proposed for the EXIST sector query retrieves a superset of the generalized tuples which intersect the sector.

8.8.1.1 ALL sector queries

For simplicity we consider only downward oriented ALL sector queries. In this case, the query generalized tuple has the form:

$$t_{Q^+} \equiv Y \leq aX + b \wedge Y \leq cX + d.$$

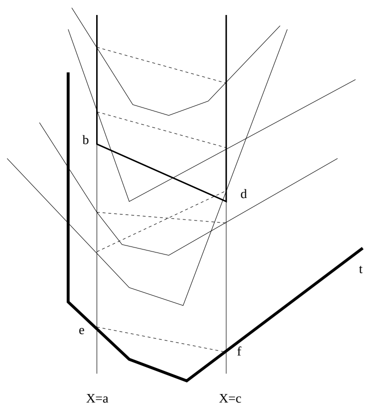


Figure 8.16: The representation of an ALL sector query problem with respect to $Y \leq aX + b \wedge Y \leq cX + d$ in the dual plan.

Suppose that $a \in S, c \in S$. Without leading the generality of the discussion, we assume that $a \leq c$. Each generalized tuple of this kind is represented in the dual plane as shown in Figure 8.16.

From the results presented in Section 8.4, it follows that a generalized tuple t_P satisfies an ALL sector query with respect to t_{Q^+} if $UP(Q^+) \subseteq UP(P)$. This condition is verified if the intersection of $UP(P)$ with slab $a \leq X \leq c$ is *under* the segment connecting points (a, b) and (c, d) (see Figure 8.16). This condition is also equivalent to establishing whether, being e and f the intersections of $UP^-(P)$ with $X = a$ and $X = c$, $e \leq b$ and $f \leq d$.

Proposition 8.5 *Let t_P be a generalized relational tuple. Let e be the intersection of $UP^-(P)$ with $X = a$ and let f be the intersection of $UP^-(P)$ with $X = c$. Then, $All(t_{Q^+}, t_P)$ is true iff $e \leq b$ and $f \leq d$.* □

Checking the previous condition for all the generalized tuples contained in a generalized relation corresponds to a 2-dimensional 2-sided range searching problem, introduced in Chapter 6 (see Figure 8.17). Among the techniques that have been proposed, *path caching* [120] allows us to perform 2-sided queries in $O(\log_B n)$, with $O(n \log_2 B \log_2 \log_2 B)$ space. Assuming we use this technique, the following result holds.

Theorem 8.7 *Let r be a generalized relation containing N generalized tuples. Let $t_{Q^+} \equiv Y \leq aX + b \wedge Y \leq cX + d$. Let T be the cardinality of the set $ALL(t_{Q^+}, r)$. If a and c are contained in a predefined set of cardinality k, there is an indexing structure*

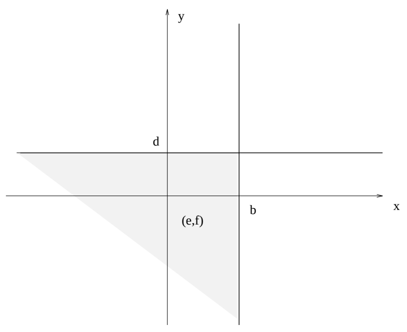


Figure 8.17: Reduction of the ALL sector query problem to a point location problem.

for storing r in $O(k\ n\ \text{loglog}B)$ *pages such that* $ALL(t_{Q^+}, r)$ *selection is performed in* $O(\log_B n + t)$ *time. Updates are executed in* $O(k\ \log_B n)$*, amortized.* □

Note that the k factor in space and update complexity is due to the fact that $2k$ data structures has to be maintained, one for each pair of adjacent values in the increasing order of S.

8.8.1.1.1 Extension to an arbitrary d-dimensional space The extension of the solution proposed for ALL sector queries to a d-dimensional space is immediate. In a d-dimensional space, a sector query is defined by the intersection of two d-dimensional half-planes. Assuming that the normal vectors corresponding to such half-planes belong to the predefined set S, as required by Theorem 8.7, the reasoning done for the 2-dimensional case still holds for the generic d-dimensional case. The number of B$^+$-trees can be reduced by maintaining only values for "adjacent" normal vectors, i.e., normal vectors whose angles formed with plane $X_d = 0$ are consecutive in the total order of such angles.

8.8.1.2 EXIST sector queries

For simplicity we consider only downward oriented EXIST sector queries. In this case, the query generalized tuple has the form:

$$t_{Q^+} \equiv Y \leq aX + b \wedge Y \leq cX + d.$$

Suppose that $a \in S, c \in S$. Without leading the generality of the discussion, we assume that $a \leq c$. Thus, each query generalized tuple of this kind is represented in the dual plane as shown in Figure 8.18.

From the results presented in Section 8.4, it follows that a generalized tuple t_P satisfies an EXIST sector query with respect to t_{Q^+} if $UP(Q^+) \cap DOWN(P) = \emptyset$. This condition is verified if the intersection of $DOWN^-(P)$ with this slab is *under* the segment connecting points (a,b) and (c,d) (see Figure 8.18). The intersection of $DOWN^-(P)$ with this slab is a chain of segments, downward oriented. This means that the previous condition cannot be reduced, as for the case of ALL sector queries, to check the position of the segment, connecting the intersections of $DOWN(P)$ with $X = a$ and $X = c$, with respect to the position of the segment connecting points (a,b) and (c,d). As an example, consider chains (1) and (2) in Figure 8.18. The highest Y coordinate of chain (1) is over segment $(a,b) - (c,d)$, thus the corresponding generalized tuple does not satisfy the EXIST selection, whereas the highest Y coordinate of (2) is under segment $(a,b) - (c,d)$, thus the corresponding generalized tuple satisfies the EXIST selection. However, the extreme points of both chains in slab $a \leq X \leq c$ are under segment $(a,b) - (c,d)$.

Condition $UP(Q^+) \cap DOWN(P) = \emptyset$ can however be checked by considering the maximum Y coordinate of $DOWN(P)$ in slab $a \leq X \leq c$, as the following result shows.

Proposition 8.6 *Let t_P be a generalized relational tuple. Let $t_{Q^+} \equiv Y \leq aX + b \wedge Y \leq cX + d$. Let e be the intersection of $DOWN^-(P)$ with $X = a$ and let f be the intersection of $DOWN^-(P)$ with $X = c$. Let $(m_x, m_y) \in DOWN(P)$ such that m_y is the maximum Y value of $DOWN(P)$ in slab $a \leq X \leq c$. Then, $Exist(t_{Q^+}, t_P)$ is satisfied iff $e \leq b$, $f \leq d$, and $m_x \frac{b-d}{a-c} + b - a\frac{b-d}{a-c} \geq m_y$.* □

Proof: It follows from the previous considerations and results presented in Section 8.4. □

Differently from the ALL case, the condition proposed by Proposition 8.6 does not correspond to any well-known geometric problem. Therefore, in order to execute an EXIST sector query, only approximated solutions can be proposed. In particular, from Figure 8.18 we can observe that:

- If the maximum Y-coordinate of a tuple t_P is *over* segment connecting (a,b) and (c,b), then $Exist(t_{Q^+}, t_P)$ is not satisfied.
- If the maximum Y-coordinate of a tuple t_P is *under* segment connecting (a,d) and (c,d), then $Exist(t_{Q^+}, t_P)$ is satisfied.
- If the maximum Y-coordinate of a tuple t_P is *between* segment connecting (a,b) and (c,b) and segment connecting (a,d) and (c,d), then $Exist(t_{Q^+}, t_P)$ may or may not be satisfied.

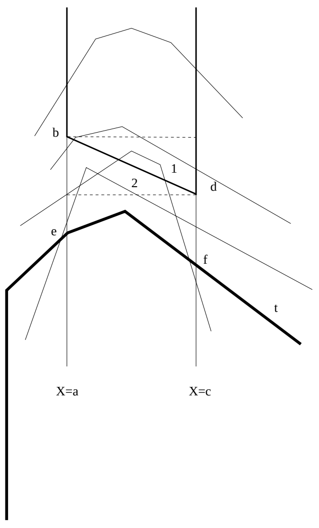


Figure 8.18: The representation of an EXIST sector query problem with respect to $Y \leq aX + b \wedge Y \leq cX + d$ in the dual plane.

Thus, all generalized tuples t_P whose maximum y coordinate of $DOWN(P)$ in slab $a \leq X \leq c$ is not greater than b, represent a superset of the generalized tuples satisfying the EXIST sector query. A B$^+$-tree, storing the maximum Y value of each generalized tuple in the considered slab, can be used to determine this superset.

Theorem 8.8 *Let S be a set of angular coefficients. Let r be a generalized relation containing N generalized tuples. Let $t_{Q^+} \equiv Y \leq aX + b \wedge Y \leq cX + d$. Let $a \in S$, $c \in S$. Let T be the cardinality of the set $EXIST(t_{Q^+}, r)$. There is an indexing structure for storing r in $O(k\ N/B)$ pages such that $EXIST(t_{Q^+}, r)$ selection is performed in $O(\log_B N/B + T_1/B)$ time, where T_1 represents the number of generalized tuples returned by the new query constructed as before, and generalized tuple update operations are performed in $O(k\ \log_B N/B)$ time.* □

Also in this case, the k factor in space and update complexity is due to the fact that $2k$ data structures has to be maintained, one for each pair of adjacent values in the increasing order of S.

8.8.1.2.1 Extension to an arbitrary d-dimensional space The technique proposed for 2-dimensional EXIST sector queries can be extended to a d-dimensional space by maintaining for each pair of normal vectors in S and for each generalized

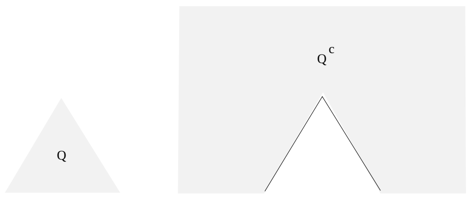


Figure 8.19: A sector domain and its complement in the 2-dimensional space.

tuple t_P the maximum X_d coordinate of $DOWN(P)$ in the slab represented by the considered normal vectors. These values can then be indexed by using a B$^+$-tree. The number of B$^+$-trees can be reduced by maintaining only values for "adjacent" normal vectors, i.e., normal vectors whose angles formed with plane $X_d = 0$ are consecutive in the total order of such angles.

Given a query sector, the value to search in such a B$^+$-tree is given by the maximum between the d-th coordinates of the two points representing the hyperplanes supporting the given sector query in the dual plane.

8.8.2 Approximating half-plane queries by sector queries

In the following, we show how sector queries can be used to execute half-plane queries. Given a half-plane query, the basic approach is to replace the original query with two new queries, constructed as described in Section 8.6. using the constructed query half-planes, a sector query is constructed and solutions proposed for sector queries are also used, by complementation of the obtained results, to answer the original half-plane query.

8.8.2.1 EXIST half-plane queries

Let q be a query half-plane. Suppose that, by applying the technique presented in Section 8.6, q is approximated by two new half-planes $q_1 \equiv Y \geq aX + b$ and $q_2 \equiv Y \geq cX + d$ such that $a \in S$, $c \in S$ (a similar discussion holds if $q_1 \equiv Y \leq aX + b$ and $q_2 \equiv Y \leq cX + d$). Let $t_Q \equiv Y < aX + b \wedge Y < cX + d$. Let $t_{Q^+} \equiv Y \leq aX + b \wedge Y \leq cX + d$. Further, given a domain Q, let Q^c represent the set of points not contained in Q (see Figure 8.19).

It is trivial to prove that $EXIST(t_{Q^c}, r) = r \setminus ALL(t_Q, r)$. This means that the result of an EXIST selection with respect to a domain Q^c, coincides with the set of generalized tuples not satisfying the ALL selection with respect to Q.

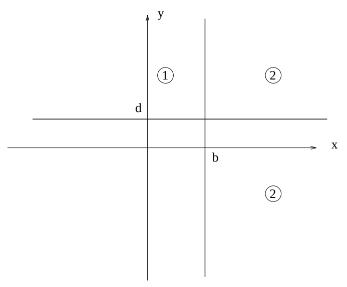


Figure 8.20: The two 2-sided range queries corresponding to the original EXIST half-plane query.

Proposition 8.7 *Let r be a generalized relation. Let Q be a domain contained in the Euclidean plan and let Q^c be the complementary region. Then, $r \setminus ALL(t_Q, r) = EXIST(t_{Q^c}, r)$.* □

By Proposition 8.5, the execution schema proposed for ALL sector queries can be used to answer EXIST half-plane queries as follows.

Theorem 8.9 *Let $t_Q \equiv Y < aX + b \wedge Y < cX + d$. Let t_P be a generalized relational tuple. Let e be the intersection of $UP(P)$ with $X = a$ and let f be the intersection of $UP(P)$ with $X = c$. Then, $Exist(t_{Q^c}, t_P)$ is true iff $e \geq b$ or $f \geq d$, i.e., iff $e \geq b$ or $e < b \wedge f \geq d$.*

Proof: This result follows from Proposition 8.5 and from the fact that if $e = b$ then $Y = aX + b$ is a supporting hyperplane for P and if $f = d$, $Y = cX + d$ is a supporting hyperplane for P. □

Conditions stated by Theorem 8.9 correspond to two 2-dimensional 2-sided range searching problems, as shown in Figure 8.20. Therefore, due to results about path caching presented in [120], the following result holds.

Theorem 8.10 *Let S be a set of angular coefficients. Let r be a generalized relation containing N generalized tuples. Let q be a query half-plane. Let T be the cardinality of the set $EXIST(t_{Q^c}, r)$. There is an indexing structure for storing r in $O(k\ N/B)$ pages such that $EXIST(t_{Q^c}, r)$ selection is performed in $O(\log_B N/B + T_1/B + T_2/B)$ time, where T_1 and T_2 represent the number of generalized tuples returned by the new queries constructed as before, and generalized tuple update operations are performed in $O(k \log_B N/B)$ time.* □

8.8.2.1.1 Extension to an arbitrary d-dimensional space The proposed technique can be extended to any d-dimensional space. The same limitations existing for T1 hold.

8.8.2.2 ALL half-plane queries

By using the notation introduced in Subsection 8.8.2.1, it is trivial to prove that a generalized tuple is contained in $ALL(t_{Q^c}, r) = r \setminus EXIST(t_Q, r)$. This means that the result of an ALL selection with respect to a domain Q^c, coincides with the set of generalized tuples not satisfying the EXIST selection with respect to Q.

Proposition 8.8 *Let r be a generalized relation. Let Q be a domain contained in the Euclidean plan and let Q^c be the complementary region. Then, $r \setminus EXIST(t_Q, r)$ $= ALL(t_{Q^c}, r)$.* □

By Proposition 8.6, the execution schema proposed for EXIST sector queries can be used to answer ALL half-plane queries. In particular, the following result holds.

Theorem 8.11 *Let $t_Q \equiv Y < aX + b \wedge Y < cX + d$. Let t_P be a generalized relational tuple. Let e be the intersection of $DOWN^-(P)$ with $X = a$ and let f be the intersection of $DOWN^-(P)$ with $X = c$. Let $(m_x, m_y) \in DOWN(P)$ such that m_y is the maximum Y value of $DOWN(P)$ in slab $a \leq X \leq c$. Then, $All(t_{Q^c}, t)$ is satisfied iff one of the following conditions holds:*

- $e \geq b$
- $f \geq d$
- $m_x \frac{b-d}{a-c} + b - a\frac{b-d}{a-c} < m_y$.

Proof: The result follows from Proposition 8.6 and from the fact that if $e = b$, $Y = aX + b$ is a supporting hyperplane for P and if $f = d$, $Y = cX + d$ is a supporting hyperplane for P. □

The previous conditions cannot be reduced to a 2-dimensional 2-sided range searching problem, as in the case of EXIST half-plane queries. However, following the approach proposed for ALL sector queries, from Figure 8.18 we can observe that:

- If a maximum Y-coordinate of a tuple t_P is *over* segment connecting (a, b) and (c, b), then $All(t_{Q^c}, t_P)$ is satisfied.

- If a maximum Y-coordinate of a tuple t is *under* segment connecting (a, d) and (c, d), then $All(t_{Q^c}, t_P)$ is not satisfied.

- If the maximum Y-coordinate of a tuple t_P is *between* segment connecting (a, b) and (c, b) and segment connecting (a, d) and (c, d), $ALL(t_{Q^c}, t_P)$ may or may not be satisfied.

Thus, the generalized tuples t_P whose maximum Y coordinate of $DOWN(P)$ in slab $a \leq X \leq c$ is not lower than d, represent a superset of the generalized tuples satisfying the EXIST sector query. A B$^+$-tree, storing the maximum Y value of each generalized tuple, can be used to determine this approximated set.

The following result holds.

Theorem 8.12 *Let S be a set of angular coefficients. Let r be a generalized relation containing N generalized tuples. Let q be a query half-plane. Let T be the cardinality of the set $ALL(t_{Q^c}, r)$. There is an indexing structure for storing r in $O(k\ N/B)$ pages such that $ALL(t_{Q^c}, r)$ selection is performed in $O(\log_B N/B + T_1/B)$ time, where T_1 represents the number of generalized tuples returned by the new query constructed as before, and generalized tuple update operations are performed in $O(k \log_B N/B)$ time.* □

8.8.2.2.1 Extension to an arbitrary d-dimensional space The same considerations presented in Subsection 8.8.1.2 hold.

8.9 Theoretical comparison

In the following, the three proposed techniques are theoretically compared with respect to several parameters:

- The type of the used data structures.

- The number of the used data structures.

- The freedom: with freedom we mean the available alternative choices to approximate a given half-plane query, given a predefined set S.

- The space complexity.

- The time complexity of searching in the index structure.

- The update complexity.

	T1	T2	T3
Used D.S.	B^+-trees	B^+-trees	priority search trees
Number of used D.S.	$2k$	$2k$	$2k$
False hits	yes	yes	yes
Duplicates	yes	no	no
Freedom	any point on the query line	one for each direction line	any point on the query line
Space complexity	$O(2kn)$	$O(2kn)$	$O(2kn\log\log B)$
Time complexity	$O(2\log_B n)$	$O(\log_B n)$	$O(\log_B n)$
Update complexity	$O(2k \log_B n)$	$O(2k \log_B n)$	$O(2k\log_B n)$ amortized

Table 8.5: Comparison for EXIST selections.

- The number of generated false hits.
- The number of generated duplicates.

All the techniques are compared with respect to the 2-dimensional case. In order to analyze the behavior of the techniques in an arbitrary d-dimensional space, we consider a further parameter:

- The applicability to an arbitrary d-dimensional space: this parameter allows us to determine which techniques better scale to higher dimensions.

The analysis is performed by considering first EXIST selections and then ALL selections.

8.9.1 EXIST selections

In the following, we compare the performance of the three techniques we have proposed to solve an EXIST selection with respect to a half-plane query whose angular coefficient does not belong to a predefined set. Table 8.5 summarizes this comparison.

- *Used data structures.* T1 and T2 are based on the data structure presented in Subsection 8.5.1 to answer half-plane queries whose angular coefficient belongs to a given set.

T3 uses priority search trees, therefore different techniques must be used to answer half-plane queries whose angular coefficient belongs to a given set S and half-plane queries whose angular coefficient does not belong to S.

Therefore, with respect to the used data structure, T3 has the worst behavior.

- *Number of used data structures.* If k is the cardinality of the predefined set S, all techniques require $2k$ data structures, two (one for TOP^P values and one for BOT^P values) for each value in S. Under this point of view, the techniques are therefore equivalent.
- *False hits.* All techniques generate false hits when the angular coefficient of the query half-plane does not belong to the predefined set. For the same set S, the set of generated false hits depends on the chosen point in techniques T1 and T3 on the considered direction line for technique T2. By considering the same point, techniques T1 and T3 generate the same set of false hits.
- *Duplicates.* Duplicates can be generated only by technique T1. Since technique T3 can be seen as an improvement of technique T1 removing the generation of duplicates, from a theoretical point of view we can assess that the cost of T3 is always lower than the cost of T1.
- *Freedom.* In technique T1, each point lying on the line supporting the query half-plane can be chosen to generate the approximating queries. The same holds for technique T3. In technique T2 different results can be obtained by considering different direction lines. Moreover, T2 and T3 can be applied only in particular cases.
- *Space complexity.* The space complexity is higher in T3, since the number of pages required to store a priority search tree is higher than the number of pages required to store a B^+-tree containing the same number of elements.
- *Time complexity.* All time complexities are in $O(\log_B n)$. However, technique T1 requires two index scans whereas all other techniques require only one index scan.
- *Update complexity.* T1 and T2 have the same update complexity; technique T3 has the same update bound than T1 and T2 but amortized.
- *Extension to a d-dimensional space.* The proposed techniques can be used to approximate half-plane queries in an arbitrary d-dimensional space ($d > 2$) only in some specific cases. Detection of techniques that can always be applied to answer half-plane queries in a d-dimensional space is a topic left to future work.

	T1	T2	T3
Used D.S.	B^+-trees	B^+-trees	B^+-trees
Number of used D.S.	$2k$	$2k$	$2k$
False hits	yes	yes	yes
Duplicates	yes	no	no
Freedom Space complexity	any point on the query line $O(2kn)$	one for each direction line $O(2kn)$	no freedom $O(2kn)$
Time complexity	$O(2\log_B n)$	$O(\log_B n)$	$O(\log_B n)$
Update complexity	$O(2k \log_B n)$	$O(2k \log_B n)$	$O(2k \log_B n)$ amortized

Table 8.6: Comparison for ALL selections.

8.9.2 ALL selections

Table 8.6 summarizes the characteristics of T1, T2, and T3 with respect to ALL selections. In general, considerations similar to those presented for EXIST queries hold. The only difference is related to the used data structures. Indeed, for answering ALL selections, technique T3 uses B^+-trees which however contain different information with respect to the B^+-trees used to answer half-plane queries whose angular coefficient belongs to a given set. Thus, with respect to the used data structure, T3 has the worst behavior.

8.10 Preliminary experimental results

Some preliminary experiments have been carried out in order to compare the performance of the proposed techniques in the 2-dimensional space. In particular, we have performed two different groups of experiments. The aim of the first group of experiments is to compare techniques T1, T2, and T3 with respect to the number of page accesses and the number of generated false hits. The aim of the second group is to compare T1 and T2 with respect to the R-tree, a well known spatial data structure [71].

In all the considered techniques, the refinement step has been applied directly on $UP(P)$ and $DOWN(P)$ polygons. In particular, each B^+-tree is associated with a file (called UP-file) containing $t_{UP(P)}$ and another file (called DOWN-file) containing $t_{DOWN(P)}$, for each generalized tuple t_P belonging to the input generalized relation.

Such generalized tuples are ordered following the ordering induced by the corresponding B$^+$-tree. By assuming that $a \in S$, each half-plane query $E(Y\ \theta\ aX + b, r)$ can be answered by first looking for b in a B$^+$-tree corresponding to a; this value is associated with a specific offset either in the corresponding UP- or DOWN- file. Starting from this offset, all tuples contained in the file and preceding or following this offset, depending on the specific query, have to be checked for refinement. Thus, only one leaf node per search is accessed in the B$^+$-tree structure. A similar approach has been taken for implementing refinement in the R-tree. Note that, even if this approach increases the redundancy of the data representation, since generalized tuples are replicated k times, it improves the query time, since only one leaf node per search is accessed. As a final remark note that, even if this solution could be not feasible from the point of view of the space occupancy, it does not alter the results of the comparison.

The experiments have been performed on a PC Pentium 100, with 16 Mb RAM. The page size is 1k. The program has been written in C++. The considered generalized relations contain respectively 500, 2000, 4000, 8000, and 12000 generalized tuples; each generalized tuple contains at most 30 constraints.

8.10.1 Comparing the performance of T1, T2, and T3

The first group of experiments concerns techniques T1, T2, and T3. Such techniques have been applied to two different groups of generalized relations, the first containing closed generalized tuples (closed relations) and the second containing also open generalized tuples (open relations). In the first case, direction lines for the considered relation have been assumed to define the minimum bounding box containing the whole relation extension. For technique T1 and T3, point P has been chosen inside such rectangle. In the second case, open generalized tuples have been constructed in such a way to guarantee the existence of at least one direction line. In particular, the extension of the generalized tuples contained in the considered generalized relation is represented in Figure 8.21.

The aim of the experiments is to analyze the trade-off existing among T1, T2, and T3, in order to assess the impact of duplicate and false hits on the search. In doing that, we have mainly focused on the influence of the cardinality of the set of angular coefficients S. In particular, we have assumed that the set S contains angular coefficients of lines dividing the space in $2k$ equal sectors. In the performed experiments we have chosen $k = 2, 4, 8$.

Several experiments have been performed by considering different object sets and different queries. We have observed that the trade-off between the techniques does not change by changing the selectivity of the query. For this reason, all results we

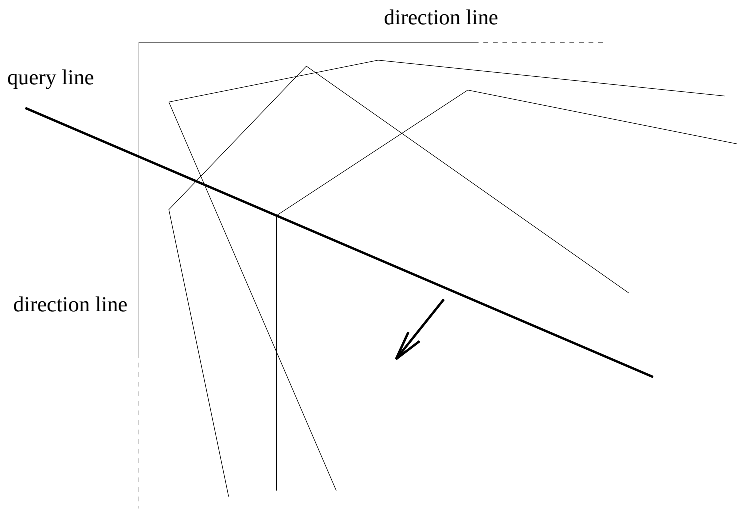


Figure 8.21: Shapes of the open generalized tuples contained in the considered generalized relation.

report here are related to a single query. Moreover, similar results have been obtained for generalized relations containing closed or open objects. Since closed relations will be considered in Subsection 8.10.2, here we report results obtained for open relations, i.e., relations containing at least one open generalized tuple. Both ALL and EXIST selections have been investigated, with respect to the same query half-plane.

In the following, experimental results are presented in three groups:

- The first group of results shows how the number of duplicates generated by technique T1 changes by changing k.
- The second group of results shows how the number of false hits generated by techniques T1, T2, and T3 changes by changing k.
- The third group of results shows the behavior of the three techniques with respect to the number of page accesses.

8.10.1.1 Duplicates

We have compared the number of duplicates generated by T1 for different cardinalities of set S on data sets containing open generalized tuples. Similar results have been obtained for closed and open generalized relations. Figure 8.22 shows that the number of duplicates increases for increasing values of k. Indeed, for higher values of k, the

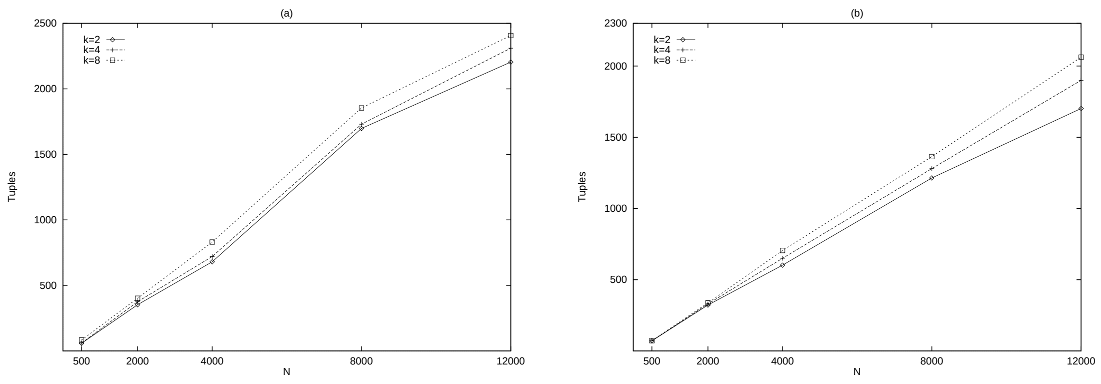


Figure 8.22: Duplicates generated by T1 (a) for an EXIST selection, (b) for an ALL selection.

common area of the two new half-planes increases. Therefore, more generalized tuples are returned twice.

8.10.1.2 False hits

Figures 8.23 and 8.24 show that the number of false hits generated by T1, T2, and T3 decreases for increasing values of k. This behavior is reasonable since higher values for k correspond to smaller false hits areas. A similar result has been obtained for closed relations. Note that T2 generates the lowest number of false hits. This is mainly due to the particular type of generalized relations we used in these experiments (in Subsection 8.10.2 we will see that, for closed relations, this difference is not so clear).

