# Efficient Searching with Linear Constraints*

Pankaj K. Agarwal† Lars Arge‡ Jeff Erickson§

Paolo G. Franciosa¶ Jeffrey Scott Vitter‖

**Abstract**

We show how to preprocess a set $S$ of points in $\mathbb{R}^d$ into an external memory data structure that efficiently supports *linear-constraint* queries. Each query is in the form of a linear constraint $x_d \leq a_0 + \sum_{i=1}^{d-1} a_i x_i$; the data structure must report all the points of $S$ that satisfy the constraint. This problem is called *halfspace range searching* in the computational geometry literature. Our goal is to minimize the number of disk blocks required to store the data structure and the number of disk accesses (I/Os) required to answer a query. For $d = 2$, we present the first data structure that uses linear space and answers linear-constraint queries using an optimal number of I/Os in the worst case. For $d = 3$, we present a near-linear size data structure that answers queries using an optimal number of I/Os on the average. We present linear-size data structures that can answer $d$-dimensional linear-constraint queries (and even more general $d$-dimensional simplex queries) efficiently in the worst case. For the $d = 3$ case, we also show how to obtain tradeoffs between space and query time.

---

*An extended abstract of this paper appeared in Proceedings of the 15th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems [1].

†Center for Geometric Computing, Department of Computer Science, Duke University, Box 90129, Durham, NC 27708–0129; pankaj@cs.duke.edu; http://www.cs.duke.edu/~pankaj. Supported in part by National Science Foundation research grants EIA–9870724, and CCR–9732787, by Army Research Office MURI grant DAAH04–96–1–0013, by a Sloan fellowship, by a National Science Foundation NYI award and matching funds from Xerox Corporation, and by a grant from the U.S.-Israeli Binational Science Foundation.

‡Center for Geometric Computing, Department of Computer Science, Duke University, Box 90129, Durham, NC 27708–0129; large@cs.duke.edu; http://www.cs.duke.edu/~large. Supported in part by U.S. Army Research Office MURI grant DAAH04–96–1–0013 and by National Science Foundation research grant ESS EIA–9870724.

§Center for Geometric Computing, Department of Computer Science, Duke University, and Department of Computer Science, University of Illinois, Urbana, IL 61801; jeffe@cs.uiuc.edu; http://www.uiuc.edu/~jeffe. Supported in part by National Science Foundation grant DMS–9627683 and by U.S. Army Research Office MURI grant DAAH04–96–1–0013.

¶Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza," Via Salaria 113, 00198, Rome; pgf@dis.uniroma1.it; http://www.dis.uniroma1.it/~pgf. Supported in part by EU Project no. 20244 (Alcom-IT). Part of this work was done while visiting Duke University under the Short Term Mobility program of CNR.

‖Center for Geometric Computing, Department of Computer Science, Duke University, Box 90129, Durham, NC 27708–0129; jsv@cs.duke.edu; http://www.cs.duke.edu/~jsv. Supported in part by U.S. Army Research Office MURI grant DAAH04–96–1–0013 and by National Science Foundation grants CCR–9522047 and CCR–9732787. Part of this work was done while on sabbatical at INRIA in Sophia Antipolis, France.

# 1 Introduction

In order to be successful, any data model in a large database requires efficient external memory (secondary storage) support for its language features. Range searching and its variants are problems that often need to be solved efficiently. In relational database systems and in SQL, for example, one-dimensional range searching is a commonly used operation [34, 44]. A number of special cases of two-dimensional range searching are important for the support of new language features, such as constraint query languages [34] and class hierarchies in object-oriented databases [34]. In spatial databases such as geographic information systems (GIS), range searching obviously plays an extremely important role, and a large number of external data structures for answering such queries have been developed (see, for example, [42, 46]). While most attention has been focused on *isothetic* or *orthogonal* range searching, in which a query is a $d$-dimensional axis-aligned hyper-rectangle, the importance of non-isothetic queries has also been recognized, most recently in [20, 26]. Many of the proposed data structures can be used to answer non-isothetic queries.

In this paper we develop efficient data structures for what in the computational geometry community is called *halfspace range searching*, where a query is a *linear constraint* of the form $x_d \leq a_0 + \sum_{i=1}^{d-1} a_i x_i$ and we wish to report all input points that satisfy this constraint. Our goals are to minimize the number of disk blocks required to store the data structure and to minimize the number of disk accesses required to answer a query. Halfspace range searching is the simplest form of non-isothetic range searching and the basic primitive for more complex queries.

## 1.1 Problem statement

The *halfspace range searching* problem is defined as follows:

> Preprocess a set $S$ of $N$ points in $\mathbb{R}^d$ into a data structure so that all points satisfying a query constraint $x_d \leq a_0 + \sum_{i=1}^{d-1} a_i x_i$ can be reported efficiently.

A halfspace range query corresponds to reporting all points below a query hyperplane $h$ defined by $x_d = a_0 + \sum_{i=1}^{d-1} a_i x_i$. An example of a halfspace range query is the following [26]: Given a relation

$$\mathit{Companies}(\mathit{Name}, \mathit{PricePerShare}, \mathit{EarningsPerShare}),$$

retrieve the names of all companies whose price/earnings ratio is less than 10. In SQL the query can be expressed as follows:

SELECT *Name* FROM *Companies*
WHERE (*PricePerShare* $-$ 10 $*$ *EarningsPerShare* $<$ 0)

If we interpret each ordered pair ($\mathit{EarningsPerShare}, \mathit{PricePerShare}$) as a point in the plane, the result of the query consists of all such points that satisfy the linear constraint line $y - 10x \leq 0$. Several complex queries can be viewed as reporting all points lying within a given convex query region. Such queries can in turn be viewed as the intersection of a number of halfspace range queries.

As our main interest is minimizing the number of disk blocks used to store the points and the number of disk accesses needed to answer a halfspace range query, we will consider the problem in the standard external memory model. This model assumes that each disk access transmits a contiguous block of $B$ units of data in a single *input/output operation* (or just *I/O*). The efficiency of a data structure is measured in terms of the amount of disk space it uses (measured in units of disk blocks) and the number of I/Os required to answer a halfspace range query. As we are interested in solutions that are *output sensitive*, our query I/O bounds are not only expressed in terms of $N$, the number of points in $S$, but also in terms of $T$, the number of points reported by the query. Note that the minimum number of disk blocks we need to store $N$ points is $\lceil N/B \rceil$. Similarly, at least $\lceil T/B \rceil$ I/Os are needed to report $T$ output points. We refer to these bounds as "linear" and introduce the notation $n = \lceil N/B \rceil$ and $t = \lceil T/B \rceil$.

## 1.2 Previous results

The computational geometry community has made tremendous progress on non-isothetic range searching in recent years; see the surveys by Agarwal and Erickson [3] and Matoušek [38] and the references therein. As mentioned, halfspace range searching is the simplest form of non-isothetic range searching, and the problem has been extensively studied. Unfortunately, all the results are obtained in the internal memory models of computation, where I/O efficiency is not considered.

The practical need for I/O support has led to the development of a large number of external data structures in the spatial database community. B-trees and their variants [8, 17] have been an unqualified success in supporting one-dimensional range queries. B-trees occupy $O(n)$ space and answer queries in $O(\log_B n + t)$ I/Os, which is optimal. Numerous structures have been proposed for range searching in two and higher dimensions, for example, grid files [41], quad-trees [46, 47], k-d-B-trees and variants [45, 31], hB-trees [23, 35], and R-trees and variants [9, 10, 29, 33, 48]. (More references can be found in the surveys [3, 28, 32, 42].) Although these data structures have good average-case query performance for common geometric searching problems, their worst-case query performance is much worse than the $O(\log_B n + t)$ I/O bound obtained in one dimension using B-trees. One key reason for this discrepancy is the important practical restriction that the structures must use near-linear space. Recently, some progress has been made on the construction of structures with provably good performance for two-dimensional [5, 6, 34, 44, 49] and three-dimensional [51] isothetic range searching. (See [52] for a survey.)

Even though the practical data structures mentioned above are often presented as structures for performing isothetic range searching, most of them can easily be modified to answer non-isothetic queries and thus also halfspace range queries. However, the query performance often seriously degrades. For example, even though we can answer halfspace range queries for uniformly distributed points in the plane in $O(\sqrt{n} + t)$ I/Os using data structures based on quad trees, the query performance can be as bad as $\Omega(n)$ I/Os even for reasonable distributions. The latter number of I/Os is required, for example, if $S$ consists of $N$ points on a diagonal line $\ell$ and the query halfplane is bounded by a line obtained by a slight perturbation of $\ell$. In this case, $\Omega(n)$ nodes of the tree are visited by the query algorithm. Similar performance degradation can be shown for the other mentioned structures.

In the internal memory model, a two-dimensional halfspace query can be answered in time $O(\log_2 N + T)$ time using $O(N)$ space [14], but in the external memory model a query may require $O(\log_2 N + T)$ I/Os using this data structure. The only known external memory data structure with provably good query performance works in two dimensions, where it uses $O(n\sqrt{N})$ disk blocks of space and answers queries using optimal $O(\log_B n + t)$ I/Os [24, 25].

## 1.3 Our results

In Section 3, we present the first optimal data structure for answering two-dimensional halfspace range queries in the worst case, based on the geometric technique called *filtering search* [12, 13, 15]. It uses $O(n)$ blocks of space and answers a query using $O(\log_B n + t)$ I/Os. It is simple enough to be efficient in practice.

| $d$ | Query I/Os | Space |
|---|---|---|
| 2 | $O(\log_B n + t)$ | $O(n)$ |
| 3 | $O(\log_B n + t)$ | $O(n \log_2 n)$ |
| | $O(n^\varepsilon + t)$ | $O(n \log_B n)$ |
| | $O((n/B^a)^{2/3+\varepsilon} + t)$ | $O(n \log_2 B)$ |
| | $O(n^{2/3+\varepsilon} + t)$ | $O(n)$ |
| $d$ | $O(n^{1-1/\lfloor d/2 \rfloor+\varepsilon} + t)$ | $O(n \log_B n)$ |
| | $O(n^{1-1/d+\varepsilon} + t)$ | $O(n)$ |

**Table 1.** Our main results.

In Section 4, we describe a data structure that uses $O(n \log_2 n)$ disk blocks and answers a three-dimensional halfspace range query using $O(\log_B n + t)$ expected I/Os. In the conference version of this paper [1], we described another data structure with optimal worst-case query time $O(\log_B n + t)$, but using $O(N(\log_2 n) \log_B n)$ space. As part of our result we also develop a data structure that uses $O(n \log_2 n)$ space to store $N$ points in the plane and that can be used to find the $k$ nearest neighbors of a query point in $O(\log_B n + k/B)$ expected I/Os.

