Large Constraint Joins and Disjoint Decompositions

Alexander Brodsky Victor E. Segal

Department of Information and Software Engineering
George Mason University, Fairfax, VA 22030

March 2001

Abstract

This paper is devoted to the problem of evaluation of a conjunction of N disjunctions of linear constraints in R^d (constraint join), which is a common constraint database operation. Existing methods are exponential in N. Here combinatorial geometry methods are applied to establish conditions for a polynomial size of the join output. A polynomial method to compute a join for an *arbitrary* input is then given. As part of the solution, a new algorithm for convex decomposition of an arbitrary non-convex polyhedron is developed. The H-tree is proposed - a new data structure combining constraint storage and indexing. H-tree and R-tree implementations of the algorithm are given, and analytical considerations regarding the performance are provided.

1 Introduction

To implement a (sub-)query, it is often desirable to use an external efficient algorithm that cannot be matched in terms of performance by the standard evaluation algorithms present in the framework. Algorithms, therefore, play an important role in any extensible database system, and, specifically, constraint algorithms are very important for constraint databases.

In this paper we consider the problem of evaluating a conjunction of N disjunctive constraints $R_1 \wedge \ldots \wedge R_N$, where each R_k is represented by a

disjunction of the form $\vee_i \wedge_j C_{ij}$, and each C_{ij} is a linear arithmetic constraint. Adhering to the constraint database terminology, we say that we want to compute a *constraint join* of N constraint relations. The spatial interpretation of the constraint join is the spatial intersection $R_1 \cap \ldots \cap R_N$.

We consider the case where the space dimension d is bounded, while the number of relations N, and the number of tuples in each relation can grow. Within the CQL framework of [KKR90] this problem is considered for the case when the query size is fixed (i.e. when N is bounded), which guarantees a polynomial CPU evaluation. A similar problem has been examined in spatial databases, where it is referred to as the multiway spatial join. It is interesting that, while the binary (i.e. $N = 2$) spatial join has been studied considerably ([BKS93, Ore86, KS97, BJM93]), the multiway join received much less attention. The work [MP99] considers processing the multiway join by integration of pairwise join algorithms. The work [PM99] exploits similarity between multiway joins and binary constraint satisfaction problems (CSP-s). Both works assume that R-tree indices are available for the input relations. Still, both methods assume that N is small. The situation where N can grow, however, typically arises in more complex applications ([BSCE97]). Consider a situation when a large number of disjunctive constraint objects are processed, and a resulting object is computed using the conjunction (and possibly other operations) over the input objects. Here the overall constrain expression representing the result (the constraint query) is proportional to the number of input objects, and so is allowed to grow. Computing a negation of a disjunction $\neg \vee \wedge C = \wedge \vee \neg C$ is another example that results in an expression containing a conjunction of a large number of disjunctions. The evaluation methods from the CQL framework, as well as the multiway join methods do not guarantee polynomiality in the general case. All of the existing methods compute the result as a union of the N-combinations of tuples (i.e. one tuple from each relation) that have a non-empty conjunction. However, in certain cases the number of all the non-empty N-combinations is exponential in N, which would result in an exponential time evaluation.

The contributions of this paper are as follows. First, we apply methods from combinatorial geometry to estimate the size of the constraint join output. This is done in order to understand when the output is polynomial. We examine specific sets of tuples from the input relations, which we term *bundles*. A bundle is a set of mutually consistent tuples containing at least one tuple from each and every relation. We show that the behavior of bundles determines whether the size of the constraint join output is polynomial or

exponential. Specifically, we formulate a condition that states that, for any bundle, there is a global bound on the number of relations supplying at least *two* tuples into the bundle. We prove that this condition is not only necessary, but also *sufficient* for the polynomial size of the constraint join output (under the assumption that space dimension is limited). Consequently, when the existence of such a bound is known from the application, it follows that the time to compute the join using the existing methods is polynomial. However, testing for the boundness appears to be very expensive, and so, if the boundness is not known from the application, the condition has little practical use as stated. Therefore, we then derive a simpler sufficient condition for a polynomial number of the non-empty N-combinations, which can be tested in a polynomial number of linear problems. Specifically, the condition requires that every constraint relation in the input contain only pairwise disjoint tuples. Next, we use this condition to develop a polynomial constraint join algorithm that works on *arbitrary* input relations. To achieve that, we shift our focus to a problem of decomposing an arbitrary relation into a spatially equivalent one with pairwise disjoint tuples. While convex decompositions in low dimensions have been studied in computational geometry ([CD85]), our algorithm focuses on the general situation when the input is an arbitrary disjunction, i.e. a non-convex, possibly unbounded polyhedron represented by linear constraints in R^d. We also present a new data structure, which we call the *H-tree*, which serves as the main processing mechanism of the algorithm. An H-tree represents a hyperplane-driven space decomposition, and, if coupled with an incremental linear constraint solver, acts like a spatial index which facilitates spatial overlap queries. We give the core version of the decomposition algorithm first, and then present its implementation that utilizes the H-tree-based space indexing. While the experimental verification of the H-tree performance is beyond the scope of this work, we give analytical considerations regarding its behavior. In particular, we explain that the H-tree is expected to perform well when the average number of hyperplanes comprising each individual tuple is small, or when the degree of mutual overlap between the tuples is high. The H-tree is examined in the context of in-memory processing, which is applicable in situations with a large number of small disjunctions. When the relations do not fit into main memory, we also present an implementation of the decomposition algorithm that utilizes the R-tree indexing. Furthermore, we present a heuristic for optimization of both the H-tree and R-tree based decomposition algorithms. Finally, we outline ways to compute the constraint join itself when the input relations

are represented via H-trees or R-trees.

1.1 Definitions

A *constraint tuple* of arity d is a finite conjunction of linear arithmetic constraints $T = C_1 \wedge \ldots \wedge C_K$ over variables $x_1, \ldots, x_d$. The spatial interpretation of a constraint tuple is a convex polyhedron in R^d (not necessarily fully-dimensional).

A *constraint relation* of arity d is a finite set $R = (T_1, \ldots, T_K)$, where each T_i is a constraint tuple of arity d. The logical formula that corresponds to a constraint relation is a DNF formula $T_1 \vee \ldots \vee T_K$. The spatial interpretation of a constraint relation is a union of polyhedra $T_1 \cup \ldots \cup T_K$. [1]

In the sequel we use terms relation and tuple to denote a constraint relation and tuple of fixed arity d over the variables $x_1, \ldots, x_d$.

Given input relations $R_1, \ldots, R_N$, a *constraint join* is a formula $R_1 \wedge \ldots \wedge R_N$ evaluated in the DNF form, i.e. in the form of a relation again. The spatial interpretation of the constraint join is the spatial intersection $R_1 \cap \ldots \cap R_N$.

Our goal is to compute the constraint join, i.e. to produce the output constraint relation. Our manipulations with constraint entities will be driven by the corresponding spatial transformations, which aim at computing the spatial intersection of $R_1 \cap \ldots \cap R_N$. We assume each relation is supplied to us as a collection (disjunction) of tuples, and each tuple is a collection (conjunction) of linear arithmetic inequalities. We assume that the input tuples are closed, i.e. the inequality sign in each input constraint is non-strict. We assume no equalities are present in the input (if there were, we assume they have been replaced by the two opposite non-strict inequalities). Unless specifically noted, we use the same symbols to denote both constraints and the corresponding spatial objects, and we often use the $\wedge$ and $\vee$ symbols to denote the spatial operations ($\cap$ and $\cup$), to emphasize that a constraint representation is used. We occasionally use the words *overlap* and *cross* synonymously with intersection, when it is more natural to say that way.

[1] Unless a misinterpretation arises, we use the same symbol to denote both the constraint object and its corresponding spatial counterpart

1.2 Arrangements of Hyperplanes

Since our approach in drawing conclusions is based on examining arrangements of hyperplanes that comprise the tuples of the input relations, we briefly introduce the basic notation, borrowing from [Ede87].

