Building An Optimal Point-Location Structure in $O(sort(n))$ I/Os

Xiaocheng Hu Cheng Sheng Yufei Tao

Chinese University of Hong Kong
Hong Kong

September 9, 2018

Abstract

We revisit the problem of constructing an external memory data structure on a planar subdivision formed by n segments to answer point location queries optimally in $O(\log_B n)$ I/Os. The objective is to achieve the I/O cost of $sort(n) = O(\frac{n}{B} \log_{M/B} \frac{n}{B})$, where B is the number of words in a disk block, and M being the number of words in memory. The previous algorithms are able to achieve this either in expectation or under the tall cache assumption of $M \geq B^2$. We present the first algorithm that solves the problem deterministically for all values of M and B satisfying $M \geq 2B$.

Keywords: Point Location Queries, Bulkloading, External Memory, Computational Geometry

Corresponding Author's Address
Yufei Tao
Department of Computer Science and Engineering
Chinese University of Hong Kong
Shatin, New Territories, Hong Kong
Email: *taoyf@cse.cuhk.edu.hk*
Tel: +852-39438437

1 Introduction

In the *point location problem*, we want to preprocess a subdivision of $\mathbb{R}^2$ formed by n segments such that, given a query point q, we can find efficiently the face of the subdivision containing q. See Figure 1a for an example. This is one of the most fundamental problems in computer science, and plays a vital role in numerous applications of spatial databases.

In the literature, nearly all the methods (e.g., [3, 5, 6, 7, 8, 11, 15, 21], to mention just a few) solving this problem target the *vertical ray shooting* (VRS) *problem* defined as follows. The input is a set $\mathcal{S}$ of n *non-intersecting* line segments in $\mathbb{R}^2$, namely, for any two segments $s_1, s_2 \in \mathcal{S}$: (i) either their intersection is empty, or (ii) the intersection is an endpoint of s_1 or s_2. Given a point q, a VRS query finds the highest segment $s \in S$ "below" q. Formally, if we shoot a ray downwards from q, s is the first segment of $\mathcal{S}$ hit by the ray; see Figure 1b. We want to store $\mathcal{S}$ in a data structure such that all such queries can be answered efficiently.

1.1 Computation Model

Our discussion will focus on the standard *external memory* (EM) model [4]. A machine is equipped with M words of memory, and a disk that has been formatted into *blocks* of size B words. An *I/O operation* either reads a disk block into the memory, or writes B words in memory to a disk block. The *cost* of an algorithm is measured in the number of I/Os performed (CPU calculation is for free). The *space* of a structure is the number of disk blocks occupied. The values of M and B satisfy $M \geq \mu B$, where $\mu \geq 2$ is a sufficiently large constant.[1] Finally, define $sort(n) = O(\frac{n}{B} \log_{M/B} \frac{n}{B})$, which is the I/O cost of sorting n elements [4].

1.2 The Quest for An Optimal Structure Constructible in $O(sort(n))$ I/Os

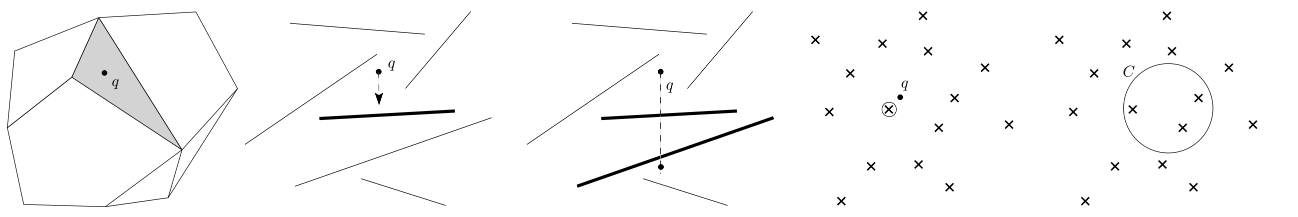
Adapting an internal-memory structure [23], Arge, Danner, and Teh [7] described a structure of linear space $O(n/B)$ that answers a VRS query in $O(\log_B n)$ I/Os. The same structure can also be used to solve the point location problem with the same space and query complexities (by tagging each segment with the face above it). Both complexities are optimal[2].