The number of false hits generated by the three techniques can also be used to compare their degree of filtering. From Figures 8.25 and 8.26 we can see that the number of generated false hits is higher for techniques T1 and T3. This is mainly due to the chosen generalized relation and therefore to the shape of the extension of open generalized tuples (see Figure 8.21) and to the choice of point P. Moreover, T1 and T3 have a similar behavior. Indeed, T3 can be seen as an optimization of T1, not generating duplicates. However, T3 usually generates more false hits than T1. For EXIST selections, this is mainly due to the path caching implementation (see Section 8.8.2.1). Indeed, not all tuples associated with the pages accessed in the corresponding data structure belong to the sector result. These tuples represent additional false hits for technique T3. For ALL selections, this is due to the fact that T3 approximates a half-plane query by further approximating the corresponding sector query (see Section 8.8.2.2). The great difference between T2 and T1, T3 is

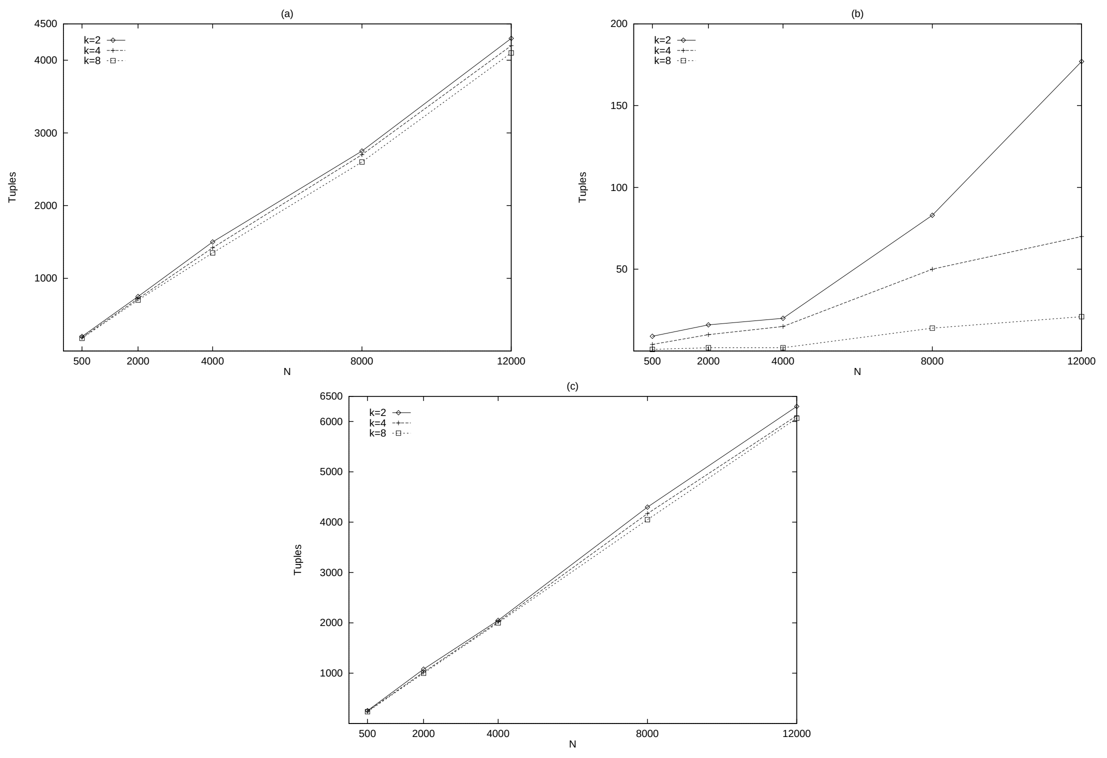


Figure 8.23: False hits generated for an EXIST selection by (a) T1, (b) T2, (c) T3.

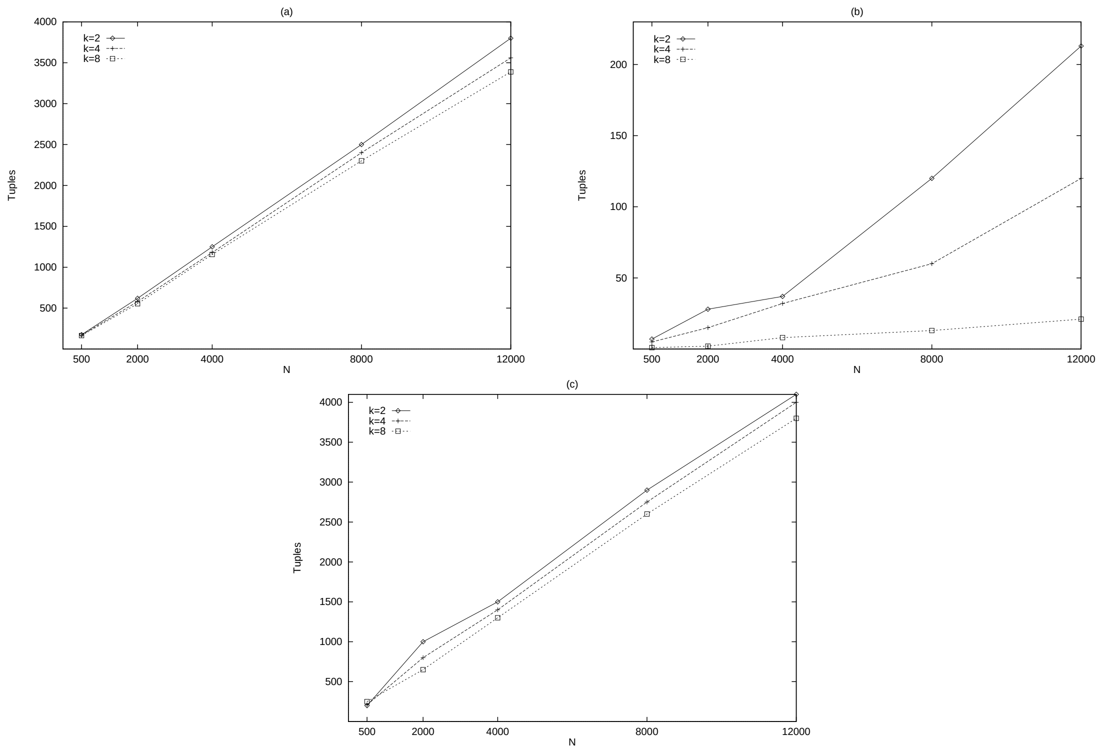


Figure 8.24: False hits generated for an ALL selection by (a) T1, (b) T2, (c) T3.

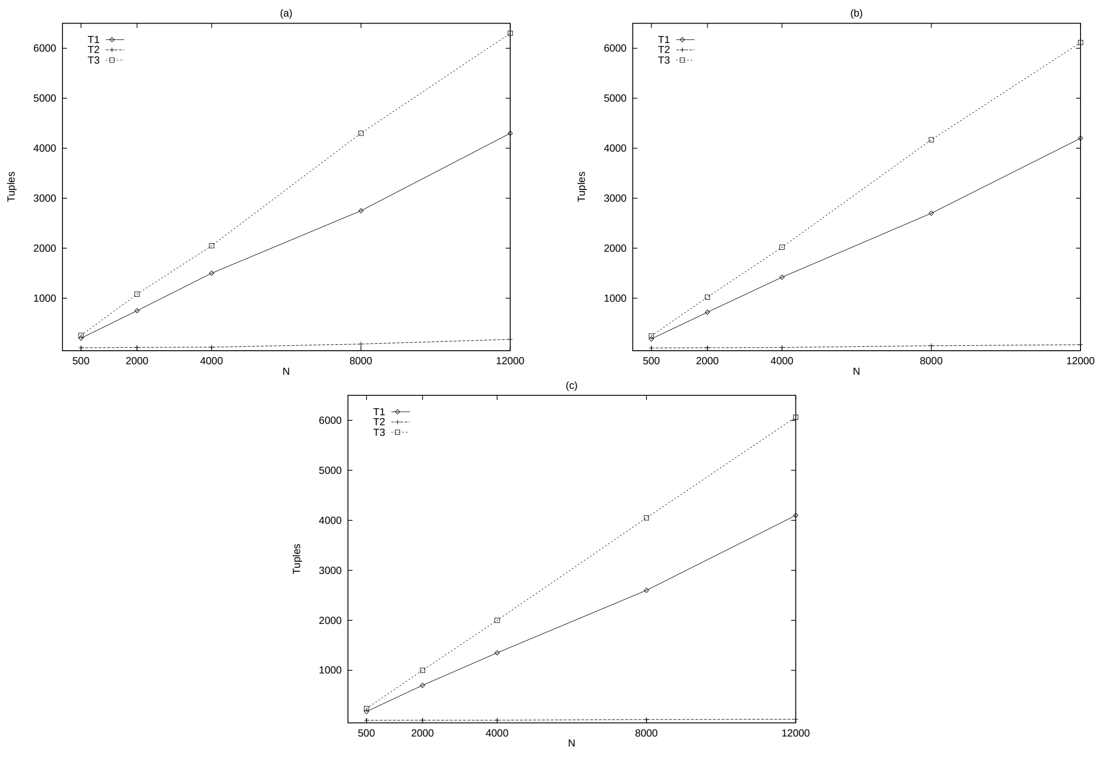
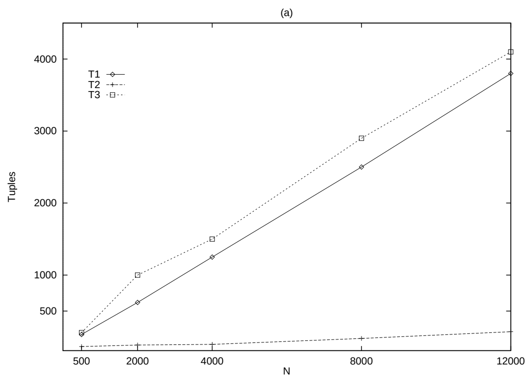


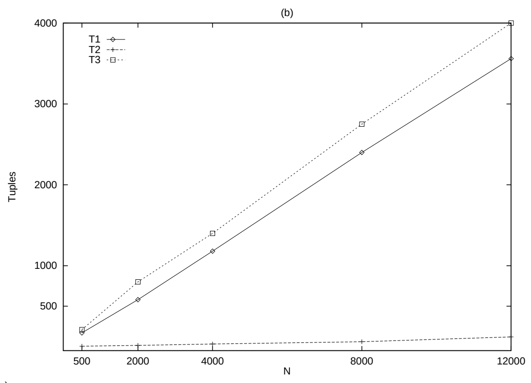
Figure 8.25: False hits generated by T1, T2, and T3 for an EXIST selection and (a) $k = 2$, (b) $k = 4$, and (c) $k = 8$.

mainly due to the choice of point P. By choosing point P on the direction line, the number of false hits would have been almost equal.

8.10.1.3 Comparison with respect to page accesses

T1, T2, and T3 have been compared with respect to the number of pages accessed in the B$^+$-tree and in refining the retrieved generalized tuples. Similar results have been obtained for all k-values. Figure reports results obtained for $k = 2$. It can be observed that technique T2 performs better than techniques T1 and T3, in almost all cases. This is mainly due to the choice of point P. From the same figures, we can also see that the number of page accessed by T1 is always lower than the number of page accessed by T3, even if higher than the number of page accessed by T2.





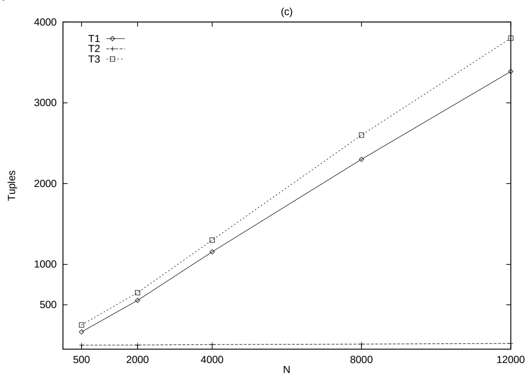
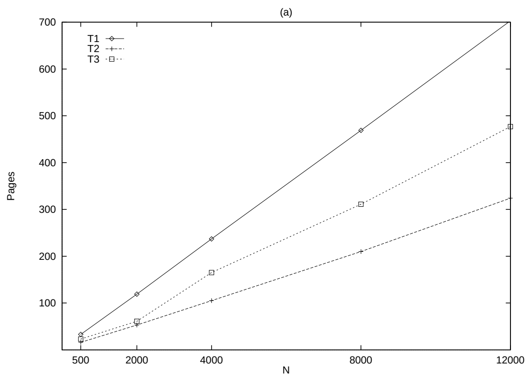


Figure 8.26: False hits generated by T1, T2, and T3 for an ALL selection and (a) $k = 2$, (b) $k = 4$, and (c) $k = 8$.



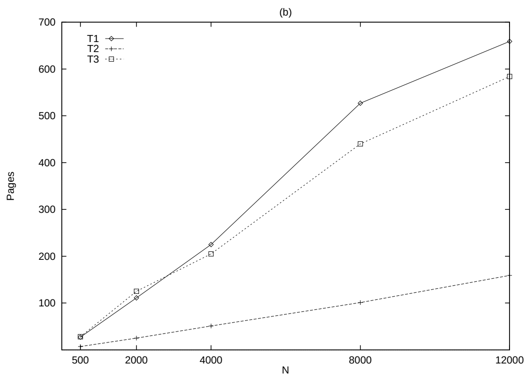


Figure 8.27: Comparison of the number of page accesses for $k = 2$ and (a) an EXIST selection, (b) an ALL selection.

8.10.2 Comparing the performance of T1, T2, and R-trees

In order to establish the practical applicability of the proposed techniques, we have compared their performance with respect to the performance of the R-tree [71], a well known spatial data structure for closed objects. The R-tree has been compared with respect to T1 and T2, i.e., with the techniques guaranteeing the worst and the best performance among those we have proposed. T3 has not been considered since its behavior on closed relations is very similar to the behavior of T2 and since, on closed relations, T2 can always be applied.

The R-tree is a direct extension of B-trees in k-dimensions. The data structure is a height-balanced tree which consists of intermediate and leaf nodes. Data objects are stored in leaf nodes. Each data object is approximated by its minimum bounding rectangle and intermediate nodes are built by grouping rectangles at the lower level. Thus, each intermediate node is associated with some rectangle which completely encloses all rectangles that correspond to lower level nodes.

The R-tree can be used to answer EXIST and ALL queries. The search starts from the root. If the rectangle associated with the root satisfies the query, the query is checked against rectangles corresponding to the root sons. This approach is then recursively applied to all nodes whose associated rectangle satisfies the query until leaf nodes are reached and data objects are directly checked. Note that, differently from B^+-trees, more than one path may be accessed during a single search.

This approach is safe for EXIST queries. However, for ALL queries, it is safe only if the query object is a rectangle. If it is not, the search may not be safe. Indeed, since data objects are approximated by rectangles, some rectangles may not satisfy the ALL query even if the original object does. Therefore, some (sub)-paths of the tree may not be accessed, even if they are associated with some generalized tuples belonging to the result. In order to safely execute an ALL selection, the selection has to be replaced by the corresponding EXIST selection; the result has then to be refined with respect to the original ALL query. Since in our case the query object is a half-plane, this is the method to be applied.

Based on the previous assumption, several experiments have been performed, by varying the following parameters:

- The average size of the considered objects.

  Three different groups of relations have been considered. The first group contains *large* rectangles, i.e., rectangles intersecting almost all other rectangles; the second group deals with *medium* rectangles, i.e., rectangles whose area does not exceed half the area of the bounding rectangle containing all stored ones; finally, the third group deals *small* rectangles, i.e., rectangles with a very small area with respect to the bounding rectangle containing all stored ones. All objects are uniformly distributed in the space.

Since spatial databases typically deal with small objects, the size of the considered objects is a good parameter to analyze how the performance of R-trees change by changing the average size of the considered objects.

- The cardinality of the indexed generalized relation.

  We have considered five different groups of relations, containing 500, 2000, 4000, 8000, and 12000 generalized tuples, respectively.

- The selectivity of the considered queries.

  We have considered six ALL queries and six EXIST queries with different selectivity. The considered selectivity are:1-3%, 3-10%, 30-40%, 40-60%, 60-80%, 90-100%. Note that in comparing the proposed techniques with R-trees, selectivity is very important since different selectivities correspond to a different number of internal tree nodes accesses in the R-tree.

Experiments have been performed by combining in all possible ways the parameter values described above. In performing these experiments, we have taken $k = 2$. This assumption allows us to compare R-trees with respect to the proposed techniques in the case when they have the worst performance (see Subsection 8.10.1).

In the following, we discuss the obtained results with respect to the number of generated false hits and the number of page accesses. The technique supported by the R-tree data structure will be denoted by R.

8.10.2.1 EXIST selections

False hits. We have first analyzed the number of false hits generated by T1, T2, and R. From the performed experiments, it follows that R almost always generates the lowest number of false hits. For selectivity very low ($< 10\%$), the number of false hits generated by T2 is very close to the number of false hits generated by R. Often T2 generates less false hits than T1, but this mainly depends on the chosen relation and on the choice of point P. These results can be observed from Figure 8.28 and Figure 8.29.

From the same figures, we can see that, by increasing the selectivity, the number of false hits generated by T1 decreases whereas the number of false hits generated by T2 and R increases. In R, the number of false hits increases because, by augmenting the selectivity, the number of tree paths to be searched increases. In T1, the number of false hits decreases since the false hits area decreases whereas in T2 increases, thus increasing the number of generated false hits (see Figure 8.30).

From Figure 8.28 and Figure 8.29, we can also observe that R generates the lowest number of false hits when rectangles are small. This is the typical case in spatial databases. This number increases by increasing the area of the rectangles and therefore the number of their intersections. Indeed, it can be shown that in such a case the number of R-tree paths to be analyzed increases. A similar situation arises for T1 and T2. However, in this case, the reason is different. In particular, in those cases the number of false hits increases because, by augmenting the area of the objects, the probability that one of such objects intersects or is contained in the false hits area determined by the new queries increases.

These considerations point out an important difference between R-trees and the proposed data structures: the performance of a search based on R-trees depends on the size of the query object. On the other hand, the performance of T1 and T2 depends on the size of the false hits area generated by the approximation. Thus, by choosing a good approximation, similar performance can be obtained when executing queries with different selectivity.

Page accesses. Different results have been obtained by considering the number of page accesses. In this case, T2 almost always performs better than R. This is in contrast with the result deriving from the analysis of false hits and is mainly due to the number of tree paths that have to be analyzed in the R-tree. Indeed, in T2, always a single path of a B^+-tree has to be analyzed. In the performed experiments, this corresponds to at most 3 page accesses. On the other hand, each single query may require the analysis of several paths in the R-tree, depending on the query selectivity. From the experimental results, it follows that the number of additional page accesses required to search the R-tree is higher than the number of additional pages that have to be analyzed in T2 for the additional false hits. These results can be observed from Figure 8.31 and Figure 8.32. From the same figures it also follows that R is better than T1 for low selectivity ($< 10\%$) or for very small relations. Finally, in the performed experiments, T2 is better than T1. T1 and T2 have a similar behavior for very high selectivity. This, as already explained, is due to the choice of point P.

The reported results show that, similarly to the analysis of false hits, by augmenting the selectivity, the number of pages accessed by T2 and R increases. However, differently from the result obtained by the analysis of false hits, the number of pages accessed by T1 increases by increasing the selectivity. This is mainly due to the fact that T1, besides the generation of false hits, also generates duplicates. This aspect, together with the fact that the number of tuples belonging to the result increases by increasing the selectivity, increments the number of page accesses.

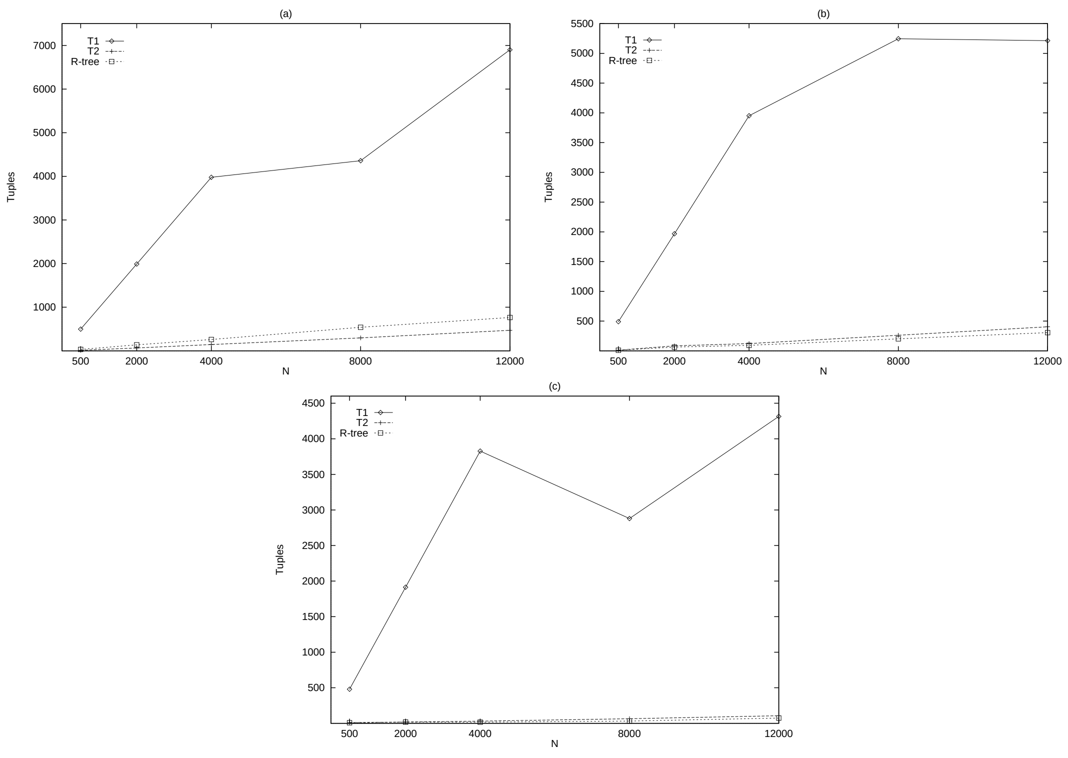


Figure 8.28: Comparison of the number of false hits for an EXIST selection, a selectivity lower than 10% and (a) large rectangles, (b) medium rectangles, (c) small rectangles.

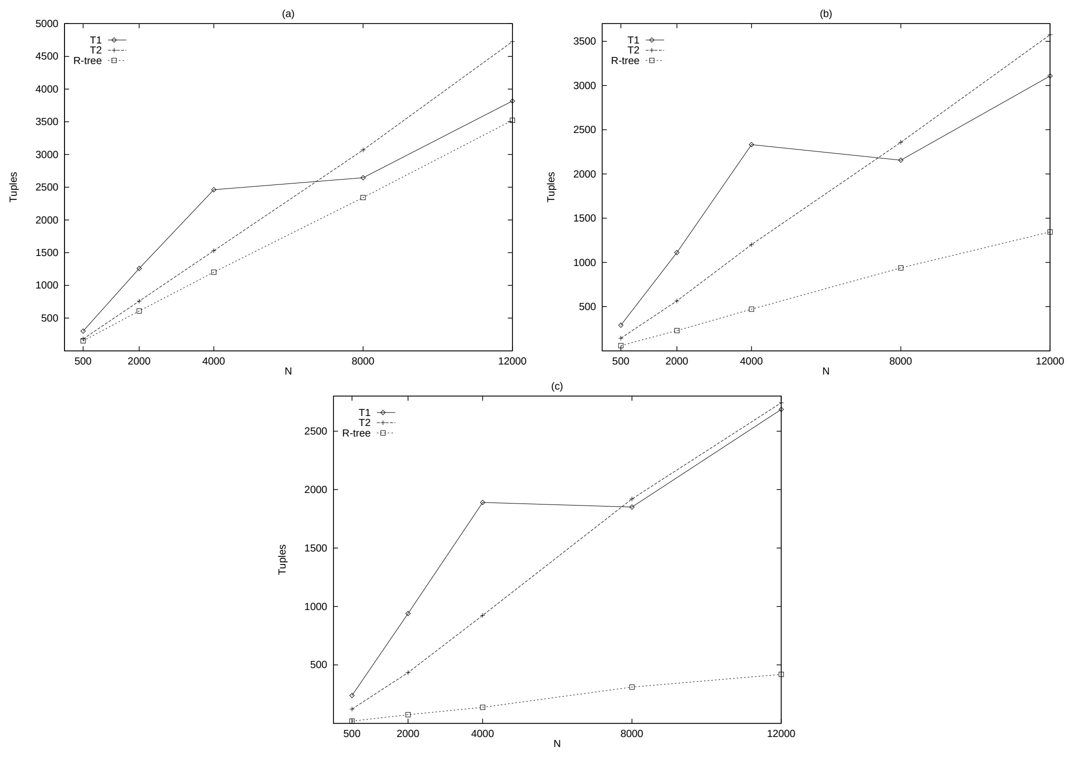


Figure 8.29: Comparison of the number of false hits for an EXIST selection, a selectivity of about 40% and (a) large rectangles, (b) medium rectangles, (c) small rectangles.

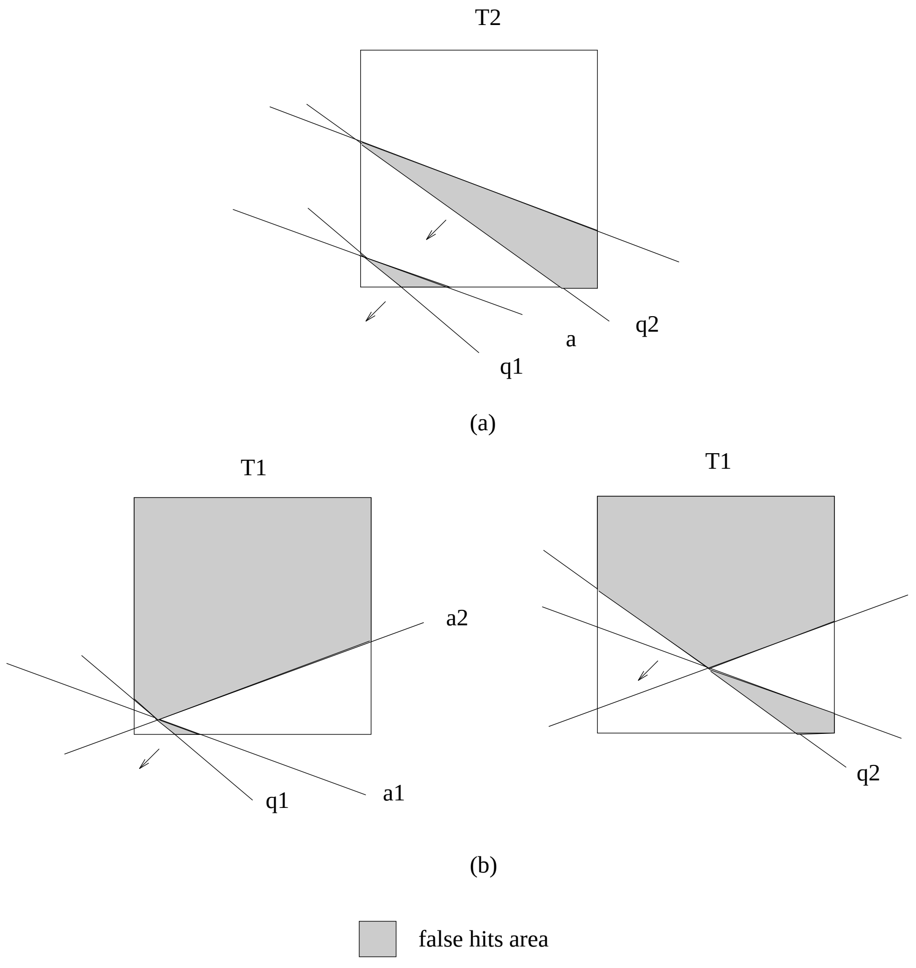


Figure 8.30: False hits area generated by (a) technique T2, (b) technique T1, with respect to queries having different selectivity. In the figure, q1 represents a high selectivity query and q2 represents a low selectivity query. a,a1, and a2 represents lines associated with the approximating query half-planes.

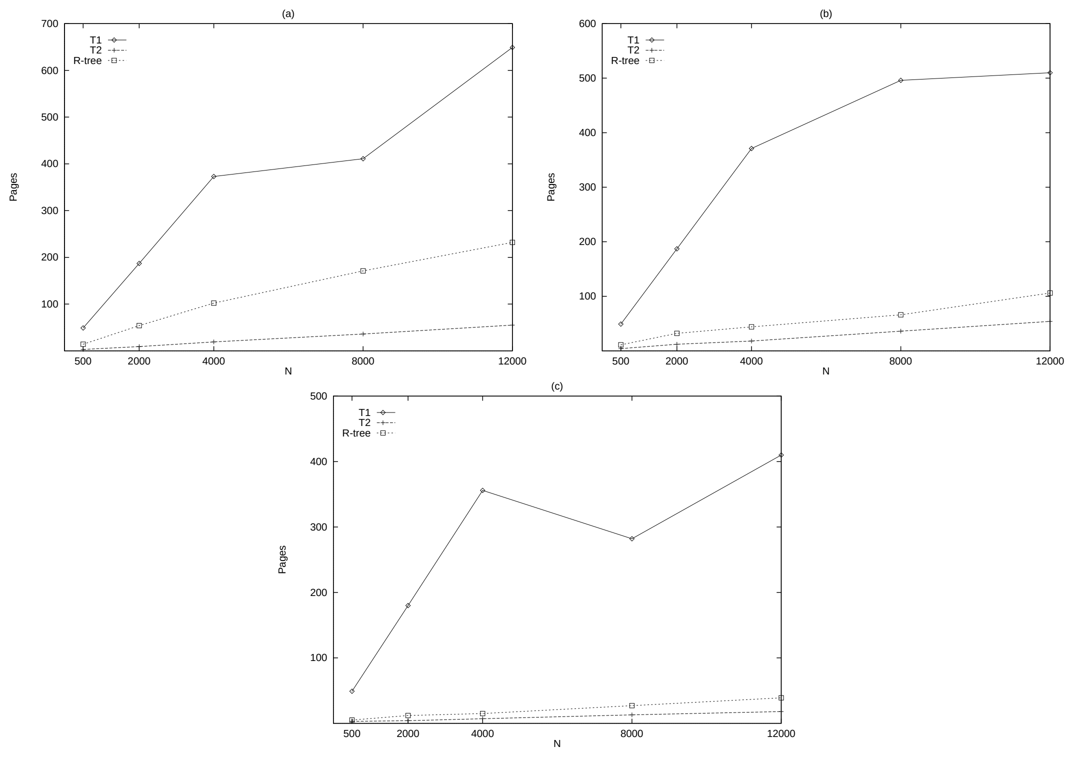


Figure 8.31: Comparison of the number of page accesses for an EXIST selection, a selectivity lower than 10% and (a) large rectangles, (b) medium rectangles, (c) small rectangles.