A hyperplane h is a set of dimension $d-1$ in R^d which can be represented as a set of points $x = (x_1, \ldots x_d)$ satisfying $h(x) = 0$, where $h(x) = a_0 + a_1 x_1 + \ldots + a_d x_d$. Each hyperplane h also defines two half-spaces, $h^+ = \{x | h(x) > 0\}$ and $h^- = \{x | h(x) < 0\}$.

A finite set $\mathcal{H}$ of hyperplanes in R^d defines a dissection of R^d into connected pieces of various dimensions. We call this dissection the *arrangement* of hyperplanes.

For each hyperplane $h_i \in \mathcal{H}$ and for each point $p \in R^d$ we can specify the location of the point relative to hyperplane, namely $l_i(p) = +1$ if $p \in h_i^+$, $l_i(p) = -1$ if $p \in h_i^-$, and $l_i(p) = 0$ if $p \in h_i$. If $l_i(p) = l_i(q)$ for all i we call points p and q equivalent, and so the arrangement $\mathcal{H}$ defines classes of equivalence which we call *faces* of the arrangement. A face is called a *k-face* if its dimension is k. A 0-face is also called a *vertex*, a 1-face is called an *edge*, and a d-face is called a *cell*. The bold lines in Figure 1 show a cell bordered by 3 edges and 3 vertexes.

The key fact that will guarantee us polynomial evaluation comes from the combinatorial geometry (for the proof we refer to [Ede87]):

Fact. The maximum number of all faces created by an arrangement $\mathcal{H}$ of h hyperplanes in R^d is $\theta(h^d)$ [2] (the maximum is taken among all possible arrangements).

A constraint defines a half-space, and so for each hyperplane there are two constraints associated with it. Unless a misunderstanding arises, we will also say that the input tuples are described using the hyperplanes. The constraints for input relations are all in variables $x_1, \ldots, x_d$, and d is assumed to be a constant.

We denote the number of tuples in each R_i be K_i, and $K = \max K_i$. We also denote H_i to be the number of hyperplanes used to describe tuples in R_i. Note, each K_i can be any number in the range from 1 (all hyperplanes describe just one tuple) to H_i (each tuple is a half-space). This is under the assumption that the tuples in the input do not share hyperplanes, which is

[2] following the standard notation for computational bounds $\theta(f(x))$ denotes a function $g(x)$ such that there exist constants C_1, C_2 and x_0 so that $C_1 f(x) \leq g(x) \leq C_2 f(x)$ when $x \geq x_0$

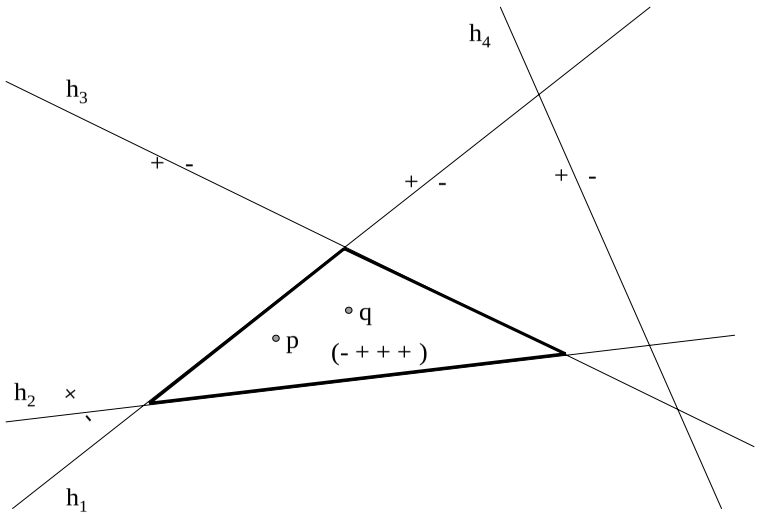

Figure 1: Arrangement of hyperplanes

usually the case. Let $H = \sum_{i=1}^{N} H_i$. Note that $N < H$ under the non-sharing assumption. [3]

We will also need the following obvious fact:

Observation 1. Let $\mathcal{H}$ be the set of hyperplanes describing tuples in all relations. Then $R_1 \wedge \ldots \wedge R_N$ can be described using the hyperplanes from $\mathcal{H}$.

In other words, $R_1 \wedge \ldots \wedge R_N$ can be represented as a union of some faces from the arrangement. In the general case those faces are a mix of faces of some sub-dimension $k \leq d$, since the spatial object $R_1 \wedge \ldots \wedge R_N$ (or some of its parts) is not necessarily fully-dimensional.

[3] Under the assumption that the tuples share hyperplanes, K_i range from 1 to $\theta(H_i^d)$, and N can go up to $\theta(H^d)$ in degenerate cases

2 Towards Polynomial Evaluation

2.1 Bounds on Output Size

In this section we consider $\mathcal{R} = (R_1, \ldots, R_N)$ to be a dynamic collection of constraint relations, i.e. where N is allowed to grow. We denote by I an instance of $\mathcal{R}$, and so I determines some fixed values of N and K, where K is the maximum relation size (among all the relations in I). We will use $\cup$ as a set operation, not as spatial intersection in this section; for example, $\cup_{i=1}^{N} R_i$ denotes the set of all the constraint tuples from $\mathcal{R}$.

Definition. Given an instance I, an *N-combination* of constraint tuples over I is a N-tuple $(t_1, \ldots, t_N)$, where $t_i \in R_i$ for each $i = 1, \ldots, N$.

If no misinterpretation arises, we will also use the term N-combination to denote the conjunction $t_1 \wedge \ldots \wedge t_N$. We call an N-combination *non-empty* if the conjunction is satisfiable (i.e. the corresponding spatial intersection is not empty).

The existing evaluation methods ([KKR90, PM99, MP99]) do not guarantee polynomiality in the general case. Those methods compute the result as a union of all the non-empty N-combinations.

However, situations are possible when the number of all the non-empty N-combinations is exponential in N. Indeed, consider a set of tuples from $\cup_{i=1}^{N} R_i$ such that (1) it contains at least two tuples from each and every relation, and (2) the tuples have a non-empty intersection (see Figure 2). In this case the number of the non-empty N-combinations that can be produced by that set alone is at least 2^N.

An interesting question is what other situations contribute to an exponential number of the non-empty N-combinations. In this section we show that situations similar to the one depicted in Figure 2 are the only ones that give rise to exponentiality in the number of the non-empty N-combinations. We next formally define what we mean by such situations.

Definition. For a given instance I, we call a set of tuples $B \subset \cup_{i=1}^{N} R_i$ a *bundle* over I if (1) it contains at least one tuple from each and every relation from $\mathcal{R}$, and (2) the tuples in B have a non-empty intersection.

For a given bundle B, we will denote by X_B the number of relations that supply at least *two* tuples into B. Let $output(B)$ denote the set of all the non-empty N-combinations produced by B. Obviuosly,

$$2^{X_B} \leq |output(B)| \leq K^{X_B} \tag{1}$$

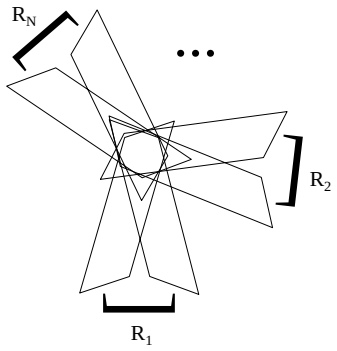

Figure 2: Exponential Number of N-Combinations

For a given I, let $\mathcal{B}(I)$ be the set of all bundles over I. Let

$$X_I = max_{B \in \mathcal{B}(I)} X_B$$

We are now ready to formulate the main result of this section.

Theorem 1. Let I be an instance of $\mathcal{R}$, and let $output(I)$ denote the set of all the non-empty N-combinations produced by I. Then

$$2^{X_I} \leq |output(I)| \leq \theta(H^d) K^{X_I} \tag{2}$$

Proof.