As mentioned earlier, practical considerations often prohibit the use of more than linear space. In Section 5, we present the first linear-size structure for higher-dimensional halfspace range queries with provably good worst-case query performance. Our basic data structure answers $d$-dimensional queries in $O(n^{1-1/d+\varepsilon} + t)$ I/Os for any constant $\varepsilon > 0$. It can also report points lying inside a $d$-dimensional simplex query within the same I/O bound.[1]

In Section 6, we describe how to trade space for query performance in $\mathbb{R}^3$ by combining the data structures presented in the previous sections. We can answer a three-dimensional halfspace range query in $O((n/B^a)^{2/3+\varepsilon} + t)$ I/Os, for any constant $a > 0$, using slightly superlinear space $O(n \log_2 B)$, or in $O(n^\varepsilon + t)$ I/Os using $O(n \log_B n)$ space. This last data structure generalizes to answer $d$-dimensional halfspace queries in $O(n^{1-1/\lfloor d/2 \rfloor+\varepsilon} + t)$ I/Os in the same space bound.

Our main results are summarized in Table 1.

## 2 Geometric Preliminaries

In order to state our results we need some concepts and results from computational geometry.

### 2.1 Duality

Duality is a popular and powerful technique used in geometric algorithms; it maps each point in $\mathbb{R}^d$ to a hyperplane in $\mathbb{R}^d$ and vice-versa. We use the following duality transform: The dual of a point $(a_1, \ldots, a_d) \in \mathbb{R}^d$ is the hyperplane $x_d = -a_1 x_1 - \cdots - a_{d-1} x_{d-1} + a_d$, and the dual of a hyperplane $x_d = b_1 x_1 + \cdots + b_{d-1} x_{d-1} + b_d$ is the point $(b_1, \ldots, b_d)$. Let $\sigma^*$ denote the dual of an object (point or hyperplane) $\sigma$; for a set of objects $\Sigma$, let $\Sigma^* = \{\sigma^* \mid \sigma \in \Sigma\}$.

An essential property of duality is that it preserves the above-below relationship between points and hyperplanes; see Figure 1.

**Lemma 2.1.** *A point $p$ is above (resp., below, on) a hyperplane $h$ if and only if the dual hyperplane $p^*$ is above (resp., below, on) the dual point $h^*$.*

[1]We define a $d$-dimensional simplex to be the intersection of $d+1$ $d$-dimensional halfspaces.

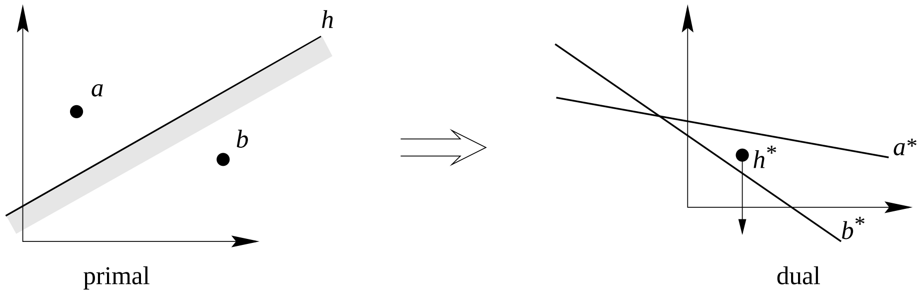


**Figure 1.** The duality transform in two dimensions.

By Lemma 2.1, the points in $S$ that lie below a hyperplane $h$ dualize to hyperplanes in $S^*$ that lie below the point $h^*$. Thus, the halfspace range searching problem has the following equivalent "dual" formulation:

> Preprocess a set $H$ of $N$ hyperplanes in $\mathbb{R}^d$ so that the hyperplanes of $H$ lying below a query point $p$ can be reported efficiently.

## 2.2 Arrangements

Let $H$ be a set of $N$ hyperplanes in $\mathbb{R}^d$. The *arrangement* of $H$, denoted $\mathcal{A}(H)$, is the decomposition of $\mathbb{R}^d$ into cells of dimensions $k$, for $0 \le k \le d$, each cell being a maximal connected set of points contained in the intersection of a fixed subset of $H$ and not intersecting any other hyperplane of $H$. For example, a set of lines in the plane induces a decomposition of the plane into vertices, edges, and two-dimensional faces; see Figure 2(i). The *combinatorial complexity* of $\mathcal{A}(H)$ is the total number of cells of all dimensions in the decomposition. It is well known that, for any fixed $d$, the complexity of $\mathcal{A}(H)$ is $O(N^d)$ [21].

## 2.3 Levels

The *level* of a point $p \in \mathbb{R}^d$ with respect to $H$ is the number of hyperplanes of $H$ that lie (strictly) below $p$. All the points in a single cell of arrangement $\mathcal{A}(H)$ lie above the same subset of hyperplanes of $H$, so we can define the level of a cell of $\mathcal{A}(H)$ to be the level of any point in that cell. For any $0 \le k < N$, the *$k$-level* of $\mathcal{A}(H)$, denoted $\mathcal{A}_k(H)$, is the closure of all the $(d-1)$-dimensional cells whose level is $k$; it is a monotone piecewise-linear $(d-1)$-dimensional surface. For example, the $k$-level in an arrangement of lines is a $x$-monotone polygonal chain. Figure 2(ii) depicts the 2-level in an arrangements of lines in the plane. The 0-level $\mathcal{A}_0(H)$ is also called the *lower envelope* of $H$. The lower envelope is the boundary of an unbounded convex polyhedron in $\mathbb{R}^d$.

An algorithm by Edelsbrunner and Welzl [22] can compute a level $\Lambda$ of an arrangement of lines with $\nu$ edges in $O(N \log_2 N + \nu \log_2^2 N)$ time, as follows. For a point $x \in \Lambda$, let $L^+(x)$ and $L^-(x)$ denote the set of lines in $L$ that lie respectively above and below $x$. Their algorithm traverses $\Lambda$ from left to right, stopping at every vertex of $\Lambda$, and maintains the two sets $L^+(x)$ and $L^-(x)$ during this traversal. These sets change only at the vertices of $\Lambda$. It stores $L^+$ in a dynamic data structure $\Psi(L^+(x))$ of Overmars and van Leeuwen [43] so that a line can be inserted or deleted in $O(\log_2^2 N)$ time and the first intersection point of a ray, emanating from a point lying above all the lines or below all the lines, with $L^+$ can be computed in $O(\log_2^2 N)$ time. Initializing this data structure takes $O(N \log_2 N)$ time. A similar structure is stored for $L^-$. Assuming that the algorithm has traversed $\Lambda$ up to a vertex $v$ and has constructed $\Psi(L^+(v))$ and $\Psi(L^-(v))$, the

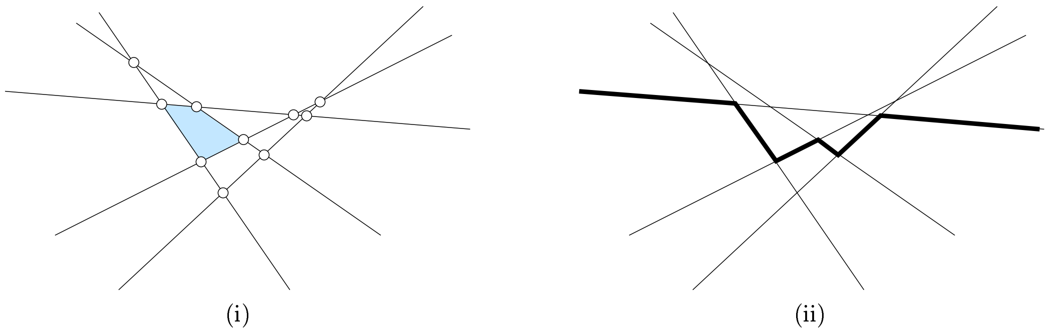

**Figure 2.** (i) An arrangement of lines, with one cell shaded; (ii) the 2-level of the arrangement.

next vertex $w$ of $\Lambda$ can be computed in $O(\log_2^2 N)$ time and $\Psi(L^+(w))$ and $\Psi(L^-(w))$ can also be computed within the same time bound; see the original paper for details. The data structure by Overmars and van Leeuwen [43] uses a two-level red-black tree. Replacing the second level red-black tree with a $B$-tree, the number of I/Os required to compute the level can be reduced to $O(N \log_2 N + \nu(\log_2 N) \log_B N)$; again, the $O(N \log_2 N)$ term is the time require to initialize the data structures.

Little is known about the worst-case complexity of $k$-levels. A recent result of Dey [19] shows that the maximum number of vertices on the $k$-level in an arrangement of $N$ lines in the plane is $O(Nk^{1/3})$. For $d = 3$, the best known bound on the complexity of $\mathcal{A}_k(H)$ is $O(Nk^{5/3})$ [2]. Neither of these bounds is known to be tight; the best lower bound known is $\Omega(N^{d-1}2^{\sqrt{\log k}})$ for all $k \le N/2$ [50]. However, if we choose a *random* level of $\mathcal{A}(H)$, a better bound follows from a result of Clarkson and Shor [16]; see, for example, Agarwal *et al.* [4].