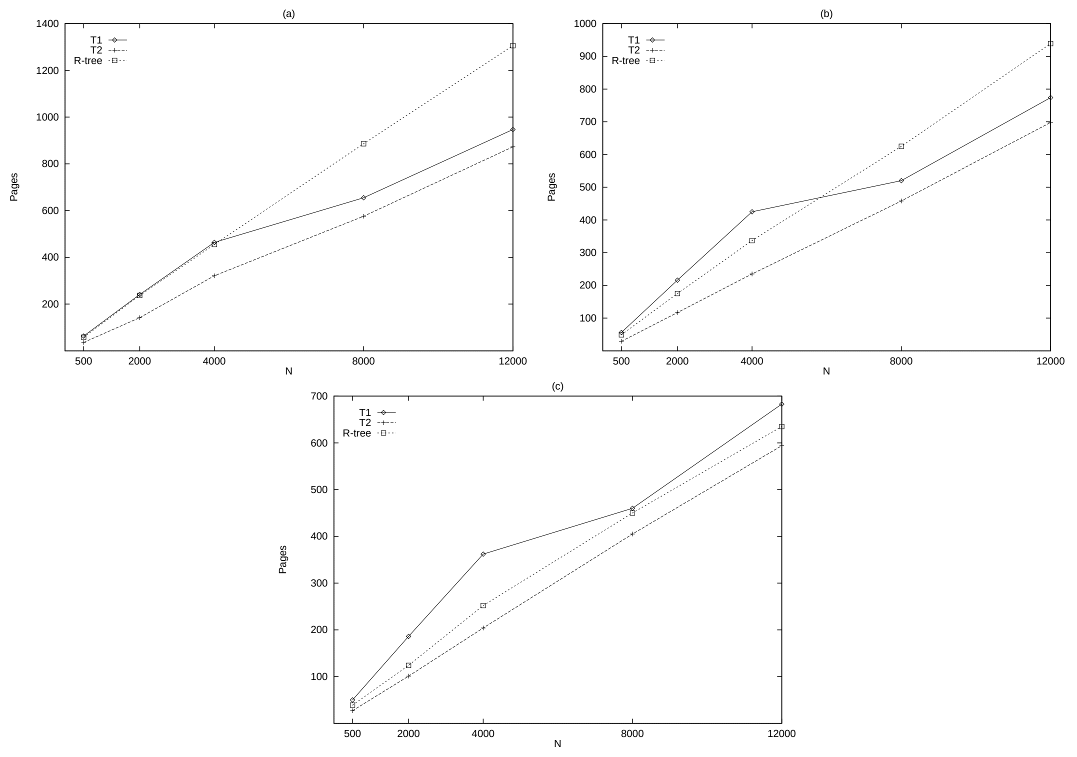


Figure 8.32: Comparison of the number of page accesses for an EXIST selection, a selectivity of about 40% and (a) large rectangles, (b) medium rectangles, (c) small rectangles.

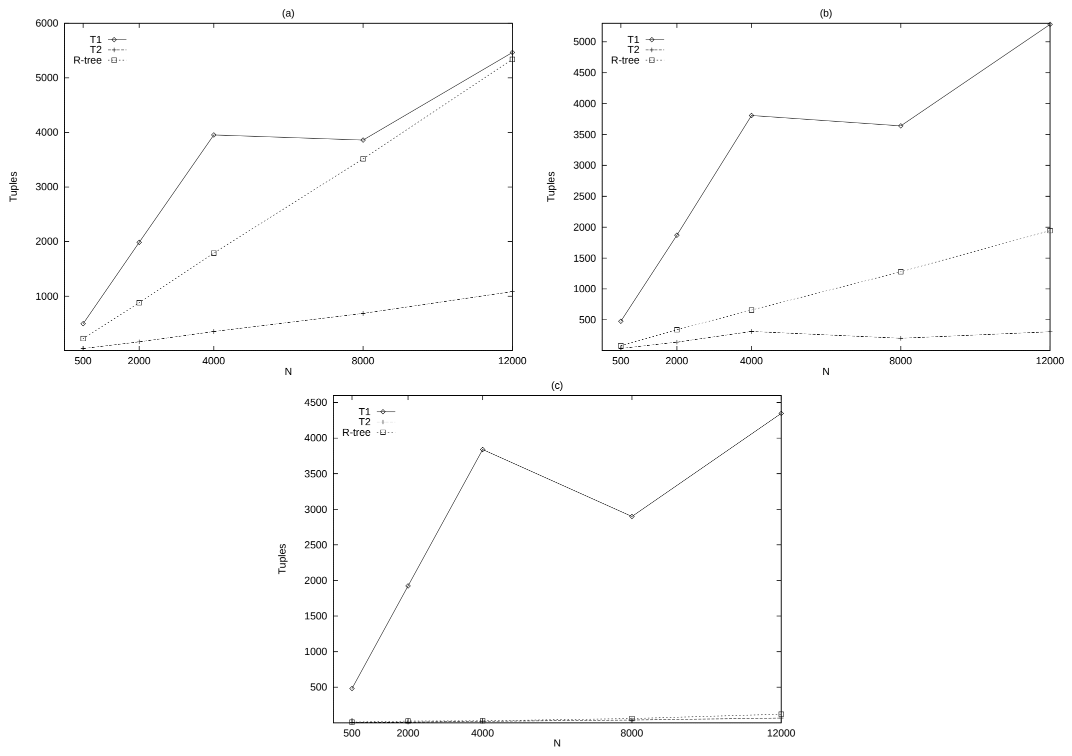


Figure 8.33: Comparison of the number of false hits for an ALL selection, a selectivity lower than 10% and (a) large rectangles, (b) medium rectangles, (c) small rectangles.

8.10.2.2 ALL selections

False hits. By analyzing the number of false hits generated by T1, T2, and R with respect to ALL selections, we can see that T2 almost always generates the lowest number of false hits. On the other hand, the performance of R is much worst for ALL selections than for EXIST selections, with the same selectivity. This is due to the fact that the search in the tree for an ALL selection coincides with a search in the tree for a corresponding EXIST selection and therefore the degree of filtering is lower. As a particular case, for large rectangles and low selectivity, R is much more close to T1 than in the corresponding EXIST case. For high selectivity and large generalized relations, T1 is better than R. These results can be observed from Figure 8.33 and Figure 8.34.

Page accesses. Considerations presented for the analysis of false hits still holds for

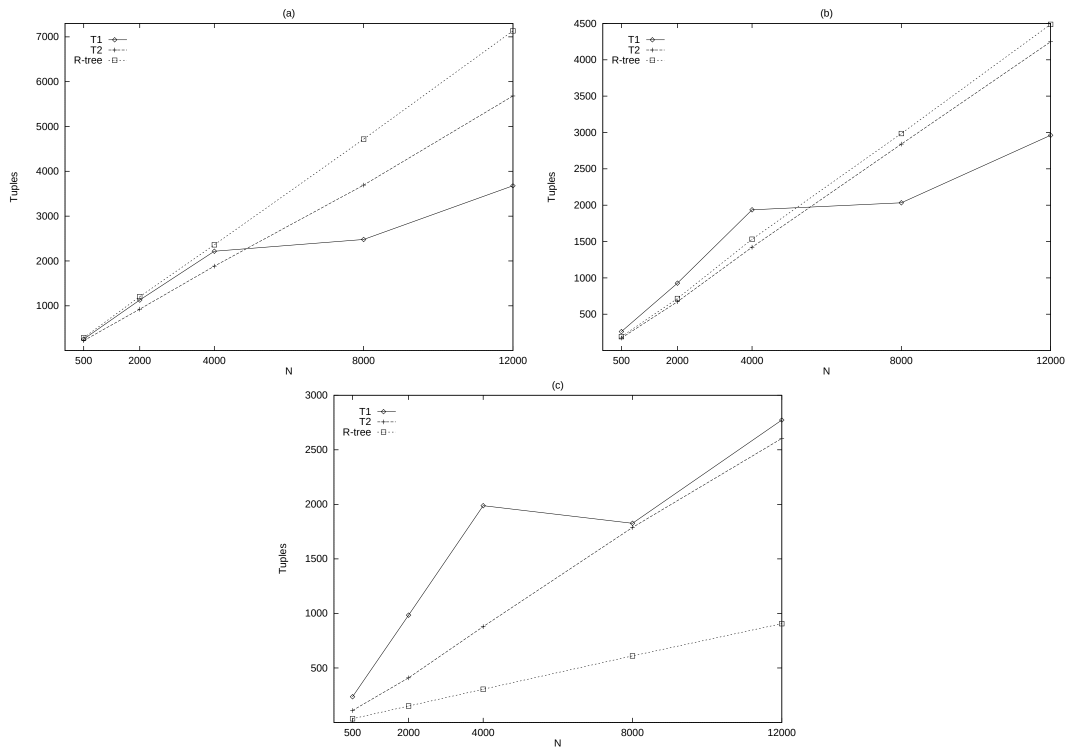


Figure 8.34: Comparison of the number of false hits for an ALL selection, a selectivity of about 40%, and (a) large rectangles, (b) medium rectangles, (c) small rectangles.

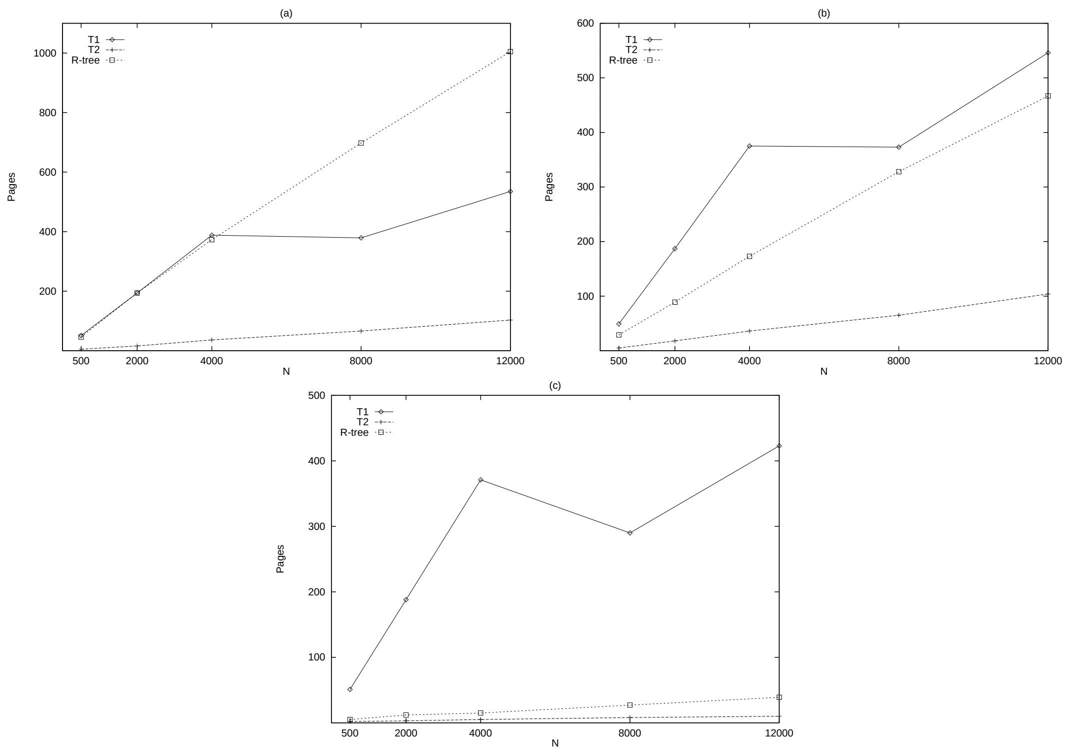


Figure 8.35: Comparison of the number of page accesses for an ALL selection, a selectivity lower than 10% and (a) large rectangles, (b) medium rectangles, (c) small rectangles.

the analysis of page accesses. Also in this case, R performs better with respect to EXIST selections. These results can be observed from Figure 8.35 and Figure 8.36.

8.11 Concluding remarks

In this chapter, we have analyzed the use of a dual representation for spatial objects to index constraint databases. The main advantage of such an approach is that the dual representation is defined whatever the space dimension is. Based on this representation, we have shown how EXIST and ALL selections can be performed in optimal time and space with respect to a half-plane having a fixed direction. When this condition is not satisfied, we have presented three approximation techniques, approximating the original query by applying a filtering-refinement approach.

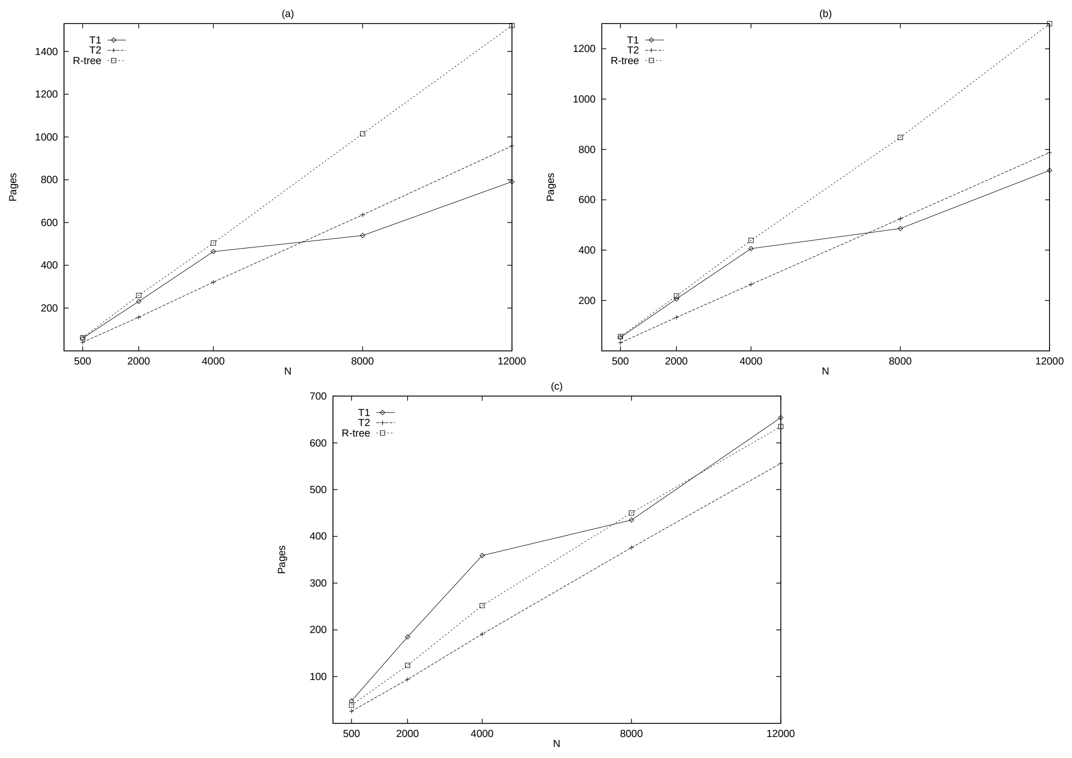


Figure 8.36: Comparison of the number of page accesses for an ALL selection, a selectivity of about 40% and (a) large rectangles, (b) medium rectangles, (c) small rectangles.

The proposed techniques have been implemented and compared. They have also been compared with respect to R-trees, a well-known spatial data structures. The obtained results show that, on relations containing unbound objects, where R-trees cannot always be applied, T2 has the better performance.

On closed generalized relations, T2 has the better time performance. However, on relations containing objects with small size, R-tree performs very well and its performance is very close to that of T2. Note that this is the typical assumption in spatial databases. However, in a general constraint database setting, this is not a reasonable assumption. In this case, T2 guarantees the better performance independently of the object size.

The performed experiments also point out an important difference between R-trees and the proposed data structures: the performance of a search based on R-trees depends on the size of the query object. On the other hand, the performance of the proposed techniques depend on the size of the false hits area generated by the approximation.

It is important to remark that, since experiments have been performed by considering $k = 2$, only two B^+-trees are constructed and the space overhead of using T2 instead of R is not remarkable. Better results for T2 can be obtained by increasing the number of sectors, for example by considering $k = 4$. On the other hand, R-tree performance can be improved by considering some R-tree variant such as R^+-trees or R^*-trees [130].

Chapter 9

An indexing technique for segment databases

As we have seen in Chapter 6, sometimes spatial objects are internally represented as the set of their boundary segments. Often, segments are not crossing but possibly touching (also called *NCT segments*). NCT segments find several applications in geographical information systems, since they often represent the basic representation for geographical data. Since, as we have seen in Chapter 2, each generalized tuple with d variables can be seen as the symbolic representation of a d-dimensional spatial object, an interesting issue is to analyze how indexing data structures defined for segment databases can be applied to constraint databases.

Few approaches have been proposed to index segment databases with good worst-case complexities. In particular, as we have seen in Chapter 6, the stabbing query problem, related to the detection of all segments intersecting a given vertical line, has been deeply investigated and an optimal dynamic solution has been recently proposed [7]. The aim of this chapter is to propose a close to optimal complexity solution for a more general problem for segment databases. The investigated problem concerns the detection of all NCT segments intersecting a given segment, having a fixed direction. Two solutions are proposed to solve this problem. The second solution uses the fractional cascading technique to improve the time complexity of the first proposed approach. The obtained complexity bounds are very close to the optimal ones.

Since the proposed approach relies on a segment representation of spatial objects representing the extension of generalized tuples, before introducing the proposed techniques, we analyze how indexing techniques for NCT segments can be applied to constraint databases, retaining good complexity results in the number of generalized tuples stored in the database.

The chapter is organized as follows. In Section 9.1, the relationships between segment and constraint databases are investigated. In Section 9.2 the new considered problem is introduced, together with some motivations, and the basic idea of our approach is outlined. In Section 9.3, a preliminary data structure is developed, whereas in Sections 9.4 and 9.5 two different solutions to the considered problem are presented. Finally, Section 9.6 presents some preliminary experimental results.

9.1 Segment and constraint databases

As we have already pointed out, the extension of each generalized tuple with d variables on LPOLY represents a, possibly open, polyhedra in a d-dimensional space. In the following, we assume that the considered generalized database contains only generalized tuples with two variables, thus representing 2-dimensional object, representing convex objects (i.e., disjunction is not used).

Each generalized tuple of the previous type can be represented as a set of segments, constituting the boundary of the generalized tuple extensions. The number of edges of such polygon corresponds to the number of non-redundant constraints[1] in the generalized tuple. In particular, it has been observed in [66] that for each generalized tuple t, expressed by using LPOLY and containing k variables, there exists a generalized tuple t' such that $t \equiv_r t'$ and t' contains at most $(k+1)$ constraints. Thus, t' has been generated from t by removing redundant constraints.

By assuming this representation, if the generalized relation contains N generalized tuples and each generalized tuple contains at most k constraints, the generalized database can be seen as a segment database containing $O(kN)$ segments. Such segments are not necessarily NCT segments, since they may intersect. We can, however, generate NCT segments by creating four new segments for each pair of intersecting segments, corresponding to the non-intersecting parts of the two segments. After this process, the number of segments (denoted by $TI(D)$, for a relational constraint database D) is $O(N^2)$. Indexing techniques for segment databases can then be used for indexing spatial objects and therefore constraint databases. This approach, even if appealing, introduces two main problems:

- each generalized tuple is represented at most k times inside the indexing structure;
- in the worst case, the number of indexed segments is quadratic in the number of generalized tuples. This means that, for example, indexing techniques that

[1]Let t be a generalized tuple and C a constraint. C is redundant with respect to t if $t \equiv t \wedge C$

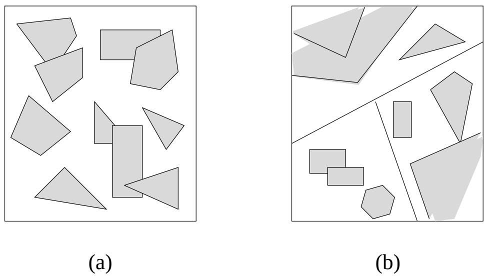


Figure 9.1: Example of $O(N)$ databases: a) SCP database: b) DG database.

guarantee a linear space with respect to the number of segments contained in the database require a quadratic space in the number of generalized tuples. In spatial and constraint databases this often cannot be considered acceptable.

From the previous discussion it follows that indexing techniques defined for segment databases can be safely and efficiently applied to constraint databases only in some particular cases, for example, when the number of intersections among the NCT segments is $O(N)$. In this case, we retain a linear space complexity in the number of the generalized tuples, i.e., in the number of spatial objects.

In general, $TI(D)$ varies between $O(1)$ and $O(N^2)$. $TI(D)$ is $O(1)$ when most tuples are disjoint, whereas is $O(N^2)$ if each tuple intersects most of the other tuples. However, in many applications $TI(D)$ is far from the extreme cases. For example, if the objects associated with generalized tuples are partitioned into disjoint groups, or they are closed and of small size, the number of intersection is very likely to be $O(N)$. This leads to the definition of an interesting class of $O(N)$-*databases*. In particular, a $O(N)$-*database* is a database where the number of intersections among tuples is $O(N)$.

Table 9.1 lists two classes of constraint $O(N)$-databases. In the table K represents the length of the *database field* side. Given a constraint relational database D, the *database field* of D is a rectangular domain in the plane which contains *all* the intersections among the generalized tuples in D. For the sake of simplicity, we assume that the field is a square $K \times K$. A graphical representation of each class is presented in Figure 9.1. The class of *small closed polygons* (SCP for short) characterizes generalized relations in which generalized tuple extensions have a small size. On the other hand, the class of *disjoint group* generalized relations (DJ for short) characterizes generalized relations in which tuple extensions can be partitioned into non intersecting groups, containing a fixed number of generalized tuples.

The following theorem holds.

Relation type	**Conditions**
SCP (Small Closed Polygons)	1. K is fixed 2. Each tuple fits into a rectangle of size $O((K/N)^{1/4}) \times O((K/N)^{1/4})$ 3. Tuples are uniformly distributed in the database field
DG (Disjoint Groups)	1. The database field $K \times K$ can be split into some regions such that each tuple is completely contained into a single region 2. Each region contains up to a fixed number m of elements

Table 9.1: Characterization of some $O(N)$-databases.

Theorem 9.1 *Any SCP and DG generalized relation has $O(N)$ tuple intersections.*

Proof:

- *SCP databases.* Under the hypothesis, the probability that two polygons, corresponding to the extension of two generalized tuples, intersect is equal to the probability that their projections on the X-axis intersect and the projections on the Y-axis intersect. The probability that their projections on the X-axis intersect is equal to the probability that their projections on the Y-axis intersect and it is equal to $O((K/N)^{1/2})$. Thus, the probability that two polygons intersect is $O(K/N)$. This means that each polygon intersects $O(K)$ polygons and therefore, since K is fixed, the total number of intersections is $O(N)$.

- *DG databases.* Since m is a constant not depending on N, each region generates $O(m^2)$ intersections and the total number is $O(m^2) \times O(N/m) = O(Nm) = O(N)$. □

Several applications can be represented by using the classes of $O(N)$-databases introduced above. For example, all applications modeled by planar subdivisions (typical of GIS) can be represented by SCP $O(N)$-databases. For such classes of databases, indexing techniques proposed for segment databases can be efficiently applied. In particular, if the considered technique guarantees a linear space complexity in the number of NCT segments, the application of the same technique on the segment database, corresponding to the original constraint database, requires a linear space in the number of generalized tuples; it can therefore be considered efficient from a space complexity point of view.

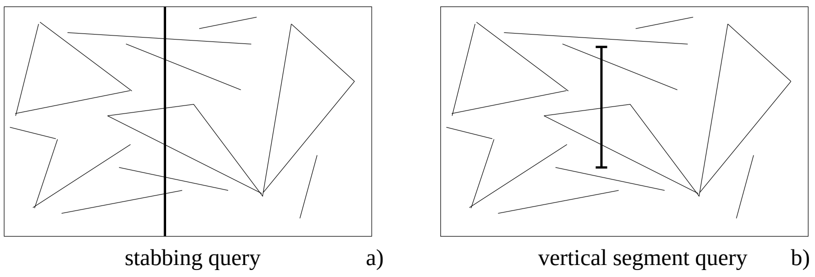


Figure 9.2: Vertical line queries vs vertical segment queries.

9.2 Towards optimal indexing of segment databases

As we have seen in Chapter 6, only few approaches have been proposed to answer queries on segment databases with optimal worst-case complexity. The most interesting problem for which an optimal worst-case complexity data structure has been proposed [7] is the *stabbing query problem*. Given a vertical line, a stabbing query determines all segments which are intersected by this line.

A more general and relevant problem in segment databases (especially for GIS) is to determine all segments intersecting a given query segment. In the following we go one step towards the solution of this problem by investigating a weaker problem, consisting in determining all segments intersected by a given generalized query segment (a line, a ray, a segment), having a fixed angular coefficient. Without leading the generality of the discussion, we consider vertical query segments.[2] The corresponding query is called *VS query* (see Figure 9.2).

There exists a difference in the optimal time complexity between VS and stabbing queries even in internal memory. A space optimal solution for solving VS queries in internal memory has $O(\log^2 N + T)$ query complexity, uses $O(N)$ space and performs updates in $O(\log N)$ time [42] (with update we mean the insertion/deletion of a segment non-crossing, but possibly touching, the already stored ones). On the other hand, the optimal solution for solving stabbing queries in internal memory has $O(\log N + T)$ query complexity, uses $O(N)$ space and performs updates in $O(\log N)$ time [43]. VS queries are therefore inherently more expensive than stabbing queries. In the following, we propose a $O(n \log_2 B)$ space solution having query I/O complexity very close to $O(\log_B^2 n + t)$. The solution is proposed for static and semi-dynamic (thus, allowing insertion) cases.

The data structures we propose to solve VS queries are organized according to two levels. At the top level, we use a primary data structure (called *first-level data*

[2]If the query segment is not vertical, coordinate axes can be appropriately rotated.

structure). One or more auxiliary data structures (called *second-level data structures*) are associated with each node of the first-level data structure. The second-level data structures are tailored to efficiently execute queries on a special type of segments, called *line-based segments*. A set of segments is line-based if all segments have an endpoint lying on a given line and all segments are positioned in the same half-plane with respect to such line. Thus, our main contributions can be summarized as follows:

1. We propose a data structure to store and query line-based segments, based on *priority search trees* (PST for short) [43, 76], similar to the internal memory data structure proposed in [42]. The proposed data structure is then extended with the P-range technique presented in [136], to reduce time complexity.

2. We propose two approaches to the problem of VS queries:

   - In the first approach, the first-level structure is a binary tree, whereas the second-level structure, associated with each node of the first-level structure, is a pair of priority search trees, storing line-based segments in secondary storage. This solution uses $O(n)$ blocks of secondary storage and answers queries in $O(\log n(\log_B n + IL^*(B)) + t)$ I/O's. We also show how updates on the proposed structures can be performed in $O(\log n + \frac{\log_B^2 n}{B})$ amortized time.[3]

   - In the second approach, to improve the query time complexity of the first solution, we replace the binary tree at the top level with a secondary storage interval tree [7]. The second-level structures are based on priority search trees for line-based segments and segment trees, enhanced with fractional cascading [40]. This solution uses $O(n \log_2 B)$ space and answers queries in $O(\log_B n(\log_B n + \log_2 B + IL^*(B)) + t)$ time. We also show how the proposed structure can be made semi-dynamic, performing insertions in $O(\log_B n + \log_2 B + \frac{\log_B^2 n}{B})$ amortized time.

9.3 Data structures for line-based segments

Let S be a set of segments. S is *line-based* if there exists a line l (called *base line*) such that each segment in S has (at least) one endpoint on l and all the segments in S having only one end-point on l are located in the same half-plane with respect to l.

In the following, we construct a data structure for storing line-based segments in secondary storage and retrieving all segments intersected by a query segment q which

[3]If a sequence of m operations takes total time t, the amortized time complexity of a single operation is t/m.

is parallel to the base line. More precisely, the query object q may be a segment, a ray, or a line.[4] Since a segment query represents the most complex case, in the following we focus only on such a type of queries. Moreover, without loss of generality, through out Section 9.3, we restrict the presentation to horizontal base lines. This choice simplifies the description of our data structure making it consistent with the traditional way of drawing data structures. The query thus becomes a horizontal segment as well.

The solution we propose for storing a set of line-based segments is based on the fact that there exists an obvious relationship between a segment query against a set of line-based segments on the plane and a 3-sided query against a set of points (see Figure 9.3). Given a set of points in the plane and a rectangular region open in one of its sides, the corresponding 3-sided query returns all points contained in the (open) rectangle. A segment query defines in a unique way a 3-sided query on the point database corresponding to all segment endpoints not belonging to the base line. On the other hand, the bottom segment of a 3-sided query on such point database corresponds to a segment query on the segment database. However, these corresponding queries do not necessarily return the same answers. Indeed, although both queries often retrieve the same data (segment 1 in Figure 9.3), this is not always true. The intersection of a segment with the query segment q does not imply that the segment endpoint is contained in the 3-sided region (segment 2 in Figure 9.3). Also, the presence of a segment endpoint in the 3-sided region does not imply that the query segment q intersects the segment (segment 3 in Figure 9.3). Despite these differences, solutions developed for 3-sided queries can be successfully applied to line-based segments as well.