The lower bound is obvious. Indeed, $\exists B_0 \in \mathcal{B}(I)$ such that $X_I = X_{B_0}$ (B_0 is the bundle on which the maximum is achieved). Then $|output(I)| \geq |output(B_0)| \geq 2^{X_{B_0}} = 2^{X_I}$.

The upper bound is the essential point here. First, we will establish the bound on the number of elements of any set of mutually consistent N-combinations. Let $c_1, \ldots, c_Y$ be any set of mutually consistent (and, therefore, non-empty) N-combinations, i.e. $c_1 \wedge \ldots \wedge c_Y \neq \emptyset$. Let $B = \cup_{i=1}^{Y} c_i$

be the set of all tuples from them. Note that (1) B contains at least one tuple from each and every relation, and (2) the tuples from B have a non-empty intersection. It means that B is a bundle, and now (1) implies that $Y \leq K^{X_B} \leq K^{X_I}$.

Now, according to `Observation 1`, any N-combination can be described as a union of some faces from the arrangement $\mathcal{H}$ of hyperplanes from I. Intuitively, if all non-empty N-combinations were pairwise disjoint, their number would be bounded by the number of all the faces $\theta(H^d)$. It is when the N-combinations start to overlap that their number can grow out of the polynomial bounds; to deal with it we will use our bound on the number of overlapping N-combinations. Formally, for each face $f \in \mathcal{F}$ from the set of all faces $\mathcal{F}$ of the arrangement, let C_f be the set of those N-combinations that contain f. Clearly, $output(I) = \cup_{f \in \mathcal{F}} C_f$. Then

$$|output(I)| \leq \sum_{f \in \mathcal{F}} |C_f| \leq \sum_{f \in \mathcal{F}} K^{X_I} \leq \theta(H^d) K^{X_I}$$

Q.E.D.

Corollary 1. A constraint join over $\mathcal{R}$ has a polynomial output size if and only if X_I is globally bounded across all instances I of $\mathcal{R}$.

The corollary follows directly from (2). Note that by the unboundness we mean that X_I grows at least linearly with N here.

2.2 Polynomial Output Size with Arbitrary Relations

`Corollary 1` states that the behavior of bundles determines whether the size of the constraint join is polynomial or exponential. It guarantees polynomiality when the boundness of X_I is known from the application. [4] However, testing for the boundness at each instance (such when the boundness is suspected, and some global constant was chosen for testing) appears to be expensive, and so if the boundness is not known from the application, `Corollary 1` has little practical use as stated. In this subsection we derive a simpler sufficient condition for a polynomial size of the output.

Corollary 2. If the tuples in each R_i are pairwise disjoint, then the size of the constraint join output is polynomial.

[4] note that, since the bounds are themselves exponential in X_I, if $X_I \leq C$ then the real savings in output size should be expected for $N >> C$

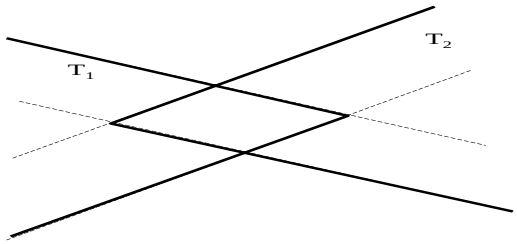

Figure 3: a DPR for two tuples

Proof. Pairwise disjointness implies that $X_I = 0$ for all I, since no disjoint relation can supply more than one tuple into a bundle. *Q.E.D.*

Note that for each instance, disjointness can be tested in a polynomial number of linear problems. However, we pursue a different direction here. Specifically, we would like to use `Corollary 2` to develop a polynomial constraint join algorithm which works on *arbitrary* input relations, i.e. even when X_I is unbounded and the number of the non-empty N-combinations is exponential.

To achieve that, note that we can reduce the general situation to the disjoint case by decomposing each relation R into another relation R', such that $R = R'$ (spatial equivalence), and R' contains pairwise disjoint tuples. We say that R' represents a *disjoint polyhedral representation (DPR)* of R. For example, Figure 3 shows a DPR for two tuples T_1 and T_2, which consists of the faces of the arrangement that are within $T_1 \cup T_2$.

3 Decomposing Relation Into Disjoint Polyhedra

In this section we present a practical DPR algorithm for a decomposition of an arbitrary relation into a spatially equal one containing pairwise disjoint tuples.

For low dimensions, minimal convex decompositions of bounded non-convex polygons were studied in [CD85]. In contrast, the algorithm that we present below (1) focuses on the situation when the input is an arbitrary disjunction, i.e. a multidimensional, non-convex, and possibly unbounded polyhedron represented by linear constraints, and (2) is simpler to implement, compared to the elaborate methods of [CD85]. [5]

3.1 Algorithm Description

Let R be a relation (disjunction) of tuples $T_1, \ldots, T_k$ none of which stands alone (i.e. if we construct a graph whose nodes represent the tuples, and edges represent the overlap, then the graph will be connected).

Figure 3 suggests that we can compute a disjoint partition of R as a union of the faces from the arrangement that are inside R. Since the number of all faces is polynomial, it is possible to do it in polynomial time. The most straightforward way is the naive enumeration, where we iterate over all faces, and then check each face against the relation. [6] However, representing the output as a union of faces is not compact, since some faces can usually be combined into bigger convex polyhedra. Also, examining each and every face of the arrangement is not a good way either, since the actual number of faces that comprise the relation is typically much smaller. In contrast, the algorithm that we give below makes a start from the tuples themselves, and attempts to produce bigger polyhedra in the result.

From now on, given a hyperplane h, we will denote by h^+ the half-space corresponding to the $\leq$ inequality sign, and by h^- the half-space corresponding to the $>$ inequality sign. Note that while the $+$ halfspace is closed, the $-$ one must be open to ensure a disjoint split.

The algorithm works incrementally, processing tuples one by one. After

[5] our algorithm does not produce the minimal number of convex pieces

[6] provided we can iterate over all faces in polynomial time. The H-tree, presented in the following section, can be used directly to achieve that

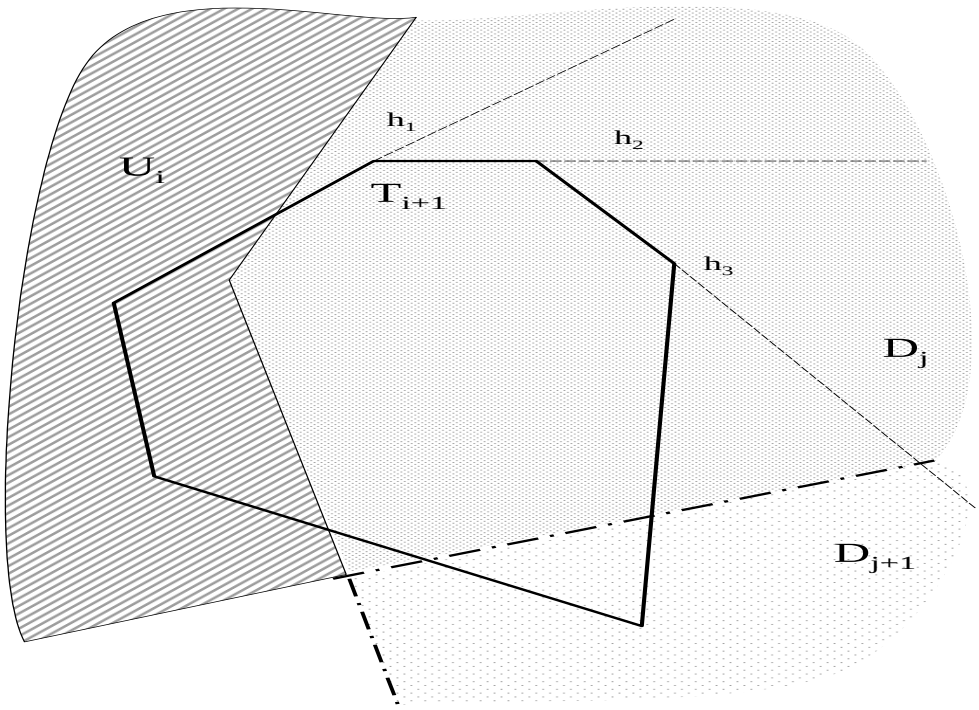

Figure 4: Processing of new tuple

the i-th tuple is processed the algorithm produces a DPR of $U_i = T_1 \cup \ldots \cup T_i$. Consequently, when the last tuple is processed, the resulting DPR of R is obtained, $R = U_k$.