**Lemma 2.2.** *Let $H$ be a set of $N$ hyperplanes in $\mathbb{R}^d$. For any $1 \le i \le \lfloor N/2 \rfloor$, if we choose a random integer $k$ between $i$ and $2i$, the expected complexity of $\mathcal{A}_k(H)$ is $O(N^{\lfloor d/2 \rfloor} k^{\lceil d/2 \rceil - 1})$.*

**Corollary 2.3.** *If only $M$ hyperplanes in $H$ contain a point on or below $\mathcal{A}_{2i}(H)$, then the expected complexity of a random level between $i$ and $2i$ is $O(M^{\lfloor d/2 \rfloor} k^{\lceil d/2 \rceil - 1})$.*

# 3 An Optimal Data Structure in 2D

In this section, we describe an optimal data structure for halfspace range searching in the plane that requires $O(n)$ disk blocks and answers a query using $O(\log_B n + t)$ I/Os. We will describe the data structure in the dual setting, *i.e.*, we wish to preprocess a set $L$ of $N$ lines in the plane so that the lines of $L$ lying below a query point can be reported efficiently. Our data structure is an extension of the halfspace range searching data structure by Chazelle *et al.* [14]. Roughly speaking, we partition $L$ into a family of subsets $\langle L_1, L_2, \ldots, L_m \rangle$, each of size at least $B \log_B n$, so that, for all $i > 1$, any point $p \in \mathbb{R}^2$ lies above a line of $L_i$ only if it lies above at least $B \log_B n$ lines of $L_{i-1}$. We store each $L_i$ into a linear-size data structure so that all $T_i$ lines of $L_i$ lying below a query point can be reported using $O(\log_B n + T_i/B)$ I/Os. To report the lines of $L$ lying below a query point $p$, we visit $L_i$'s in increasing order $i = 1, 2, \ldots$, stopping when we reach an $i$ so that at most $B \log_B n$ line of $L_i$ lie below $p$. Thus the total number of I/Os used in answering a query is $O(\log_B n + t)$.

In order to present our data structure we first describe a method for representing a level of $\mathcal{A}(L)$ compactly.

### 3.1 Compressing a level

For any point $p$ in the plane, let $x(p)$ denote its $x$-coordinate, and let $L_p$ denote the set of lines passing strictly below $p$. For two vertices $v$ and $v'$ on $\mathcal{A}_k(L)$ with $x(v) < x(v')$, we define the *cluster* $C \subseteq L$ induced by $v$ and $v'$ to be the set $\bigcup_p L_p$, where the union is taken over all points on $\mathcal{A}_k(L)$ between $v$ and $v'$. In other words, $C$ is the set of lines intersecting the interior of the polygon bounded by the portion of $\mathcal{A}_k(L)$ between $v$ and $v'$ and the vertical rays emanating downward from $v$ and $v'$; see Figure 3. We say that $C$ is *relevant* for a point $p$ if $x(v) \le x(p) < x(v')$.

Let $W = \langle w_0, w_1, w_2, \ldots, w_u \rangle$ be a subsequence of vertices of $\mathcal{A}_k(L)$, sorted from left to right, where $w_0$ and $w_u$ are the points on $\mathcal{A}_k(L)$ at $x = -\infty$ and $x = +\infty$, respectively. The *clustering* defined by $W$ is the family $\Gamma = \{C_1, C_2, \ldots, C_u\}$, where $C_i$ is the cluster induced by $w_{i-1}$ and $w_i$. We call the vertices $w_0, w_1, \ldots, w_u$ the *boundary points* of $\Gamma$. A line in $L$ may belong to several clusters in $\Gamma$. We call $\Gamma$ a *$b$-clustering* if every cluster contains $b$ or fewer lines. The *size* of a clustering $\Gamma$ is $u$, the number of clusters. For any point $p$ in the plane, exactly one cluster in $\Gamma$ is relevant for $p$.

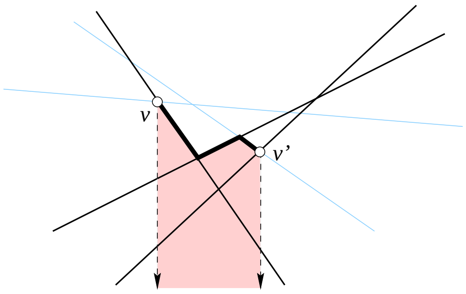


**Figure 3.** A cluster induced by two vertices of the 2-level.

The following lemmas form the basis of the data structures described in this section.

**Lemma 3.1.** *Let $L$ be a set of $N$ lines in the plane, let $\Gamma$ be a clustering of $\mathcal{A}_k(L)$ for some $1 \le k < N$, let $p$ be a point in the plane, and let $C \in \Gamma$ be the cluster relevant for $p$. If $p$ is above fewer than $k$ lines in $C$, then every line in $L$ that lies below $p$ is in $C$.*

**Proof:** Suppose $p$ is above fewer than $k$ lines in $C$. Let $\bar{p}$ be the intersection of $\mathcal{A}_k(L)$ with the vertical line through $p$. By definition, every line below $\bar{p}$ is in the cluster $C$. To prove the lemma, it suffices to show that $p$ is either on or directly below $\bar{p}$.

If $\bar{p}$ is a convex (downward) vertex of $\mathcal{A}_k(L)$, then $\bar{p}$ is above exactly $k-1$ lines, all in the cluster $C$. Moreover, at least one of the two lines through $\bar{p}$ is also in $C$ (in fact both lines will be in $C$ unless $\bar{p}$ is a boundary point, which is impossible for the clusterings we construct). Thus, if $p$ were above $\bar{p}$, then $p$ would be above at least $k$ lines in $C$, contradicting our original assumption.

If $\bar{p}$ is not a convex vertex of $\mathcal{A}_k(L)$, then $\bar{p}$ is above exactly $k$ lines, all in the cluster $C$. So in this case, $p$ must lie strictly below $\bar{p}$. □

**Lemma 3.2.** *Let $L$ be a set of $N$ lines in the plane. For any $1 \le k < N$, there exists a $3k$-clustering of $\mathcal{A}_k(L)$ of size at most $N/k$. Given $\mathcal{A}_k(L)$, this clustering can be computed using $O(\nu_k)$ I/Os, where $\nu_k$ is the complexity of $\mathcal{A}_k(L)$.*

**Proof:** Let $V = \langle v_0, v_1, \ldots, v_{\nu_k} \rangle$ be the complete sequence of vertices of $\mathcal{A}_k(L)$, sorted from left to right. We construct the clustering $\Gamma$ incrementally using the following greedy algorithm. Suppose we have already computed the clusters $C_1, \ldots, C_{i-1}$ and their boundary points $w_0, w_1, \ldots, w_{i-1}$. To construct $C_i$, we initially set $C_i = L_{w_{i-1}}$ and then scan through $V$ from left to right, starting with $w_{i-1}$. At each vertex $v_j \in V$, we either set $w_i = v_j$ and start a new cluster $C_{i+1}$ or continue with the cluster $C_i$ and add a new line to $C_i$ if necessary. Specifically, we process each vertex $v_j$ as follows. If $j = \nu_k$, then we are done—$C_i$ is induced by $w_{i-1}$ and $w_i = v_{\nu_k}$, and $C_i$ is the last cluster in $\Gamma$. If $v_j$ is a concave (upward) vertex (*e.g.*, vertices $a_h$ and $a_f$ in Figure 4), there is nothing to do, since the set $L_p$ remains the same for all points $p \in \mathcal{A}_k(L)$ in a sufficiently small neighborhood of $v_j$. Finally, suppose $v_j$ is a convex (downward) vertex (*e.g.*, vertex $w_{i-1}$ in Figure 4). Let $\ell \in L$ be the line through $v_j$ with minimum slope; $\ell$ lies below $\mathcal{A}_k(L)$ just to the right of $v_j$. If $\ell$ is already in $C_i$, there is nothing to do. If $\ell \notin C_i$ and $C_i$ already has $3k$ lines, we set $w_i = v_j$ and begin building $C_{i+1}$. Finally, if $\ell \notin C_i$ and $|C_i| < 3k$, we add $\ell$ to $C_i$ and continue scanning.

In order to check whether $\ell \in C_i$ in one I/O, we maintain a bit $b(\ell)$ for each line $\ell \in L$ indicating whether or not $\ell \in C_i$. After we find the boundary point $w_i$, for each line $\ell \in C_i$ that lies above $w_i$, we reset the bit $b(\ell)$ to 0; this requires at most $|C_i|$ I/Os. Thus, given $\mathcal{A}_k(L)$, our greedy algorithm builds the clustering $\Gamma$ in $O(\nu_k)$ I/Os.

To finish the proof, it suffices to show that there are at least $k$ lines in each cluster $C_i \in \Gamma$ that do not belong to any later cluster $C_j$ with $j > i$. Fix a cluster $C_i$. For each line $\ell \in C_i$, define its *exit point* $a_\ell$ to be the rightmost point of $\ell$ between $w_{i-1}$ and $w_i$ whose level is at most $k$. If $\ell$ lies below $w_i$, then $a_\ell$ lies on the vertical line passing through $w_i$; otherwise, $a_\ell$ is a concave (upward) vertex of $\mathcal{A}_k(H)$. Refer to Figure 4. Let $h \in C_i$ be a line such that the exit points of at least $2k$ lines in $C_i$ lie to the right of $a_h$. There are exactly $k$ such lines. We claim that $h$ lies above at least $k$ lines to the right of $w_i$, and thus $h$ cannot appear in a later cluster.

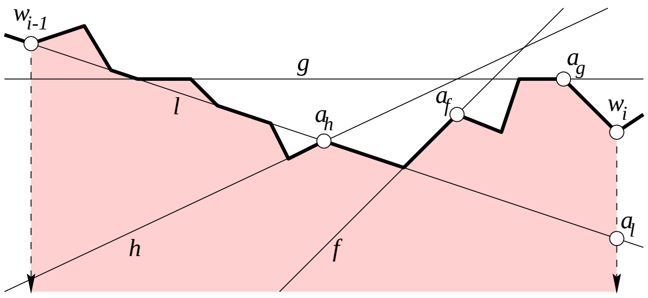


**Figure 4.** Proof of Lemma 3.2: Four lines in $C_i$ and their exit points; $g$ and $\ell$ lie below $f$ and $h$ to the right of $w_i$.

Exactly $k$ lines pass below the exit point $a_h$, since $a_h$ is a concave vertex of $\mathcal{A}_k(L)$. Thus, there are at least $k$ lines in $C_i$ that lie above $a_h$ and whose exit points lie to the right of $a_h$. Let $g$ be such a line. Since $a_h$ lies on or below $g$ and $a_g$ lies below $h$, the slope of $g$ is less than the slope of $h$, and $x(a_h) \le x(h \cap g) < x(a_g) \le x(w_i)$. These two observations imply that $g$ lies below $h$ to the right of $w_i$, proving our claim. See Figure 4. $\square$

We call the clustering built by our greedy algorithm the *greedy* $3k$*-clustering* of $\mathcal{A}_k(L)$.