In internal memory, priority search trees [101] are used to answer 3-sided queries in optimal space, query, and update time. All proposals to extend priority search trees, for use in secondary storage, do not provide both query and space optimal complexities [76, 120, 136]. In [76], a solution with $O(n)$ storage and $O(\log n + t)$ query time was developed. Two techniques have been defined to improve these results. *Path-caching* [120] allows us to perform 3-sided queries in $O(\log_B n)$, with $O(n \log_2 B \log_2 \log_2 B)$ space. A space optimal solution to implement secondary storage priority search trees is based on the *P-range tree* [136] and uses $O(n)$ blocks, performing 3-sided queries in $O(\log_B n + IL^*(B) + t)$ and updates in $O(\log_B n + \frac{log^2_B n}{B})$, where $IL^*(B)$ is a small constant, representing the number of times we must repeatedly apply the $\log^*$ function to B before the result becomes ≤ 2.[5]

[4]We assume that the query object belongs to the same half-plane where segment endpoints belong. Otherwise, no segment intersects the query.

[5]Unless otherwise stated all logarithms are given with respect to base 2.

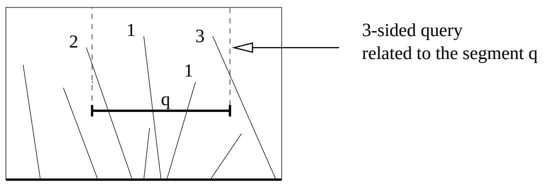


Figure 9.3: A segment query on a set of line-based segments vs a 3-sided query on the endpoint set of the same segments.

As in the approach presented in [120, 136] for point databases, in order to define priority search trees for a set of line-based segments, a binary decomposition is first used and algorithms for retrieving all segments intersected by the query segment are developed. As a result, we obtain a binary tree structure in secondary storage of height $O(\log\ n)$, which meets all conditions required in [120, 136] for applying any advanced technique between path-caching and P-range tree.

Data structure. Let S be a set of N line-based segments. We first select a number B of segments from S with the topmost y-value endpoints and store them in the root r of the tree Tr under construction, ordered with respect to their intersections with the base line.[6] The set containing all other segments is partitioned into two subsets containing an equal number of elements. The top segment in each subset is then copied into the root. These segments are denoted respectively by $left$ and $right$. A separator low is also inserted in the root, which is a horizontal line separating the selected segments from the others. Line low for a generic node v is denoted by $v.low$. A similar notation is used for $left$ and $right$ (see Figure 9.4). If $v.left.y$ ($v.right.y$) denotes the top y-value of segment $v.left$ ($v.right$) and $v.low.y$ denotes the y-value of line $v.low$, then $v.left.y \leq v.low.y$ and $v.right.y \leq v.low.y$.

The decomposition process is recursively repeated for each of the two subsets. Like external priority search trees in point databases, the resulting tree Tr is a balanced binary tree of height $O(\log\ n)$, occupying $O(n)$ blocks in external storage. The difference, however, is that no subtree in Tr defines a rectangular region in the plane. Indeed, in a point database, a vertical line is used as a separator between points stored in left and right subtrees. Instead, in a segment database, the line separating segments stored in left and right subtrees is often biased (see Figure 9.4).

Search algorithm. Let q be a horizontal query segment. We want to find all segments intersecting q. The search algorithm is based on the comparison of q with stored segments.[7] The search is based on two functions $Find$ and $Report$. Function

[6]Note that the construction guarantees that each node is contained in exactly one block.

[7]Note that this is different from the approach usually used in PST for point databases. In that

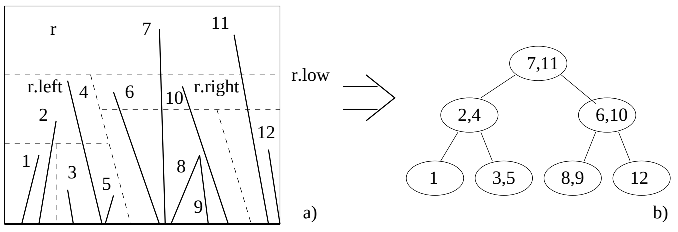


Figure 9.4: External PST for line-based segments (a) and corresponding binary tree (b), assuming $B = 2$. Non-horizontal dashed lines do not exist in the real data structure and are given only for convenience. Moreover segments $v.left$ and $v.right$, as well as line $v.low$, are shown only for the root r.

Find is to locate the deepest-leftmost (deepest-rightmost) segment intersected by the query q, with respect to its storage position in Tr, and the node in Tr where the segment is located. Function *Report* then uses the result of function *Find* to retrieve all segments in Tr intersected by q, starting from the deepest-leftmost and deepest-rightmost segments intersecting the query. The algorithms for such functions are presented in Figures 9.5 and 9.6. Figure 9.7 illustrates some different cases, considered in function *Find*. Figure 9.8 shows all nodes visited by the *Report* algorithm. The algorithms are similar to those presented in [42] for the internal memory intersection problem, and satisfy the following properties:

- Function *Find* maintains in a queue Q the nodes that should be analyzed to find the deepest-leftmost (the deepest-rightmost) segment intersecting the query segment. It can be shown that Q contains at most two nodes for each level of Tr, thereby assuring that the answer is found in $O(\log n)$ steps. Moreover, a constant space $O(1)$ is sufficient to store Q [42].

- Function *Report* determines the deepest-leftmost and the deepest-rightmost segments intersecting the query, using function *Find*. Then, it visits the subtree rooted at the common ancestor of the nodes containing such segments and the path from such ancestor to the root of the tree. It can be proved that the number of nodes of such subtree, containing at least one segment non-intersecting q, is $O(\log n + t)$ [42].

The following lemma follows from the previous considerations.

case, the comparison is performed against region boundaries. Such an approach is not possible for line-based segments, since no rectangular region is related to any subtree of Tr.

```
Algorithm 9.1
input : a PST $T$ for a set of $N$ line-based segments
        a query segment $q$
output : the deepest-leftmost segment intersected by the query segment $q$,
          with respect to its storage position in $T$
        the node in $T$ where the segment is located
begin
let $q$ be the segment $q.l \leq x \leq q.r$, $y = q.y$
let $(v.left.x, v.left.y)$ be the upper endpoint of segment $v.left$
let $(v.right.x, v.right.y)$ be the upper endpoint of segment $v.right$
let $Q$ be the empty queue
Initialize $Q$ with the tree root
$answer_n \leftarrow \infty$; $answer_s \leftarrow \infty$;
repeat
  Extract a block $v$ from the front of $Q$
  if some segments in $v$ intersect $q$ then
    select the leftmost segment in $v$ and $answer_s$ and assign it to $answer_s$
    select the block of $Tr$ containing $answer_s$ and assign it to $answer_n$
    $Q$ is updated according to the following cases:
      if $q.y \geq v.low$ then
        no segments in the blocks having $v$ as ancestor in $Tr$ can intersect $q$ and the queue
          remains unchanged
      else

        a) if $q.y > v.left.y$ and $q.y > v.right.y$ then $Q$ remains unchanged (Figure 9.7.a)
              endif
        b) if $q.y \leq v.left.y$ and $q.l$ is on the left of $v.left$ then the left son of $v$ is added
              at the end of $Q$ (Figure 9.7.b). In symmetric case if $q.y \leq v.right.y$ and $q.l$
              is on the right of $v.right$, the right son of $v$ is added at the end of $Q$ endif
        c) if $q.y \leq v.left.y$, $q.y > v.right.y$ and $q.l$ lies between $v.left$ and $v.right$ then the
              left son of $v$ is added at the end of $Q$ (Figure 9.7.c). Symmetric case is
              treated similarly endif
        d) if $q.y \leq v.left$, $q.y \leq v.right$ and $q.l$ lies between $v.left$ and $v.right$ then we
              empty queue $Q$ and then insert both sons in it (Figure 9.7.d) endif

      endif
until $Q$ is empty
return $answer_s$ and $answer_n$
end
```

Figure 9.5: Function **Find**.

Lemma 9.1 *[42] Let S be a set of line-based segments. Let q be a query segment. Let Tr be the priority search tree for S, constructed as above. Then:*

1. *Function Find returns the deepest-leftmost (the deepest-rightmost) segment bl_l (bl_r) in S intersected by q and the node it belongs to in $O(\log n)$ I/O's.*

2. *All T segments in S intersected by q can be found from Tr, bl_l and bl_r, in $O(\log n + \frac{T}{B})$ I/O's.* □

The following result summarizes the costs of the proposed data structure.

Lemma 9.2 *[42] N line-based segments can be stored in a secondary storage priority search tree having the following costs: (i) storage cost is $O(n)$; (ii) horizontal segment query cost is $O(\log n + t)$, where t is the number of the detected intersections.* □

```
Algorithm 9.2
input : a PST T for a set of N line-based segments
        a segment query q
output : all segments stored in Tr and intersected by q
begin
let q be the segment $q.l \leq x \leq q.r, y = q.y$
Apply function Find to Tr and q
let $s_l$ and $bl_l$ be the segment and the block containing $s_l$ located by function Find, respectively
if $bl_l = \infty$ then q does not intersect any segment
else
  Apply the symmetric version of function Find (Find') to Tr and q,
    retrieving the deepest-rightmost segment intersected by q and the node where it is contained
  let $s_r$ and $bl_r$ be the segment and the block containing $s_l$ located by function Find', respectively
  let lca be the lowest common ancestor of $bl_l$ and $bl_r$ in Tr
  let $P_l$ and $P_r$ be the paths from $bl_l$ to lca and from $bl_r$ to lca, respectively
  Walk up $P_l$
  for each node v in $P_l$ do
    retrieve all segments in v intersected by q
    if (case 1) $v = bl_l$ or (case 2) $v \neq lca$ and the predecessor of v on $P_l$ is a left child of v then
      if (case 1) then $z = v$ else (case 2) $z = v$'s right child endif
      perform a preorder traversal of the sub-tree rooted at z
      for each visited node w do
        retrieve all segments in w which intersect q
        if $w.low < q.y$ or w does not contain any segment intersecting q then
          do not proceed the traversal in w's children endif
      endfor
    endif
  endfor
  The above steps are repeated also on $P_r$, with "left" and "right" interchanged
  let $\overline{P}$ be the path from lca to the root of the tree
  Walk up $\overline{P}$
  for each node v in $\overline{P}$ do
    retrieve all segments in v intersected by q
  endfor
endif
end
```

Figure 9.6: Function **Report**.

Despite the difference in the query results between a 3-sided query on a point database and a segment query on a segment database (see Figure 2), either the path-caching [120] or the P-range tree [136] methods can be applied for reducing the search time in a segment database, using an external PST. Since we will use an external PST on each level of the first-level data structure we are going to develop (see next section), we choose a linear memory solution based on P-range trees, obtaining an optimal space complexity in the data structure for storing line-based segments.

The application of the P-range tree technique to an external PST for line-based segments requires only one minor modification of the technique described in [136]. As for PST, the vertical line separator should be replaced by the queue Q and several procedures needed for the queue maintenance. Then, a comparison of a query point against a vertical line separator in a point database is replaced by the check of at most two nodes in queue Q, during the search in a segment database. Since the detection of the next-level node and the queue maintenance in a segment database takes O(1) time, this substitution does not influence any properties of the P-range tree technique.

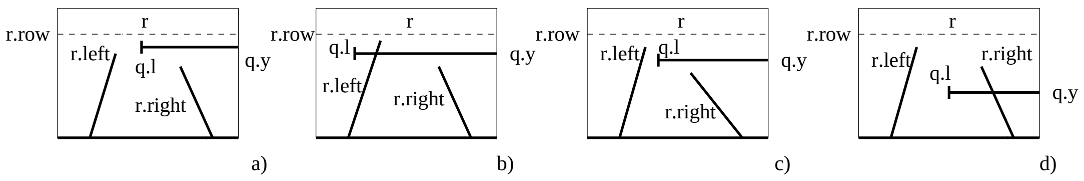


Figure 9.7: Different cases in function *Find*.

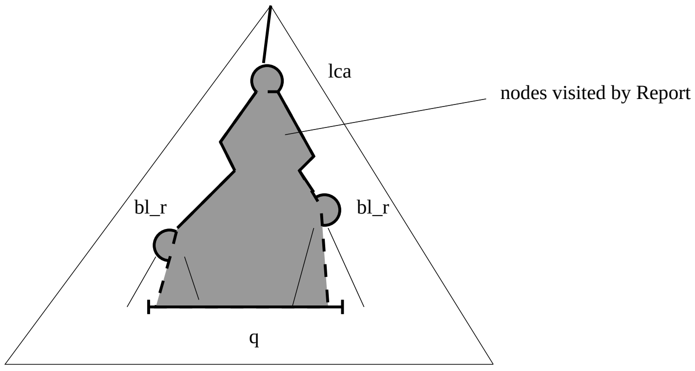


Figure 9.8: Search space visited by the *Report* algorithm.

This proves the following lemma.

Lemma 9.3 *N line-based segments can be stored in a secondary storage data structure having the following costs: (i) storage cost is* $O(n)$*; (ii) horizontal segment query cost is* $O(\log_B n + IL^*(B) + t)$ *I/O's; (iii) update amortized cost is* $O(\log_B n + \frac{\log_B^2 n}{B})$. □

9.4 External storage of NCT segments

In order to determine all NCT segments intersecting a vertical segment, we propose two secondary storage solutions, based on two-level data structures (denoted by 2LDS). Second-level data structures are based on the organization for line-based segments presented in Section 9.3. In the following, we introduce the first proposed data structure; the second one will be presented in Section 9.5.

First-level data structure. The basic idea is to consider a binary tree as first-level structure. With each node v of the tree, we associate a line $bl(v)$ (standing for *base line* of v) and the set of segments intersected by the line. More formally, let N be a set of NCT segments. We order the set of endpoints corresponding to such segments in ascending order according to their x-values. Then, we determine a vertical line partitioning such ordered set in two subsets of equal cardinality and we associate such vertical line with the base line $bl(r)$ of the root. All segments intersecting $bl(r)$ are associated with the root whereas all segments which are on the left (right) of $bl(r)$ and do not intersect it, are passed to the left (right) subtree of the root. The decomposition recursively continues until each leaf node contains B segments and fits as a whole in internal memory. The construction of base lines guarantees that the segments in a node v are intersected by $bl(v)$ but are not intersected by the base line of the parent of v. The tree height is $O(\log n)$.

Second-level data structures. Because of the above construction, each segment in an internal node v either lies on $bl(v)$ or intersects it. The segments which lie on the base line are stored in $C(v)$, an external interval tree [7] which requires a linear number of storage blocks and performs a VS query in $O(\log_B n + t)$ I/O's. Each segment which is intersected by $bl(v)$ has left and right parts. Left and right parts of all the segments are collected into two sets, called $L(v)$ and $R(v)$, respectively. Each of these sets contains line-based segments and can be efficiently maintained in secondary storage using the technique proposed in Section 9.3. Totally, each segment is represented at most twice inside the two-level data structure. Therefore, the tree stores N segments in $O(n)$ blocks in secondary storage. Figure 9.9 (b) illustrates the organization and content of the proposed 2LDS, for the set of segments presented in Figure 9.9 (a).

Search algorithm. Given a query segment of the form $x = x_0, a \leq y \leq b$, the search is performed on the first-level tree as follows. We scan the tree and visit exactly one node v for each level. In each node v, we first verify if x_0 equals the x-coordinate of the vertical line $bl(v)$. In such a case, all segments in $C(v)$, $L(v)$ and $R(v)$ intersected by q are retrieved and the search stops. Otherwise, if x_0 is lower than the x-coordinate of $bl(v)$, we visit only $L(v)$ and move to the left son of v. If x_0 is greater than the x-coordinate of $bl(v)$, we visit only $R(v)$ and move to the right son. The search for all segments T' inside one node intersected by the query requires $O(\log_B n + IL^*(B) + \frac{T'}{B})$ time. Since the height of the first-level data structure is $O(\log n)$ and each segment is reported at most twice,[8] the I/O complexity of the total search is $O(\log n(\log_B n + IL^*(B)) + t)$.

Updates. If updates are allowed, the binary tree should be replaced by a dynamic search-tree, for which efficient rebalancing methods are known. To maintain insertions and deletions of line-based segments in the data structure described above, we replace the binary tree with a $BB[\alpha]$-tree [42, 108], $0 < \alpha < 1 - 1/\sqrt{2}$. We store balance

[8]A segment is reported twice only if it intersects q and it is contained in a node v such that x_0 equals the x-coordinate of $bl(v)$.

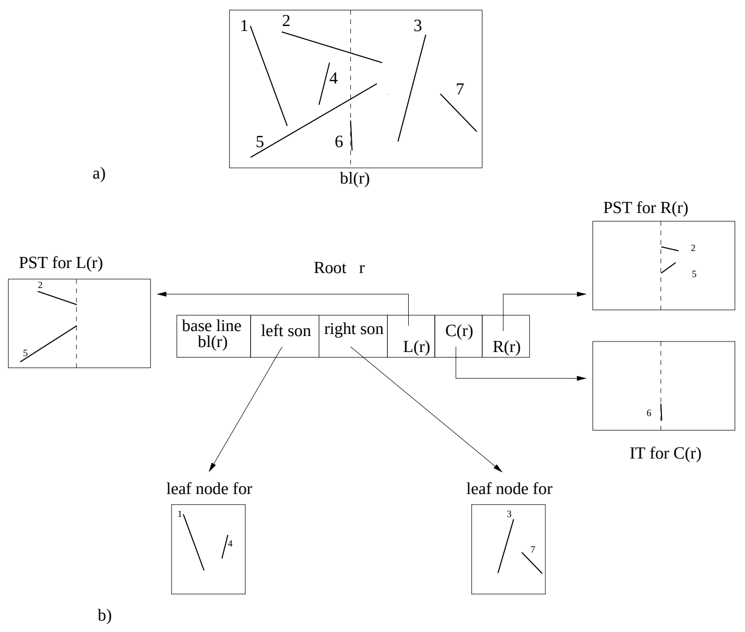


Figure 9.9: a) A set of 7 NCT segments; b) the corresponding data structure ($B = 2$, PST stands for priority search tree, IT stands for interval tree).

values in internal nodes of the $BB[\alpha]$-tree and maintain the optimal $O(\log\ n)$ height of the tree by performing $O(\log\ n)$ single or double rotations during an update. The update cost consists of $O(\log\ n)$ operations for the search and balance maintenance in the first-level tree and $O(\log_B n + \frac{\log^2_B n}{B})$ operations for updating the second-level data structures. Therefore, the total update cost is $O(\log\ n + \frac{\log^2_B n}{B})$. The cost is $O(\log\ n)$ for all real values of n (more exactly, for $n \in O(2^B)$).

Theorem 9.2 *N NCT segments can be stored in a secondary storage data structure having the following costs: (i) storage cost is $O(n)$; (ii) VS query time is $O(\log\ n(\log_B n + IL^*(B)) + t)$; (iii) update time is $O(\log\ n + \frac{\log^2_B n}{B})$.* □

9.5 An improved solution to query NCT segments

In order to improve the complexity results obtained in the previous section, a secondary storage interval tree, designed for solving stabbing queries [7], is used as first-level data structure, instead of the binary tree. This modification, together with the use

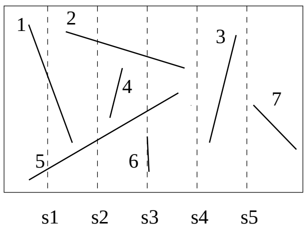


Figure 9.10: Partition of the segments by lines s_i.

of the fractional cascading technique [40], improves the wasteful factor log n in the complexity results presented in Theorem 9.2, but uses $O(n \log_2 B)$ space.

9.5.1 First-level data structure

The *interval tree* is a standard dynamic data structure for storing a set of 1-dimensional segments [7, 52], tailored to support stabbing queries. The tree is balanced over the segment endpoints, has a branching factor b, and requires $O(n)$ blocks for storage. Segments are stored in secondary structures associated with the internal nodes of the tree.

As first level data structure, we use an external-memory interval tree and we select b equal to $B/4$. The height of the first-level structure is therefore $O(\log_b n) = O(\log_B n)$. The first level of the tree partitions the data into $b + 1$ slabs separated by vertical lines $s_1, \ldots, s_b$. In Figure 9.10, such lines are represented as dashed lines. In the example, b is equal to 5. Multislabs are defined as contiguous ranges of slabs such as, for example, $[1 : 4]$. There are $O(b^2)$ multislabs in each internal node. Segments stored in the root are those which intersect one or more dashed lines s_i. Segments that intersect no line are passed to the next level (segments 3, 4 and 7 in Figure 9.10). All segments between lines s_{i-1} and s_i are passed to the node corresponding to the i-th slab. The decomposition continues until each leaf represents B segments.

9.5.2 Second-level data structures

In each internal node of the first-level tree, we split all segments which do not lie on dashed lines s_i into *long* and *short* fragments. A long fragment spans one or more slabs and has both its endpoints on dashed lines. A short fragment spans no complete slab and has only one endpoint on the dashed line. Segments are split as follows (see Figure 9.11). If a segment completely spans one or more slabs, we split it into one long (central) fragment and at most two short fragments. The long fragment is obtained by splitting the segment on the boundaries of the largest multislab it spans. After

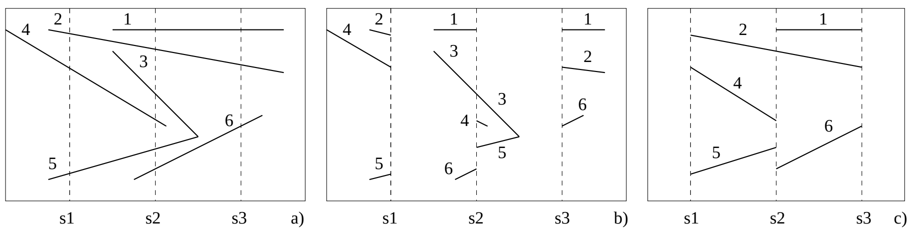


Figure 9.11: The splitting of segments: a) segments associated with a node; b) short fragments; c) long fragments.

this splitting, at most two additional short segments are generated. If a segment in the node intersects only one dashed line and spans no slab, it is simply split into two short fragments. In total, if k segments are associated with a node, the splitting process generates at most k long and $2k$ short fragments.

As before, segments lying on a dashed line s_i are stored in an external interval tree C_i. Short ad long fragments are stored as follows.

Short fragments. All short fragments are naturally clustered according to the dashed line they touch. Note that short fragments having one endpoint on line s_i are line-based segments and can be maintained in an external priority search tree as described in Section 9.3. Short line-based fragments which are located on the left of s_i are stored in an external PST L_i. Symmetrically, short fragments on the right side of s_i are stored in an external PST R_i. Totally, an internal node of the first-level structure contains $2b$ external PSTs for short fragments.

Long fragments. We store all long segments in an additional structure G which is essentially a segment tree [5, 43] based on dashed lines s_i, $i = 1, \ldots, b$. G is a balanced binary tree with $b - 2$ internal nodes and $b - 1$ leaves. Thus, in total it has $O(B)$ nodes.

Each leaf of the segment tree G corresponds to a single slab and each internal node v is associated with the multislab $I(v)$ formed by the union of the slabs associated with the leaves of the subtree of v. The root of G is associated with the multislab $[1 : b]$. Given a long fragment l which spans many slabs, the *allocation nodes* of l are the nodes v_i of G such that l spans $I(v_i)$ but not $I(par(v_i))$, where $par(v)$ is the parent of v in G. There are at most two allocation nodes of l at any level of G, so that, since the height of the segment tree is $\log_2 B$, l has $O(\log_2 B)$ allocation nodes [43].

Each internal node v of G is associated with a multislab $[i : j]$ and is associated with the ordered list (called *multislab list* $[i : j]$) of long fragments having v as allocation node, cut on the boundaries of $I(v)$. A B$^+$-tree is maintained on the list for fast retrieval and update.

Since the segment tree G contains $O(B)$ nodes, each containing a pointer to a B$^+$-tree in addition to standard node information, it can be stored in $O(1)$ blocks. In total, each segment may be stored in at most three external-memory structures. That is, if a segment spans the multislab $[i : j]$, the segment is stored in data structures L_i, R_j, and in $O(\log_2 B)$ allocation nodes of G. Since $b = B/4$, an internal node of the first-level structure has enough space to store all references to b structures C_i, b structures L_i, b structures R_i and one structure G. Thus, in total, the space utilization is $O(n \log_2 B)$.

Search algorithm. Given a query segment $x = x_0, a_1 \leq y \leq a_2$, a lookup is performed on the first-level tree from the root, searching for a leaf containing x_0. For each node, if x_0 equals the x-coordinate of any s_i, the interval tree C_i is searched together with the second-level structures R_i and L_i to retrieve the segments lying on s_i and short fragments intersected by the query segment. Otherwise, if x_0 hits the i-th slab, that is $s_i < x_0 < s_{i+1}$, then we check second-level structures R_i and L_{i+1}.

In both cases, we have to check also the second-level structures G which contain multislabs spanning the query value x_0 and retrieve all the long fragments intersected by the query. When visiting G, we scan from the root of G to a leaf containing the value x_0. In each visited node, we search the ordered list associated with the node. Finally, if x_0 does not coincide with any s_i in the node, the search continues on the next level, in the node associated with the slab containing x_0.

Although any segment may be stored in three different external structures, it is clear that each segment intersected by the query q is retrieved at most twice. Moreover, for each internal node, during the search we visit exactly two structures for short fragments and structure G for long ones. This proves the following lemma.

Lemma 9.4 *N NCT segments can be stored in a secondary storage data structure having the following costs: (i) storage cost is $O(n \log_2 B)$; (ii) VS query time is $O(\log_B n(\log_B n \log_2 B + IL^*(B)) + t)$.* □

To further reduce the search time, we extend this approach with fractional cascading [40] between lists stored on neighbor levels of structures G.

9.5.3 Fractional cascading

Fractional cascading [40] is a technique supporting the execution of a sequence of searches at a constant cost (in both internal memory and secondary storage) per search, except for the first one. The main idea is to construct a number of "bridges" among lists. Once an element is found in one list, the location of the element in other lists is quickly determined by traversing the bridges rather than applying the general search. Fractional cascading has been extensively used in internal-memory algorithms [40, 43, 102]. Recently, the technique has been also applied to off-line external-memory algorithms [6]. Our approach can be summarized as follows.

Data structure supporting fractional cascading. The idea is to create bridges between nodes on neighbor levels of the G structure, stored in one node of the first-level data structure. In particular, for an internal node of G associated with a multislab $[i:j]$, two sets of bridges are created between the node and its two sons associated with multislabs $[i:\frac{i+j}{2}]$ and $[\frac{i+j}{2}:j]$ (see Figure 9.12). Each fragment in the multislab list $[i:j]$ keeps two references to the nearest fragments in the list, which are bridges to left and right sons.

For multislabs $[i:j]$ and $[i:\frac{i+j}{2}]$ (and similarly for multislabs $[i:j]$ and $[\frac{i+j}{2}:j]$), the bridges are created in such a way that the following *d-property* is satisfied: *the number s of fragments in both multislab lists $[i:j]$ and $[i:\frac{i+j}{2}]$ between two sequential bridges is such that $d \leq s \leq 2d$, where d is a constant ≥ 2.*

The bridges between two multislab lists $[i:j]$ and $[i:\frac{i+j}{2}]$ are generated as follows. First we merge the two lists in one. All fragments in the joined list do not intersect each other and either touch or intersect line $s_{\frac{i+j}{2}}$. We scan the joined list given by the order of segment intersections with line $s_{\frac{i+j}{2}}$ and select each $d+1$-th fragment from the list as a bridge. If the fragment is from $[i:j]$ (like fragment 7 in Figure 9.12), we cut it on line $s_{\frac{i+j}{2}}$ and copy it in the multislab list $[i:\frac{i+j}{2}]$. Otherwise, if the fragment is from $[i:\frac{i+j}{2}]$ (like fragment 4 in Figure 9.12), we copy it in the multislab list $[i:j]$. Such a copy of the bridge is called *augmented bridge fragment*; in Figure 9.12 these fragments are marked with "*". [9] The position of the augmented bridge fragment in $[i:j]$ is determined by its intersection with line $s_{\frac{i+j}{2}}$. Analogously, the bridges are created between multislabs $[i:j]$ and $[\frac{i+j}{2}:j]$. Bridge fragments from a multislab list $[i:j]$ are copied (after the cutting) in the multislab list $[\frac{i+j}{2}:j]$ while bridge fragments from the multislab list $[\frac{i+j}{2}:j]$ are copied to $[i:j]$.

After bridges from the multislab list $[i:j]$ to both lists $[i:\frac{i+j}{2}]$ and $[\frac{i+j}{2}:j]$ are generated, the list $[i:j]$ contains original fragments (some of them are bridges to left or right son) and augmented bridge fragments copied from lists $[i:\frac{i+j}{2}]$ and $[\frac{i+j}{2}:j]$. In Figure 9.12, the list $[i:j]$ contains three augmented bridge fragments, respectively fragments 3, 4, and 9. All the fragments in $[i:j]$ are ordered by the points in which they intersect or touch line $s_{\frac{i+j}{2}}$. Each non augmented bridge fragment maintains a pointer to the next and to the previous non augmented bridge fragment in the list as

[9] Note that augmented bridge fragments are only used to speed up the search, they are never reported in the query reply.