At step $i + 1$, having a DPR of U_i, the algorithm obtains a DPR of $U_{i+1} = U_i \cup T_{i+1}$ in the form of $U_i \cup (T_{i+1} \setminus U_i)$. To do that, the algorithm leaves U_i as is while producing a DPR of $T_{i+1} \setminus U_i$. Note that $T_{i+1} \setminus U_i$ is typically non-convex. Figure 4 shows the current set U_i (the dashed region), and the new tuple T_{i+1} (the solid bold lines). Note that, in contrast to what is shown in the figure, T_{i+1} may not intersect U_i, or U_i may be composed of more than one piece (depending on the order in which the tuples are processed). Our subsequent statements hold true for the both cases (see also section 4).

In turn, to produce a DPR of $T_{i+1} \setminus U_i$, we keep track of the complement of U_i, i.e. $U_i^c = R^d \setminus U_i$. We keep it represented as a union of (convex) poly-

hedra $\mathcal{D} = \cup\, D_j$ (U_i^c itself is typically non-convex). We call them dormant regions, because at this point they are outside the current set U_i, but can be 'awakened' if T_{i+1} intersects them. In Figure 4, D_j (densely dotted) and D_{j+1} (sparsely dotted) are some of the dormant regions.

Then a DPR of $T_{i+1} \setminus U_i$ can be computed as all non-empty intersections of the form $D_j \cap T_{i+1}$, for all j. We also need to update the set of dormant regions $\mathcal{D}$ with respect to T_{i+1}. Let $D_1, \ldots, D_a$ be the dormant regions that intersect T_{i+1}. Then $D_1, \ldots, D_a$ must be deleted from $\mathcal{D}$ since they are not within U_{i+1}^c (they were awakened). At the same time, new dormant regions appear. Let $h_1, \ldots, h_m, \ldots, h_q$ be the hyperplanes that comprise T_{i+1}. To simplify the exposition, we assume that h_m^+ always corresponds to the half-space looking inward the tuple, and h_m^- always corresponds to the halfspace looking outward the tuple. (i.e. the inequality signs in the input tuples are $\leq$; if not, we can multiply by -1). To compute the new dormant regions, we construct q areas $T_{i+1}^{(s)} = (\wedge_{m=1}^{s-1} h_m^+) \wedge h_s^-$, $s = 1, \ldots, q$, which we call *slices*. Note that Figure 4 shows the hyperplanes traversed in the clockwise direction (in 2D), but the conclusions remain the same for any order of hyperplane traversal in R^d. The new dormant regions are then computed as all non-empty intersections of the form $T_{i+1}^{(s)} \cap D_j$, for all $s = 1, \ldots, q$ and $j = 1, \ldots, a$. The core algorithm is listed in Figure 5, where U stands for the current set U_i at each iteration.

`CORE-DPR` is likely to perform significantly better than the naive enumeration for the following reasons. The naive enumeration piles up the hyperplanes from all tuples of the relation into the arrangement, and then checks each face against the relation. `CORE-DPR`, on the other hand, examines each tuple individually. The current tuple T_{i+1} is interposed with the existing set $U_i = T_1 \cup \ldots \cup T_i$, and the portion of the tuple $T_{i+1} \cap U_i$ gets "swallowed" by U_i, and so the hyperplanes that comprise that part of the tuple (and only that part) do not contribute to the arrangement. Moreover, when the rest of hyperplanes of T_{i+1} is then examined, `CORE-DPR` does not produce all faces created by them; rather, the slices are constructed. Each slice is typically a union of some faces, and so the number of slices is much smaller than the number of faces.

3.2 Using H-tree for Decomposition Algorithm

We next introduce the *H-tree*, which we use as the main data structure to implement the algorithm. The purpose of the H-tree is twofold. Note that in

CORE-DPR(Relation: R)
****

1. **foreach** tuple $T \in R$ **do**
2. **foreach** $D \in \mathcal{D}$ **do**
3. **if** $D \cap T$ **then**
4. U.Add $(D \wedge T)$
5. $\mathcal{D}$.Delete (D)
6. **foreach** $s = 1$ **to** q **do**
7. $T^{(s)} = (\wedge_{m=1}^{s-1} h_m^+) \wedge h_s^-$
8. **if** $T^{(s)} \cap D$ **then**
9. $\mathcal{D}$.Add $(T^{(s)} \wedge D)$
10. **end if**
11. **end do**
12. **end if**
13. **end do**
14. **end do**

Figure 5: Procedure CORE-DPR

`CORE-DPR`, U_i gets augmented with all intersections of the form $D \wedge T$, for all D (line 4). If conjunctive constraints are implemented simply as collections of linear constraints, each of the resulting conjunctions $D \wedge T$ will contain identical subsets of constraints from D. The same happens with the new dormant regions $D \wedge T^{(s)}$ (line 9), and each slice $T^{(s)}$ also contains the same constraints as $T^{(s-1)}$, plus one more (line 7). As the number of iterations increases, same constraints become stored multiple times (removing redundant constraints at each step somewhat improves on that; see the R-tree implementation in section 3.5). Consequently, the first purpose of the H-tree is to provide a compact constraint storage structure by significantly reducing the number of constraints that are stored more than once. [7] The second purpose of the H-tree is to provide space indexing capabilities which facilitate spatial overlap queries (see section 3.4).

We explain the structure of the H-tree below. An H-tree is a binary tree and its nodes represent polyhedra. A node of the H-tree has either no

[7] but not completely eliminating them; the H-tree, as presented below, does not present the optimal storage solution, but improves significantly compared to the collection-based implementation

children, or both left and right sons. If a node does have children, they represent a split of the node (polyhedron) performed by a hyperplane h. The two children are marked with the hyperplane that performed the split, plus the sign '+' or '-', to distinguish between the two children polyhedra. The '+' and '-' signs correspond to the two half-spaces defines by the hyperplane. When it is clear from the context, we will also use h^+ to denote the node of the tree corresponding to the the +-side split of the parent node, or, interchangeably, the constraint corresponding to the + halfspace, or, finally, the polyhedron corresponding to that node. When it is necessary to avoid a misunderstanding, we will use $C(h^+)$ or $P(h^+)$ to clarify that the constraint or polyhedron is meant, respectively. The actual constraints describing the polyhedron corresponding to a node are collected by going up the tree from this node towards the root, and collecting all the constraints, i.e. pairs (hyperplane,sign), along the way. The root of the tree represents the entire space R^d.

Both the polyhedra that constitute U_i and the dormant polyhedra are kept as leaves of the H-tree. We mark each leaf of the H-tree either as *dormant*, *finished*, or *active*. A dormant node represents a dormant region which is part of U_i^c. A finished node represents a polyhedron that is a part of U_i. The use of an active node will be explained in a moment.