**Corollary 3.3.** *Let $L$ be a set of $N$ lines in the plane, let $\Gamma$ be the greedy $3k$-clustering of $\mathcal{A}_k(L)$ for some $1 \le k < N$, and let $\ell$ be a line that appears in a cluster $C_i$ of $\Gamma$. If $\ell$ appears in another cluster of $\Gamma$ to the right of $C_i$, then it also appears in $C_{i+1}$.*

**Proof:** Assume that $\ell \notin C_{i+1}$. Let $a_\ell$ be the exit point of $\ell$ in $C_i$; this point must be on $\mathcal{A}_k(L)$ to the left of the boundary point $w_i$. Since $|C_{i+1}| = 3k$ and exactly $k$ lines lie below $a_\ell$, at least

$2k$ lines of $C_{i+1}$ must lie above $a_\ell$. Each of these $2k$ lines must be below $\mathcal{A}_k(L)$, and thus below $\ell$, somewhere between $w_i$ and $w_{i+1}$. These $2k$ lines must lie below $\ell$ to the right of $w_{i+1}$, which implies that $\ell$ cannot appear in any cluster $C_j$ with $j > i+1$. $\square$

### 3.2 Constructing the data structure

Our construction algorithm partitions $L$ into a family of $m$ disjoint subsets $L_1, L_2, \ldots, L_m$ and builds a data structure for each subset using the following iterative process. Let $\beta = B \log_B n$. Let $H_1 = L$, and for each $i > 1$, let $H_i = H_{i-1} \setminus L_{i-1}$. For each $i$, choose a random integer $\lambda_i$ between $\beta$ and $2\beta$, and construct the greedy $3\lambda_i$-clustering $\Gamma_i$ of $\Lambda_i = \mathcal{A}_{\lambda_i}(H_i)$. Recall that our greedy algorithm first computes $\Lambda_i$ using the Edelsbrunner-Welzl algorithm described in Section 2.3 and then computes the greedy clustering $\Gamma_i$ of $\Lambda_i$. $L_i$ is the union of the clusters in $\Gamma_i$, or equivalently, the set of lines that pass below some point on $\Lambda_i$. The data structure for $L_i$ is the lines in each cluster of $\Gamma_i$ sorted in increasing slope order (or by any other total order on $L$), as well as a $B$-ary tree $T_i$ (specifically, a $B^+$-tree) over the $x$-coordinates of the boundary points of $\Gamma_i$. The process stops when $L_i = H_i$.

Let $N_i = |L_i|$, and let $n_i = N_i/B$. Since each of the at most $N_i/\lambda_i$ clusters in $\Gamma_i$ can be stored in at most $\lceil 3\lambda_i/B \rceil$ blocks (Lemma 3.2), we can store the entire clustering in $O(n_i)$ blocks. The tree $T_i$ requires only $O(n_i/\lambda_i)$ blocks. Since $N_i > \beta$, the total number of clusterings $m$ is at most $N/\beta = n/\log_B n$. Thus, the total space used by our data structure is $\sum_{i=1}^{m} O(n_i) = O(n+m) = O(n)$.

Without any further modification, Edelsbrunner and Welzl's algorithm requires $O(|H_i| \log_2 |H_i| + |\Lambda_i| (\log_2 |H_i|) \log_B |H_i|)$ I/Os to construct the level $\Lambda_i$. We can speed up the construction by maintaining a data structures from one invocation of the algorithm to the next, as follows. We maintain a $B^+$-tree $T^*$ storing the lines in $H_i$, sorted by slope. At the beginning of the overall algorithm, we construct $T^*$ to store the entire set of lines $H_1 = L$ in $O(N \log_B n)$ I/Os. Let $y$ be the point on $\Lambda_{i-1}$ at $x = +\infty$. Recall that when the Edelsbrunner-Welzl algorithm finishes computing $\Lambda_{i-1}$, it has also computed the data structures $\Psi(H_{i-1}^+(y))$ and $\Psi(H_{i-1}^-(y))$. Before we can compute $\Lambda_i$, we first need to construct $\Psi(H_i^+(z))$ and $\Psi(H_i^-(z))$, where $z$ is the point on $\Lambda_i$ at $x = -\infty$.

Note that lines appear in increasing slope order along the positive $y$-direction at $x = +\infty$, but in decreasing slope order at $x = -\infty$. In particular, $H_{i-1}^-(y)$ consists of the lines with the $\lambda_{i-1}$ smallest slopes in $H_{i-1}$ and $H_i^-(z)$ consists of the $\lambda_i$ lines with largest slopes in $H_i$. We can extract $H_i^-(z)$ from $T^*$ in $O(\lambda_i/B)$ I/Os, after which we can construct $\Psi(H_i^-(z))$ in $O(\lambda_i (\log_2 \lambda_i) \log_B \lambda_i) = O(\lambda_i (\log_2 |H_i|) \log_B |H_i|)$ I/Os. Rather than extracting the lines in $H_i^+(z)$ from $T^*$ and constructing $\Psi(H_i^+(z))$, we use the following identity to obtain $\Psi(H_i^+(z))$:

$$H_i^+(z) = (H_{i-1}^+(y) \cup H_{i-1}^-(y)) \setminus L_{i-1} \setminus H_i^-(z).$$

To construct $\Psi(H_i^+(z))$, we start with $\Psi(H_{i-1}^+(y))$, insert the lines in $H_{i-1}^-(y)$, delete the lines in $L_{i-1}$, and finally delete the lines in $H_i^-(z)$. The lines in $H_{i-1}^-(y)$ and $H_i^-(z)$ can be extracted from $T^*$ in $O(\lambda_{i-1}/B)$ and $O(\lambda_i/B)$ I/Os, respectively. The total number of insertions and deletions is at most $\lambda_{i-1} + N_{i-1} + \lambda_i \leq 4N_{i-1}$, since $\lambda_i \leq 2\beta \leq 2\lambda_{i-1}$ and $\lambda_{i-1} \leq N_{i-1}$. Each insertion or deletion requires $O((\log_2 |H_i|) \log_B |H_i|)$ I/Os, so the total time to construct $\Psi(H_i^+(z))$ and $\Psi(H_i^-(z))$ is $O(N_{i-1} (\log_2 |H_i|) \log_B |H_i|)$ I/Os.

Once these two data structures have been built, we need $O(|\Lambda_i| (\log_2 |H_i|) \log_B |H_i|)$ I/Os to construct $\Lambda_i$, and then $O(|\Lambda_i|)$ I/Os to build the clustering $\Gamma_i$ and the subset $L_i$. Finally, we can delete the lines in $L_{i-1}$ from $T^*$ in $O(N_{i-1} \log_B |H_i|)$ I/Os. Thus, the total number of I/Os used for the $i$th phase of our preprocessing algorithm is $O((N_{i-1} + |\Lambda_i|)(\log_2 |H_i|) \log_B |H_i|) = O((N_{i-1} + |\Lambda_i|)(\log_2 N) \log_B n)$.

Let $\overline{L}_i$ be the set of lines in $H_i$ that pass below some point on $\mathcal{A}_{2\beta}(H_i)$, and let $\overline{N}_i = |\overline{L}_i|$. The expected number of vertices $|\Lambda_i|$ in $\Gamma_i$ is $O(\overline{N}_i)$ by Corollary 2.3. We easily observe that $\overline{L}_i \subseteq L_i \cup L_{i+1}$, so $\overline{N}_i \leq N_i + N_{i+1}$. It follows that the total expected number of I/Os to construct our data structure is $O(N \log_B n) + \sum_{i=2}^{m} O((N_{i-1} + |\Lambda_i|)(\log_2 N) \log_B n) = O(N(\log_2 N) \log_B n)$. In the worst case, each $\Gamma_i$ can have $O(\overline{N}_i \lambda_i^{1/3})$ vertices, so the worst-case construction time is $O(NB^{1/3}(\log_2 N) \log_B^{4/3} n)$.

## 3.3 Answering a query

Let $p$ be a query point and let $C_{i\,..\,j}$ denote $\bigcup_{k=i}^{j} C_k = C_i \cup C_{i+1} \cup \cdots \cup C_j$. To report all the lines lying below $p$, we visit the clusterings $\Gamma_i$ in increasing order $i = 0, 1, 2, \ldots$, stopping when at most $\lambda_i$ lines of $L_i$ lie below $p$. For each $i$, we determine which cluster $C_j \in \Gamma_i$ is relevant for $p$ in $O(\log_B n)$ I/Os, using the $B$-ary tree $T_i$. Next, we scan through $C_j$, counting the lines that lie below $p$. If there are fewer than $\lambda_i$ such lines, we report them and halt; by Lemma 3.1, every line in $H_i$ (and thus in $L_{i\,..\,m}$) that lies below $p$ is actually reported in this case. Otherwise, we visit the clusters $C_{j+1}, C_{j+2}, \ldots$ in order from left to right, stopping when we reach a cluster $C_r$ so that more than $\lambda_i$ lines in $C_{j+1\,..\,r}$ lie *above* $p$. Since the lines in each cluster are stored in sorted order by slope, we can keep track of the lines above $p$ in $O(\lambda_i/B) = O(\log_B n)$ I/Os per cluster. Next, we visit clusters from right to left in a similar manner, starting with $C_{j-1}$ and stopping when we reach a cluster $C_l$ so that more than $\lambda_i$ distinct lines in $C_{l\,..\,j-1}$ lie above $p$. For each cluster $C_k$ we visit, we report the lines of $C_k$ that lie below $p$.

The correctness of this procedure follows from the following lemma.