The grand challenge, however, is construction. Arge, Danner, and Teh [7] gave an algorithm to build their structure using $O(n \log_B n)$ I/Os. Reducing this cost to $O(sort(n))$ has been an intriguing problem ever since, especially given the fundamental nature of the point location problem: a solution will immediately improve the structures for several other problems. Somewhat surprisingly, this problem still remains open in its most generic form, but several solutions have come very close to its final settlement:

- Bender et al. [12] developed an algorithm in the *cache-oblivious model* that solves the problem in $O(sort(n))$ I/Os, under the *tall-cache assumption*, namely, $M \geq B^2$.

[1]In the original model formuation in [4], M can be as small as $2B$. However, any algorithm that works on $M = \mu B$ with constant $\mu > 2$ can be adapted to work on $M = 2B$ with only a constant blowup in space and I/O cost. For this purpose, it suffices to treat each block as μ "micro-blocks", each with B/μ words. Each "logical I/O" now reads or writes a micro-block. A memory of $2B$ words can accommodate μB "micro-blocks", plus B more words that can be used to perform the "physical I/Os" (which are still done in B words each). Whenever a logical I/O is needed on a micro-block, a physical I/O occurs on the block containing the micro-block. Hence, any algorithm with I/O complexity $O(sort(n))$ under $M = \mu B$ now incurs $O(\frac{\mu n}{B} \log_{\frac{\mu M}{B}} \frac{\mu n}{B}) = O(sort(n))$ I/Os on $M = 2B$.

[2]An $\Omega(\log_B n)$ query lower bound can be established via a reduction from *predecessor search* [22].



(a) Point Location (b) VRS (c) Segment intersection (d) NN search (e) Circular Counting

Figure 1: Indexing problems considered in this paper: in (a), the answer is the shaded region; in (b) and (c), the answers are the bold segments; in (d), the answer is the circled cross; in (e), the answer can be any number between $3(1-\epsilon)$ and $3(1+\epsilon)$.

- Perhaps as a folklore, by resorting to ϵ-*nets* [19], one may obtain a randomized algorithm with expected cost $O(sort(n))$, by combining random sampling with I/O-efficient algorithms for computing the *trapezoidal map* and solving the *red-blue segment intersection* problem [9].
- More subtly, the problem can be settled in $O(sort(n))$ I/Os, provided that it is possible to find a so-called M-*partition* on a planar graph (which is a balanced multiway separator; see definition in [20]) of n edges with the same I/O complexity. Unfortunately, the known algorithms [20, 24] for computing M-partitions all make the tall-cache assumption.

In summary, currently there does not exist a *deterministic* algorithm for building an optimal VRS structure in $O(sort(n))$ I/Os for all values of M and B satisfying $M \geq 2B$ (thus eliminating the tall-cache assumption). Note that, besides the obvious scientific merits, overcoming the tall-cache constraint also benefits the data structures that depend on fast VRS construction—now those structures fail to hold unconditionally in the EM model: they must all carry the unpleasant condition "only if $M \geq B^2$".

Meanwhile, $O(sort(n))$-cost deterministic construction (of an optimal VRS structure) are known for special types of inputs. When all the segments in $\mathcal{S}$ are horizontal (i.e., parallel to the x-axis), two algorithms—due to Goodrich et al. [18] and Achakeev and Seeger [1], respectively—are available for this purpose. On the other hand, de Berg et al. [16] explained how to do so when the segments in $\mathcal{S}$ (i) have "low-density", or (ii) define a "fat" subdivision of $\mathbb{R}^2$ (see [16] for the meanings of the quoted terms). Unfortunately, the techniques in [1, 16, 18] rely heavily on the special nature of $\mathcal{S}$, and are not powerful enough to deal with general $\mathcal{S}$.[3]

1.3 Our Results

We present a deterministic algorithm to construct an optimal VRS (hence, optimal point-location) structure in $O(sort(n))$ I/Os on general n non-intersecting segments for all values of M and B allowed by the EM model, settling the open problem.

A crucial idea is to convert, using topological segment sorting [9], the input set $\mathcal{S}$ of segments to a set $\mathcal{S}^H$ of horizontal segments in a rank space. This appears quite natural in retrospect: we already know from the literature how to efficiently create an optimal VRS structure on horizontal segments. The main issue, however, is that a query point in the original space cannot be directly compared to the segments in $\mathcal{S}^H$. We resolve the issue by creating light-weight secondary structures that allow us to carry out the comparisons indirectly in the original space, and then, map the

[3]In particular, as pointed out in [12], the algorithm of [18] incurs $O(n \log_B n)$ I/Os on a general $\mathcal{S}$.

comparison results back to the rank space. Interestingly, we achieve the purpose by using the expensive-to-construct structure of [7] directly as a black box.