Procedure `CORE-DPR` now translates to the following. For an incoming tuple, T_{i+1}, we first check T_{i+1} on intersection with all dormant leaves. We awaken those dormant leaves that intersect T_{i+1}, thus marking them active. Next, we compute the actual intersections of T_{i+1} with all the active nodes. Those intersections will become the newly added finished nodes. To do that, we traverse the hyperplanes of T_{i+1}. For each hyperplane h_m we check if it crosses any of the active nodes. If it crosses an active node (a polyhedron) , we span two new nodes corresponding to the split performed by h_m. The child node that corresponds to the side of h_m (i.e halfspace) that looks inward T_{i+1} is marked as active, while the child node that corresponds to the side that looks outward of T_{i+1} is marked as dormant. So h_m^+ becomes active, while h_m^- becomes dormant. We proceed in this manner until all hyperplanes of T_{i+1} are exhausted. In the next section we formally prove that at this point all active leaves represent a DPR of $T_{i+1} \setminus U_i$. We then mark the active leaves as finished, thus making them part of U_i, and then proceed on to the next tuple. Finally, when all tuples are exhausted, all finished leaves represent the final set U_k, which is the resulting DPR of the original relation.

The tree building process for 3 simple tuples is illustrated in Figure 6.

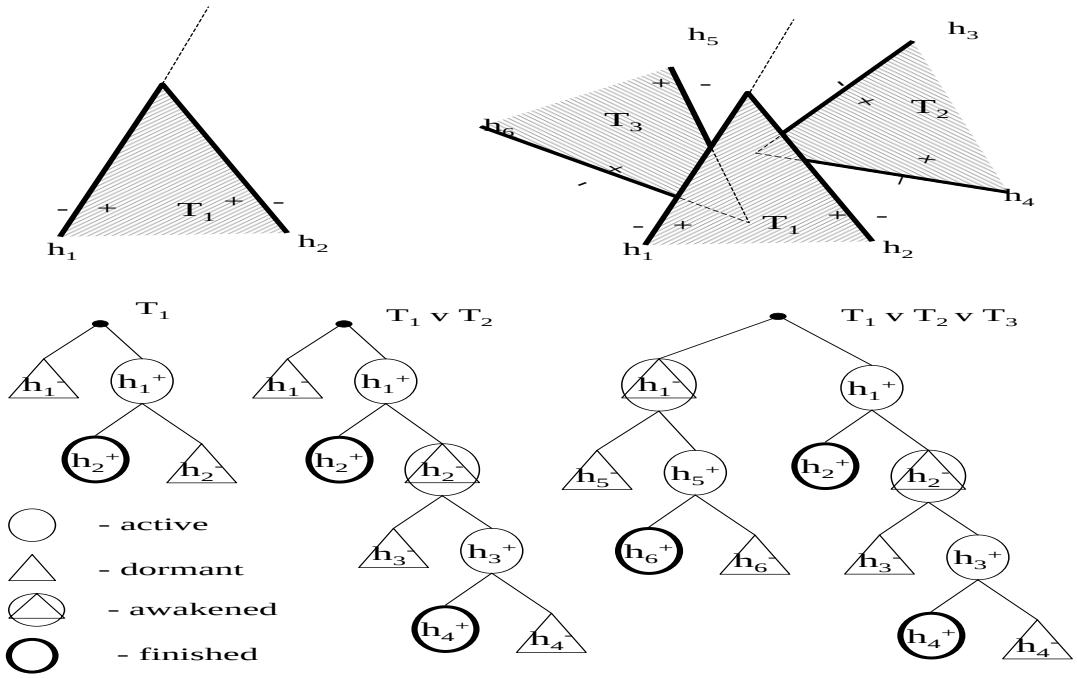

Figure 6: Example of building H-tree

The first tree corresponds to the state when the first tuple has been processed. There are total of 3 regions at this moment: two dormant (h_1^+ and $h_1^- \wedge h_2^-$) and one finished ($h_1^- \wedge h_2^+$) (remember that the $-$ conjuncts correspond to the open boundaries). Consequently, the tree contains the corresponding 3 leaves. Note that the finished leaf represents the tuple itself. The second tree represents $T_1 \cup T_2$, and the last tree represents the final DPR of $T_1 \cup T_2 \cup T_3$ (the bold lines border the dashed areas that correspond to the finished regions).

The pseudo-code of the algorithm is shown in Figure 7. Note that the intersection predicate in lines 4 and 10 tests if the two polyhedra intersect (note, in the second loop this procedure accepts a hyperplane as one of the input polyhedra). The intersection procedure requires the actual constraints for the input leaf, which are collected by going up the tree from this leaf towards the root and collecting all constraints along the way.

Note that, when we collect the constraints for the given leaf, we will

```
H-DPR(Relation: R)
**********
1. H-Tree = root node
2. foreach tuple T ∈ R do // awaken dormant leaves
3.     foreach dormant leaf D do
4.         if D ∩ T then
5.             mark D as active
6.         end if
7.     end do
8.     foreach hyperplane h ∈ T do // traverse hp-s
9.         foreach active leaf L do
10.            if h ∩ L then
11.                span h+ (active) and h− (dormant) from L
12.            end if
13.        and do
14.    end do
15.    foreach active leaf L do //mark active leaves finished
16.        mark L as finished
17.    end do
18.end do
```

Figure 7: Procedure H-DPR

typically have some redundant constraints. For example, in Figure 6 the node h_4^- will be represented by the following conjunction of constraints: $h_4^- \wedge h_3^- \wedge h_2^- \wedge h_1^-$, out of which the last conjunct is redundant. Another typical situation that guarantees to produce a redundant constraint is when the tuple has parallel hyperplanes.

If during the processing of the current tuple there is an awakened leaf that does not get crossed by any hyperplane from the tuple, that means that the leaf lies entirely inside the tuple. Consequently, it gets marked as finished at the end of tuple processing.

We do not give any more implementation details here, since an improved version of this procedure immediately follows after we prove the correctness.

3.3 Correctness of the Decomposition Algorithm

We next prove that the procedure `H-DPR` indeed produces a DPR of the input relation R.

The following lemma is obvious, and we will need it later:

Lemma 2. Given two overlapping polyhedra P and Q, let $q_1, ..., q_k$ be constraints of Q corresponding to those hyperplanes of Q that cross P. Then, $P \wedge Q = P \wedge q_1 \wedge ... \wedge q_k$ (spatial equality)

Theorem 2. The algorithm `H-DPR` is correct, i.e. after the last tuple has been processed, the finished leaves of the H-tree represent a DPR of $T_1 \cup ... \cup T_k$.

Proof.

To prove the theorem, we will validate the following (even more general) proposition:

Proposition. After tuple T_i has been processed,

1. the finished leaves of the tree form a DPR of $U_i = T_1 \cup ... \cup T_i$, and
2. the dormant leaves form a DPR of $R^d \setminus U_i$.

We will validate the proposition by induction on the number of tuples.

Base. There are no tuples. Then there are no finished leaves, but there is just the root of the tree which is one and only dormant leaf representing R^d.

Induction step. We assume the finished leaves represent a DPR of U_i, and the dormant leaves represent a DPR of $R^d \setminus U_i$. To validate the induction step we need to show that when we add tuple T_{i+1}, in the end of its processing

I1: all the finished leaves will represent a DPR of U_{i+1}

I2: all the dormant leaves will represent a DPR of $R^d \setminus U_{i+1}$.

In turn, to show that **I1** holds, we will prove the following statement:

Statement A. The newly added finished leaves, i.e the finished leaves that are added to the tree after T_{i+1} has been processed, will form a DPR of $T_{i+1} \setminus U_i$.

Indeed, if we show that **A** holds, **I1** will follow immediately from the following fact:

$$U_{i+1} = U_i \cup (T_{i+1} \setminus U_i)$$

. We will also get the proof of **I2** while we will be validating **A**.

Let $D_1, \ldots, D_j, \ldots$ be the dormant nodes of the current tree. We will need the following statement:

Statement B. If D_j is a dormant node that was awakened by T_{i+1}, then:

B1: the finished leaf of the subtree that grows out of D_j (as the result of processing of the current tuple T_{i+1}) represents $T_{i+1} \cap D_j$ (note there is only one finished leaf in this subtree)

B2: the dormant nodes of that subtree represent a DPR of $D_j \setminus T_{i+1}$.