**Lemma 3.4.** *Suppose $C_j \in \Gamma_i$ is relevant for a point $p$. If more than $\lambda_i$ lines in $C_{l\,..\,j-1}$ lie above $p$, then no line of $C_{1\,..\,l-1} \setminus C_{l\,..\,j}$ lies below $p$. If more than $\lambda_i$ lines in $C_{j+1\,..\,r}$ lie above $p$, then no line of $C_{r+1\,..\,|\Gamma_i|} \setminus C_{j\,..\,r}$ lies below $p$.*

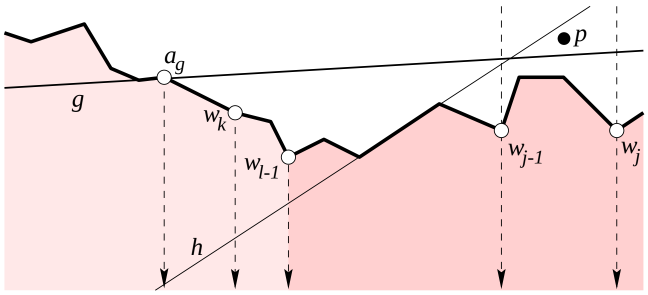


**Figure 5.** Proof of Lemma 3.4.

**Proof:** Suppose a line $g \in C_k \setminus C_{l\,..\,j}$ lies below $p$, for some $k < l$. Without loss of generality, assume $g \notin C_{k+1}$. Then Corollary 3.3 implies that $g$ is not in any cluster to the right of $C_{k+1}$. Let $a_g$ be the exit point of $g$, *i.e.*, the rightmost point of $g$ on $\Lambda_i$; see Figure 5. The point $a_g$ is strictly to the left of the boundary point $w_k$, so $g$ lies above $\Lambda_i$ to the right of $w_k$. Any line $h \in C_{l\,..\,j}$ that lies above $p$ must intersect $g$ between $a_g$ and $p$, which implies that $h$ lies below $a_g$. But exactly $\lambda_i$ lines lie below $a_g$. It follows that at most $\lambda_i$ lines in $C_{l\,..\,j}$ lie above $p$.

The second half of the lemma follows by a similar argument. □

Altogether, our query procedure uses $O(\log_B n+(r-l+1)\lambda_i/B)$ I/Os to search the clustering $\Gamma_i$. Let $T_i$ be the number of lines in $L_i$ that lie below $p$. For each cluster visited by the query algorithm, at least $\lambda_i$ lines do not appear in any cluster further to the right; see the proof of Lemma 3.2. It follows that the algorithm visits at least $(r-l+1)\lambda_i$ distinct lines. Among these, at most $10\lambda_i$ lines can lie above $p$—specifically, at most $2\lambda_i$ lines from $C_j$, at most $\lambda_i$ lines from each of $C_{l+1\,..\,j-1}$ and $C_{j+1\,..\,r-1}$, and at most $3\lambda_i$ lines from each of $C_l$ and $C_r$. Hence, $T_i \geq (r-l-9)\lambda_i$, and the number of I/Os required to search $\Gamma_i$ is $O(\log_B n + T_i/B)$.

If our query algorithm visits clusterings $\Gamma_1, \dots \Gamma_u$, then it requires $O(u\log_B n + \sum_{i=1}^{u} T_i/B) = O(u\log_B n + t)$ I/Os. Since $T_i > \lambda_i \geq B\log_B n$ for all $i \leq u-1$, we have, $t \geq (u-1)\log_B n$. Thus, the total number of I/Os required to answer a query is $O(\log_B n + t)$.

**Theorem 3.5.** *Let $S$ be a set of $N$ points in the plane. We can store $S$ in a data structure that uses $O(n)$ blocks so that a halfspace range query can be answered in $O(\log_B n + t)$ I/Os. The expected number of I/Os used to construct the data structure is $O(N(\log_2 N)\log_B n)$.*

Although our query procedure is asymptotically optimal, it may report some lines several times, since a line can appear in several clusters within the same clustering. We can detect and remove these duplications by exploiting Corollary 3.3 as follows. Suppose our algorithm visits clusters $C_l, \dots, C_j, \dots, C_r$. We report all the lines that lie below $p$ in the leftmost cluster $C_l$ as usual. For each other cluster $C_k$ to the right of $C_l$, we report only the lines in $C_k \setminus C_{k+1}$ that lie below $p$. Since the lines in each cluster are sorted by their slopes, the set $C_k \setminus C_{k+1}$ can be computed using $O(\lambda_i/B)$ I/Os and thus our modified query algorithm reports each line exactly once, still using only $O(\log_B n + t)$ I/Os.

## 4 Halfspace Range Searching in 3D

In this section, we describe our three-dimensional data structure, which generalizes a recent internal memory result of Chan [11] to the external memory setting. As in the previous section, we solve the dual version of the problem; given a set $H$ of $N$ planes, we build a data structure that allows us to quickly compute all the planes lying below an arbitrary query point.

The general idea in our structure is to store $\mathcal{A}_k(H)$ for an exponentially increasing series of values of $k$ ($k = 2^j B\log_B n$ for $j = 0, 1, 2, \dots$). These levels partition the space into "layers". To answer a query with point $p$, we test whether $p$ lies above or below $\mathcal{A}_k(H)$ for the exponentially increasing values of $k$ until we find a $k$ such that $p$ lies below $\mathcal{A}_k(H)$. Once we know which "layer" contains $p$, we simply report the planes that lie below $p$. Since the complexity of a level in $\mathcal{A}_k(H)$ can be quite large, we actually approximate $\mathcal{A}_k(H)$ by computing the 0-level, or *lower envelope*, of an appropriate subset of $H$, and use the lower envelope to determine whether a query point is above or below $\mathcal{A}_k(H)$. The lower envelope of a set of $N$ planes is the boundary of an unbounded convex polyhedron with at most $N$ convex polygon facets and, by Euler's formula, $O(N)$ edges and vertices. Because lower envelopes have a much simpler structure than arbitrary levels, we can compute them much more easily; however, because we do not store the levels exactly, we need an additional data structure to determine whether a query point lies above a certain level. We now describe the data structure in detail.

### 4.1 Finding the $k$ lowest planes along a vertical line

Let $H$ be a set of $N$ planes in $\mathbb{R}^3$. For any vertical line $\ell$ and any integer $k$, the *$k$ lowest planes in $H$ along $\ell$* are the planes in $H$ whose intersections with $\ell$ have the $k$ smallest $z$-coordinates. Following

Chan [11], we first develop a data structure that stores $H$ so that we can efficiently report the $k$ lowest planes in $H$ along $\ell$ for any $k$ and $\ell$.

Let $\beta = B \log_B n$. We choose a random permutation $\langle h_1, h_2, \ldots, h_N \rangle$ of the planes in $H$, and for each $i$ between 1 and $\lceil \log_2(N/\beta) \rceil$, we define $R_i = \{h_1, h_2, \ldots, h_{2^i}\} \subset H$. Each subset $R_i$ is a uniformly-distributed random sample of $H$ of size $2^i$. Our data structure consists of $O(\log_2(N/\beta)) = O(\log_2 n)$ layers, one for each random sample $R_i$. To simplify our notation, let $r_i = 2^i/B$ denote the number of blocks required to store $R_i$.

We construct the $i$th layer of our data structure as follows. First, we construct the lower envelope of $R_i$, in $O(r_i \log_B r_i)$ expected I/Os, using the external 3D halfspace intersection algorithm of Crauser *et al.* [18].[2] In fact, this algorithm computes a triangulation of the lower envelope, decomposing each non-triangular facet into several interior-disjoint triangles. We denote this triangulation $\Delta(R_i)$. By Euler's formula, $\Delta(R_i)$ contains $O(2^i)$ triangles and thus can be stored in $O(r_i)$ blocks.

Next, we construct an external point-location structure for the planar map obtained by projecting $\Delta(R_i)$ orthogonally onto the $xy$-plane. This structure allows us to find the triangle in $\Delta(R_i)$ directly above or below a query point in $O(\log_B r_i) = O(\log_B n)$ I/Os. This point location structure can be constructed in $O(r_i \log_B r_i)$ I/Os and stored in $O(r_i)$ blocks [7, 27].

Finally, we say that a plane $h \in H \setminus R_i$ *conflicts* with a triangle $\triangle \in \Delta(R_i)$ if it lies below some point in $\triangle$. The set of planes that conflict with a triangle $\triangle$ is called the triangle's *conflict list* and denoted $K(\triangle)$. We store the conflict list of each triangle in $\Delta(R_i)$ in one contiguous set of blocks. This allows us to scan a single conflict list $K(\triangle)$ in $\lceil |K(\triangle)|/B \rceil$ I/Os. The halfspace intersection algorithm of Crauser *et al.* [18] can be used to construct these conflict lists in $\Delta(R_i)$ in $O(n \log_B r_i)$ expected I/Os.

Our analysis relies on the following probabilistic lemma of Clarkson and Shor [16].

**Lemma 4.1.** *Let $1 \le r \le N$, and let $R$ be a random sample of $H$ of size $r$.*

*(a)* $E\left[\sum_{\triangle \in \Delta(R)} |K(\triangle)|\right] = O(N)$.

*(b) For any vertical line $\ell$, the expected size of $K(\triangle)$, where $\triangle$ is the triangle in $\Delta(R_i)$ that intersects $\ell$, is $O(N/r)$.*

The first part of this lemma implies that the expected number of blocks required to store all the conflict lists for each sample $R_i$ is $O(n)$. Since this is larger than the space requirement for $\Delta(R_i)$ and its point-location structure, we conclude that each layer can be stored in $O(n)$ expected blocks. Thus, the expected size of the entire data structure is $O(n \log_2 n)$ blocks. The total expected preprocessing time of all $O(\log_2 n)$ layers is $O(n(\log_2 n) \log_B n)$ I/Os.

Given an integer $k$, a vertical line $\ell$, and a third parameter $0 < \delta < 1$, the following procedure *usually* finds the $k$ lowest planes in $H$ along $\ell$. The parameter $\delta$ controls the probability of failure.

[2]The algorithm by Crauser *et al.* actually uses $O(r_i \log_{M/B} r_i)$ I/Os where $M$ is the size of the internal memory. We make the realistic assumption that the internal memory is capable of holding more than $B$ blocks, that is, we have $M/B > B$ and thus $O(r_i \log_{M/B} r_i) = O(r_i \log_B r_i)$.