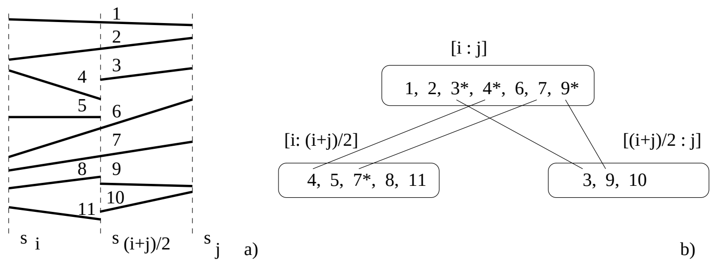


Figure 9.12: "Bridges" in G. a) Long fragments stored in the node associated with multislab $[i : j]$ and in its two sons, associated with multislabs $[i : \frac{i+j}{2}]$ and $[\frac{i+j}{2} : j]$. b) Lists of fragments associated with nodes of the G structure. The lists are extended with bridge segments (d=2). Bridges are shown by lines and new bridge fragments inserted in the nodes are marked with '*'.

well as a pointer to the next and to the previous augmented bridge fragment. The following result holds.

Proposition 9.1 *Multislab lists $[i : j]$, $[i : \frac{i+j}{2}]$, and $[\frac{i+j}{2} : j]$ satisfy the d-property.*

Proof: Consider multislab lists $[i : j]$ and $[i : \frac{i+j}{2}]$. Bridge fragments can be either fragments from the multislab list $[i : j]$, cut on line $s_{\frac{i+j}{2}}$, or fragments from the multislab list $[i : \frac{i+j}{2}]$. Consider the first case. These bridge fragments are already contained, as fragments, in the multislab list $[i : j]$ and are inserted as augmented bridge fragments in the multislab list $[i : \frac{i+j}{2}]$. In the second case, bridge fragments are already contained, as fragments, in the multislab list $[i : \frac{i+j}{2}]$ and are inserted as augmented bridge fragments in the multislab list $[i : j]$. This means that bridge fragments are contained both in the father and in the child node. By construction, in the ordered multislab lists exactly d fragments appear between two bridges. These fragments may be either from the multislab list $[i : j]$ or from the multislab list $[i : \frac{i+j}{2}]$. Therefore the sum of the number of fragments in both multislab lists between two bridges is at least d. A similar proof holds for multislab lists $[i : j]$ and $[\frac{i+j}{2} : j]$.

During insertion/deletions, the number of fragments between two sequential bridges may change. To guarantee the satisfaction of the d-property after an update is executed, specific rebalancing operators are applied to split or merge the nodes in order to still satisfy the d-property. These operations origins from 2-4-trees and B-trees [40, 102]. □

Given an internal node v of G, associated with the multislab $[i : j]$, a B$^+$-tree is built from the multislab list $[i : j]$, after bridges to both sons are generated and copied in the list. The following result holds.

Proposition 9.2 *After including augmented bridge fragments in all nodes of G, the space complexity is still $O(n \log_2 B)$.*

Proof: Without bridges, the space complexity is $O(n\log_2 B)$. Each insertion of a bridge fragment in the structure results in inserting a copy of an existing fragment into a list. Moreover, the insertion of pointers can only add a constant to the space. Therefore, the insertion of augmented bridge fragments cannot alter the space complexity. □

Search algorithm. Let q be the vertical segment of the form $x = x_0, a_1 \leq y \leq a_2$. The VS query q is performed as described in Subsection 9.5.2, by modifying the search in G as follows. First we search in the B$^+$-tree associated with the root of G and detect the leftmost segment fragment f_l^1 intersected by q and associated with the root. This takes $O(\log_B n)$ steps. Then, the leaves of the B$^+$-tree are traversed and all fragments in the root of G intersected by q (except for the augmented bridge fragments) are retrieved. As a second step, if x_0 is lower than $s_{\frac{i+j}{2}}$, the bridge to the left son, nearest to f_l^1, is determined, otherwise the bridge to the right son, nearest to f_l^1, is determined. Following the appropriate bridge, a leaf node in the B$^+$-tree associated with a second level node of G is reached. Because of the d-property of bridges, the leftmost segment fragment f_l^2, contained in the reached leaf node and intersected by q, can be found in $O(1)$ I/O's. Then, the leaves of the B$^+$-tree are traversed and all fragments intersected by q (except for the augmented bridge fragments) are retrieved. The same procedure for bridge navigation and fragment retrieval is repeated on levels $3, \ldots, \log_2 b$ of G. The following result holds.

Proposition 9.3 *Let f_l^i be the leftmost fragment intersected by q and associated with a node v stored at level i of G. Let b_i be the bridge fragment to the left son, nearest to f_l^i. The leftmost fragment intersected by q and associated with a son of v can be found from b_i in $O(1)$ I/O'.*

Proof: Suppose that x_0 is lower than $s_{\frac{i+j}{2}}$. We show that the number of fragments in the left son between f_l^{i+1} and b_i is at most $2d$, thus f_l^{i+1} can be reached in $O(1)$ I/O. Let d_i' be the bridge fragment nearest to b_i, and such that if b_i is lower than f_l^i, then b_i' is greater than f_l^i and vice versa. Suppose that b_i is lower than f_l^i (a similar proof can be given if b_i is greater than f_l^i). There are several cases:

- $b_i \leq f_l^i \leq b'_i \leq f_l^{i+1}$. Since bridge fragments are stored both in the father and in the son and since f_l^{i+1} and f_l^i intersect q, this means that f_l^{i+1} is not the leftmost fragment intersected by q and this contradict the hypothesis.

- $b_i \leq f_l^i \leq f_l^{i+1} \leq b'_i$. Since b'_i can be determined from b_i in $O(1)$, the same holds for f_l^{i+1}.

- $b_i \leq f_l^{i+1} \leq f_l^i \leq b'_i$. Since b'_i can be determined from b_i in $O(1)$, the same holds for f_l^{i+1}.

- $f_l^{i+1} \leq b_i \leq f_l^i \leq b'_i$. Since bridge fragments are stored both in the father and in the son and since f_l^{i+1} intersects q, this means that f_l^i is not the leftmost fragment intersected by q and associated with v and this contradicts the hypothesis. □

With the use of bridges, searching for the leftmost fragment intersecting q on all levels of G takes $O(\log_B n + \log_2 B)$ steps. Together with searching in L_i and R_{i+1} for short fragments, the search time for one internal node a of the first-level structure is $O(\log_B n + \log_2 B + IL^*(B) + \frac{T'}{B})$, where T' is the number of segments in node a intersected by the query. Since any segment is stored in only one node of the first-level tree (whose height is $O(\log_B n)$) and each segment intersected by the query is reported only once, reporting all segments intersected by the query takes $O(\log_B n(\log_B n + \log_2 B + IL^*(B)) + t)$.
Insertions. The 2LDS proposed above has been designed for the static case. To extend the schema to the semi-dynamic case when segment insertions are allowed together with queries, we have to extend both first- and second-level structures to manage insertion of segments.

First, we replace a static interval tree, used as a first-level structure, with a weighted-balanced B-tree [7] (see Chapter 6). Updates on such structure can be performed in $O(\log_B n)$ amortized time. The second-level structures C_i, R_i and L_i are dynamic and need not any extension. However, in order to store long fragments, a $BB[\alpha]$-tree [42, 108], $0 < \alpha < 1 - 1/\sqrt{2}$ can be used as the second-level structure G for long fragments. The last issue is how to maintain bridges between neighbor levels of the G structure, when a segment is inserted and the corresponding long fragment may violate the d-property. To ensure $O(1)$ I/O navigation complexity neighbor levels of the G structure, we provide some additional operations on multislab lists (similar to those presented in [102]). The linear bridge structure when each multislab list $[i : j]$ contains bridges to at most two multislabs $[i : \frac{i+j}{2}]$ and $[\frac{i+j}{2} : j]$ allows us to retain the $O(1)$ I/O amortized complexity of bridge navigation. Such extensions allow the execution of insertions in $O(\log_B n + \log_2 B + \frac{\log_B^2 n}{B})$ amortized time.

Theorem 9.3 *N NCT segments can be stored in a secondary storage data structure having the following costs: (i) storage cost is $O(n\ \log_2 B)$; (ii) VS query time is $O(\log_B n(\log_B n + \log_2 B + IL^*(B)) + t)$; (iii) insertion amortized time is $O(\log_B n + \log_2 B + \frac{\log_B^2 n}{B})$.* □

9.6 Preliminary experimental results

Some preliminary experiments have been carried out in order to investigate the performance of the proposed technique. The performed experiments deal with a simplified implementation of the technique presented in Section 9.4 (hereafter denoted by T). In particular, the P-range tree technique has not been applied to the data structures supporting the search on a set of line-based segments (see Section 9.3).

The experiments have been performed on a PC Pentium 100, with 16 Mb RAM. The page size is 1k. The program has been written in C++. The considered generalized relations contain respectively 5000, 10000, 15000, 20000, and 31000 NCT segments, uniformly distributed in the Euclidean plane. Segments are assumed to be ordered in the data file with respect their leftmost X-coordinate. The query segment is assumed to be vertical.

The aim of the performed experiments is to compare the number of page accessed by T during the execution of VS queries having different selectivity. This number has been compared with the number of page accessed in executing the same queries by applying a particular type of sequential search, that we call *clever sequential search* (hereafter denoted by CS). Since segments are orderly stored with respect to their leftmost X-coordinate, a sequential search can be interrupted as soon as a segment is retrieved such that its leftmost X-coordinate is greater than the X-coordinate of the query segment. This approach greatly reduces the number of segments that have to be analyzed in a sequential scan. Since the performance of CS depends on the distance of the query segment from the leftmost border of the minimum bounding box containing the stored segments, we have considered query segments differing not only in their length but also in this additional parameter. In particular, the reported experiments deal with three different query segments, Q1, Q2, and Q3; they are graphically represented in Figure 9.14. Q1 is the lowest selective query object whereas Q3 is the highest selective query object.

From the performed experiments, we have observed that the number of page accesses for T increases by decreasing the selectivity. This is due to the fact that queries with lower selectivity retrieve more segments and therefore require the analysis of more tree nodes. This result can be observed from Figure 9.15. Note that the considered selectivities are very low. Indeed, due to the form of the query ob-

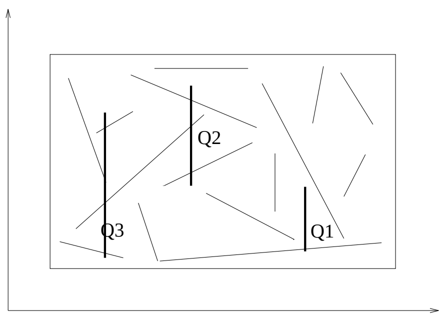


Figure 9.13: Position of the query segment with respect to the minimum bounding box containing all stored segments.

ject, selectivity greater than 10% can only be achieved by considering specific data distributions. On the other hand, experiments have been performed by assuming an uniform distribution of segments in the space.

From Figure 9.15 it also follows that the number of page accessed by CS does not depend on the query selectivity. In the case of the reported experiments, such number decreases by decreasing the selectivity. However, from Figure 9.13 we can see that CS has very good performance when the X-coordinate of the query segment is close to the leftmost border of the minimum bounding box containing the stored segments. In particular, the number of page accessed by CS decreases by decreasing such distance. On the other hand, T performance does not depend on this parameter.

9.7 Concluding remarks

In this chapter we have proposed two techniques to solve a vertical (or having any other fixed direction) segment query on segment databases. The more efficient technique has $O(n \log_2 B)$ space complexity and time complexity very close to $O(\log_B^2 n + t)$. These are the optimal bounds that can be achieved by using the (non-optimal) solution for the secondary storage implementation of priority search trees we have developed. The hypothesis under which the proposed techniques can be efficiently applied to constraint databases have also been pointed out, introducing the class of $O(N)$-databases.

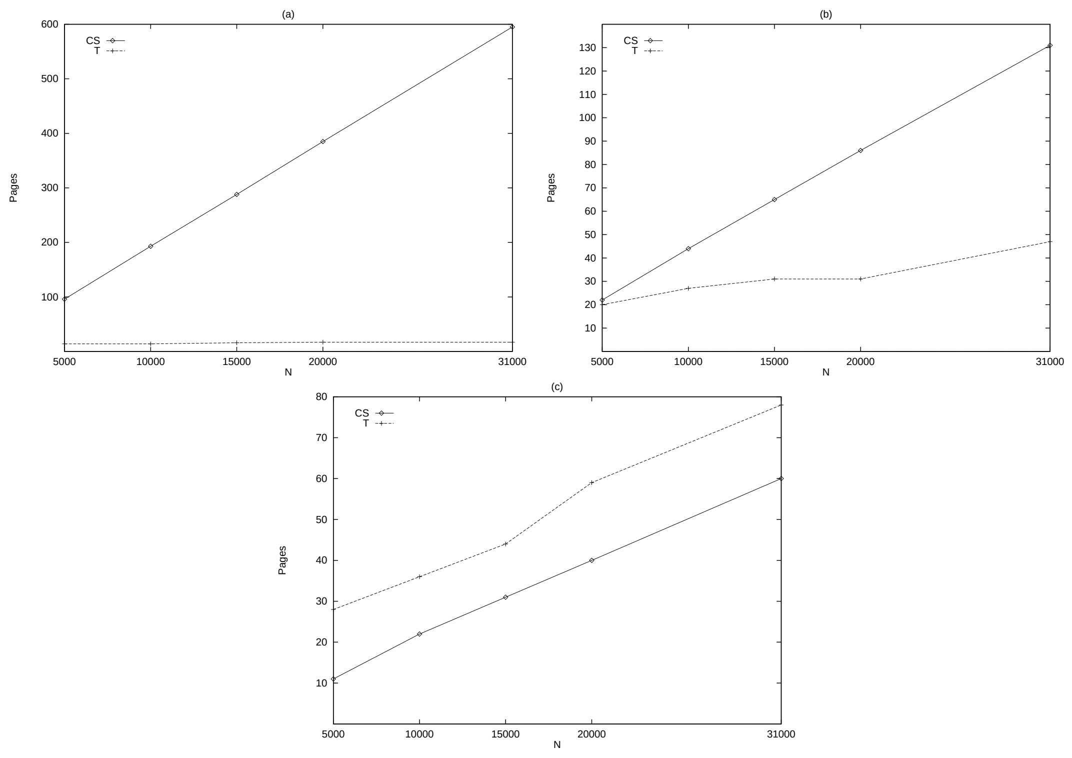


Figure 9.14: Number of page accesses with respect to the database size, by considering queries (a) Q1, (b) Q2, (c) Q3.

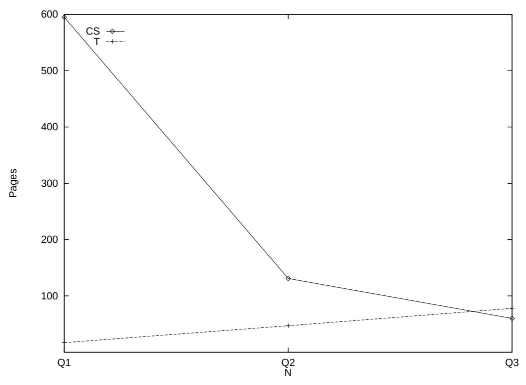


Figure 9.15: Number of page accesses with respect to different selectivities and $N = 31000$.

Chapter 10

Conclusions

In this dissertation, we have investigated modeling and optimization topics in the context of constraint databases. In particular, after a brief introduction to constraint databases (Chapter 1), the first part of the dissertation (Chapters 2–5) concerned modeling aspects whereas the second part (Chapters 6–9) concerned optimization issues. In this last chapter, we shortly summarize the contributions of this dissertation and we outline some topics left to future work.

10.1 Summary of the contributions

Data modeling. Contributions with respect to data modeling can be summarized as follows. After a brief survey of the main results that have been proposed in the context of constraint database modeling (Chapter 2), we have investigated the definition of new data manipulation languages for relational constraint databases. In particular, in Chapter 3 we have introduced an extended generalized algebra (EGRA) and an extended generalized calculus (ECAL) for relational constraint databases; in Chapter 4 an update language has also been proposed to complete the definition of a data manipulation language for relational constraint databases.

The main difference between the proposed languages and the classical languages proposed for relational constraint databases can be summarized as follows:

- Algebraic operators deal with generalized relations as they were finite sets of possibly infinite sets. By considering this semantics, which is slightly different from the standard one, two different groups of operators have been proposed. The first group includes the standard generalized relational algebra operators; the second group includes operators treating each generalized tuple as a single object. The proposed algebra is equivalent to the standard one when generalized tuple identifiers are introduced, but it allows typical user requests to be expressed in a more convenient way.

- The calculus, similarly to the algebra, is defined by using two different types of variables, those representing generalized tuples and those representing atomic values. The calculus has been defined by following Klug's approach [88].

- Both the algebra and the calculus have been extended to deal with external functions. As far as we know, this is the first approach to integrate constraint database languages with external primitives.

The algebra and the calculus have been proved to be equivalent. The proof, which is very technical, follows the approach taken by Klug [88] to prove the equivalence between the relational algebra and the relational calculus extended with aggregate functions.

As a second contribution with respect to data modeling, in Chapter 5 we have introduced a formal model and a query language for nested relational constraint databases, overcoming some limitations of the previous proposals. Indeed, the proposed model is characterized by a clear formal foundation, a low data complexity, and the ability to model any degree of nesting. The proposed language is obtained by extending $\mathcal{NRC}$ [148] to deal with possibly infinite relations, finitely representable by using POLY. Its definition is based on structural recursion and on monads. We would like to recall that the proposed language is *not* a new language. Rather, it represents a new formal definition of already existing languages. We claim that this formalism could be useful to investigate further properties of relational constraint query languages.

Optimization issues. The contributions with respect to optimization issues can be summarized as follows. After a brief survey of the main results that have been proposed in the context of constraint database optimization (Chapter 6), in Chapter 7 we have investigated the definition of rewriting rules for EGRA expressions. In particular, following the approach taken in [61], simplification and optimization rules for EGRA expressions have been proposed, pointing out the differences with respect to the typical rules used in the relational context. The proposed rules can be used not only to optimize EGRA expressions but, due to the equivalence between GRA and EGRA, they can also be used to improve the efficiency of GRA optimizers. The basic issues in designing such an optimizer have also been discussed.

In Chapter 8 we have investigated the use of a dual representation for polyhedra to index constraint databases. Under this representation, each generalized tuple is transformed into two unbound polyhedra in a dual plane. We have shown that intersection and containment problems with respect to a half-plane query with a predefined direction can be solved in logarithmic time and linear space. When the query half-plane does not satisfy this condition, some approximated solutions have also be proposed. Experimental results show that such techniques often perform better than R-trees, a well-known spatial data structure [71, 130].

In Chapter 9, we have proposed a close to optimal technique for segment databases, allowing the detection of all segments which are intersected by a given vertical segment. This result is an improvement with respect to the classical stabbing query problem, determining all segments intersecting a given line. We have also discussed how techniques proposed for segment databases can be applied to constraint and spatial databases and we have introduced some classes of databases in which such techniques can be efficiently used.

10.2 Topics for further research

The research described in this dissertation can be extended along several directions.

Data modeling. With respect to data modeling, we feel that it may be interesting to continue in the investigation of models for complex objects. In particular, a quite interesting topic is the definition of a complex object model for the representation of planning reasoning and partial information. Planning reasoning refers to the support of decision activities with respect to non yet existing entities. A typical example is the analysis of the impact that some buildings will have on the environment. Partial information refers to the ability of representing incomplete information with respect to complex objects. Note that this topic is different from the use of generalized tuples to model imprecise values [91, 133]. In this case, we would like to express partial information not only with respect to atomic objects, but also with respect to complex ones, represented by using constraints.

Optimization issues. With respect to the investigated optimization issues, we feel that it may be interesting to investigate the following topics:

- With respect to the logical optimization, the development of a prototype is required in order to establish if the logical optimization of relational constraint query languages can really benefit from the use of the new introduced rules. The design of a logical optimizer for the generalized relational algebra, based on the guidelines we have proposed, is another interesting issue.

- With respect to the indexing techniques based on the dual representation, an important issue is the definition of indexing techniques with optimal complexity for arbitrary half-plane queries. The use of the dual representation as the basis for the development of further indexing techniques is another topic of great interest. Finally, the definition of more general techniques for arbitrary d-dimensional tuples is another issue that should be investigated.

- With respect to the indexing technique based on the segment representation, a first aspect requiring further investigation is the comparison, in terms of performance, of the proposed technique with R-trees and their variants, in order to establish the real applicability of the proposed approach. Further, a fundamental issue is the development of indexing techniques to retrieve all segments intersecting a given segment having an arbitrary direction.

Bibliography

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[2] S. Abiteboul and P. Kanellakis. Query Languages for Complex Object Databases. *SIGACT News*, 21(3):9–18, 1990.

[3] F. Afrati, S.S. Cosmadakis, S. Grumbach, and G.M. Kuper. Linear vs. Polynomial Constraints in Database Query Languages. In A. Borning, editor, *LNCS 874: Proc. of the 2nd Int. Workshop on Principles and Practice of Constraint Programming*, pages 181–192, 1994. Springer Verlag.

[4] W.G. Aref and H. Samet. Extending a Database with Spatial Operations. In O. Günther and H.J. Schek, editors, *LNCS 525: Proc. of the 2nd Int. Symp. on Advances in Spatial Databases*, pages 299–319, 1991. Springer Verlag.

[5] L. Arge. The Buffer Tree: A New Technique for Optimal I/O Algorithms. In S.G. Akl, F.K.H.A. Dehne, J.R. Sack, and N. Santoro, editors, *LNCS 955: Proc. of the 4th Int. Workshop on Algorithms and Data Structures*, pages 334–345, 1995. Springer Verlag.

[6] L. Arge, D.E. Vengroff, and J. S. Vitter. External-Memory Algorithms for Processing Line Segments in Geographic Information Systems. In P.G. Spirakis, editor, *LNCS 979: Proc. of the 3rd Annual European Symp. on Algorithms*, pages 295–310, 1995.

[7] L. Arge and J.S. Vitter. Optimal Dynamic Interval Management in External Memory. In *Proc. of the Int. Symp. on Foundations of Computer Science*, pages 560–569, 1996.

[8] J.L. Balcazar, J. Diaz, and J. Gabarro. *Structural Complexity I.* Springer Verlag, 1989.

[9] M. Baudinet, J. Chomicki, and P. Wolper. Temporal Databases: Beyond Finite Extensions. In *Proc. of the Int. Workshop on Infrastructure for Temporal Databases*, 1993.

[10] M. Baudinet, M. Niezette, and P. Wolper. On the Representation of Infinite Temporal Data and Queries. In *Proc. of the 10th ACM SIGACT-SIGMOD-SIGART Int. Symp. on Principles of Database Systems*, pages 280–290, 1991. ACM Press.

[11] R. Bayer and E. McCreight. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica*, 1:173–189, 1972. Springer Verlag. Springer Verlag.

[12] A. Belussi, E. Bertino, M. Bertolotto, and B. Catania. Generalized Relational Algebra: Modeling Spatial Queries in Constraint Databases. In G.M. Kuper and M. Wallace, editors, *LNCS 1034: Proc. of the 1st Int. CONTESSA Database Workshop, Constraint Databases and their Applications*, pages 40–67, 1995. Springer Verlag.

[13] A. Belussi, E. Bertino, and B. Catania. An Extended Algebra for Constraint Databases. *IEEE Trans. on Knowledge and Data Engineering.* To appear. IEEE Computer Society Press. Also Technical Report n. 211–98, University of Milano, 1996.

[14] A. Belussi, E. Bertino, and B. Catania. Manipulating Spatial Data in Constraint Databases. In M. Scholl and A. Voisard, editors, *LNCS 1262: Proc. of the 5th Symp. on Spatial Databases*, pages 115–141, 1997. Springer Verlag.

[15] M. Benedikt, G. Dong, L. Libkin, and L. Wong. Relational Expressive Power of Constraint Query Languages. In *Proc. of the 15th ACM SIGACT-SIGMOD-SIGART Int. Symp. on Principles of Database Systems*, pages 5–16, 1996. ACM Press.

[16] E. Bertino and B. Catania. Query Refinement in Constraint Multimedia Databases. In *Proc. of the Principles of Multimedia Information Systems Workshop*, 1995.

[17] E. Bertino and B. Catania. Constraints and Optimization in Database Systems: a Survey and Research Issues. *Int. Journal of Information Technology*, 1(2): 111–143, 1996. World Scientific.

[18] E. Bertino and B. Catania. A Constraint-based Approach to Shape Management in Multimedia Databases. *ACM Multimedia Journal.* 6(1):2–16, 1998. ACM Springer.

[19] E. Bertino, B. Catania, and B. Shidlovsky. Towards Optimal Two-Dimensional Indexing for Constraint Databases. *Information Processing Letters*, 64(1):1–8, 1997. Elsevier Science Publishers.

[20] E. Bertino, B. Catania, and B. Shidlovsky. Towards Optimal Indexing for Segment Databases. In *Proc of the 6th Int. Conf on Extended Database Technology*, 1998. To appear. Springer Verlag.

[21] E. Bertino, B. Catania, and L. Wong. Finitely Representable Nested Relations. Submitted for publication.

[22] E. Bertino and L. Martino. *Object-Oriented Database Systems - Concepts and Architectures.* Addison-Wesley, 1993.

[23] A.H. Borning. The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory. *ACM Transactions on Programming Languages and Systems*, 3(3):353–387, 1981. ACM Press.

[24] T. Brinkhoff, H.-P. Kriegel, and R. Schneider. Comparison of Approximations of Complex Objects Used for Approximation-based Query Processing in Spatial Database Systems. In *Proc. of the IEEE Int. Conf. on Data Engineering*, pages 40–49, 1993. IEEE Computer Society Press.

[25] A. Brodsky. Constraint Databases: Promising Technology or Just Intellectual Exercize?. *Constraints Journal.* To appear. Kluwer Academic Publishers.

[26] A. Brodsky, D. Goldin, and V. Segal. On Strongly Polynomial Projections in d-monotone Constraint Databases. In *Proc. of the Int. Workshop on Constraints and Databases, 2nd Int. Conf. on the Principles and Practice of Constraint Programming*, 1996.

[27] A. Brodsky, J. Jaffar, and M.J. Maher. Toward Practical Constraint Databases. In R. Agrawal, S. Baker, and D.A. Bell, editors, *Proc. of the 19th Int. Conf. on Very Large Data Bases*, pages 567–579, 1993. Morgan Kaufmann Publishers.

[28] A. Brodsky and Y. Kornatzky. The $\mathcal{L}yri\mathcal{C}$ Language: Querying Constraint Objects. In M.J. Carey and D.A. Schneider, editors, *Proc. of the 1995 ACM SIGMOD Int. Conf. on Management of Data*, pages 35–46, 1995. ACM Press.

[29] A. Brodsky, C. Lassez, J.L. Lassez, and M.J. Maher. Separability of Polyhedra and a New Approach to Spatial Storage. In *Proc. of the 14th ACM SIGACT-SIGMOD-SIGART Int. Symp. on Principles of Database Systems*, pages 7–11, 1995. ACM Press.

[30] A. Brodsky and V. Segal. The C^3 Constraint Object-Oriented Database System: An Overview. In V. Gaede, A. Brodsky, O. Günther, D. Srivastava, V. Vianu, and M. Wallace, editors, *LNCS 1191: Proc. of the 2nd Int. CONTESSA Database Workshop, Constraint Databases and their Applications*, pages 134–159, 1997. Springer Verlag.

[31] A. Brodsky and X. S. Wang. On Approximation-Based Query Evaluation, Expensive Predicates and Constraint Objects. In *Proc. of the ILPS Post-Conference Worskshop on Constraints, Databases and Logic Programming*, 1995.

[32] K.Q. Brown. *Geometric Transformations for Fast Geometric Algorithms*, Ph.D. Dissertation, Carnegie-Mellon University, Pittsburgh, Pa., December 1979.

[33] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension Syntax. *SIGMOD Record*, 23(1):87–96, 1994.

[34] P. Buneman, S. Naqvi, V. Tannen, and L. Wong. Principles of Programming with Complex Objects and Collection Types. *Theoretical Computer Science*, 149(1):3–48, 1995. Elsevier Science Publishers.

[35] J. Byon and P.Z. Revesz. DISCO: A Constraint Database System with Sets. In G.M. Kuper and M. Wallace, editors, *LNCS 1034: Proc. of the 1st Int. CONTESSA Database Workshop, Constraint Databases and their Applications*, pages 68–83, 1995. Springer Verlag.

[36] A.K. Chandra and D. Harel. Computable Queries for Relational Data Bases. *Journal of Computer and System Sciences*, 21(2):156–178, 1980. Academic Press.

[37] A.K. Chandra and D. Harel. Structure and Complexity of Relational Queries. *Journal of Computer and System Sciences*, 25(1):99–128, 1980. Academic Press.

[38] C.C. Chang and H.J. Keisler. *Model Theory.* North-Holland, 1973.

[39] B. Chazelle. Filtering Search: A New Approach to Query-Answering. *SIAM Journal of Computing*, 15(3): 703–724, 1986. SIAM Press.

[40] B. Chazelle and L.J. Guibas. Fractional Cascading : I. A Data Structuring Technique. *Algorithmica*, 1(2):133–162, 1986. Springer Verlag.