The finding yields new *semi-dynamic* structures—which support insertions but not deletions—with insertion cost $O(\frac{1}{B^{1-\delta}} \text{polylog}_B\, n)$, where $\delta < 1$ can be an arbitrarily small positive constant. The following is a short representative list of such problems that are of immediate interest to database systems.

VRS (and Point Location). For this problem, we obtain a structure that uses $O(n/B)$ space, answers a query in $O(\log_B^2 n)$ I/Os, and supports an insertion in $O(\frac{1}{B^{1-\delta}} \log_B^2 n)$ I/Os amortized.

Currently, the best fully dynamic (i.e., supporting both insertions and deletions) structure [6] uses $O(n/B)$ space, ensures $O(\log_B^2 n)$ query cost, and performs an update in $O(\log_B n)$ I/Os amortized, while no improvement is known for the semi-dynamic case.

Vertical Segment Intersection. In this problem, we want to store a set $\mathcal{S}$ of n *non-intersecting* segments in $\mathbb{R}^2$ such that, given a vertical query segment q (i.e., parallel to the y-axis), we can report all the segments in $\mathcal{S}$ intersecting q efficiently. See Figure 1c for an example, and [14] for a detailed account of the problem's database applications.

We obtain a semi-dynamic structure of $O(n/B)$ space that answers a query in $O(\log_B^2 n + k/B)$ I/Os (where k is the number of reported segments), and supports an insertion in $O(\frac{1}{B^{1-\delta}} \log_B^2 n)$ I/Os amortized.

Currently, the best semi-dynamic structure [14] uses $O((n/B)\log B)$ space, answers a query in $O(\log_B^2 n + \log(n/B) + IL^*(B) + k/B)$,[4] supports an insertion in $O(\log_B n + \log B + \frac{1}{B}\log_B^2 n)$ I/Os.

Nearest Neighbor (NN) Search. This is one of the most important problems in spatial databases. We want to store a set $\mathcal{P}$ of n points in $\mathbb{R}^2$ so that, given a query point q, we can efficiently report the point $p \in \mathcal{P}$ whose Euclidean distance to q is the smallest; see Figure 1d.

We obtain a semi-dynamic structure of $O(n/B)$ space that answers a query in $O(\log_B^2 n)$ I/Os, and supports an update in $O(\frac{1}{B^{1-\delta}} \log_B^2 n)$ I/Os amortized.

Currently, the best semi-dynamic structure obtained by combining [8, 18] uses $O(n/B)$ space, answers a query in $O(\log_B^3 n)$ I/Os, and supports an update in $O(\frac{1}{B^{1-\delta}} \log_B^2 n)$ I/Os amortized.

Approximate Circular Counting. In this problem, we want to store a set $\mathcal{P}$ of n points in $\mathbb{R}^2$ so that, given a circle C, we can efficiently report the size $k = |\mathcal{P} \cap C|$ up to an error of ϵk, where $\epsilon < 1$ is a fixed positive value; see Figure 1e. These queries' relevance to spatial databases is evident; e.g., in urban planning we would be interested in: "*how many hospitals are there within 50km from the town center?*".

We obtain a semi-dynamic structure that uses $O_\epsilon((n/B)\,\log n)$ space[5], answers a query with high probability in $O_\epsilon(\log_B^2 n \cdot \log n)$ I/Os, and supports an insertion in $O_\epsilon(\frac{1}{B^{1-\delta}} \log_B^3 n)$ expected I/Os amortized.

In spite of its importance, the problem does not seem to have been studied previously under the EM model. Adapting the best semi-dynamic structure in RAM (combining [2, 13]), one can get a

[4] $IL^*(B)$ is the number of times that we need to repeatedly apply $\log^*$ operation on B before the value becomes $O(1)$.