TryLowestPlanes$(k, \ell, \delta)$:

$\rho = \lceil \log_2(N\delta/k) \rceil$
Find the triangle $\triangle \in \Delta(R_\rho)$ intersecting $\ell$
if $|K(\triangle)| \leq k/\delta^2$ then
  scan $K(\triangle)$
  if $\geq k$ planes in $K(\triangle)$ cross $\ell$ below $\ell \cap \triangle$ then
    return $k$ lowest planes in $K(\triangle)$ along $\ell$
  else fail
else fail

Recall that $R_\rho$ is a random sample of $H$ of size $2^\rho < 2N\delta/k$. We can find the triangle $\triangle$ that intersects $\ell$ in $O(\log_B 2^\rho) = O(\log_B n)$ I/Os using the external point-location structure for $\Delta(R_\rho)$. We only scan the conflict list $K(\triangle)$ if its length is less than $k/\delta^2$, so the scan requires at most $\lceil k/(\delta^2 B) \rceil$ I/Os. Thus, the overall running time of TryLowestPlanes is $O(\log_B n + k/(B\delta^2))$ I/Os, regardless of its success or failure.

Lemma 4.1(b) implies that the expected size of $K(\triangle)$ is $O(N/2^\rho) = O(k/\delta)$, so by Markov's inequality, the probability that the size of $K(\triangle)$ exceeds $k/\delta^2$ is $O(\delta)$. There are fewer than $k$ planes in $K(\triangle)$ below $\ell \cap \triangle$ if and only if the plane containing $\triangle$ is one of the $k$ lowest planes along $\ell$. This event occurs with probability at most $k2^\rho/N = O(\delta)$, since each of the $k$ lowest planes has probability $2^\rho/N$ of being in the random sample $R_\rho$. Thus, the probability that TryLowestPlanes fails is $O(\delta)$.

We can reduce the failure probability to $O(\delta^3)$ by building and querying three completely independent data structures. This triples the space and preprocessing time, of course, but is necessary to achieve optimal query time.[3]

To compute the $k$ lowest planes along $\ell$ with complete certainty, we invoke TryLowestPlanes using all three data structures, with $\delta = 2^{-1}, 2^{-2}, 2^{-3}, \ldots$, stopping as soon as some invocation succeeds. Let $X_i$ be the 0-1 random variable whose value is 1 if all three calls to TryLowestPlanes fail when $\delta = 2^{-i}$. Then the total number of I/Os used by this procedure is at most

$$\sum_{i \geq 1} X_{i-1} \cdot O(\log_B n + 4^i k/B)$$

By our earlier argument, $E[X_i] = O((2^i)^3) = O(8^{-i})$, so the expected number of I/Os is

$$\sum_{i \geq 0} \frac{O(\log_B n + 4^{i+1} k/B)}{8^i} = \sum_{i \geq 0} O\left(\frac{\log_B n}{8^i} + \frac{4k/B}{2^i}\right) = O(\log_B n + k/B).$$

**Theorem 4.2.** *Let $H$ be a set of $N$ planes in $\mathbb{R}^3$. We can store $H$ in a data structure with expected size $O(n \log_2 n)$ blocks so that, for any vertical line $\ell$ and any integer $1 \leq k \leq N$, we can find the $k$ lowest planes in $H$ along $\ell$ in $O(\log_B n + k/B)$ expected I/Os. The expected number of I/Os used to build the data structure is $O(n(\log_2 n) \log_B n)$.*

**Remark.** Using the fact that $R_{i-1} \subset R_i$, we can reduce the expected number of I/Os used to construct the data structure to $O(n \log_2 n)$.

By a standard lifting argument [21], our data structure can also be used to answer $k$ nearest neighbor queries for points in the plane with the same space, preprocessing time, and query time

[3] Building three independent data structures may not be necessary in practice, but the best expected query time we can prove using just one data structure is $O(\log_B n + (k/B) \log_2(N/k))$ I/Os.

bounds. Given any set $S$ of $N$ points in the plane, we can lift it to a set of $N$ planes $\hat{S}$ in $\mathbb{R}^3$ by mapping each point $(a,b) \in S$ to the plane $z = a^2 + b^2 - 2ax - 2by$. Then the nearest $k$ neighbors in $S$ to a query point $(p,q) \in \mathbb{R}^2$ correspond precisely to the $k$ lowest planes in $\hat{S}$ along the vertical line through the point $(p,q,0)$.

**Theorem 4.3.** *Let $S$ be a set of $N$ points in the plane. We can store $S$ in a data structure with expected size $O(n \log_2 n)$ blocks so that, for any point $p \in \mathbb{R}^2$ and any integer $1 \le k \le N$, we can find the $k$ nearest neighbors of $p \in S$ in $O(\log_B n + k/B)$ expected I/Os. The expected number of I/Os used to build the data structure is $O(n(\log_2 n)\log_B n)$.*

### 4.2 Finding all the planes below a query point

To find the $T$ planes lying below a query point $p$, we only need to report the $T$ lowest planes along the vertical line $\ell$ through $p$. Since we do not know $T$ in advance, we find the $k$ lowest planes along $\ell$ for successively larger and larger values of $k$, halting when at least one of the $k$ lowest planes is above $p$ and then reporting only those planes that are actually below $p$. Specifically, in the $j$th iteration, we use the value $k = 2^j\beta = 2^j B \log_B n$.

If $T < \beta$, we halt after the very first iteration, spending $O(\log_B n + \beta/B) = O(\log_B n)$ expected I/Os. Otherwise, we halt after $\mu = \lceil \log_2(T/\beta) \rceil$ iterations, and since the $j$th iteration requires $O(\log_B n + 2^j\beta/B) = O(2^j\beta/B)$ I/Os on average, the total expected number of I/Os is

$$\sum_{i=0}^{\mu} O(2^j\beta/B) = O(2^\mu\beta/B) = O(T/B) = O(t).$$

**Theorem 4.4.** *Let $S$ be a set of $N$ points in $\mathbb{R}^3$. We can store $S$ in a data structure with expected size $O(n \log_2 n)$ blocks so that a halfspace range query can be answered in $O(\log_B n + t)$ expected I/Os. The expected number of I/Os used to build the data structure is $O(n(\log_2 n)\log_B n)$.*

## 5 A Linear-Size Data Structure

In this section we present a $d$-dimensional halfspace range-searching data structure that uses only $O(n)$ blocks, for any constant $d$. We will describe the data structure in the primal setting. Let $S$ be a set of $N$ points in $\mathbb{R}^d$. A *simplicial partition* of $S$ is a set of pairs $\Pi = \{(S_1,\Delta_1),(S_2,\Delta_2),\ldots,(S_r,\Delta_r)\}$, where $S_i$'s are disjoint subsets of $S$, and each $\Delta_i$ is a simplex containing $S_i$. Note that a point of $S$ may lie in many simplices, but it belongs to only one $S_i$; see Figure 6. The size of $\Pi$, here denoted $r$, is the number of pairs. A simplicial partition is *balanced* if each subset $S_i$ contains between $N/r$ and $2N/r$ points.

**Theorem 5.1 (Matoušek [36]).** *Let $S$ be a set of $N$ points in $\mathbb{R}^d$, and let $1 < r \le N/2$ be a given parameter. For some constant $\alpha$ (independent of $r$), there exists a balanced simplicial partition $\Pi$ of size $r$, so that any hyperplane crosses at most $\alpha r^{1-1/d}$ simplices of $\Pi$.*

We use this theorem to build a range-searching data structure for $S$ called a *partition tree*. Partition trees are one of the most commonly used internal memory data structures for geometric range searching [3, 30, 36, 53]; our construction closely follows the one by Matoušek [36]. Each node $v$ in a partition tree $T$ is associated with a subset $S_v \subseteq S$ of points and a simplex $\Delta_v$. For the root $u$ of $T$, we have $S_u = S$ and $\Delta_u = \mathbb{R}^d$. Let $N_v = |S_v|$ and $n_v = \lceil N_v/B \rceil$. For each node $v$, we construct the subtree rooted at $v$ as follows. If $N_v \le B$, then $v$ is a leaf and

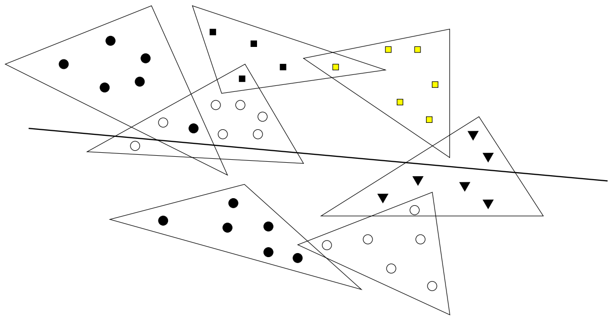

**Figure 6.** A balanced simplicial partition of size 7.

we store all points of $S_v$ in a single block. Otherwise, $v$ is an internal node of degree $r_v$, where $r_v = \min\{cB, 2n_v\}$, for some constant $c \geq 1$ to be specified later. We compute a balanced simplicial partition $\Pi_v = \{(S_1, \Delta_1), \ldots, (S_{r_v}, \Delta_{r_v})\}$ for $S_v$, as described in Theorem 5.1, and then recursively construct a partition tree $T_i$ for each subset $S_i$. For each $i$, we store the vertices of $\Delta_i$ and a pointer to $T_i$; the root of $T_i$ is the $i$th child of $v$, and it is associated with $S_i$ and $\Delta_i$. We need $O(c) = O(1)$ blocks to store any node $v$. Since $r_v$ was chosen to be $\min\{cB, 2n_v\}$, every leaf node contains $\Theta(B)$ points. Thus the total number of nodes in the tree is $O(n)$, so the total size of the partition tree is $O(n)$. A closer look at Matoušek's algorithm reveals that $\Pi_v$ can be constructed in $O(N \log_2 r_v) = O(N \log_2 B)$ expected I/Os, so the total expected number of I/Os to construct $T$ is $O(N \log_2 B \log_B N) = O(N \log_2 N)$.

To find all points below a query hyperplane $h$, we visit $T$ in a top down fashion. Suppose we are at a node $v$. If $v$ is a leaf, we report all points of $S_v$ that lie below $h$ in a single I/O. Otherwise, we test each simplex $\Delta_i$ of $\Pi_v$. If $\Delta_i$ lies above $h$, we ignore $\Delta_i$; if $\Delta_i$ lies below $h$, we report all points in $S_i$ by traversing the $i$th subtree of $v$; finally, if $h$ crosses $\Delta_i$, we recursively visit the $i$th child of $v$. Note that each point is reported only once.