[41] C.M. Chen and N. Roussopoulos. Adaptive Selectivity Estimation Using Query Feedback. In R.T. Snodgrass and M. Winslett, editors, *Proc. of the 1994 ACM SIGMOD Int. Conf. on Management of Data*, pages 161–174, 1994.

[42] S.W. Cheng and R. Janardan. Efficient Dynamic Algorithms for Some Geometric Intersection Problems. *Information Processing Letters*, 36(5):251–258, 1990. Elsevier Science Publishers.

[43] Y.-J. Chiang and R. Tamassia. Dynamic Algorithms in Computational Geometry. *Proc. IEEE*, 80(9):1412–1434, 1992. IEEE Computer Society Press.

[44] J. Chomicki, D. Goldin, and G. Kuper. Variable Independence and Aggregation Closure. In *Proc. of the 15th ACM SIGACT-SIGMOD-SIGART Int. Symp. on Principles of Database Systems*, pages 40–48, 1996. ACM Press.

[45] J. Chomicki and G. Kuper. Measuring Infinite Relations. In *Proc. of the 14th ACM SIGACT-SIGMOD-SIGART Int. Symp. on Principles of Database Systems*, pages 78–94, 1995. ACM Press.

[46] E. Clementini, P. Di Felice, and P. van Oosterom. A Small Set of Formal Topological Relationships Suitable for End-User Interaction. In D.J. Abel and B.C. Ooi, editors, *LNCS 692: Proc. of the 3rd Int. Symp. on Advances in Spatial Databases*, pages 277–295, 1993. Springer Verlag.

[47] E.F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 6(13):377–387, 1970. ACM Press.

[48] A. Colmerauer. An Introduction to Prolog III. *Communications of the ACM*, 33(7):69–90, 1980. ACM Press.

[49] D. Comer. The Ubiquitous B-tree. *Computing Surveys*, 11(2):121–138, 1979. ACM Press.

[50] L. De Floriani, P. Marzano, and E. Puppo. Spatial Queries and Data Models. In A.U. Frank, editor, *LNCS 716: Spatial Information Theory: a Theoretical Basis for GIS*, pages 123–138, 1993. Spinger Verlag.

[51] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. In ICOT, editor, *Proc. of the Fifth Generation Computer Systems*, pages 693–702, 1988. Springer Verlag.

[52] H. Edelsbrunner. A New Approach to Rectangular Intersection. Part I. *Int. Journal of Comp. Mathematics*, 13:209–219, 1983.

[53] H. Edelsbrunner. Algorithms in Combinatorial Geometry. *EATCS Monographs on Theoretical Computer Science*, Vol. 10, 1987.

[54] M.J. Egenhofer. Reasoning about Binary Topological Relations. In O. Günther and H.J. Schek, editors, *LNCS 525: Proc. of the Int. Symp. on Advances in Spatial Databases*, pages 143–160, 1991. Springer Verlag.

[55] E. Freuder. Synthesizing Constraint Expressions. *Communication of the ACM*, 21(11):958-966, 1978. ACM Press.

[56] V. Gaede and O. Günther. Constraint-Based Query Optimization and Processing. In G.M. Kuper and M. Wallace, editors, *LNCS 1034: Proc. of the 1st Int. CONTESSA Database Workshop, Constraint Databases and their Applications*, pages 84–101, 1995. Springer Verlag.

[57] M. Gargano, E. Nardelli, and M. Talamo. Abstract Data Types for the Logical Modeling of Complex Data. *Information Systems*, 16(6):565–583, 1991. North Holland.

[58] D.Q. Goldin. Constraint Query Algebras. Ph.D. Thesis, Brown University, 1996.

[59] D.Q. Goldin and P.C. Kanellakis. Constraint Query Algebras. *Constraints Journal.* To appear. Kluwer Academic Publishers.

[60] S. Grumbach and G. Kuper. Tractable Query Languages for Geometric and Topological Data. In *Proc. of 13th ACM SIGACT-SIGMOD-SIGART Int. Symp. on Principles of Database Systems*, pages 289–300, 1994. ACM Press.

[61] S. Grumbach and Z. Lacroix. Computing Queries on Linear Constraint Databases. In P. Atzeni, editor, *Proc. of the Int. Workshop on Database Programming Languages*, 1995.

[62] S. Grumbach, P. Rigaux, M. School, and L. Segoufin. DEDALE, A Spatial Constraint Database. In *Proc. of the Int. Workshop on Database Programming Languages*, 1997.

[63] S. Grumbach, P. Rigaux, and L. Segoufin. The DEDALE System for Complex Spatial Queries. In *Proc. of the 1998 ACM SIGMOD Int. Conf. on Management of Data*, 1998. To appear. ACM Press.

[64] S. Grumbach and J. Su. Finitely Representable Databases. In *Proc. of 13th ACM SIGACT-SIGMOD-SIGART Int. Symp. on Principles of Database Systems*, pages 289–300, 1994. ACM Press.

[65] S. Grumbach and J. Su. Dense-Order Constraint Databases. In *Proc. of the 14th ACM SIGACT-SIGMOD-SIGART Int. Symp. on Principles of Database Systems*, pages 66–77, 1995. ACM Press.

[66] S. Grumbach, J. Su, and C. Tollu. Linear Constraint Query Languages. Expressive Power and Complexity. In D. Leivant, editor, *LNCS 960: Proc. of the Int. Workshop on Logic and Computation*, pages 426–446, 1994. Springer Verlag.

[67] O. Günther. *Efficient Structures for Geometric Data Management.* Springer Verlag, 1988.

[68] O. Günther. The Design of the Cell Tree: An Object-Oriented Index Structure for Geometric Databases. In *Proc. of the 5th Int. Conf. on Data Engineering,* pages 598–605, 1989. IEEE Computer Society Press.

[69] A. Gupta, Y. Sagiv, J.D. Ullman, J. Widom. Constraint Checking with Partial Information. In *Proc. of the 3th ACM SIGMOD-SIGACT-SIGART Int. Symp. on Principles of Database Systems*, pages 45–55, 1994. ACM Press.

[70] R.H. Güting and M. Schneider. Realm-Based Spatial Data Types: The ROSE Algebra. *VLDB Journal*, 4(2):243–286, 1995. Springer Verlag.

[71] A. Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. In B. Yormark, editor, *Proc. of the 1984 ACM SIGMOD Int. Conference on Management of Data*, pages 47–57, 1984. ACM Press.

[72] M.R. Hansen, B.S. Hansen, P. Lucas, and P. van Emde Boas. Integrating Relational Databases and Constraint Languages. *Computer Languages*, 14(2):63–82, 1989. Elsevier Science Publishers.

[73] L. Hermosilla and G. Kuper. Towards the Definition of a Spatial Object-Oriented Data Model with Constraints. In G.M. Kuper and M. Wallace, editors, *LNCS 1034: Proc. of the 1st Int. CONTESSA Database Workshop, Constraint Databases and their Applications*, pages 120–131, 1995. Springer Verlag.

[74] D.S. Hochbaum and J. Naor. Simple and Fast Algorithms for Linear and Integer Programs with Two Variables per Inequalities. *SIAM Journal of Computing,* 23(6):1179–1192, 1994. SIAM Press.

[75] R. Hull and J. Su. On the Expressive Power of Database Queries with Intermediate Types. *Journal of Computer and System Sciences*, 43(1):219–267, 1991. Academic Press.

[76] C. Icking, R. Klein, and T. Ottmann. Priority Search Trees in Secondary Memory. In *LNCS 314: Proc. of the Int. Workshop on Graph-Theoretic Concepts in Computer Science*, pages 84–93, 1988. Springer Verlag.

[77] J. Jaffar and J.L. Lassez. Constraint Logic Programming. In *Proc. of the 14th ACM Symp. on Principles of Programming Languages*, pages 111–119, 1987. ACM Press.

[78] M. Jarke and Jurgen Koch. Query Optimization in Database Systems. *ACM Computing Surveys*, 16(2): 111–150. ACM Press.

[79] F. Kabanza, J.M. Stevenne, and P. Wolper. Handling Infinite Temporal Data. In *Proc. the 9th ACM SIGACT-SIGMOD-SIGART Int. Symp. on Principles of Database Systems*, pages 392–403, 1990. ACM Press.

[80] P.C. Kanellakis. Constraint Programming and Database Languages: a Tutorial. In *Proc. the 14th ACM SIGACT-SIGMOD-SIGART Int. Symp. on Principles of Database Systems*, pages 46–53, 1995. ACM Press.

[81] P.C. Kanellakis. Elements of Relational Database Theory. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, Chapter 17, 1990.

[82] P.C. Kanellakis and D.Q. Goldin. Constraint Programming and Database Query Languages. In *LNCS 789: Proc. of the Int. Symp. on Theoretical Aspects of Computer Software*, pages 96–120, 1994. Springer Verlag.

[83] P.C. Kanellakis, G. Kuper, and P. Revesz. Constraint Query Languages. *Journal of Computer and System Sciences*, 51(1):25–52, 1995. Academic Press.

[84] P.C. Kanellakis and S. Ramaswamy. Indexing for Data Models with Constraints and Classes. *Journal of Computer and System Sciences*, 52(3):589–612, 1996. Academic Press.

[85] H. Katsumo and A.O. Mendelzon. On the Difference Between Updating a Knowledge Base and Revising it. *Belief Revision*, Cambridge Tracts in Theoretical Computer Science, 1992. Cambridge University Press.

[86] D.B. Kemp, K. Ramamohanarao, I. Balbin, and K. Meenakshi. Propagating Constraints in Recursive Deductive Databases. In E.L. Lusk and R.A. Overbeek, editors, *Proc. of the North American Conf. on Logic Programming*, pages 981–998, 1989.

[87] M. Kifer, W. Kim, and Y. Sagiv. Querying Object-Oriented Databases. In M. Stonebraker, editor, *Proc. of the 1992 ACM SIGMOD Int. Conf. on Management of Data*, pages 393–402, 1992. ACM Press.

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

[89] P. Kolaitis and C.H. Papadimitriou. Why not Negation by Fixpoint?. In *Proc. the 7th ACM SIGACT-SIGMOD-SIGART Int. Symp. on Principles of Database Systems*, pages 231–239, 1988. ACM Press.

[90] M. Koubarakis. Representation and Querying in Temporal Databases: the Power of Temporal Constraints. In *Proc. of the Int. Conf. on Data Engineering*, pages 327–334, 1993. IEEE Computer Society Press.

[91] M. Koubarakis. Database Models for Infinite and Indefinite Temporal Information. *Information Systems*, 19(2):141–173, 1994. North Holland.

[92] G.M. Kuper. On the Expressive Power of the Relational Calculus with Arithmetic Constraints. In S. Abiteboul and V. Vianu, editors, *LNCS 470: Proc. of the 3th Int. Conf. on Database Theory*, pages 202–211, 1990. Springer Verlag.

[93] G.M. Kuper. Aggregation in Constraint Databases. In *Proc. of the 1st Int. Workshop on Principles and Practice of Constraint Programming*, pages 166–173, 1993.

[94] J.L. Lassez. Querying Constraints. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Int. Symp. on Principles of Database Systems*, pages 288–298, 1990. ACM Press.

[95] A. Levy and Y. Sagiv. Constraints and Redundancy in Datalog. In *Proc. of the 11th ACM SIGACT-SIGMOD-SIGART Int. Symp. on Principles of Database Systems*, pages 67–80, 1992. ACM Press.

[96] L. Libkin and L. Wong. Conservativity of Nested-Relational Calculi with Internal Generic Functions. *Information Processing Letters*, 49(6):273–280, 1994. Elsevier Science Publishers.

[97] L. Libkin and L. Wong. On Representation and Querying Incomplete Information in Databases with Multisets. *Information Processing Letters*, 56:209–214, 1995. Elsevier Science Publishers.

[98] D. Lomet and B. Salzberg. The hB-tree: A Multiattribute Indexing Method with Good Guaranteed Performance. *ACM Transactions on Database Systems*, 15(4):625–658, 1990. ACM Press.

[99] A.K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8(1), 1977.

[100] K. Marriott, and P.J. Stuckey. The 3 R's of Optimizing Constraint Logic Programs: Refinement, Removal and Reordering. In *Proc. of the 20th Annual ACM SIGPLAN-SIGACT Int. Symp. on Principles of Programming Languages*, pages 334–344, 1993. ACM Press.

[101] E.McCreight. Priority Search Trees. *SIAM Journal of Computing.* 14(2): 257–276, 1985. SIAM Press.

[102] K. Mehlhorn and S. Naher. Dynamic Fractional Cascading. *Algorithmica*, 5(2):215–241, 1990. Springer Verlag.

[103] E. Moggi. Notions of Computation and Monads. *Information and Computation*, 93(1):55–92, 1991. Academic Press.

[104] E.E. Moise. *Geometric Topology in Dimension Two and Three.* Springer Verlag, 1977.

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

[106] I.S. Mumich, S.J. Finkelstein, H. Pirahesh, and R. Ramakrishnan. Magic Conditions. In *Proc. of the 9th ACM SIGACT-SIGMOD-SIGART Int. Symp. on Principles of Database Systems*, pages 314–330, 1990. ACM Press.

[107] J. Nievergelt, H. Hinterberger, and K. Sevcik. The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Transactions on Database Systems*, 9(1):257–276, 1984. ACM Press.

[108] J. Nievergelt and E. M. Reingold. Binary Search Tree of Bounded Balance. *SIAM J. Computing*, 2(1):33–43, 1973. SIAM Press.

[109] M. Niezette and J.M. Stevenne. An Efficient Representation of Periodic Time. In *Proc. of the Int. Conf. on Information and Knowledge Management*, pages 161–168, 1992.

[110] J. Orenstein. Spatial Query Processing in an Object-Oriented Database System. In C. Zaniolo, editor, *Proc. of the 1986 ACM SIGMOD Int. Conference on Management of Data*, pages 326–336, 1986. ACM Press.

[111] G. Ozsoyoglu, Z.M. Ozsoyoglu, and V. Matos. Extending Relational Algebra and Relational Calculus with Set-Valued Attributes and Aggregated Functions. *ACM Transactions on Database Systems*, 12(4):566–592, 1987. ACM Press.

[112] C.H. Papadimitriou, D. Suciu, and V. Vianu. Topological Queries in Spatial Databases. In *Proc. of the 15th ACM SIGACT-SIGMOD-SIGART Int. Symp. on Principles of Database Systems*, pages 81–92, 1996. ACM Press.

[113] C.H. Papadimitriou and M. Yannakakis. On the Complexity of Database Queries. In *Proc. of the 16th ACM SIGACT-SIGMOD-SIGART Int. Symp. on Principles of Database Systems*, pages 12–19, 1997. ACM Press.

[114] J. Paredaens. Spatial Databases, The Final Frontier. In G. Gottlob and M.Y. Vardi, editors, *LNCS 893: Proc. of the 5th Int. Conf. on Database Theory*, pages 14–31, 1995. Springer Verlag.

[115] J. Paredaens, J. Van den Bussche, and D. Van Gucht. Towards a Theory of Spatial Database Queries. In *Proc. of the 13th ACM SIGACT-SIGMOD-SIGART Int. Symp. on Principles of Database Systems*, pages 279–288, 1994. ACM Press.

[116] J. Paredaens and D. Van Gucht. Possibilities and Limitations of Using Flat Operators in Nested Algebra Expressions. In *Proc. of the 7th ACM SIGACT-SIGMOD-SIGART Int. Symp. on Principles of Database Systems*, pages 29–38, 1988. ACM Press.

[117] J. Paredaens and D. Van Gucht. Converting Nested Relational Algebra Expressions into Flat Algebra Expressions. *ACM Transactions on Database Systems*, 17(1):65–93, 1992. ACM Press.

[118] F. P. Preparata and M.I. Shamos. *Computational Geometry - an Introduction*, Springer Verlag, New York, 1985.

[119] S. Ramaswamy. Efficient Indexing for Constraints and Temporal Databases. In F.N. Afrati and P. Kolaitis, editors, *LNCS 1186: Proc. of the 6th Int. Conf. on Database Theory*, pages 419–431, 1997. Springer Verlag.

[120] S. Ramaswamy and S. Subramanian. Path-Caching: A Technique for Optimal External Searching. In *Proc. of the 13th ACM SIGACT-SIGMOD-SIGART Int. Symp. on Principles of Database Systems*, pages 25–35, 1994. ACM Press.

[121] P.Z. Revesz. A Closed Form for Datalog Queries with Integer Order. In S. Abiteboul and P.C. Kanellakis, editors, *LNCS 470: Proc. of the Int. Conf. on Database Theory*, pages 187–201, 1990. Springer Verlag.

[122] P.Z. Revesz. Datalog Queries of Set Constraint Databases. In G. Gottlob and M.Y. Vardi, editors, *LNCS 893: Proc. of the 5th Int. Conf. on Database Theory*, pages 424–438, 1995. Springer Verlag.

[123] P.Z. Revesz. Model-Theoretic Minimal Change Operators for Constraint Databases. In F.N. Afrati and P. Kolaitis, editors, *LNCS 1186: Proc. of the 6th Int. Conf. on Database Theory*, pages 447-460, 1997. Springer Verlag.

[124] R.T. Rockafellar. *Convex Analysis.* Princeton University Press, Princeton, NJ, 1970.

[125] M.A. Roth, H.F. Korth, and A. Silberschatz. Theory of Non-First Normal Form Relational Databases. *ACM Transactions on Database Systems*, 1(3):189–222, 1976. ACM Press.

[126] N. Roussopoulos, C. Faloutsos, and T. Sellis. An Efficient Pictorial Database System for PSQL. *IEEE Transactions on Software Engineering*, 14(5):639–650, 1988. IEEE Computer Society Press.

[127] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison Wesley, 1989.

[128] M. Scholl and A. Voisard. Thematic Map Modeling. In A.P. Buchmann, O. Günther, and T.R. Smith, editors, *LNCS 409: Proc. of the Int. Symp. on the Design and Implementation of Large Spatial Databases*, pages 167–190, 1989. Springer Verlag.

[129] A. Schriver. Theory of Linear and Integer Programming. Interscience Series in *Descrete Mathematics and Optimization.* John Wiley, 1986.

[130] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+-tree: A Dynamic Index for Multi-Dimensional Objects. In P.M. Stoker, W. Kent, and P. Hammersley, editors, *Proc. of the 13th Int. Conf. on Very Large Data Bases*, pages 507–518, 1987. Morgan Kaufmann Publishers.

[131] D. Srivastava. Subsumption and Indexing in Constraint Query Languages with Linear Arithmetic Constraints. *Annals of Mathematics and Artificial Intelligence*, 8(3-4):315–343, 1993. Baltzer Press.

[132] D. Srivastava and R. Ramakrishnan. Pushing Constraint Selections. In *Proc. of the 11th ACM SIGACT-SIGMOD-SIGART Int. Symp. on Principles of Database Systems*, pages 301–315, 1992. ACM Press.

[133] D. Srivastava, R. Ramakrishnan, and P.Z. Revesz. Constraint Objects. In A. Borning, editor, *LNCS 874: Proc. of the 2nd Int. Workshop on Principles and Practice of Constraint Programming*, pages 218–228, 1994. Springer Verlag.

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

[135] P.J. Stuckey and S. Sudarshan. Compiling Query Constraints. In *Proc. of the 14th ACM SIGACT-SIGMOD-SIGART Int. Symp. on Principles of Database Systems*, pages 56–67, 1995. ACM Press.

[136] S. Subramanian and S. Ramaswamy. The P-Range Tree: A New Data Structure for Range Searching in Secondary Memory. In *Proc. of the ACM-SIAM Symp. on Discrete Algorithms*, pages 378–387, 1995. ACM Press.

[137] D. Suciu. Bounded Fixpoints for Complex Objects. In C. Beeri, A. Ohori, and D. Shasha, editors, *Proc. of the Int. Workshop on Database Programming Languages*, pages 263–281, 1993.

[138] D. Suciu. Domain-Independent Queries on Databases with External Functions. In G. Gottlob and M.Y. Vardi, editors, *LNCS 893: Proc. of the 5th Int. Conf. on Database Theory*, pages 177–190, 1995. Springer Verlag.

[139] P. Svensson. GEO-SAL: a Query Language for Spatial Data Analysis. In O. Günther and H.J. Schek, editors, *LNCS 525: Proc. of the Int. Symp. on Advances in Spatial Databases*, pages 119–140, 1991. Springer Verlag.

[140] J.D. Ullman. *Principles of Database and Knowledge-Base Systems*, Vol. 1 and 2. Computer Science Press, 1989.

[141] J. Van den Bussche. Complex Object Manipulation through Identifiers - an Algebraic Perspective. Techical report 92-41, University of Antwerp, Belgium, 1992.

[142] L. Vandeurzen, M. Gyssens, and D. Van Gucht On the Desirability and Limitations of Linear Spatial Database Models. In M.J. Egenhofer, editor, *LNCS 951: Proc. of the Int. Symp. on Advances in Spatial Databases*, pages 14–28, 1995. Springer Verlag.

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

[144] M.Y. Vardi. The Complexity of Relational Query Languages. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 137–146, 1982. ACM Press.

[145] P. Wadler. Comprehending Monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.

[146] A. Wallance. *An Introduction to Algebraic Topology.* Pergamon Press, 1967.

[147] W.L. Wilson. *Operations Research.* Int. Thomson Publishing under licence of Wadsworth Publishing Company, 1993.

[148] L. Wong. Normal Forms and Conservative Extension Properties for Query Languages over Collection Types. *Journal of Computer and System Sciences*, 52(3):495–505, 1996. Academic Press.

[149] L. Wong. Normal Forms and Conservative Properties for Query Languages over Collection Types. In *Proc. of the 13th ACM SIGACT-SIGMOD-SIGART Int. Symp. on Principles of Database Systems*, pages 26–36, 1993. ACM Press.

Appendix A

Selected proofs of results presented in Chapter 3

Proposition 3.2 *Let L_1 and L_2 be two constraint query languages. Let Φ be a decidable theory admitting variable elimination and closed under complementation. Let $t \in \{r, n\}$. The following facts hold:*

1. *If $L_i(\Phi)$ is t-based, then for all $S(\Phi, \Sigma_1), S(\Phi, \Sigma_2)$, $S(\Phi, \Sigma_1) \subseteq_t S(\Phi, \Sigma_2)$ iff $L_i(\Phi, \Sigma_1) \subseteq_t L_i(\Phi, \Sigma_2)$.*

2. *If $L_1(\Phi, \Sigma_1) \equiv_t L_2(\Phi, \Sigma_2)$ then $S(\Phi, \Sigma_1) \equiv_t S(\Phi, \Sigma_2)$.*

Proof:

1. We only prove the case $t = r$. The case $t = n$ trivially follows from $t = r$, using function *nested* instead of function *rel*.

   $\Rightarrow$ Suppose that $S(\Phi, \Sigma_1) \subseteq_t S(\Phi, \Sigma_2)$. Consider $q \in L_i(\Phi)$. Consider $r_1, ..., r_n \in S(\Phi, \Sigma_1)$. By hypothesis, there exist $r'_1, ..., r'_n \in S(\Phi, \Sigma_2)$ such that $r_i \equiv_r r'_i$, $i = 1, ..., n$. Since $L_i(\Phi)$ is r-based, there exists a relational algebra query q' such that:

   $$rel(q(r_1, ..., r_n)) = q'(rel(r_1), ..., rel(r_n))$$
   $$rel(q(r'_1, ..., r'_n)) = q'(rel(r'_1), ..., rel(r'_n)).$$

   But, since $rel(r_i) = rel(r'_i)$, then $q'(rel(r_1), ..., rel(r_n)) = q'(rel(r'_1), ..., rel(r'_n))$. Therefore, $rel(q(r_1, ..., r_n)) = rel(q(r'_1, ..., r'_n))$, thus $q(r_1, ..., r_n) \equiv_r q(r'_1, ..., r'_n)$.

   $\Leftarrow$ Directly from Definition 3.6.

2. Directly from Definition 3.6. □

Proposition 3.3 *Let L be a constraint query language. Let Φ be a decidable theory admitting variable elimination and closed under complementation. Let $t \in \{r, n\}$. Let $S(\Phi, \Sigma_1)$ and $S(\Phi, \Sigma_2)$ be two EGR supports. If $L(\Phi)$ is t-based, for all $q \in L(\Phi)$, for all $r_1, ..., r_n \in S(\Phi, \Sigma_1)$ and for all $r'_1, ..., r'_n \in S(\Phi, \Sigma_2)$, such that $r'_i \equiv_t r_i$, $q(r_1, ..., r_n) \equiv_t q(r'_1, ..., r'_n)$ holds.*

Proof: We only prove the case $t = r$. The case $t = n$ trivially follows from $t = r$, using function *nested* instead of function *rel*.

Consider $r_i \in S(\Phi, \Sigma_1)$, $i = 1, ..., n$ and $r'_i \in S(\Phi, \Sigma_2)$, $i = 1, ..., n$, such that $r_i \equiv_r r'_i$, $i = 1, ..., n$. Since $L(\Phi)$ is r-based, for each query $q \in L(\Phi)$ there exists a relational algebra query q' such that $rel(q(r_1, ..., r_n)) = q'(rel(r_1), ..., rel(r_n))$ and $rel(q(r'_1, ..., r'_n)) = q'(rel(r'_1), ..., rel(r'_n))$. Moreover, since $r_i \equiv_r r'_i$, $i = 1, ..., n$, $q'(rel(r_1), ..., rel(r_n)) = q'(rel(r'_1), ..., rel(r'_n)))$. Therefore, $rel(q(r_1, ..., r_n)) = rel(q(r'_1, ..., r'_n))$ and this concludes the proof. □

Theorem 3.1 *EGRA(Φ) is n-based.*

Proof: In order to show that EGRA(Φ) is n-based, we construct, for each EGRA expression a nested-relational algebra expression for which the condition of Definition 3.5 holds..

Let D be a domain of values. The nested-relational model deals with objects of type:

$$\tau ::= D \mid \langle A_1 : \tau, ..., A_n : \tau \rangle \mid \{\tau\}$$

where $A_1, ..., A_n$ are attributes names. A basic nested-relational algebra consists of the following operators:

1. the classical relational operators extended to nested-relations: union ($\cup$), difference ($\setminus$), selection (σ), projection (Π), and join ($\bowtie$);

2. two restructuring operators: nest and unnest.

   The unnest operator transforms a relation into one which is less deeply nested by concatenating each element in the set attribute being unnested to the remaining attributes in the relation. Thus, if R is a nested-relation of type $\{\langle A_1 : \tau 1, ..., A_n : \tau_n \rangle\}$:

$$unnest_{A_j} = \{\langle A_1 : x_1, ..., A_{j-1} : x_{j-1}, B_1 : y_1, ..., B_m : y_m, A_{j+1} : x_{j+1}, \\ ..., A_n : x_n \rangle \mid \langle A_1 : x_1, ..., A_n : x_n \rangle \in R, \langle B_1 : y_1, ..., B_m : y_m \rangle \in x_j\}.$$

The nest operator creates partitions based on the formation of equivalence classes. Two tuples are equivalent if they have the same values for the attributes which are not being nested. For each equivalence class, a single tuple is placed into the result. The attributes being nested are used to generate a nested-relation containing all tuples in the equivalence class for those attributes. Thus, if R is a nested-relation of type $\{\langle A_1 : \tau 1, ..., A_n : \tau_n \rangle\}$:

$$nest_{B=\langle A_{k+1},...,A_n\rangle} = \{\langle A_1 : x_1, ..., A_k : x_k, B = y\rangle \mid \\ \{\langle A_{k+1} : x_{k+1}, ..., A_n : x_n\rangle \mid \langle A_1 : x_1, ..., A_n : x_n\rangle \in R\} = y \neq \emptyset\}.$$

Let Φ be a decidable logical theory, admitting elimination of quantifier and closed under complementation. Let D be the domain of Φ. Table A.1 shows for each EGRA expression the corresponding nested-relational algebra expression. In the table:

- For all name of generalized relation R_i, R_i' represents the name of a nested-relation. If the generalized relation associated with R_i during the evaluation is r, then the nested-relation associated with R_i' is $nested(r)$.
- If the generalized relation R_i has schema $\{X_1, ..., X_n\}$, then R_i' has type $\{\langle A_i : \{\langle X_1 : D, ..., X_n : D\rangle\}\rangle\}$.
- Given a generalized tuple P with schema $\{X_1, ..., X_n\}$, $r(P)$ denotes the nested-relation $\{\langle A_P : \langle X_1 : a_1, ..., X_n : a_n\rangle\rangle \mid X_1 = a_1 \wedge ... \wedge X_n = a_n \in ext(P)\}$. Note that the type of $r(P)$ is $\{\langle A_P : \langle X_1 : D, ..., X_n : D\rangle\rangle\}$.
- Given a generalized tuple P with schema $\{X_1, ..., X_n\}$, $n(P)$ denotes the nested-relation containing only one element, represented by the set $ext(P)$. Thus, $n(P)$ coincides with the set $\{\langle A_P : \{\langle X_1 : a_1 \wedge ... \wedge X_n : a_n\rangle\}\rangle \mid X_1 = a_1 \wedge ... \wedge X_n = a_n \in ext(P)\}$. The type of $n(P)$ is $\{\langle A_P : \{\langle X_1 : D, ..., X_n : D\rangle\}\rangle\}$.
- $t\_tot(\alpha(R))$ represents the generalized tuple whose extension contains all possible relational tuples with schema $\alpha(R)$ that can be constructed on Φ. For example, if $\alpha(R) = \{X, Y\}$, in POLY one possible $t\_tot(\alpha(R))$ is $X + Y \leq 2 \vee X + Y \geq 2$.
- When a Cartesian product is used, we assume to rename the attributes names $A_j, X_1, ..., X_n$ of the relation appearing in the i-th position of the product $(i > 1)$ as $A^i, X_1^i, ..., X_n^i$.