[5] O_ϵ hides a factor polynomial to $1/\epsilon$.

structure that uses $O_\epsilon(n/B)$ space, but answers a query in $O_\epsilon(\log^2 n)$ I/Os (base is 2), and supports an update in $O_\epsilon(\log^2 n)$ expected I/Os amortized[6]. Although our structure does not always improve this result, it gives an alternative tradeoff which may be interesting in the realistic scenario where n is no more than a polynomial of B.

2 Preliminaries

Conventions on the Input. Let $\mathcal{S}$ be the input set of n non-intersecting segments in $\mathbb{R}^2$. We assume that $\mathcal{S}$ contains the segment $(-\infty, \infty) \times -\infty$, so that no vertical ray-shooting query can return an empty answer. For simplicity, we assume that the endpoints of the segments in $\mathcal{S}$ are in *general position*: no two endpoints share the same x-coordinate. This assumption can be eliminated with standard tie-breaking methods (more specifically: breaking ties by y-coordinate). We further assume that, each segment $s \in \mathcal{S}$ is "open" on its right endpoint, namely, the right endpoint does *not* belong to s. For example, if s has endpoints $(1, 2)$ and $(3, 6)$, we regard s as the set of points $\{(t, 2t) \mid 1 \le t < 3\}$. Given a query point $q = (x, y)$, we ensure returning the correct answer (on the original input) by moving q to the left by an infinitesimal distance δ, and returning the answer with respect to the resulting point $(x - \delta, y)$.

B-Tree. Let S be a set of n real values. We define a *B-tree* T on S as follows. T is a tree parameterized by a *leaf capacity* b and *internal fanout* f. Each leaf node z of T contains between b/c and b elements of S (for some constant $c \ge 4$), unless z is the root of T, in which case the number of elements in z can be anywhere from 1 to b. Every element of S appears in one and only one leaf.

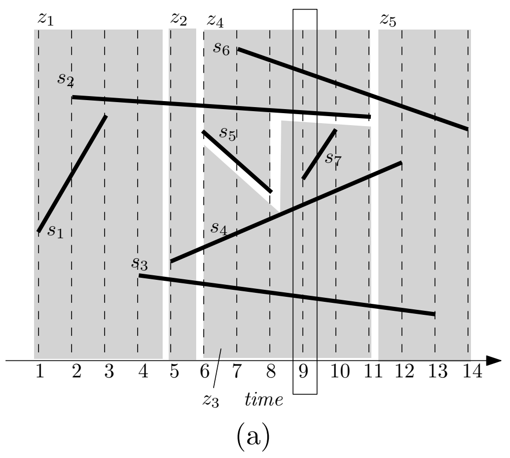
Consider an internal node u of T with g child nodes $v_1, v_2, ..., v_g$. If u is the root, it must hold that $2 \le g \le f$; otherwise, $f/c \le g \le f$. Denote by $sub(u)$ the subtree rooted at u. Node u stores g *routers* $e_1, e_2, ..., e_g$ satisfying:

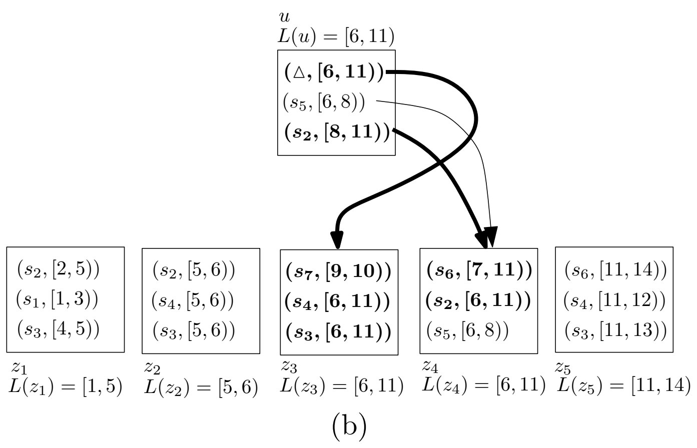
- e_1 is always a *dummy* placeholder $\triangle$.
- Each e_i for $i \ge 2$ must be an element in S; we refer to $e_2, e_3, ..., e_g$ as *non-dummy* routers.
- For each $i \in [2, g]$: all the leaf elements in $sub(v_{i-1})$ are *strictly* smaller than e_i, while all the leaf elements in $sub(v_i)$ are at least e_i.