To bound the number of I/Os used to perform a query, first note that if $\Delta_v$ lies below $h$, then all points of $S_v$ lie below $h$ and we spend $O(n_v)$ I/Os to report these points. Since each point is reported only once, $\sum_v n_v \leq t$, where the sum is taken over all nodes $v$ visited by the query procedure for which $\Delta_v$ lies below $h$. Let $\mu$ denote the number of visited nodes $v$ for which $\Delta_v$ intersects $h$, or in other words, the number of recursive calls to the query procedure. Since we require $O(1)$ I/Os at each such node, the overall query procedure requires $O(\mu + t)$ I/Os.

What remains is to bound the value of $\mu$. For each node $v \in T$, let $\Sigma(N_v)$ be the maximum number of descendants of $v$ (including $v$) that are recursively visited by the query procedure. (Recall that $N_v$ points are stored in the subtree rooted at $v$.) If $v$ is a leaf, only $v$ itself is visited. If $v$ is an interior node, then, by Theorem 5.1, we recursively visit at most $\alpha r_v^{1-1/d}$ children of $v$. Since each child of $v$ is associated with at most $2N_v/r_v$ points, we obtain the following recurrence:

$$\Sigma(N_v) \leq \begin{cases} 1 + \alpha r_v^{1-1/d}\, \Sigma(2N_v/r_v) & \text{if } N_v > B, \\ 1 & \text{if } N_v \leq B. \end{cases}$$

For any constant $\varepsilon > 0$, we prove by induction on $N_v$ that

$$\Sigma(N_v) \leq A n_v^{1-1/d+\varepsilon}, \tag{1}$$

where $A$ is a constant, provided we choose $c = c(\varepsilon)$ sufficiently large in the definition of $r_v$. Indeed, (1) is obviously true if $N_v \leq B$. If $B < N_v \leq cB^2/2$, then $r_v = 2n_v$ and

$$\Sigma(N_v) \;\leq\; 1 + \alpha(2n_v)^{1-1/d}\Sigma(B) \;=\; 1 + \alpha(2n_v)^{1-1/d} \;\leq\; A n_v^{1-1/d}$$

provided $A > 2^{1-1/d}\alpha + 1$. Finally, if $N_v > cB^2/2$, then by the induction hypothesis,

$$\begin{aligned}\Sigma(N_v) &\leq 1 + \alpha r_v^{1-1/d} A(2n_v/r_v)^{1-1/d+\varepsilon}\\ &\leq 1 + 2\alpha A n_v^{1-1/d+\varepsilon}/r_v^{\varepsilon}\\ &\leq \left(\frac{2}{AcB} + \frac{2\alpha}{(cB)^{\varepsilon}}\right) A n_v^{1-1/d+\varepsilon}\\ &\leq A n_v^{1-1/d+\varepsilon},\end{aligned}$$

assuming that we choose $c > (2\alpha)^{1/\varepsilon}$. Hence, we conclude the following.

**Theorem 5.2.** *Given a set $S$ of $N$ points in $\mathbb{R}^d$ and a constant $\varepsilon > 0$, we can preprocess $S$ into a data structure of size $O(n)$ blocks so that a $d$-dimensional halfspace range query can be answered using $O(n^{1-1/d+\varepsilon} + t)$ I/Os. The expected number of I/Os required to construct the data structure is $O(N \log_2 N)$.*

**Remarks.**

(i) The same data structure can also be used to report points lying inside a query *simplex* $\tau$ in time $O(n^{1-1/d+\varepsilon} + t)$. Basically, at each interior node $v$ visited by the query procedure, we check whether $\tau$ intersects $\triangle_i$, for each $\triangle_i \in \Pi_v$. If $\triangle_i \subseteq \tau$, we report all points in $S_v$. If $\triangle_i \not\subseteq \tau$ but $\triangle_i \cap \tau \neq \emptyset$, we recursively visit the $i$th child of $v$. Since each each facet of $\tau$ intersects $O(r_v^{1-1/d})$ simplices of $\Pi_v$, the query time is as desired. If the query object is a polyhedron with $m$ faces of all dimensions, we triangulate it into $O(m)$ simplices and query the data structure with each simplex. The total query time is $O(mn^{1-1/d+\varepsilon} + t)$ I/Os.

(ii) If we use a somewhat more sophisticated algorithm [36] for computing $\Pi_v$, the expected number of I/Os required to construct the above data structure can improved to $O(n \log_2 n)$ without affecting its asymptotic size and query time.

(iii) Using the above preprocessing algorithm and the standard partial-reconstruction method for dynamizing a data structure [39], a point can be inserted into or deleted from the data structure in $O((\log_2 n) \log_B n)$ amortized expected number of I/Os.

(iv) In our query algorithm, at each node $v$, we used a brute force method to determine which simplices of $\Pi_v$ cross the query hyperplane $h$. If we use a more sophisticated procedure to determine these simplices [36], we can choose $r_v = N_v^{\delta}$ for some constant $\delta < 1/2$. The query bound improves to $O(\sqrt{n} \log_2^{O(1)} n + t)$, and the space used by the data structure remains $O(n)$.

## 6 Trading Space for Query Time in 3D

We can combine Theorem 5.2 with Theorem 4.4 in order to improve the query time for 3-dimensional halfspace range query at the expense of space. The idea is to use the same recursive procedure to construct a tree as in the previous section, but stop the recursion when $N_v \leq B^a$ for some constant $a > 1$. In that case we preprocess $S_v$ into a data structure of size $O(n_v \log_2 n_v) = O(aB^a \log_2 B)$ using Theorem 4.4. The total size of the data structure is $O(an \log_2 B)$. A query is answered as in the previous section except that when we reach a leaf of the tree, we use the query procedure described in Section 3. The query procedure now visits $O((n/B^{a-1})^{2/3+\varepsilon})$ nodes of the tree.

**Theorem 6.1.** *Given a set $S$ of $N$ points in $\mathbb{R}^3$ and constants $\varepsilon > 0$ and $a > 1$, we can preprocess $S$ into a data structure of size $O(n \log_2 B)$ blocks so that a 3-dimensional halfspace range query can be answered using $O((n/B^{a-1})^{2/3+\varepsilon} + t)$ expected I/Os.*

If we allow $O(n \log_B n)$ space, the asymptotic query time can be improved considerably. A plane $h$ is called *$k$-shallow* with respect to $S$, for $k < N$, if at most $k$ points of $S$ lie below $h$.

**Theorem 6.2 (Matoušek [37]).** *Let $S$ be a set of $n$ points in $\mathbb{R}^3$, and let $1 < r \leq n/2$ be a given parameter. For some constant $\beta > 1$ (independent of $r$), there exists a balanced simplicial partition $\Pi$ of $S$ so that any $(N/r)$-shallow plane crosses at most $\beta \log_2 r$ simplices of $\Pi$.*

Using this theorem, we construct a so-called *shallow partition tree* $\Psi$ essentially as in the previous section, except for one additional twist. Each node of $\Psi$ is also associated with a subset $S_v \subseteq S$ of $N_v$ points and a simplex $\Delta_v$. If $N_v \leq B$, then $v$ is a leaf and we store all points of $S_v$ in a single block. Otherwise, $v$ is an internal node of degree $r_v$, where $r_v = \min\{cB, 2n_v\}$. We compute a balanced partition $\Pi_v = \{(S_1, \Delta_1), \ldots, (S_{r_v}, \Delta_{r_v})\}$ of $S$ using Theorem 6.2, recursively construct a shallow partition tree $\Psi_i$ for each $S_i$, and attach $\Psi_i$ as the $i$th subtree of $v$. We also construct a (non-shallow) partition tree $T_v$ on $S_v$, as described in the previous section, and store it as a secondary structure of $v$. Since we need $O(n_v)$ blocks to store $T_v$, the total size of $\Psi$ is $O(n \log_B n)$.

A query is answered by traversing $\Psi$ in a top down fashion. Suppose we are at an interior node $v$. As previously, we check which of the simplices of $\Pi_v$ cross the query plane $h$. If more than $\beta \log_2 r_v$ simplices cross $h$, we conclude that $h$ is not $(N_v/r)$-shallow with respect to $S_v$, and we use the secondary structure $T_v$ to report all points of $S_v$ lying below $h$ using $O(n_v^{2/3+\varepsilon} + t_v)$ I/Os, where $t_v$ is the number of points of $S_v$ lying below $h$. Since $t_v \geq N_v/r \geq n_v/c$, we have that $O(n_v^{2/3+\varepsilon} + t_v) = O(t_v)$. Otherwise, if at most $\beta \log_2 r$ simplices of $\Pi_v$ cross $h$, we report all points in simplices in $\Pi_v$ that lie below $h$, by traversing the corresponding subtrees, and recursively visit all simplices in $\Pi_v$ that cross $h$.

We say that a node $v$ visited by the query procedure is *shallow* if $h$ is $(N_v/r)$-shallow with respect to $S_v$. If the query visits $\mu$ shallow nodes, then the query procedure requires $O(\mu + t)$ I/Os. Let $\overline{\Sigma}(N_v)$ be the maximum number of shallow descendants of a shallow node $v$ (including $v$). Then, as in the previous section, we obtain the recurrence

$$\overline{\Sigma}(N_v) \leq \begin{cases} 1 + \beta(\log_2 r_v)\, \overline{\Sigma}(2N_v/r_v) & \text{if } N_v > B, \\ 1 & \text{if } N_v \leq B. \end{cases}$$

Solving this recurrence, as in the previous section, we can prove that $\mu = \overline{\Sigma}(N) = O(n^{\varepsilon})$ for any constant $\varepsilon > 0$, provided we choose $c = c(\varepsilon)$ sufficiently large.