It is simple to show that the proposed expressions satisfy Definition 3.5. □

EGRA expression	**NRA expression**
R_1	R'_1
$R_1 \cup R_2$	$R'_1 \cup R'_2$
$R_1 \setminus^s R_2$	$R'_1 \setminus R'_2$
$\sigma^s_{(P,t,\subseteq)}(R)$	$\Pi_{[A_1]}(\sigma_{A_P \subseteq A_1}(R'_1 \times n(P)))$ where
	$\sigma_{A_P \subseteq A_1}$ is defined in [1]
$\sigma^s_{(P,t,\bowtie \neq \emptyset)}(R_1)$	$\Pi_{[A_1]}(\sigma_{A_P \in A_1}(R'_1 \times r(P)))$
$\neg^s R_1$	$\Pi_{[B]}(nest_{B=\langle X^2_1,\ldots,X^2_n\rangle}(R_3 \setminus R_2))$ where
	$R_2 = unnest_{A^2}(R'_1 \times R'_1)$
	$R_3 = unnest_{A_{t\_tot}}(R'_1 \times n(t\_tot(\alpha(R'_1))))$
$\varrho_{[A\|B]}(R_1)$	see [1]
$\Pi_{[X_{i_1},\ldots,X_{i_m}]}(R_1)$	$\Pi_{[B]}(R_3)$ where
	$R_2 = \Pi_{[A_1,X^2_{i_1},\ldots,X^2_{i_m}]}(unnest_{A^2}(R'_1 \times R'_1))$
	$R_3 = nest_{B=\langle X^2_{i_1},\ldots,X^2_{i_m}\rangle}(R_2)$
$\neg R_1$	$nest_{B=\langle X_1,\ldots,X_n\rangle}(R_3 \setminus R_2)$ where
	$R_2 = unnest_{A_1}(R'_1)$
	$R_3 = unnest_{A_{t\_tot}}(n(t\_tot(\alpha(R'_1))))$
$\sigma_P(R_1)$	$\Pi_{[B]}(nest_{B=\langle X^2_1,\ldots,X^2_n\rangle}(R_4))$ where
	$R_2 = unnest_{A^2}(R'_1 \times R'_1)$
	$R_3 = unnest_{A_P}(R_2 \times n(P))$
	$R_4 = \Pi_{[A_1,X_1,\ldots,X_n,X^2_1,\ldots,X^2_n]}(\sigma_{X^2_1=X^3_1 \wedge \ldots \wedge X^2_n=X^3_n}(R_3))$
$R_1 \bowtie R_2$	$\Pi_{[B]}(R_8)$ where
	$R_3 = R'_1 \times R'_2 \times R'_1$
	$R_4 = R'_1 \times R'_2 \times R'_2$
	$R_5 = unnest_{A^3}(R_3)$
	$R_6 = unnest_{A^3}(R_4)$
	$R_7 = R_5 \bowtie R_6$
	$R_8 = nest_{B=\langle X^3_1,\ldots,X^3_n\rangle}(R_7)$

Table A.1: Nested-relational algebra expressions corresponding to EGRA expressions.

Theorem A.1 *Let $\mathcal{F}$ be a set of admissible functions. Any EGRA expression can be translated into an equivalent ECAL expression.*

Proof: In the following we prove only the translations that are different from those presented in [88] and [111].

1. $T_{ac}(R_i) = R_i$.

   See [88].

2. $T_{ac}(\sigma_P(e)) = ((x : g(x), t_P(x) :) : \alpha(g) :)$

   where t_P is the target alpha representing the generalized tuple P.

   Suppose that $u \in \sigma_P(e)(I)$. This means that there exists a generalized tuple $u_1 \in e(I)$ such that $u = u_1 \wedge P$ and $ext(u) = ext(u_1) \cap ext(P)$. By induction $u_1 \in \alpha(I)$ and $ext(P) = t_P$. Therefore, $ext(u) = (x : g(x), t_P(x) :)(I)$ when g is bound to u_1 and this concludes the proof.

3. $T_{ac}(\Pi_{[X]}(e)) = (t[X] : r_1, ..., r_h : (\exists r_{h+1})...(\exists r_m)\psi)$

   where $t[X]$ contains free variables $v_1, ..., v_h$ ranging over $r_1, ..., r_h$. Variables of α that are not included in the projection list range over $r_{h+1}, ..., r_m$.

   If α is a target alpha, $t[X]$ is well defined. If α is a general alpha, t is either a target alpha or a set term. In the first case, if $t = (t_1 : r'_1, ..., r'_n : \psi)$, $t[X]$ is defined as $(t_1[X] : r'_1, ..., r'_n : \psi)$, in the second case $t[X]$ is defined as $(v[X] : t(v) :)$.

   The proof follows from [88] by extending the projection on target alphas and set terms.

4. $T_{ac}(e_1 \bowtie e_2) = (((v_1, v_3) : g_1(v_1), g_2(v_2) : \wedge_{k=1,n} v_1[X_{i_k}] = v_2[X_{j_k}]) : \alpha_1(g_1), \alpha_2(g_2) :)$

   where each pair (X_{i_k}, X_{j_k}) represents a pair of variables on which natural join is performed. and v_3 is the tuple formed by all column of v_2 except $X_{j_1}, ..., X_{j_n}$.

   Suppose that $u \in e_1 \bowtie e_2(I)$. This means that there exists a generalized tuple $u_1 \in e_1(I)$ and a generalized tuple $u_2 \in e_2(I)$ and $u = u_1 \wedge u_2$. Moreover, $ext(u) = ext(u_1) \bowtie ext(u_2)$. By inductive hypothesis, $u_1 \in \alpha_1(I)$ and $u_2 \in \alpha_2(I)$. Moreover, it is simple to show that $ext(u) = ((v_1, v_3) : g_1(v_1), g_2(v_2) : \wedge_{k=1,...,n} v_1[X_{i_k}] = v_2[X_{j_k}])(I)$ when g_1 is bound to u_1 and g_2 is bound to u_2 and this concludes the proof.

5. $T_{ac}(\neg e) = ((v : D^n(v) : (\nexists \alpha(g))\ g(v)) : :)$.

   Suppose that $u = (\neg e)(I)$. This means that $u = \neg u_1 \wedge ... \wedge \neg u_n$, and $e(I) = \{u_1, ..., u_n\}$. By induction, $u_i \in \alpha(I)$, $i = 1, ..., n$, and $ext(u) = D^n_t \setminus (ext(u_1) \cup ... \cup ext(u_n)$, where D^n_t represents the generalized tuple whose extension contains all possible relational tuples with n arguments, constructed on domain D. Therefore, $ext(u) = (v : D^n(v) : (\nexists \alpha(g))\ g(v))(I)$ and this concludes the proof.

6. $T_{ac}(\neg^s(e)) = ((v : D^n(v) : \neg g(v)) : \alpha(g) :)$.

   Suppose that $u \in (\neg e)(I)$. This means that there exists a generalized tuple $u_1 \in e(I)$ such that $u = \neg u_1$ and $ext(u) = ext(D^n_t) \setminus ext(u_1)$ where D^n_t represents the generalized tuple whose extension contains all possible relational tuples with n arguments, constructed on domain D. By induction, $u_1 \in \alpha(I)$ and $ext(D^n_t) = D^n$. Therefore, $ext(u) = (v : D^n(v) : \neg g(v))(I)$ when t is bound to u_1 and this concludes the proof.

7. $T_{ac}(e_1 \cup e_2) = (g : \alpha_1(g) \vee \alpha_2(g) :)$.

   See [88].

8. $T_{ac}(e_1 \setminus^s e_2) = (g : \alpha_1(g) : \neg\alpha_2(g))$.

   See [88].

9. $T_{ac}(e_1 \setminus e_2) = ((v : g_1(v) : \neg\exists\alpha_2(g_2)g_2(v)) : \alpha_1(g_1) :)$.

   Suppose that $u \in (e_1 \setminus^t e_2)(I)$. This means that there exists a generalized tuple $u_1 \in e_1(I)$ such that $u = u_1 \wedge \neg u_2 \wedge ... \wedge \neg u_n$, $e_2(I) = \{u_2, ..., u_n\}$ and $ext(u) = ext(u_1) \setminus (ext(u_2) \cup ... \cup ext(u_n)$. By induction, $u_1 \in \alpha_1(I)$ and $u_i \in \alpha_2(I)$, $i = 2, ..., n$. Therefore, $ext(u) = (v : g_1(v) : \neg\exists\alpha_2(g_2)t_2(v))(I)$ when g_1 is bound to u_1 and this concludes the proof.

10. $T_{ac}(\sigma^s_{(Q_1,Q_2,\theta)}(e)) = \quad (g : \alpha(g) : ((\exists\ g_1)T_{ac}(Q'_1)(g_1))$
    $(((\exists\ g_2)T_{ac}(\Pi_{[\alpha(Q_1)]}(Q'_2))(g_2))\ g_1\theta g_2))$

    where Q'_i is obtained from Q_i by replacing constant t with relation $\{t\}$. Note that $T_{ac}(Q'_i) = \{g_i\}$.

    Suppose that $u \in (\sigma^s_{(Q_1,Q_2,\theta)}(e))(I)$. This means that $u \in e(I)$ and $ext(Q_1(u))\theta ext(\Pi_{[\alpha(Q_1)]}(Q_2(u)))$ is true, i.e., $(Q'_1(\{u\})) = u_1$, $(\Pi_{[\alpha(Q'_1)]}(Q'_2(\{u\}))) = u_2$ and $ext(u_1)\theta ext(u_2)$. By induction $u \in \alpha(I)$, $\{u_1\} = T_{ac}(Q'_1(\{u\}))$, and $\{u_1\} = T_{ac}(\Pi_{[\alpha(Q'_1)]}(Q'_2(u)))$. Therefore

    $u \in (g : \alpha(g) : ((\exists\ g_1)T_{ac}(Q'_1)(g_1))(((\exists\ t_2)T_{ac}(\Pi_{[\alpha(Q_1)]}(Q'_2))(g_2))\ g_1\theta g_2))(I)$

    and this concludes the proof.

11. $T_{ac}(AT_f(e)) = (f(g) : \alpha(g) :)$.

    Suppose that $u \in (AT_f(e))(I)$. This means that there exists a generalized tuple $u_1 \in e(I)$ such that $u = f(u_1)$. By induction $u_1 \in \alpha(I)$ and therefore $u \in (f(g) : \alpha(g) :)(I)$ and this concludes the proof.

12. $T_{ac}(AT_f^{\tilde{X}}(e)) =$
    $(((v_2[\tilde{X} \setminus los(f)], v_1) : (f(g))(v_1), g(v_2) : v_2[\tilde{X} \cap los(f)] = v_1[\tilde{X} \cap los(f)]) : \alpha(g) :)$.

    Suppose that $u \in (AT_f^X(e))(I)$. This means that there exists a generalized tuple $u_1 \in e(I)$ such that $u = \Pi_{[X]}(u_1) \bowtie f(u_1)$ and $ext(u) = ext(\Pi_{[X]}(u_1)) \bowtie ext(f(u_1))$. By induction $u_1 \in \alpha(I)$, and $ext(u) = (v_2[X], v_1) : (f(g))(v_1), g(v_2) : v_2[v] = v_1[\tilde{X} \cap los(f)])(I)$ when g is bound to u_1 and this concludes the proof.

□

Theorem A.2 *Let $\mathcal{F}$ be a set of admissible functions. Any ECAL expression can be translated into an equivalent EGRA expression.*

Proof: As in [88], we define several notational conventions using free attribute lists. In the following, L_1, L_1', L_1'' are simple free attribute lists, L_2, L_2', L_2'' are set free attribute lists, e is an expression, and x is a valuation. Let $\langle I, S, X, \rangle$ be a model.

1. $L_1 = X$.

   If $\langle v_i[A], c\rangle \in L_1$, then $c = x_i[A]$[1] is a component of $L_1 = X$.

2. $L_2 = X$.

   If $\langle g_i, sc\rangle \in L_2$, then $(x_i, \Pi_{[sc]}(t), =)$ belongs to $L_2 = X$.

3. L_1^r.

   If $\langle v_i[A], c\rangle \in L_1$, then $c = A$ is a component of L_1^r. Note that in $c = A$, A symbolically represents the column number corresponding to A in the resulting expression.

4. L_2^r.

   If $\langle g_i, sc\rangle \in L_2$, then $(\Pi_{[sc]}(t), \Pi_{[\alpha(g_i)]}, =)$ is a component of L_2^r. Note that in $(\Pi_{[sc]}(t), \Pi_{[\alpha(g_i)]}, =)$, $\alpha(g_i)$ symbolically represents the set of column numbers corresponding to g_i attributes in the resulting expressions.

[1]We recall that we assume that $\langle v_1, x_i\rangle \in X$.

5. L_1^s. If $\langle v_i[A], c\rangle \in L_1$, then $c \in L_1^s$.
6. L_2^s. If $\langle g_i, sc\rangle \in L_2$, then $sc \subseteq L_2^s$.
7. $L_1(i) = \{\langle v_i[A], c\rangle | \langle v_i[A], c\rangle \in L_1\}$.
8. $L_2(i) = \{\langle g_i, sc\rangle | \langle g_i, sc\rangle \in L_2\}$.
9. $L_1(\neg i) = \{\langle v_j[A], c\rangle | \langle v_j[A], c\rangle \in L_1, j \neq i\}$.
10. $L_2(\neg i) = \{\langle g_j, sc\rangle | \langle g_j, sc\rangle \in L_2, j \neq i\}$.
11. $L_1' L_1'' = L_1' \cup L_1''$.
12. $L_2' L_2'' = L_2' \cup L_2''$.

In the following we prove only translations that are different from those presented in [88] and [111]. Moreover, if D is an expression of degree n, we denote with D^* the expression $(\{1\} \setminus (\{1\} \times D)[1]) \times \{1\}^{n-1} \cup D$. Note that D^* is never empty and, given an instance I, $D^*(I) = D(I)$ if $D(I)$ is not empty.

Simple terms

1. *Constants.* Let $c \in D$. Then, $T_{ca}(c) = \langle e, Z, L\rangle$ where

   $$\begin{aligned} &e = \{c\} \\ &Z = 1 \\ &L = \emptyset. \end{aligned}$$

   Proof: see [88].

2. *Simple variables.* Suppose that v_i ranges over the closed alpha r, and D is the equivalent algebraic expression, i.e., $r(I) = D(I)$. Then, $T_{ca}(v_i[A]) = \langle e, Z, L\rangle$ where

   $$\begin{aligned} &e = D[A] \\ &Z = 1 \\ &L = \{\langle v_i[A], 1\rangle\} \end{aligned}$$

   Proof: see [88].

Set terms

1. *Set variables.* Suppose that $g_i \in T$ ranges over the closed alpha r and $T_{ca}(r) = D_i$. Then, $T_{ca}(g_i) = \langle e, Z, L\rangle$ where

   $$\begin{aligned} &e = D_i \\ &Z = \{1, ..., deg(r)\} \\ &L = sc, \text{ where } sc = \{\langle g_i, \{1, ..., deg(r)\}\rangle\}. \end{aligned}$$

   Proof: $\Pi_{[Z]}(\sigma^s_{L=X}(e))(I) = \sigma^s_{(s_i, \Pi_{[sc]}(t), =)}(D_i)(I) = \{s_i\} = \{t_i(I, S, X)\}$.

2. *Constants.* $T_{ca}(D^n) = \langle e, Z, L\rangle$ where

   $$\begin{aligned} &e = \{D^n\} \\ &Z = \{1, ..., n\} \\ &L = \emptyset \end{aligned}$$

   Proof: $\Pi_{[Z]}(\sigma^s_{L=X}(e))(I) = \{D^n\}(I) = \{D^n\} = \{D^n(I, S, x)\}$.

3. *External functions.* Let $T_{ca}(g_i) = \langle h, Y, J\rangle$. Then, $T_{ca}(f(g_i)) = \langle e, Z, L\rangle$ where

   $$\begin{aligned} &e = AT_{\overline{f}}(h \times h) \\ &Z = \{1, ..., deg(h)\} \\ &L = J \end{aligned}$$

   Proof:
   $$\begin{aligned} \Pi_{[Z]}(\sigma^s_{L=X}(e))(I) &= \sigma^s_{J=x}(AT_{\overline{f}}(h \times h))(I) \\ &= \sigma^s_{J=x}(AT_{f,[k+1,...,k+k]}(h \times h))(I) \qquad (1) \\ &= \sigma^s_{J=x}(\{f(\Pi_{[k+1,...,k+k]}(t) \mid t \in h \times h)\})(I) \\ &= \{f(\Pi_{[k+1,...,k+k]}(t) \mid t \in x_i \times x_i)\}(I) \\ &= \{f(x_i)\} \\ &= \{f(g_i(I, S, X))\} \\ &= \{f(g_i)(I, S, X)\} \end{aligned}$$

   Expression in (1) is obtained due to the hypothesis on the validity of the uniformity property.

Simple formulas

1. *Simple range formulas.* Suppose that $T_{ca}(g) = \langle h, Y, J\rangle$. Suppose that v_i ranges over r, corresponding to expression D_i. $T_{ca}(g(v_i)) = \langle e, Z, L\rangle$ where

$e = h$
$E = D_i$
$L = \{\langle v_i[1], 1\rangle, ..., \langle v_i[n], n\rangle\}$

Proof: $\sigma_{L=X}(e)(I) = \sigma_{1=x_i[1]\wedge...\wedge n=x_i[n]}(h)(I) = \sigma_{1=x_i[1]\wedge...\wedge n=x_i[n]}(D_i)(I)$.

(a) If $t(v_i)(I, S, X) = 1$, then $x_i \in D_i$ and $\sigma_{1=x_i[1]\wedge...\wedge n=x_i[n]}(D_i)(I) = \{x_i\} = \sigma_{L=X}(E)(I)$.

(b) If $t(v_i)(I, S, X) = 0$, then $x_i \notin D_i$ and $\sigma_{1=x_i[1]\wedge...\wedge n=x_i[n]}(D_i)(I) = \emptyset$.

(c) $\sigma_{L=X}(E)(I) = \sigma_{1=x_i[1]\wedge...\wedge n=x_i[n]}(D_i^*)(I) = \{x_i\} \neq \emptyset$.

2. *Simple constraints.* Let $T_{ca}(s_i) = \langle e_i, Z_i, L_i\rangle$, $i = 1, .., n$.
Then, $T_{ca}(\mu(s_1, ..., s_n)) = \langle e, E, L\rangle$ where

$e = \sigma_{\mu(Z_1,...,Z_n)}(e_1 \times ... \times e_n)$
$E = e_1 \times ... \times e_n$
$L = L_1 L_2 ... L_n$.

Proof: see the proof in [88].

3. *Negation.* Suppose that $T_{ca}(\psi) = \langle e_1, E_1, L_1\rangle$. Then, $T_{ca}(\neg\psi) = \langle e, E, L\rangle$ where

$e = E_1 \setminus e_1$
$E = E_1$
$L = L_1$.

Proof: see [88].

4. *Disjunction.* Suppose that $T_{ca}(\psi_i) = \langle e_i, E_i, L_i\rangle$, $i = 1, ..., 2$. Then, $T_{ca}(\neg\psi) = \langle e, E, L\rangle$ where

$e = (e_1 \times E_2) \cup (E_1 \times e_2)$
$E = E_1 \times E_2$
$L = L_1 L_2$.

Proof: see [88].

5. *Existential quantification.* Suppose that the translation of range r (i.e., the closed alpha) produces the algebra expression D and $T_{ca}(\psi) = \langle e_1, E_1, L_1\rangle$. Then, $T_{ca}((\exists r_{v_i})\psi) = \langle e, E, L\rangle$ where

$e = \Pi_{[L_1^s(\neg i)]}(\sigma_{L_1^r(i)}(e_1 \times D))$
$E = \Pi_{[L_1^s(\neg i)]}(E_1)$
$L = L_1 L_2.$

Proof: see [88].

Set formulas

1. *Set range formulas.* Suppose that $T_{ca}(\alpha) = \langle h, Y, J\rangle$. Suppose that t_i ranges over r, corresponding to an expression D_i. $T_{ca}(\alpha(g_i)) = \langle e, E, L\rangle$ where

   $e = h$
   $E = D_i$
   $L = \{\langle g_i, \{1, ..., deg(r)\}\rangle\}$

   Proof: $\sigma^s_{L=X}(e)(I) = \sigma^s_{(t,x_i,=)}(h)(I) = \sigma^s_{(t,x_i,=)}(D_i)(I)$.

   (a) If $\alpha(g_i)(I, S, X) = 1$, then $x_i \in D$ and $\sigma^s_{(t,x_i,=)}(D_i)(I) = \{x_i\} = \sigma^s_{J=X}(E)(I)$.
   (b) If $\alpha(g_i)(I, S, X) = 0$, then $x_i \notin D_i$ and $\sigma^s_{(t,x_i,=)}(D_i)(I) = \emptyset$.
   (c) $\sigma^s_{J=X}(E)(I) = \sigma^s_{(t,x_i,=)}(D_i^*)(I) = \{x_i\} \neq \emptyset$.

2. *Set constraints.* Let $T_{ca}(t_1) = \langle e_1, Z_1, L_1\rangle$ and $T_{ca}(t_2) = \langle e_2, Z_2, L_2\rangle$. Then, $T_{ca}(t_1\theta t_2) = \langle e, E, L\rangle$ where

   $e = \sigma^s_{(\Pi_{[Z_1]}(t),\Pi_{[Z_2]}(t),\theta)}(e_1 \times e_2)$
   $E = e_1 \times e_2$
   $L = L_1 L_2.$

   Proof: see [111].

3. *Negation.* Suppose that $T_{ca}(\psi) = \langle e_1, E_1, L_1\rangle$. Then, $T_{ca}(\neg\psi) = \langle e, E, L\rangle$ where

   $e = E_1 \setminus^s e_1$
   $E = E_1$
   $L = L_1.$

   Proof: $\sigma^s_{L=X}(e)(I) = \sigma^s_{L_1=x}(E_1 \setminus^s e_1)(I) = \sigma^s_{L_1=x}(E_1)(I) \setminus^s \sigma^s_{L_1=x}(e_1)(I)$.

   (a) If $\neg\psi(I, S, x) = 1$, then $\psi(I, S, x) = 0$ and $\sigma^s_{L_1=x}(E_1)(I) \setminus^s \sigma^s_{L_1=x}(e_1)(I) = \sigma^s_{L_1=x}(E_1)(I) = \sigma^s_{L_1=x}(E)(I)$.
   (b) If $\neg\psi(I, S, x) = 0$, then $\psi(I, S, x) = 1$ and $\sigma^s_{L_1=x}(E_1)(I) \setminus^s \sigma^s_{L_1=x}(e_1)(I) = \emptyset$.

(c) Moreover, $\sigma^s_{L_1=x}(E)(I) = \sigma^s_{L_1=x}(E_1)(I) \neq \emptyset$.

4. *Disjunction.* Suppose that $T_{ca}(\psi_i) = \langle e_i, E_i, L_i\rangle$, $i = 1, ..., 2$. Then, $T_{ca}(\psi_1 \vee \psi_2) = \langle e, E, L\rangle$ where

   $e = (e_1 \times E_2) \cup (E_1 \times e_2)$
   $E = E_1 \times E_2$
   $L = L_1 L_2$.

   Proof: See [88].

5. *Existential quantification.* Suppose that the translation of range r (i.e., the closed set alpha) produces the algebra expression D and $T_{ca}(\psi) = \langle e_1, E_1, L_1\rangle$. Then, $T_{ca}((\exists r_{v_i})\psi) = \langle e, E, L\rangle$ where

   $e = \Pi_{[L^s_1(\neg i)]}(\sigma_{L^r_1(i)}(e_1 \times D))$
   $E = \Pi_{[L^s_1(\neg i)]}(E_1)$
   $L = L_1 L_2$.

   Proof: See [88].

Simple alphas

1. *Target alpha.* Consider a target alpha $\alpha_1 = (t_1, ..., t_n) : r_1, ..., r_m : \psi$. Suppose that $T_{ca}(t_i) = \langle e_i, Z_i, L_i\rangle$. Let simple range formulas $r_1, ..., r_m$ produce the equivalent algebra expressions $D_1, ..., D_m$ and let $T_{ca}(\psi) = \langle e_2, E_2, L_2\rangle$. Then, $T_{ca}(\alpha_1) = \langle e, Z, L\rangle$ where

   $e = \sigma_{L^r_1 L^r_2(1,...,m)}(e_1 \times \times e_n \times D_1 \times ... \times D_m)$
   $Z = Z_1 Z_2 ... Z_n$
   $L = L_2(\neg 1 ... m)$

   Proof: See [88].

Set alphas

1. *Atomic alpha.* $T_{ca}(R_i) = \langle e, Z, L\rangle$ where

   $e = R_i$
   $Z = \{1, ..., n\}$, where $n = deg(R_i)$
   $L = \emptyset$

2. *General alpha.* Consider a general alpha $\alpha_1 = (t_1) : r_1, ..., r_m : \psi$. Suppose that $T_{ca}(t_1) = \langle e_1, Z_1, L_1 \rangle$. Suppose that the translation of range formulas $r_1, ..., r_m$ produce the equivalent algebraic expressions $D_1, ..., D_m$ and let $T_{ca}(\psi) = \langle e_2, E_2, L_2 \rangle$. Let $\alpha = T_{ca}(t_1 : :)) = \langle \overline{e}, \overline{Z}, \overline{L} \rangle$. $T_{ca}(\alpha_1) = \langle e, Z, L \rangle$ where

   $e = \sigma^s_{L_1^r L_2^r(1,...,m)}(e_1' \times e_2 \times D_1 \times ... \times D_m)$ where

   $$e_1' = \begin{cases} \Pi_{[\overline{ZL}^s]}(\overline{e}) & \text{if } t_1 \text{ is a target alpha} \\ e_1 & \text{if } t_1 \text{ is a set term} \end{cases}$$

   $Z = Z_1$
   $L = L_2(\neg 1...m)$

   Proof: We consider two cases:

   (a) (t_1) is a set term: the proof follows from the proof proposed in [88].

   (b) t_1 is a target alpha: We assume that $m = 1$ and t_1 has only one free variable v_i, ranging over $T_{ca}(r_1) = D$.

   $$\begin{aligned} &\Pi_{[Z]}(\sigma^s_{L=X}(e))(I) \\ &\quad = \Pi_{[\overline{Z},Z_2]}(\sigma^s_{L_2(\neg i)=x}(\sigma^s_{\overline{L}^r L_2^r(i)}(\Pi_{[\overline{ZL}^s]}(\overline{e}) \times e_2 \times D))) \\ &\quad = \Pi_{[\overline{Z},Z_2]}(\sigma^s_{L_2(\neg i)=x}(\cup_{t \in D(t)}(\sigma^s_{\overline{L}L_2(i)=t}(\Pi_{[\overline{ZL}^s]}(\overline{e}) \times e_2)))) \\ &\quad = \Pi_{[\overline{Z},Z_2]}(\cup_{t \in D(t)}(\sigma^s_{\overline{L}=X'}(\Pi_{[\overline{ZL}^s]}(\overline{e})) \times \sigma^s_{L_2(i)=X'}(e_2))) \\ &\quad = \Pi_{[\overline{Z},Z_2]}(\cup_{t \in D(t), \psi(I,S,X')=1}(\sigma^s_{\overline{L}=X'}(\Pi_{[\overline{ZL}^s]}(\overline{e})))) \\ &\quad = \cup_{t \in D(t), \psi(I,S,X')=1}(\Pi_{[\overline{Z}]}(\sigma^s_{\overline{L}=X'}(\Pi_{[\overline{ZL}^s]}(\overline{e})))) \\ &\quad = \cup_{t \in D(t), \psi(I,S,X')=1}(\Pi_{[\overline{Z}]}(\Pi_{[\overline{ZL}^s]}(\sigma^s_{\overline{L}=X'}(\overline{e})))) \\ &\quad = \cup_{t \in D(t), \psi(I,S,X')=1}(\Pi_{[\overline{Z}]}(\Pi_{[\overline{Z}]}(\sigma^s_{\overline{L}=X'}(\overline{e})))) \\ &\quad = \cup_{t \in D(t), \psi(I,S,X')=1}(\Pi_{[\overline{Z}]}(\sigma^s_{\overline{L}=X'}(\overline{e}))) \\ &\quad = \cup_{t \in D(t), \psi(I,S,X')=1}(\alpha(I, S, X')) \\ &\quad = \cup_{t \in D(t), \psi(I,S,X')=1}\{t_1(I, S, X')\} \\ &\quad = \alpha_1(I, S, X'). \end{aligned}$$

   where $X' = X \cup \{\langle v_i, t \rangle\}$.

□

Appendix B

Selected proofs of results presented in Chapter 5

Proposition 5.4 *For every function $f : s_1 \to s_2$ in $g\mathcal{NRC}$, there is a function $(f)' : s'_1 \to s'_2$ such that*

$$\begin{tikzcd}
s_1 \arrow[r, "id"] \arrow[d, "p_{s_1}"'] & s_1 \arrow[r, "f"] & s_2 \arrow[r, "id"] & s_2 \arrow[d, "p_{s_2}"] \\
s'_1 \arrow[r, "\sim"'] & s'_1 \arrow[u, "q_{s_1}"] \arrow[r, "(f)'"'] & s'_2 \arrow[u, "q_{s_2}"'] \arrow[r, "\sim"'] & s'_2
\end{tikzcd}$$

Proof. We note that the left and right squares commute by definitions of p_s, q_s, and $\sim$. We now construct $(f)'$ by induction on the structure of the g$\mathcal{NRC}$ expression that defines f and we argue that the middle square and thus the entire diagram commutes. To simplify notations, we omit the subscript s from p_s and q_s in the proof below. We also omit the argument for the more obvious cases.