We require that every element $e \in S$ can serve as a router *at most once*. In other words, e appears in the entire T at least once but at most twice: once definitely in a leaf node, and perhaps another time in an internal node. All leaves are at the same *level* 0. In general, the parent of a level-i node is said to be at level $i + 1$.

Persistent B-Tree. Let $\mathcal{S}$ be a set of n non-intersecting segments in $\mathbb{R}^2$. We will regard the x-axis as the *time dimension*. Let $s \in \mathcal{S}$ be a segment with left and right endpoints (x_1, y_1) and (x_2, y_2), respectively. We define $L(s) = [x_1, x_2)$, and call it the *lifespan* of s. Given an x-coordinate (a.k.a., a *timestamp*) $t \in L(s)$, we define $s(t)$ to be the y-coordinate of the intersection of s and the vertical line $x = t$. We say that s is *alive* at any timestamp $t \in L(s)$. Define $\mathcal{S}(t) = \{s(t) \mid s \in \mathcal{S} \wedge s \text{ alive at } t\}$, that is, $\mathcal{S}(t)$ collects the y-coordinates of all the intersections of the segments in $\mathcal{S}$ with the vertical line $x = t$.

[6]The structure solves a more general problem called *approximate half-space counting* in $\mathbb{R}^3$.





u
$L(u) = [6, 11)$
$(\triangle, [6, 11))$
$(s_5, [6, 8))$
$(s_2, [8, 11))$
$(s_2, [2, 5))$ $(s_1, [1, 3))$ $(s_3, [4, 5))$
$(s_2, [5, 6))$ $(s_4, [5, 6))$ $(s_3, [5, 6))$
$(s_7, [9, 10))$ $(s_4, [6, 11))$ $(s_3, [6, 11))$
$(s_6, [7, 11))$ $(s_2, [6, 11))$ $(s_5, [6, 8))$
$(s_6, [11, 14))$ $(s_4, [11, 12))$ $(s_3, [11, 13))$
z_1 $L(z_1) = [1, 5)$
z_2 $L(z_2) = [5, 6)$
z_3 $L(z_3) = [6, 11)$
z_4 $L(z_4) = [6, 11)$
z_5 $L(z_5) = [11, 14)$
(b)

Figure 2: A persistent B-tree example. Figure (a) shows an input set $\mathcal{S}$ of 7 segments: $s_1, s_2, ..., s_7$; Figure (b) shows the corresponding persistent B-tree $\mathcal{T}$. Each shaded area in Figure (a) shows the region in $\mathbb{R}^2$ that a leaf node is "responsible for". The bold elements and edges in Figure (b) illustrate $\mathcal{T}(9)$ which is a B-tree on $\mathcal{S}(9)$ (see the box with solid edges in Figure (a)).

Also parameterized by a leaf capacity b and internal fanout f, a *persistent B-tree* $\mathcal{T}$ on $\mathcal{S}$ is a directed acyclic graph with parallel edges allowed (i.e., there can be multiple edges between two nodes). Figure 2 shows an example, which the reader may consult in reading the formal definitions below. Every node u in $\mathcal{T}$ is associated with a time interval $L(u)$ (in the form $[t_1, t_2)$), called the *lifespan* of u. If u has no outgoing edges, it is a *leaf* node; otherwise, it is an *internal* node.

A leaf node u stores a set of *leaf elements* e of the form $(seg(e), L(e))$, where $seg(e)$ is a segment $s \in \mathcal{S}$, and $L(e)$ a time interval satisfying $L(e) \subseteq L(s) \cap L(u)$. Consider now an internal node u. For each outgoing edge (u, v) of u in $\mathcal{T}$, u stores an *internal element* e of the form $(seg(e), L(e), ptr(e))$, where

- $seg(e)$ can be either a segment $s \in \mathcal{S}$ or a dummy placeholder $\triangle$.
- $L(e)$ is a time interval satisfying $L(e) \subseteq L(u) \cap L(v)$. Furthermore, if $seg(e)$ is a segment $s \in \mathcal{S}$ (i.e., $seg(e)$ is not dummy), it must additionally hold that $L(e) \subseteq L(s)$.
- $ptr(e)$ a pointer referencing v.

A leaf node has at most b elements, whereas an internal node has at most f elements (this implies that an internal node has out-degree at most f). For each leaf/internal element e such that $seg(e)$ is a segment $s \in \mathcal{S}$, define $e(t) = s(t)$ for any $t \in L(e)$. Similarly, if e is an internal element with $seg(e) = \Delta$, define $e(t) = -\infty$ for any $t \in L(e)$.