**Theorem 6.3.** *Given a set $S$ of $N$ points in $\mathbb{R}^3$ and a constant $\varepsilon > 0$, we can preprocess $S$ into a data structure of size $O(n \log_B n)$ blocks so that a 3-dimensional halfspace range query can be answered using $O(n^{\varepsilon} + t)$ I/Os.*

**Remark.** The shallow partition trees can also be generalized to higher dimensions. For any $d > 3$, we obtain a data structure requiring $O(n \log_B n)$ blocks that can answer halfspace queries in $O(n^{1-1/\lfloor d/2 \rfloor + \varepsilon} + t)$ I/Os. The corresponding internal-memory data structure is described by Matoušek [37]. However, unlike the data structure described in the previous section, it cannot be extended to report points lying inside a query simplex.

## 7 Conclusions

In this paper, we have presented an optimal worst-case data structure for halfspace range searching in two dimensions and an $O(n \log_2 n)$-size average-case data structure for three-dimensional halfspace range searching that answer queries optimally. We have also presented a linear-size data structure that not only answers halfspace range queries but can also report points lying inside a query polyhedron. We conclude by mentioning a few open problems:

1. Is there a dynamic data structure for two-dimensional halfspace range searching that answers queries in $O(\log_B n + t)$ I/Os and that can be updated in $O(\log_B N)$ I/Os?

2. Can we use the techniques developed in this paper to solve more complicated problems? For example, can we develop an efficient data structure to store a set of line segments in the plane so that all the segments intersecting a query segment can be reported efficiently?

## References

[1] P. K. Agarwal, L. Arge, J. Erickson, P. G. Franciosa, and J. S. Vitter, Efficient searching with linear constraints, *Proc. 17th Annu. ACM Sympos. Principles Database Syst.*, 1998, pp. 169–178.

[2] P. K. Agarwal, B. Aronov, T. M. Chan, and M. Sharir, On levels in arrangements of lines, segments, planes, and triangles, *Discrete Comput. Geom.*, 19 (1998), 315–331.

[3] P. K. Agarwal and J. Erickson, Geometric range searching and its relatives, in *Advances in Discrete and Computational Geometry* (B. Chazelle, J. E. Goodman and R. Pollack, Eds.), AMS Press, Providence, RI, 1999, pp. 1–58.

[4] P. K. Agarwal, M. van Kreveld, and M. Overmars, Intersection queries in curved objects, *J. Algorithms*, 15 (1993), 229–266.

[5] L. Arge and J. S. Vitter, Optimal dynamic interval management in external memory, *Proc. IEEE Symp. on Foundations of Comp. Sci.*, 1996, pp. 560–569.

[6] L. Arge, V. Samoladas, and J. S. Vitter. On two-dimensional indexability and optimal range search indexing. In *Proc. ACM Symp. Principles of Database Systems*, pages 346–357, 1999.

[7] L. Arge, D. E. Vengroff, and J. S. Vitter, External-memory algorithms for processing line segments in Geographic Information Systems, *Algorithmica* (to appear in special issues on Geographical Information Systems), extended abstract appears in Proc. of Third European Symposium on Algorithms, 1995.

[8] R. Bayer and E. McCreight, Organization and maintenance of large ordered indexes, *Acta Informatica*, 1 (1972), 173–189.

[9] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, The R*-tree: An efficient and robust access method for points and rectangles, *Proc. SIGMOD Intl. Conf. on Management of Data*, 1990, pp. 322–331.

[10] S. Berchtold, D. A. Keim, and H.-P. Kriegel, The X-tree: An index structure for higher dimensional data, *Proc. 22th VLDB Conference*, 1996, pp. 28–39.

[11] T. M. Chan, Random sampling, halfspace range reporting, and construction of ($\leq k$)-levels in three dimensions, *Proc. 39th IEEE Symp. Foundations of Computer Science*, 1998, pp. 586–595

[12] B. Chazelle, Filtering search: a new approach to query-answering, *SIAM J. Comput.*, 15 (1986), 703–724.

[13] B. Chazelle, R. Cole, F. P. Preparata, and C. K. Yap, New upper bounds for neighbor searching, *Inform. Control*, 68 (1986), 105–124.

[14] B. Chazelle, L. J. Guibas, and D. T. Lee, The power of geometric duality, *BIT*, 25 (1985), 76–90.

[15] B. Chazelle and F. P. Preparata, Halfspace range search: An algorithmic application of $k$-sets, *Discrete Comput. Geom.*, 1 (1986), 83–93.

[16] K. L. Clarkson and P. W. Shor, Applications of random sampling in computational geometry, II, *Discrete Comput. Geom.*, 4 (1989), 387–421.

[17] D. Comer, The ubiquitous B-tree, *ACM Computing Surveys*, 11 (1979), 121–137.

[18] A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, and E. Ramos, Randomized external-memory algorithms for some geometric problems, *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, 1998, pp. 269–268.

[19] T. K. Dey, Improved bounds on planar $k$-sets and related problems, *Discrete Comput. Geom.*, 19 (1998), 373–382.

[20] F. Dumortier, M. Gyssens, and L. Vandeurzen, On the decidability of semi-linearity for semi-algebraic sets and its implications for spatial databases, *Proc. ACM Symp. Principles of Database Systems*, 1997, pp. 68–77.

[21] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, Springer-Verlag, Heidelberg, West Germany, 1987.

[22] H. Edelsbrunner and E. Welzl, Constructing belts in two-dimensional arrangements with applications, *SIAM J. Comput.*, 15 (1986), 271–284.

[23] G. Evangelidis, D. Lomet, and B. Salzberg, The $HB^{\Pi}$-tree: a multi-attribute index supporting concurrency, recovery and node consolidation, *The VLDB Journal*, 6 (1997), 1–25.

[24] P. G. Franciosa and M. Talamo, Time optimal halfplane search on external memory, Unpublished manuscript, 1997.

[25] P. G. Franciosa and M. Talamo, Orders, $k$-sets and fast halfplane search on paged memory, *Proc. Workshop on Orders, Algorithms and Applications*, *Lecture Notes Comput. Sci.*, Vol. 831, Springer-Verlag, 1994, pp. 117–127.

[26] J. Goldstein, R. Ramakrishnan, U. Shaft, and J.-B. Yu, Processing queries by linear constraints, *Proc. ACM Symp. Principles of Database Systems*, 1997, pp. 257–267.

[27] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter, External-memory computational geometry, *Proc. IEEE Symp. on Foundations of Comp. Sci.*, 1993, pp. 714–723.

[28] R. H. Güting, An introduction to spatial database systems, *VLDB Journal*, 4 (1994), 357–399.

[29] A. Guttman, R-trees: A dynamic index structure for spatial searching, *Proc. SIGMOD Intl. Conf. on Management of Data*, 1985, pp. 47–57.

[30] D. Haussler and E. Welzl, Epsilon-nets and simplex range queries, *Discrete Comput. Geom.*, 2 (1987), 127–151.

[31] A. Henrich, Improving the performance of multi-dimensional access structures based on $kd$-trees, *Proc. 12th IEEE Intl. Conf. on Data Engineering*, 1996, pp. 68–74.

[32] E. G. Hoel and H. Samet, A qualitative comparison study of data structures for large line segment databases, *Proc. ACM SIGMOD Conf. on Management of Data*, 1992, pp. 205–214.

[33] I. Kamel and C. Faloutsos, Hilbert R-tree: An improved R-tree using fractals, *Proceedings 20th International Conference on Very Large Databases*, 1994, pp. 500–509.

[34] P. C. Kanellakis, S. Ramaswamy, D. E. Vengroff, and J. S. Vitter, Indexing for data models with constraints and classes, *Journal of Computer and System Sciences*, 52 (1996), 589–612.

[35] D. Lomet and B. Salzberg, The hB-tree: A multiattribute indexing method with good guaranteed performance, *ACM Transactions on Database Systems*, 15 (1990), 625–658.

[36] J. Matoušek, Efficient partition trees, *Discrete Comput. Geom.*, 8 (1992), 315–334.

[37] J. Matoušek, Reporting points in halfspaces, *Comput. Geom. Theory Appl.*, 2 (1992), 169–186.

[38] J. Matoušek, Geometric range searching, *ACM Comput. Surv.*, 26 (1994), 421–461.

[39] K. Mehlhorn, *Multi-dimensional Searching and Computational Geometry*, Springer-Verlag, Heidelberg, West Germany, 1984.

[40] R. Motwani and P. Raghavan, *Randomized Algorithms*, Cambridge University Press, New York, NY, 1995.

[41] J. Nievergelt, H. Hinterberger, and K. Sevcik, The grid file: An adaptable, symmetric multikey file structure, *ACM Transactions on Database Systems*, 9 (1984), 257–276.

[42] J. Nievergelt and P. Widmayer, Spatial data structures: Concepts and design choices, in: *Algorithmic Foundations of GIS* (M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer, eds.), Springer-Verlag, Lecture Notes in Computer Science 1340, 1997.

[43] M. H. Overmars and J. van Leeuwen, Maintenance of configurations in the plane, *J. Comput. Syst. Sci.*, 23 (1981), 166–204.

[44] S. Ramaswamy and S. Subramanian, Path caching: A technique for optimal external searching, *Proc. ACM Symp. Principles of Database Systems*, 1994, pp. 25–35.

[45] J. Robinson, The K-D-B tree: A search structure for large multidimensional dynamic indexes, *Proc. ACM SIGMOD Conf. on Management of Data*, 1984, pp. 10–18.

[46] H. Samet, *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*, Addison Wesley, MA, 1989.

[47] H. Samet, *The Design and Analysis of Spatial Data Structures*, Addison Wesley, MA, 1989.

[48] T. Sellis, N. Roussopoulos, and C. Faloutsos, The $R^+$-tree: A dynamic index for multi-dimensional objects, *Proc. IEEE International Conf. on Very Large Databases*, 1987.

[49] S. Subramanian and S. Ramaswamy, The P-range tree: A new data structure for range searching in secondary memory, *Proc. ACM-SIAM Symp. on Discrete Algorithms*, 1995, pp. 378–387.

[50] G. Tóth, Point sets with many $k$-sets, in preparation.

[51] D. E. Vengroff and J. S. Vitter, Efficient 3-d range searching in external memory, *Proc. ACM Symp. on Theory of Computation*, 1996, pp. 192–201.

[52] J. S. Vitter. Online data structures in external memory. In *Proc. Annual International Colloquium on Automata, Languages, and Programming*, 1999.

[53] D. E. Willard, Polygon retrieval, *SIAM J. Comput.*, 11 (1982), 149–165.