Then if **B** holds, **B1** will imply that all the newly added finished leaves for the current tuple will represent $\cup_j (T_{i+1} \cap D_j)$. Recall now that according to the induction hypothesis all D_j-s form a DPR of $R^d \setminus U_i$. Then

$$T_{i+1} \setminus U_i = (R^d \setminus U_i) \cap T_{i+1} = (\cup_j D_j) \cap T_{i+1} = \cup_j (T_{i+1} \cap D_j)$$

which proves **A**. Also, since by the induction hypothesis the existing dormant leaves represent $R^d \setminus U_i$, **B2** will immediately imply **I2**.

So, all is left is to prove **B1** and **B2**. For that, lets focus our attention on one awakened dormant leaf D_j. Let $h_1, \ldots, h_q$ be the hyperplanes that participate in building the subtree growing from D_j as we process T_{i+1} (see Figure 4) When we get h_1, it splits D_j into two pieces, which amounts to spanning two child nodes from D_j. Then leaf h_1^- becomes dormant, while leaf h_1^+ remains active. Note that the two leaves are disjoint due to the way we assign inequality signs. Clearly, the active leaf h_1^+ represents the polyhedra $P(h_1^+) = D_j \wedge C(h_1^+)$, and the dormant leaf h_1^- represents the polyhedra

$P(h_1^-) = D \wedge C(h_1^-)$. We then proceed in the same manner taking leaf h_1^+ as the new root until the tree stops growing, i.e. all $h_1, \ldots h_q$ have been processed. Figure 4 shows the hyperplanes traversed in the clockwise direction (in 2D), but the conclusions remain the same for any order of hyperplane traversal in R^d. At this moment the last active leaf(which becomes the finished leaf) represent $P(h_q^+) = D_j \wedge \cap_j C(h_j^+)$. It follows from Lemma 2 that $P(h_q^+) = T_{i+1} \wedge D_j$, which completes the proof of **B1**. Also all the new dormant leaves correspond to the disjuncts in the following expression:

$$(D_j \wedge C(h_1^-)) \vee (D_j \wedge C(h_1^+) \wedge C(h_2^-)) \vee \ldots \vee (D_j \wedge C(h_1^+) \wedge \ldots \wedge C(h_{q-1}^+) \wedge C(h_q^-))$$

. Note now that the disjuncts of the last expression are (1) disjoint (since the + signs are not strict, while the − ones are) (2) convex (3) outside of $P(h_q^+) = T_{i+1} \wedge D_j$ (4) inside D_j, since D_j is the root of the subtree. It means that this expression represents a DPR of $D_j \setminus T_{i+1}$, which completes the proof of **B2**. Note that the formal proof of both **B1** and **B2** is again by (nested) induction on q, but we omit the formal induction steps here. *Q.E.D.*

3.4 Using H-Tree Indexing

An H-tree presents a hyperplane-driven space decomposition of R^d. Consequently, it can serve as an index facilitating retrieval of polyhedral regions which overlap a given one. The corresponding procedure `H-SEARCH` is given in Figure 8. For each node it checks if it overlaps with the input polyhedra. If the answer is positive, `H-SEARCH` calls itself recursively for both the '+' and '-' sons. When a leaf is encountered, it is output into the result.

A very important feature of the procedure is its use of an *incremental constraint solver* ([MS98]). An incremental solver is able to determine the satisfaction of a set of constraints incrementally, as the constraints are fed to it one by one. It does so by maintaining an implicit constraint store where the current constraints are stored in some partially solved form. Incremental constraint solvers have been studied in the CLP framework ([MS98]), as they are essential for CLP top-down evaluation.

To illustrate the benefits of using an incremental solver, consider the final H-tree depicted in Figure 6, and suppose the finished node h_4^+ is the only node that overlaps the input. If we used a regular simplex solver, we would solve 9 linear problems before getting to the node. Each problem would involve the same set of constraints as the previous one, plus one more, thus

introducing a redundancy. In this case a simple linear scan of the leaf nodes would be even superior to the tree search.

An incremental real arithmetic solver is described in [MS98], and can be used here. In our procedure the variable `ISolv` represents the solver. Note also that, just like in CLP, when processing of the subtree corresponding to a son of the current node is done, the solver must be restored to its previous state before processing of the other son begins. Consequently, we assume the following methods are available from the solver: `Save_Store` remembers the current store state by putting a checkmark, and `Restore_Store` restores the store to the state before the checkmark. The methods usually take use of a stack, and their efficient implementation is also studied in the CLP context ([MS98]).

Note also that the implementation shown in Figure 8 uses the variable `Result` (of type `DC_LIN`) to hold the result of the search. In practice, a pipe-line solution should be used instead. A pipe-line solution can be viewed as a CCUBE monoid, and should be implemented as a co-procedure which uses a stack to emulate recursion.

`H-SEARCH (Ht,CLIN)` procedure can now be used to facilitate `H-DPR`. We replace the first loop with procedure `H-ACTIVATE (Ht,CLIN)`, the second loop with `H-SEARCH-ACTIVE (Ht,hp)`, and the third loop with `H-DEACTIVATE (Ht)`. `H-ACTIVATE (Ht,CLIN)` marks the leaves that intersect the input area as active, `H-SEARCH-ACTIVE (Ht,hp)` performs search of the active nodes that cross the input hyperplane and outputs them in a pipe-lined mode, and `H-DEACTIVATE (Ht)` marks the active nodes as finished. All three procedures perform searches of the tree, and so they are accordingly modified versions of `H-SEARCH (Ht,CLIN)`. Specifically, `H-ACTIVATE` has the following modifications: (1) it ignores the finished leaves (2) it marks the leaves that intersect `CLIN` as active, and doesn't produce any output. (3) it specially flags all those non-leaf nodes that belong to the paths that terminate at active nodes. The latter flagging is done so that `H-SEARCH-ACTIVE (Ht,hp)` would later search only along such paths, thus avoiding unnecessary traversals of the whole tree. To implement the flagging, the recursive part of `H-ACTIVATE` can be made to return TRUE if the current node belongs to such a path, i.e. if at least one of its recursive calls for the sub-trees returns TRUE. Consequently, `H-SEARCH-ACTIVE (Ht,hp)` has the following modification: it only searches along the flagged nodes. Finally, `H-DEACTIVATE (Ht)` traverses the tree along the flagged nodes, un-flags them, and marks the leaves as finished.

The final procedure is called `H-DPR+`, and its pseudo-code is in Figure 9.

```
H-SEARCH (Htree: Ht, C_LIN: area)
*********************************
1. ISolve.Initialize-Store
2. ISolve.Add (area)
3. H-SEARCH-RECURSIVE(Ht.Root)
4. return Result

H-SEARCH-RECURSIVE (HtreeNode: Node)
************************************
1. ISolve.Add (Node.constraint)
2. if not ISolve.Sat then return
3. else
4.       if Node.isLeaf then
5.            Result.Insert (Node)
6.            return
7.       else
8.            ISolve.Save-Store //right son
9.            H-SEARCH-RECURSIVE (Node.Right)
10.           ISolve.Restore-Store
11.           ISolve.Save-Store //left son
12.           H-SEARCH-RECURSIVE (Node.Left)
13.           ISolve.Restore-Store
14.      end if
15.end if
```

Figure 8: Procedure H-SEARCH

H-DPR+ (Relation: R)
****

1. H-Tree = root node
2. **foreach** tuple $T \in R$ **do**
3. H-ACTIVATE (Htree, T) // awaken dormant leaves
4. **foreach** hyperplane $h \in T$ **do** // traverse hp-s
5. **foreach** active leaf L from H-SEARCH-ACTIVE(Htree, h) **do**
6. span h^+ (active) and h^- (dormant) from L
7. **end do**
8. **end do**
9. H-DEACTIVATE (Htree) // mark active leaves finished
10. **end do**

Figure 9: Procedure H-DPR+

While an experimental evaluation of `H-DPR+`'s performance is beyond the scope of this work, we give some analytical considerations regarding its performance here. Note that during processing of the current tuple T_{i+1} the H-tree grows as follows: each active leaf becomes the root for a linear subtree, and the number of nodes in the subtree is equal to the number of hyperplanes of T_{i+1} that intersect that active leaf. Consequently, if the average number of hyperplanes comprising each individual tuple of R is relatively small, it will result in a smaller H-tree, smaller number of the finished leaves, and therefore, faster H-tree searches.