We say that a node u (or an element e therein) is *alive* at a timestamp t if $t \in L(u)$ (or $t \in L(e)$, resp.). Likewise, we say that an edge (u, v) in $\mathcal{T}$ is alive at t if its corresponding internal element (stored in u) is alive at t. Define $\mathcal{T}(t)$ to be the subgraph of $\mathcal{T}$ that consists of only the nodes and edges alive at time t. It is required that $\mathcal{T}(t)$ must be a tree. In particular, it must be a B-tree when we look at it in the following manner. First, from each node $u \in \mathcal{T}(t)$, ignore all the elements not alive at t. Second, replace each leaf element $(seg(e), L(e))$ with value $e(t)$, and treat an internal element $(seg(e), L(e), ptr(e))$ as a router $e(t)$. After this conversion, $\mathcal{T}(t)$ must be a B-tree on $\mathcal{S}(t)$ with leaf parameter b and internal fanout f. We refer to $\mathcal{T}(t)$ as the *snapshot B-tree* of $\mathcal{T}$.

If a node u is the root of $\mathcal{T}(t)$ for an arbitrary $t \in L(u)$, then u must be the root of $\mathcal{T}(t)$ for all $t \in L(u)$—this implies that u must have indegree 0; and hence, we call it a *root* of $\mathcal{T}$. Note that there can be multiple roots (recall that $\mathcal{T}$ is a dag). That $\mathcal{T}(t)$ must be a B-tree on $\mathcal{S}(t)$ for each t implies that the roots must have disjoint lifespans (e.g., the persistent B-tree in Figure 2b has 4 roots: z_1, z_2, u, and z_5). Finally, $\mathcal{T}$ is *level consistent*: for any node u in $\mathcal{T}$, the level of u in the snapshot B-tree $\mathcal{T}(t)$ is always the same for all $t \in L(u)$. In other words, we can label the levels of $\mathcal{T}$ still in the bottom up manner (even though $\mathcal{T}$ is a dag): leaves are at level 0; and if a node v is at level i and an edge (u, v) exists, then node u is at level $i + 1$.

Lemma 1 ([7]). *We can build a persistent B-tree with leaf capacity $b \geq B$ and internal fanout $f \geq B$ on n non-intersecting segments in $\mathbb{R}^2$ using $O(n(\frac{b}{B} + \frac{f}{B}\log_f \frac{n}{b}))$ I/Os. The tree consumes $O(n/B)$ space, and has $O(n/b)$ leaf nodes and $O(n/f)$ internal nodes. It supports a VRS query in $O(\frac{b}{B} + \frac{f}{B}\log_f \frac{n}{b})$ I/Os.*

Proof. The algorithm in [7], which incrementally processes the endpoints of the segments in $\mathcal{S}$ in ascending order of x-coordinate, almost achieves the purpose (noticing that reading a leaf and an internal node now takes $O(b/B)$ and $O(f/B)$ I/Os, respectively). The only difference lies in the definition of a B-tree: while we require that every element in the input set must appear in one leaf node, this may not be the case in [7]. The difference can be easily eliminated with straightforward modification of the algorithm in [7]. $\square$

***β*-Subdivision.** Let $\mathcal{S}^H$ be a set of n *horizontal* segments in $\mathbb{R}^2$ with distinct y-coordinates. In other words, each segment in $\mathcal{S}^H$ has the form $[x_1, x_2) \times y$; and no two segments have an identical value for y. Given a value $\beta \in [B, \sqrt{MB}]$, we define a *β-subdivision* of $\mathcal{S}^H$ as a set R of rectangles satisfying the following properties:

- R has $O(n/\beta)$ rectangles of the form $r = [x_1, x_2) \times [y_1, y_2)$; note that both x- and y-projections of r are open on the right.
- The rectangles in R partition $\mathbb{R}^2$, i.e., they are mutually disjoint, and their union is $\mathbb{R}^2$.
- For each rectangle $r \in R$, we define $\mathcal{S}^H(r) = \{s \cap r \mid s \in \mathcal{S}^H \wedge s \text{ intersects } r\}$; namely, for each segment $s \in \mathcal{S}^H$ intersecting r, $\mathcal{S}^H(r)$ collects the portion of s within r. It must hold that $\mathcal{S}^H(r)$ has no more than β segments.
- For each rectangle $r \in R$, define $L(r)$ as the projection of r onto the x-axis. We say that r is *alive* at timestamp t if $t \in L(r)$. Then:
  - Either $\mathcal{S}^H(r)$ has $\Omega(\beta)$ segments,
  - or r is the only rectangle of R alive at timestamp t, for all $t \in L(r)$.