- Case $f(\vec{x}) = x_i$. Set $(f)'(O) = \{_{fr} x_i \mid (x_1, \ldots, x_n) \in_{fr} O\}$.

  For this case, suppose $q(O) = \vec{x}$. Thus $O \sim p(\vec{x})$. Then $(f)'(O) \sim p(x_i)$. Since $q \circ p = id$, we have $q((f)'(O)) = x_i$. So the middle square commutes.

- Case $f(\vec{x}) = c$. Set $(f)'(O) = \{_{fr} c\}$

- Case $f(\vec{x}) = \pi_i e$. Let $g(\vec{x}) = e$. Set $(f)'(O) = \{_{fr} x_i \mid (x_1, \ldots, x_n) \in_{fr} (g)'(O)\}$.

For this case, suppose $q(O) = \vec{x}$. Thus $O \sim p(\vec{x})$. By hypothesis, $q((g)'(O)) = e$. Thus $(g)'(O) \sim p(e)$. By definition

$p(\pi_i e) = \{_{fr} x_i \mid (x_1, \ldots, x_n) \in_{fr} p(e)\}$.

Then $(f)'(O) \sim p(\pi_i e)$. Hence $q((f)'(O)) = \pi_i e$. So the middle square commutes.

- Case $f(\vec{x}) = (e_1, \ldots, e_n)$. Let $g_i(\vec{x}) = e_i$. Set

  $(f)'(O) = \{_{fr}(x_1, \ldots, x_n) \mid x_1 \in_{fr} (g_1)'(O), ..., x_n \in_{fr} (g_n)'(O)\}$.

  For this case, suppose $q(O) = \vec{x}$. Thus $O \sim p(\vec{x})$. By hypothesis, $q((g_i)'(O)) = e_i$. Thus $(g_i)'(O) \sim p(e_i)$.

  By definition, $p((e_1, \ldots, e_n)) = \{_{fr}(x_1, \ldots, x_n) \mid x_1 \in_{fr} p(e_1), \ldots, x_n \in_{fr} p(e_n)\}$. Then $(f)'(O) \sim p((e_1, \ldots, e_n))$. Hence $q((f)'(O)) = (e_1, \ldots, e_n)$. So the middle square commutes.

- Case $f(\vec{x}) = \{\}$. Set $(f)'(O) = \{_{fr}(0, 0, \vec{0})\}$.

- Case $f(\vec{x}) = \{e\}$. Let $g(\vec{x}) = e$. Set $(f)'(O) = \{_{fr}(1, 1, x) \mid x \in_{fr} (g)'(O)\}$.

  For this case, suppose $q(O) = \vec{x}$. Thus $O \sim p(\vec{x})$. By hypothesis, $q((g)'(O)) = e$. Thus $(g)'(O) \sim p(e)$.

  By definition, $p(\{e\}) = \{_{fr}(1, 1, x) \mid x \in_{fr} p(e)\}$. Then $(f)'(O) \sim p(\{e\})$. Hence $q((f)'(O)) = \{e\}$. So the middle square commutes.

- Case $f(\vec{x}) = \mathit{empty}_{fr} e$. Let $g(\vec{x}) = e$. Set $(f)'(O) = \{_{fr} 1 \mid (0, \vec{0}) \in_{fr} (g)'(O)\} \cup_{fr} \{_{fr} 0 \mid (1, x) \in_{fr} (g)'(O)\}$.

  For this case, suppose $q(O) = \vec{x}$. Thus $O \sim p(\vec{x})$. By hypothesis, $q((g)'(O)) = e$. Thus $(g)'(O) \sim p(e)$. Thus e is empty iff $(g)'(O) = \{_{fr}(0, \vec{0})\}$. Thus $q((f)'(O)) = \mathit{empty}_{fr} e$. So the middle square commutes.

  The following variations of the emptiness test are used in subsequent cases:

  $$\mathit{myempty}(X) = \mathit{empty}_{fr}\{_{fr} 0 \mid 0 \in_{fr} (\mathit{empty}_{fr})'(X)\}$$

  $$\mathit{myempty}'(X) = \mathit{empty}_{fr}\{_{fr} 1 \mid (1, x) \in_{fr} X\}$$

- Case $f(\vec{x}) = e_1 \cup e_2$. Let $g_i(\vec{x}) = e_i$. Set

```
$(f)'(O) =$ if $myempty((g_1)'(O))$ then
                ($if$ $myempty((g_2)'\ (O))$ $then$
                  $\{_{fr}\vec{0}\}$
```

```
        else $(g_2)'(O))$
    else
        $(if\ myempty((g_2)'(O))then$
            $(g_1)'(O)$
        else $(g_1)'(O) \cup_{fr} A(O))$
```

where

$$A(O) = \{_{fr}(1, i+k+1, x) \mid (1, i, x) \in_{fr} (g_2)'(O), \\ k = \max\{_{fr} j \mid (1, j, y) \in_{fr} (g_1)'(O)\}.$$

For this case, suppose $q(O) = \vec{x}$. Thus $O \sim p(\vec{x})$.

By hypothesis, $q((g_i)'(O)) = e_i$. There are four subcases. The subcases where either e_1 or e_2 is empty are trivial. So suppose e_1 and e_2 are both not empty. First note that $A(O) \sim (g_2)'(O)$ and thus $q(A(O)) = e_2$. Next observe that for any $(1, i, x)$ in $(g_1)'(O)$ and $(1, j, y)$ in $A(O)$, it is the case that $i < j$. Thus for any h so that one of the set $X = \{_{fr}(1, i, x) \mid (1, i, x) \in_{fr} (f)'(O),\ i = h\}$, $Y = \{_{fr}(1, i, x) \mid (1, i, x) \in_{fr} (g_1)'(O), i = h\}$, $Z = \{_{fr}(1, i, x) \mid (1, i, x) \in_{fr} A(O),\ i = h\}$ is not empty, it is the case that $X = Y$ or $X = Z$. Consequently, for any o in $(f)'(O)$, we have o in $e_1 \cup e_2$ and vice versa. That is, $q((f)'(O)) = e_1 \cup e_2$. So the middle square commutes.

- Case $f(\vec{x}) = \bigcup\{e_1 \mid y \in e_2\}$. Let $g_1(\vec{x}, y) = e_1$ and $g_2(\vec{x}) = e_2$. Set

```
$(f)'(O) =$if $myempty((g_2)'\ (O))then$
        $\{_{fr}(0, 0, \vec{0})\}$
    else if $myempty'(A(O))then$
        $\{_{fr}(0, 0, \vec{0})\}$
    else $A(O)$
```

where $A(O) = \bigcup\{_{fr}\ B(O, i) \mid (1, i, y) \in_{fr} (g_2)'(O)\}$, where $B(O, i) = \{_{fr}(1, k * i + h + 1, w) \mid (1, h, w) \in_{fr} C(O, i),\ k = \max\{_{fr} h \mid (1, j, u) \in_{fr} (g_2)'(O),\ j < i,\ (1, h, v) \in_{fr} C(O, j)\}\}$, where $C(O, i) = (g_1)'(\{_{fr}(z, u) \mid z \in_{fr} O,\ (1, j, u) \in_{fr} (g_2)'(O),\ i = j\})$.

This is the most complex case. Suppose $q(O) = p(\vec{x})$. Thus $O \sim p(\vec{x})$. By hypothesis, $q((g_2)'(O)) = e_2$. Thus $(g_2)'(O) \sim p(e_2)$. Now there are two subcases. For the first subcase, suppose e_2 is empty. Then $myempty((g_2)'(O))$

is true. Then $q((f)'(O)) = \{_{fr}\} = f(\vec{x})$. So the middle square commutes in this subcase.

For the second subcase, we assume that e_2 is not empty. Then $myempty((g_2)'(O))$ is false. By the hypothesis on g_2, we know that for each i such that the set $y_i = \{_{fr} u \mid (1, j, u) \in_{fr} (g_2)'(O), i = j\}$ is not empty, we have $q(y_i)$ is an element o_i of e_2. Moreover, there is one such i for each element of e_2. Then by hypothesis on g_1, we have $q(C(O, i)) = g_1(\vec{x}, o_i)$ for each such i. It is also obvious that $q(B(O, i)) = g_1(\vec{x}, o_i)$ for each such i, provided $g_1(\vec{x}, o_i)$ is not empty. Note that if $g_1(\vec{x}, o_i)$ is empty, then $B(O, i)$ is also empty, as opposed to being a singleton zero tuple.

However, $B(O, i)$ has an advantage over $C(O, i)$ because the numbers it uses to identify the elements of $g_1(\vec{x}, o_i)$ are distinct from those of $B(O, j)$ whenever $i \neq j$. To see this, suppose $k = \max\{_{fr} h \mid (1, j, u) \in_{fr} (g_2)'(O),\ j < i,\ (1, h, v) \in_{fr} C(O, j)\}$. Then k is the maximum identifier that is used to identify elements in $g_1(\vec{x}, o_j)$, for $j < i$. This k exists because $g_1(\vec{x}, o_j)$ is finite for each o_j in e_2. Then $k * i + 1$ is greater than the cardinality of the union of $g_1(\vec{x}, o_j)$ for $j < i$.

We now have two subsubcases. For the first subsubcase, suppose $g_1(\vec{x}, o_i)$ is empty for each o_i in e_2. Then $B(O, i)$ is empty for all such o_i. Then $A(O)$ is also empty. Then $myempty'(A(O))$ is true. Then $q((f)'(O)) = \{_{fr}\} = f(\vec{x})$. So the middle square commutes in this subsubcase.

For the second subsubcase, we assume that there are o_1, ..., o_n in e_2 such that $g_1(\vec{x}, o_i)$ is not empty and $f(\vec{x}) = g_1(\vec{x}, o_1) \cup_{fr} \cdots \cup_{fr} g_1(\vec{x}, o_n)$. Then $(f)'(O) = A(O) = B(O, o_1) \cup_{fr} \cdots \cup_{fr} B(O, o_n)$. Then $q((f)'(O)) = f(\vec{x})$. This finishes the final subsubcase.

- Case $f(\vec{x}) = (e_1 = e_2)$. Let $g_i(\vec{x}) = e_i$. Set

$$\begin{aligned}(f)'(O) = {}& \mathit{if}\ \mathit{empty}_{fr}\{_{fr} 1 \mid x \in_{fr} (g_1)'(O), y \in_{fr} (g_2)'(O), x = y\}\ \mathit{then} \\ & \{_{fr}\mathit{false}\}\ \mathit{else}\ \{_{fr}\mathit{true}\}.\end{aligned}$$

For this case, suppose $q(O) = \vec{x}$. Thus $O \sim p(\vec{x})$. By hypothesis, $q((g_i)'(O)) = e_i$. So $(g_i)'(O) \sim p(e_i)$. Since $e_i : \mathbb{R}$, we have $(g_i)'(O) = \{_{fr} e_i\}$. Then it is obvious that $q((f)'(O)) = (e_1 = e_2)$. So the middle square commutes.

- Case $f(\vec{x}) = \mathit{empty}\ e$. Let $g(\vec{x}) = e$.

  Set $(f)'(O) = \{_{fr} \mathit{empty}_{fr}\{_{fr} 1 \mid (1, i, x) \in_{fr} (g)'(O)\}\}$.

- Case $f(\vec{x}) = \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3$. Let $g_i(\vec{x}) = e_i$.

  Set $(f)'(O) = \mathit{if}\ \mathit{empty}_{fr}\{_{fr}1 \mid 0 \in_{fr} (g_1)'(O)\}\ \mathit{then}\ (g_2)'(O)\ \mathit{else}\ (g_3)'(O)$.

  For this case, suppose $q(O) = \vec{x}$. Thus $O \sim p(\vec{x})$.

  By hypothesis, $q((g_1)'(O)) = e_1$. Thus $(g_1)'(O) \sim p(e_1)$. Since $e_1 : \mathbb{B}$, we have $(g_2)'(O) = \{_{fr}1\}$ if e_1 is true and $(g_2)'(O) = \{_{fr}0\}$ if e_1 is false. Then $\mathit{empty}_{fr}\{_{fr}1 \mid 0 \in_{fr} (g_1)'(O)\}$ is true iff e_1 is true. Then it follows by hypothesis on e_2 and e_3 that $q((f)'(O)) = \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3$. So the middle square commutes.

- Case $f(\vec{x}) = \{_{fr}\}$. Set $(f)'(O) = \{_{fr}\vec{0}\}$.

- Case $f(\vec{x}) = \{_{fr}e\}$. Let $g(\vec{x}) = e$. Set $(f)'(O) = \{_{fr}(1, x) \mid x \in_{fr} (g)'(O)\}$.

  For this case, suppose $q(O) = \vec{x}$. Thus $O \sim p(\vec{x})$.

  By hypothesis, $q((g)'(O)) = e$. Thus $(g)'(O) \sim p(e)$. Since $e : \mathbb{R} \times \cdots \times \mathbb{R}$, we know that $(g)'(O) = \{_{fr}e\}$. Thus $q((f)'(O)) = \{_{fr}e\}$. So the middle square commutes.

- Case $f(\vec{x}) = e_1 \cup_{fr} e_2$. Let $g_i(\vec{x}) = e_i$. Set

```
$(f)'(O) = \mathit{if}\ \mathit{myempty}((g_1)'(O))\mathit{then}$
              $(\mathit{if}\ \mathit{myempty}\ ((g_2)'(O))\mathit{then}$
                $\{_{fr}\vec{0}\}$
              $\mathit{else}\ (g_2)'(O))$
           $\mathit{else}$
              $(\mathit{if}\ \mathit{myempty}((g_2)'(O))\mathit{then}$
                $(g_1)'(O)$
              $\mathit{else}\ (g_1)'(O) \cup_{fr} (g_2)'(O))$.
```

- Case $f(\vec{x}) = \bigcup\{_{fr}e_1 \mid y \in_{fr} e_2\}$. Let $g_1(\vec{x}, y) = e_1$ and $g_2(\vec{x}) = e_2$.
  Set

```
$(f)'(O) = \mathit{if}\ \mathit{myempty}\ ((g_2)'(O))\ \mathit{then}$
              $\{_{fr}\vec{0}\}$
           $\mathit{else}$
              $\mathit{if}\ \mathit{myempty}'(A(O))\ \mathit{then}$
                $\{_{fr}(0, 0, \vec{0})\}$
              $\mathit{else}\ A(O)$
```

where

$$A(O) = \bigcup\{_{fr}\ \textit{if myempty}\ (B(O,y))\ \textit{then}\ \{_{fr}\}\ \textit{else}\ B(O,y) \mid (1,y) \in_{fr} (g_2)'(O)\}$$

and $B(O,y) = (g_1)'(\{_{fr}(z,y) \mid z \in_{fr} O\})$.

For this case, suppose $q(O) = \vec{x}$. Thus $O \sim p(\vec{x})$.

By hypothesis, $q((g_2)'(O)) = e_2$. Thus $(g_2')(O) \sim p(e_2)$. Now there are two subcases. For the first subcase, suppose e_2 is empty. Then $\textit{myempty}((g_2)'(O))$ is true. Then $q((f)'(O)) = \{_{fr}\} = \bigcup\{_{fr}e_1 \mid y \in_{fr} e_2\}$. So the middle square commutes in this subcase.

For the second subcase, we assume that e_2 is not empty. Then $\textit{myempty}((g_2)'(O))$ is false and $(g_2)'(O) = \{_{fr}(1,x) \mid x \in_{fr} e_2\}$ is forced. It is clear that $q(\{_{fr}(z,y) \mid z \in_{fr} O\}) = (\vec{x},y)$ for each y in e_2. By hypothesis, $q(B(O,y)) = g_1(\vec{x},y)$ for each y in e_2. We now have two subsubcases. For the first subsubcase, suppose $g_1(\vec{x},y)$ is empty for each y in e_2. Then $\textit{myempty}(B(O,y))$ is true for each y in e_2. Then $\textit{myempty}'(A(O))$ is true. Then $q((f)'(O)) = \{_{fr}\} = f(\vec{x})$. So the middle square commutes in this subsubcase.

For the second subsubcase, we assume that there are y_1, ..., y_n in e_2 such that $g_1(\vec{x},y_i)$ is not empty and $f(\vec{x}) = g_1(\vec{x},y_1) \cup_{fr} \cdots \cup_{fr} g_1(\vec{x},y_n)$. Then $(f)'(O) = A(O) = B(O,y_1) \cup_{fr} \cdots \cup_{fr} B(O,y_n)$. Then $q((f)'(O)) = f(\vec{x})$. This finishes the final subsubcase. So the middle square commutes.

- Case $f(\vec{x}) = e_1 \oplus e_2$, where $\oplus$ is either $+$, $-$, $\cdot$, or $\div$. Let $g_i(\vec{x}) = e_i$. Set $(f)'(O) = \{_{fr}x \oplus y \mid x \in_{fr} (g_1)'(O),\ y \in_{fr} (g_2)'(O)\}$.

  For this case, suppose $q(O) = \vec{x}$. Thus $O \sim p(\vec{x})$.

  By hypothesis, $q((g_i)'(O)) = e_i$. Thus $(g_i)'(O) \sim p(e_i)$. Since $e_i : \mathbb{R}$, we must have $(g_i)'(O) = \{_{fr}e_i\}$. Then $q((f)'(O)) = q(\{_{fr}e_1 \oplus e_2\}) = e_1 \oplus e_2$. So the middle square commutes.

- Case $f(\vec{x}) = R$. Set $(f)'(O) = \{_{fr}(1,x) \mid x \in_{fr} R\}$.

- Case $f(\vec{x}) = \bigcup\{_{fr}e_1 \mid y \in e_2\}$. Let $g_1(\vec{x},y) = e_1$ and $g_2(\vec{x}) = e_2$. Set $(f)'(O) = \textit{if myempty}((g_2)'\ (O))\ \textit{then}\ \{_{fr}(0,\vec{0})\}\ \textit{else if myempty}'(A(O))\ \textit{then}\ \{_{fr}(0,\vec{0})\}\ \textit{else}\ A(O)$, where $A(O) = \bigcup\{_{fr}\ \textit{if myempty}\ (B(O,i))\ \textit{then}\ \{_{fr}\}\ \textit{else}\ B(O,i) \mid (1,i,y) \in_{fr} (g_2)'(O)\}$, where

$B(O,i) = (g_1)'(\{_{fr}(z,u) \mid z \in_{fr} O,\ (1,j,u) \in_{fr} (g_2)'(O),\ i = j\})$.

For this case, suppose $q(O) = \vec{x}$.

Thus $O \sim p(\vec{x})$. By hypothesis, $q((g_2)'(O)) = e_2$. Thus $(g_2)'(O) \sim p(e_2)$. We have two subcases. The first is when e_2 is empty. Then $(g_2)'(O)$ is a singleton zero tuple. Then $myempty((g_2)'\ (O))$ is true. Then $(f)'(O) = \{_{fr}(0,\vec{0})\}$. Thus $q((f)'(O)) = \{_{fr}\}\ = f(\vec{x})$. So the middle square commutes in this subcase.

For the second subcase, we assume that e_2 is not empty. Then $myempty((g_2)'(O))$ is false. By the hypothesis on g_2, we know that for each i such that the set $y_i = \{_{fr} u \mid (1,j,u) \in_{fr} (g_2)'(O), i = j\}$ is not empty, we have $q(y_i)$ is an element o_i of e_2. Moreover, there is one such i for each element of e_2. Then by hypothesis on g_1, we have $q(B(O,i)) = g_1(\vec{x}, o_i)$ for each such i. We now have two subsubcases. For the first subsubcase, suppose $g_1(\vec{x}, o_i)$ is empty for each such i. Then $myempty(B(O,i))$ is true for each such i. Then $myempty'(A(O))$ is true. Then $q((f)'(O)) = \{_{fr}\}\ = f(\vec{x})$. So the middle square commutes in this subsubcase.

For the second subsubcase, we assume that there are o_1, ..., o_n in e_2 such that $g_1(\vec{x}, o_i)$ is not empty and $f(\vec{x}) = g_1(\vec{x}, o_1) \cup_{fr} \cdots \cup_{fr} g_1(\vec{x}, o_n)$. Then $(f)'(O) = A(O) = B(O, o_1) \cup_{fr} \cdots \cup_{fr} B(O, o_n)$. Then $q((f)'(O)) = f(\vec{x})$. This finishes the final subsubcase. So the middle square commutes. $\square$

Appendix C

Selected proofs of results presented in Chapter 8

Lemma 8.3 *Let* $UP^-(P) = \{(p_1, ..., p_d) | (p_1, ..., p_d) \in UP(P)$ *and* $p_d = min\{p'_d | (p_1, ..., p_{d-1}, p'_d) \in UP(P)\}$. *Let* $DOWN^-(P) = \{(p_1, ..., p_d) | (p_1, ..., p_d) \in UP(P)$ *and* $p_d = max\{p'_d | (p_1, ..., p_{d-1}, p'_d) \in DOWN(P)\}$. *Then,* $(p_1, ..., p_d) \in UP^-(P)$ *iff* $TOP^P(p_1, ..., p_{d-1}) = p_d$ *and* $(p_1, ..., p_d) \in DOWN^-(P)$ *iff* $BOT^P(p_1, ..., p_{d-1}) = p_d$.
Proof:

$\Rightarrow$ Suppose that $(p_1, ..., p_d) \in UP^-(P)$. There are two different cases:

- $(p_1, ..., p_d)$ belongs to a line which is the dual representation of a vertex, say v, of P (thus, it has been added in Steps 1 or 2 of the transformation algorithm). Since $(p_1, ..., p_d) \in UP^-(P)$, and all hyperplanes in $UP(P)$ are transformed in 1-half-planes, this also mean that there does not exist another hyperplane H' supporting $UP(P)$ such that $(p_1, ..., p_{d-1}, p'_d) \in H'$ and $p'_d > p_d$. But, since H' would correspond to another vertex v' of P, this means that $p_d = max_{v \in V_P}\{F_{D(v)}(p_1, ..., p_{d-1})\} = TOP^P(p_1, ..., p_{d-1})$.
- $(p_1, ..., p_d)$ belongs to a line which has been added in Steps 3 or 4 of the transformation algorithm. Since this line connects $d-1$ points representing in the primal plane $d-1$ unbound hyperplanes, $(p_1, ..., p_d)$ in the primal plane represents a hyperplane passing through the vertex defined by the intersection of such hyperplanes. Since $(p_1, ..., p_d) \in UP^-(P)$, and all hyperplanes in $UP(P)$ are transformed in 1-half-planes, this also mean that there does not exist another hyperplane H', besides the one on which $(p_1, ..., p_d)$ lies, supporting $UP(P)$ such that $(p_1, ..., p_{d-1}, p'_d) \in H'$ and

$p'_d > p_d$. But, since H' would correspond to another vertex v' of P, this means that $p_d = max_{v \in V_P}\{F_{D(v)}(p_1, ..., p_{d-1})\} = TOP^P(p_1, ..., p_{d-1})$.

A similar reasoning can be done for BOT^P.

$\Leftarrow$ Suppose that $TOP^P(p_1, ..., p_d) = p_d$ (thus, for Lemma 8.2, $D((p_1, ..., p_d))$ is a supporting hyperplane for P) and assume that $(p_1, ..., p_d) \notin UP^-(P)$. There are several cases:

1. $(p_1, ..., p_d) \notin UP(P) \cup DOWN(P)$. Three different situations may arise:
   (a) There exist two points $(p_1, ..., p'_d)$ and $(p_1, ..., p''_d)$ such that $p''_d < p_d < p'_d$ and such that $(p_1, ..., p'_d) \in UP^-(P)$ and $(p_1, ..., p''_d) \in DOWN^-(P)$. For the proof of $\Rightarrow$ and Lemma 8.2, it follows that in the primal plane these points represent hyperplane which are supporting with respect to P. But this also means that $D((p_1, ..., p_d))$ intersects P and is not supporting. But this leads to a contradiction, since $D((p_1, ..., p_d))$ is supported by hypothesis.
   (b) There exists a point $(p_1, ..., p'_d)$ such that $p_d < p'_d$ and $(p_1, ..., p'_d) \in UP^-(P)$ but there does not exist a point $(p_1, ..., p''_d)$ such that $p''_d < p_d$ and $(p_1, ..., p''_d) \in DOWN^-(P)$.
   For the proof of $\Rightarrow$ and Lemma 8.2 it follows that in the primal plane point $(p_1, ..., p'_d)$ represents a hyperplane which is supporting with respect to P. However, for any real number $q_d < p'_d$, $D((p_1, ..., q_d))$ is not a supporting hyperplane for P. Thus, $(p_1, ..., p_d)$ in the primal plane is a hyperplane intersecting P but is not supporting. This leads to a contradiction.
   (c) There exists a point $(p_1, ..., p'_d)$ such that $p'_d < p_d$ and such that $(p_1, ..., p'_d) \in DOWN^-(P)$ but there does not exist a point $(p_1, ..., p''_d)$ such that $p_d < p''_d$ and such that $(p_1, ..., p''_d) \in UP^-(P)$.
   The proof is similar to item (1.c).
2. $(p_1, ..., p_d) \in UP(P) \setminus UP^-(P)$. This means that a point $(p_1, ..., p'_d)$ exists such that $p'_d < p_d$ and such that $(p_1, ..., p'_d) \in UP^-(P)$. Thus, $D((p_1, ..., p'_d))$ is supporting with respect to P. Moreover, by hypothesis, $TOP^P(p_1, ..., p_{d-1}) = p_d$; this means that there does not exist $p''_d < p_d$ such that $D((p_1, ..., p''_d))$ is a supporting hyperplane for P and this leads to a contradiction.
3. $(p_1, ..., p_d) \in DOWN(P) \setminus DOWN^-(P)$.
   The proof is similar to item (2).

4. $(p_1, ..., p_d) \in DOWN^-(P)$. In this case, for the proof of $\Rightarrow$, this means that $BOT^P(p_1, ..., p_{d-1}) = p_d$. Since $(p_1, ..., p_d) \notin UP^-(P)$, this means that either there exists $p'_d > p_d$ such that $(p_1, ..., p'_d) \in UP^-(P)$ or there does not exist any p'_d such that $(p_1, ..., p'_d) \in UP^-(P)$. In the first case, for the proof of $\Rightarrow$, $D((p_1, ..., p'_d))$ is a supporting hyperplane for P; in the second case, for any real number $q_d > p'_d$, $D((p_1, ..., q_d))$ is not a supporting hyperplane for P. In both cases, $(p_1, ..., p_d)$ in the primal plane is a hyperplane intersecting P but is not supporting. This leads to a contradiction. $\square$

Lemma 8.4 *The following facts hold:*

1. *All points contained in $UP(P) \cup DOWN(P)$ represent in the primal plane hyperplanes that do not intersect P or are supporting with respect to P.*

2. *All points not contained in $UP(P) \cup DOWN(P)$ represent in the primal plane hyperplanes that intersect P but are not supporting with respect to P.*

Proof:

1. Suppose that $(p_1, ..., p_d) \in UP^-(P)$.

   In this case, by Lemma 8.3, $p_d = TOP^P(p_1, ..., p_d)$ and therefore hyperplane $D((p_1, ..., p_d))$ is supporting with respect to P.

   Now suppose that $(p_1, ..., p_d) \in UP(P) \setminus UP^-(P)$. In this case, there exists a point $(p_1, ..., p'_d)$ such that $p'_d < p_d$ and $D((p_1, ..., p'_d))$ is supporting with respect to P. This means that $D(p_1, ..., p_d)$ does not intersect P.

   A similar proof holds for points $(p_1, ..., p_d) \in DOWN^-(P)$.

2. Suppose that $(p_1, ..., p_d) \notin UP(P) \cup DOWN(P)$. For the proof of Lemma 8.3, case (1.b), $D((p_1, ..., p_d))$ is not a supporting hyperplane. $\square$

Theorem 8.1 *For all points $(X_1, ..., X_{d-1})$:*

$$TOP^P(X_1, ..., X_{d-1}) = \begin{cases} X_d & \textit{if } (X_1, ..., X_d) \in UP^-(P) \\ +\infty & \textit{otherwise} \end{cases}$$

$$BOT^P(X_1, ..., X_{d-1}) = \begin{cases} X_d & \textit{if } (X_1, ..., X_d) \in DOWN^-(P) \\ -\infty & \textit{otherwise} \end{cases}$$

Proof: We present the proof for TOP^P. A similar proof holds for BOT^P.

$\Leftarrow$ Suppose that $(x_1, ..., x_d) \in UP^-(P)$. In this case the theorem follows from Lemma 8.3.

Now suppose that for all x_d, $(x_1, ..., x_d) \notin UP^-(P)$. By Lemma 8.4, this means that for each value x_d, hyperplane $D((x_1, ..., x_d))$ intersects P but is not supporting with respect to it. Due to the definition of TOP^P, this means that $TOP^P(X_1, ..., X_{d-1}) = +\infty$.

$\Rightarrow$ Suppose that $TOP^P(x_1, ..., x_d) = +\infty$. From Lemma 8.3 it follows that there does not exists x_d such that $(x_1, ..., x_d) \in UP^-(P)$ and this concludes the proof.

Now suppose that $TOP^P(x_1, ..., x_d) = x_d$. In this case, the result follows from Lemma 8.3. $\square$