Also, we say that the degree of mutual overlap between $T_1, \ldots, T_k$ is high if, intuitively, the tuples are crowded all over each other (we do not define it formally here; one possible definition is the ratio of the volume of $\cap T_i$ to the volume of $\cup T_i$). In this case, a large portion of T_{i+1} gets "swallowed" by U_i. Consequently, less hyperplanes of T_{i+1} will participate in further tree expansion (and further space decomposition), and so searches over the H-tree will be faster as well.

3.5 Using R-tree Indexing

In the previous section we explained that the H-tree is expected to perform better when the average number of hyperplanes comprising each individual tuple is small, and/or when the degree of mutual overlap between the tuples

is high. When none of these holds, employing R-tree indexing could be a more efficient solution. Besides, the H-tree has been examined in the context of in- memory processing only, such as when we have a large number of small disjunctions. The R-tree, on the other hand, provides good I/O performance, which is something the H-tree is not yet capable of doing due to the open questions of how to pack its nodes and keep it balanced.

Consequently, we next give an implementation of the core decomposition algorithm utilizing R-trees. Procedure `R-DPR` is shown in Figure 10 and its code follows closely that of procedure `CORE-DPR`. We use two R-trees: `FRt` is maintained over the current set U_i, and `DRt` is maintained over the set of dormant regions $\mathcal{D}$. Consequently, the operations of insertion and deletion over both U_i and $\mathcal{D}$ translate to the corresponding operations over the R-trees. All the variables that hold the spatial objects are of CCUBE type `C_LIN`. We also use method `Get_MBR`, which computes the MBR of the object by solving $2d$ linear problems.

Apart from the differences in how `H-DPR+` and `R-DPR` index both the dormant and the finished regions, the two procedures present two aspects of the following trade-off: in `H-DPR` we choose to keep redundant constraints at the expense of having them in the linear problems (using an incremental constraint solver somewhat compensates for the redundancy); in `R-DPR`, on the other hand, the redundancy can be easily eliminated ([LHM89]) since the conjunctions are implemented via collections (calls to a redundancy elimination procedure should be inserted after lines 6 and 13).

Interestingly, a third solution is possible, which is a combination of the previous two. The H-tree is used for storing constraints, just like in `H-DPR`, while the two R-trees of `R-DPR` are built over the leaves of the H-tree. Thus, we can combine a compact constraint storage with an I/O efficient R-tree indexing.

3.6 Improving Space Decomposition

For a given relation the H-tree is not unique and depends on the order in which tuples are inserted, as well as on the order in which hyperplanes are read from each tuple. Consequently, different H-trees present different space decompositions. In this subsection we outline one heuristic that is expected to produce H-trees with better space decompositions, i.e. decompositions that result in faster H-tree searches. The heuristic deals with the order in which the hyperplanes are read from tuples, and it works for *bounded* tuples

```
R-DPR(Relation: R)
**********************
1. DRt.Insert (R^d)
2. foreach tuple T ∈ R do
3.   T_mbr = T.Get_MBR
4.   foreach D from DRt.Search(T_mbr) do
5.       if D ∩ T then
6.           F = D ∧ T
7.           F_mbr = F.Get_MBR
8.           FRt.Insert(F_mbr, F)
9.           DRt.Delete (D)
10.          foreach s = 1 to q do
11.              T^(s) = (∧_{m=1}^{s-1} h_m^+) ∧ h_s^-
12.              if T^(s) ∩ D then
13.                  D^(s) = T^(s) ∧ D
14.                  D_mbr^(s) = D^(s).Get_MBR
15.                  DRt.Insert (D^(s), D_mbr^(s))
16.              end if
17.          end do
18.      end if
19.  end do
20.end do
```

Figure 10: Procedure R-DPR

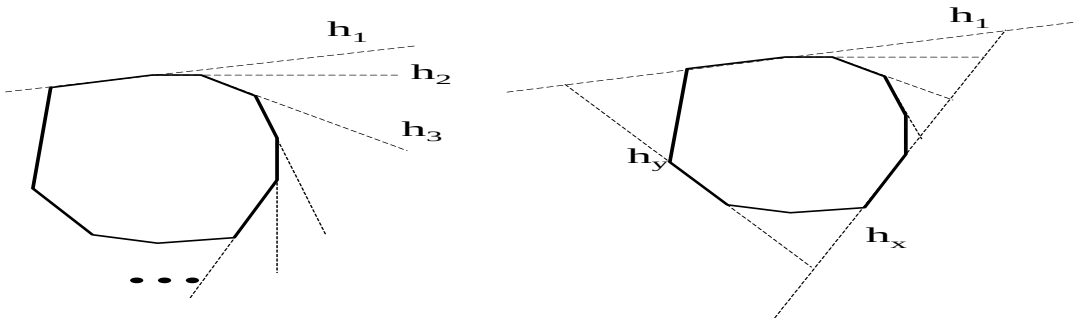

Figure 11: Order of Hyperplanes

only.

Consider a tuple which is described using a large number of hyperplanes, and suppose during the insertion of that tuple into H-tree the hyperplanes were taken in the clockwise direction $h_1, h_2, h_3, \ldots$ (see Figure 11(a)). One can see that this situation will produce a space decomposition consisting of a large number of 'narrow' chunks each of which spans across the space (unbounded). Consequently, each tuple that follows is more likely to intersect a larger number of the chunks, thus being dissected into a larger number of finer pieces and fueling faster growth of the H-tree.

In contrast, suppose the hyperplanes were read in the following order: $h_1, h_x, h_y, \ldots$ (the rest in any order) (see Figure 11(b)). One can see that in this case the decomposition consists of a smaller number of larger chunks which span across the space, and a large number of smaller chunks which are spatially bounded. Consequently, for a tuple that follows, the likelihood that it will lie inside one of the bigger chunks is greater, and so is the likelihood that the tuple will 'swallow' some of the smaller chunks entirely. In other

words, the tuple is expected to be dissected in a lesser extent than in the previous situation, and the tree is expected to grow slower as well. This is expected to speed-up the H-tree search procedure.

Intuitively, in the second situation the hyperplanes were selected in a 'balanced' way - each next hyperplane was chosen in attempt to maximize the distance from the corresponding face to any other face with already selected hyperplane. Therefore, the following heuristic is suggested: enclose the tuple into its MBR and then select 'opposite' hyperplanes for each dimension. Specifically, for tuple T in variables $x_1, \ldots, x_d$ choose variable x_1, then minimize x_1 subject to T obtaining the corresponding solution point x_1^{min} on T, and then select a hyperplane of T that intersects x_1^{min}. Note that intersection of a hyperplane with point x_1^{min} is tested by simply checking if the substitution of the points' coordinates for the variables in the hyperplane expression zeroes the result. After that, maximize x_1 subject to T, and, again, select a hyperplane that intersects the solution. This way the second hyperplane will be 'opposite' to the first one with respect to the chosen dimension. Proceed in the same manner with the second dimension (i.e. x_2), until all d dimensions are exhausted. After that, select the remaining hyperplanes in any order. Note that the heuristic assumes that the tuple is bounded. Figure 12 lists the pseudo-code of the procedure `H-REORDER` which accepts a tuple (represented as a collection of hyperplanes) and outputs the same tuple with the hyperplanes re-ordered according to the heuristics. `H-REORDER` should be called for each tuple iteration in `H-DPR+` or `R-DPR`.

4 Computing Constraint Join

Note that, to avoid dealing with stand-alone tuples in the decomposition algorithm, we assumed that all the tuples of the relation form a connected graph with respect to overlap. However, it is easy to observe that the assumption is not actually used, and the algorithm can accept stand-alone tuples as well. Those tuples will be processed exactly the same way as the overlapping ones, and integrated within the resulting H-tree (or R-tree, in `R-DPR`). Consequently, we can assume that both `H-DPR+` and `R-DPR` take an arbitrary relation in their input. Of course, it only makes sense to do so when there are only a few stand-alone tuples. Assuming this is the case, we can perform `H-DPR+` over each and every relation, so that each output relation becomes represented in the form of an H-tree. We can now imme-

```
H-REORDER-HP (T) //***T is bounded
*****************
1. T_new = new tuple
2. for i = 1 to d do
3.       x_min = MIN-POINT (x_i, T)
4.       x_max = MAX-POINT (x_i, T)
5.       foreach h ∈ T do
6.            if h(x_min) = 0 or h(x_max) = 0 then
7.                 T_new.Insert(h)
8.                 T.Delete(h)
9.            end if
10.      end do
11.end do
12.T_new.Insert (the rest of hyperplanes from T)
13.return T_new
```

Figure 12: Procedure H-REORDER-HP

diately use the H-trees to compute the join. The most straightforward way is a pairwise join integration using the left-deep plan. At each step, having computed $R_1 \wedge \ldots \wedge R_i$, we join it with R_{i+1}, and eliminate inconsistent pairs of tuples. An indexed nested loop can be used, where the inner loop (over R_{i+1}) is replaced to a call to `H-SEARCH-FINISHED`. `H-SEARCH-FINISHED` works the same way as `H-SEARCH` (section 3.4), but has the following modifications: (1) it is only interested in the finished leaves (2) it treats each linear tree segment as one big node. To clarify the last point, recall that with each new tuple an H-tree grows by means of linear segments growing out of the active leaves. After the entire relation is processed, some of the segments still remain linear, if none of their nodes became the root to a subtree. When `H-SEARCH-FINISHED` moves along a linear segment, no branching occurs, and, therefore, it does not make sense to check the satisfaction of the constraint store at each node; rather, `H-SEARCH-FINISHED` should advance along the segment, collect the constraints, and resume satisfaction checks at the end of the segment. Further CPU and I/O optimization of the H-tree-based constraint join is beyond the scope of this work. Approaches similar to those used to boost the R-tree-based multiway join ([MP99, PM99]) may be considered in the future, including an examination of different pairwise

integration plans, or a synchronous traversal of the H-trees.

When the R-tree is a better candidate to perform the decomposition (section 3.5), we can use `R-DPR` to decompose each relation, and so each output relation becomes indexed by an R-tree. Next, the existing techniques for the multiway spatial join ([MP99, PM99]) can be applied to compute the constraint join. Note however, that the above works consider spatial join queries represented using a graph whose nodes are relations, and edges are *binary* predicates between relations (the overlap, etc.), and concentrate on specific cases of query graphs, such as clique and acyclic graphs. The entire intersection, on the other hand, is an N-ary predicate. One direct way to overcome this discrepancy is to employ the algorithms of [MP99, PM99] over the clique graph, and to augment the code with an actual intersection test for each found N-combination. With this modification, the pairwise join integration of [MP99] can be applied directly. The CSP-based synchronous traversal of R-trees of [PM99] (which is superior to the pairwise integration) still requires more changes, since it relies on the assumption that N is small and stores certain data in main memory ($O(N^2)$ space is required).

Finally, if the input relations consist mostly of stand-alone tuples, performing `H-DPR+` or `R-DPR` over the entire relations may not be efficient. In this situation, we can split each relation into groups of connected tuples (connected sub-graphs), perform `H-DPR+` or `R-DPR` within each group separately, and then re-index the entire relation with a new R-tree.

5 Conclusions

we showed how to compute a constraint join of N constraint relations polynomially in N. We contributed to a better understanding of the nature of the constraint join by applying results from combinatorial geometry to estimate the size of the constraint join output. As one of the consequences, we proved that, under the assumption that the space dimension is bounded, the output is polynomial if the input relations contain pairwise disjoint tuples. Since relations in existing spatial databases typically satisfy this condition (e.g. cities, forests, rivers), we can conclude that the multiway spatial join is typically polynomial. We also examined the problem of the convex decomposition of an arbitrary d-dimensional polyhedron represented by a disjunction of linear constraints. Beside its use for a polynomial evaluation of the constraint join, the convex decomposition is a separate problem by itself; for example,

it has another direct application of computing the volume of a non-convex bounded polyhedron, since the formulae for the volume of a convex polyhedron are known. As part of the decomposition algorithm, we also introduced the H-tree - a new data structure which combines compact constraint storage with spatial indexing capabilities, and we used the H-tree representation of constraint relations to compute the constraint join.

We presented analytical considerations regarding the performance of the proposed techniques. However, as it is the case with most spatial algorithms, average estimates are very hard to determine precisely, while the worst-case estimates reflect very uncommon situations. Consequently, the most important future research track is to identify a range of applications that would provide a test bed for the proposed techniques, and to perform an implementation to validate the analytical conclusions.

Future work directions for improving the performance of the constraint join were mentioned in section 4. As far as the H-tree itself is concerned, the following questions remain open. First, other ways of improving the space decomposition could be identifiable, for instance, the possibility to union some of the finished leaves into even larger convex pieces. Secondly, the leaves of an H-tree contain redundant constraints (hyperplanes), and so efficient ways of dealing with the redundancy may be considered. Also, the important questions of how to keep the H-tree balanced, as well as how to improve its I/O performance, are open.

References

[BJM93] A. Brodsky, J. Jaffar, and M.J. Maher. Toward practical query evaluation in constraint databases. *CONSTRAINTS, Am International J., to appear. Preliminary version appeared in Proc. 19th International Conference on Very Large Data Bases (VLDB) 1993, Dublin.*, 1993.

[BKS93] T. Brinkhoff, H. Kriegel, and B. Seeger. Efficient processing of spatial joins using r-trees. *ACM Sigmod*, 1993.

[BSCE97] A. Brodsky, V. Segal, J. Chen, and P. Exarkhopoulo. The ccube constraint object-oriented database system. *CONSTRAINTS, An International J., to appear.*, 1997.

[CD85] B. Chazelle and D. Dobkin. Optimal convex decompositions. In G. Toussaint, editor, *Computational Geometry*, pages 63–135, Amsterdam, 1985. North-Holland.

[Ede87] H. Edelsbrunner. *Algorithms in Combinatorial Geometry.* Springer-Verlag, 1987.

[KKR90] P. Kanellakis, G. Kuper, and P. Revesz. Constraint query languages. *J. Computer and System Sciences, to appear. (A preliminary version appeared in* Proc. 9th PODS, pages 299–313, 1990.

[KS97] N. Koudas and K. Sevcik. Size separation spatial join. *Proc. ACM Sigmod*, 1997.

[LHM89] J-L. Lassez, T. Huynh, and K. McAloon. Simplification and elimination of redundant linear arithmetic constraints. In *Proc. North American Conference on Logic Programming*, pages 35–51, Cleveland, 1989.

[MP99] N. Mamoulis and D. Papadias. Integration of spatial join algorithms for processing multiple inputs. In *Proc. ACM SIGMOD*, Philadelphia, PA, 1999. ACM.

[MS98] K. Marriott and P. Stuckey. *Programming with Constraints: An Introduction.* The MIT Press, 1998.

[Ore86] J. Orenstein. Spatial query processing in an object-oriented database system. *Proc. ACM Sigmod*, 1986.

[PM99] D. Papadias and N. Mamoulis. Processing and optimization of multiway spatial join using r-trees. In *Proc. 18th PODS*, Philadelphia, PA, 1999. ACM.