See Figure 3 for an example with $\beta = 3$ on a segment set $\mathcal{S}^H = \{s'_1, s'_2, ..., s'_7\}$, where each shaded region is a rectangle in R.

Lemma 2. *For any $\beta \in [B, \sqrt{MB}]$, we can compute a β-subdivision of a set of n horizontal segments in $\mathbb{R}^2$ using $O(sort(n))$ I/Os. Furthermore, in the same I/O complexity, we can obtain $\mathcal{S}^H(r)$ for each $r \in R$.*

Proof. It is known (e.g., see [17]) that a β-subdivision R can be obtained from the set of leaf nodes of a persistent B-tree on $\mathcal{S}^H$ whose leaf capacity equals β. Each leaf node u defines a rectangle in

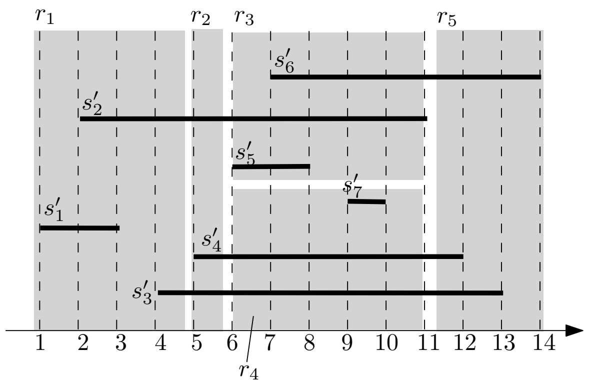


Figure 3: An example of a 3-subdivision

R. Hence, any algorithm capable of building such a persistent B-tree on *horizontal* intervals can be deployed to produce a β-subdivision. The algorithm in [1] (see also [18]) does so in $O(sort(n))$ I/Os.[7]

To prove the second sentence of the lemma, essentially we are facing the following problem: given a set $\mathcal{S}^H$ of n horizontal segments and a set R of $O(n/\beta)$ rectangles, output all pairs $(s, r) \in \mathcal{S}^H \times R$ such that s intersects r. This problem can be solved in $O(sort(n) + k/B)$ I/Os by *distribution sweep* [18], where k is the number of reported pairs. The definition of β-subdivision ensures that $k = O(n)$. □

3 The Proposed Algorithm

This section will present our solution to building an optimal VRS structure on a set $\mathcal{S}$ of n non-intersecting segments in $O(sort(n))$ I/Os.

3.1 The Easy Case: Shallow Inputs

We say that $\mathcal{S}$ is *shallow* if $|\mathcal{S}(t)| \le M/c$ for all $t \in \mathbb{R}$ where c is a sufficiently large constant. In other words, $\mathcal{S}$ contains at most M/c segments that are alive at any timestamp.

Lemma 3. *Given a shallow $\mathcal{S}$ where the endpoints of the segments in $\mathcal{S}$ have been sorted by x-coordinate, using $O(n/B)$ I/Os, we can build a persistent B-tree $\mathcal{T}$ of leaf capacity and internal fanout B on $\mathcal{S}$.*

Proof. It suffices to use the incremental-update algorithm of [7]. Let t be the timestamp of the next endpoint to be inserted/deleted. Since $\mathcal{S}$ is shallow, the entire $\mathcal{T}(t)$ (i.e., the currently alive snaptshot B-tree) fits in memory (the constant c accounts for the space overhead of $\mathcal{T}(t)$ besides storing $\mathcal{S}(t)$). Hence, we can perform the insertion/deletion in memory with no I/Os. □

3.2 Resorting to Topological Segment Sorting

Topological segment sorting on $\mathcal{S}$ is a process that assigns each segment $s \in \mathcal{S}$ a *rank*, which is a distinct integer in $[1, n]$ denoted as $rank(s)$. The ranks must be consistent with the relative ordering

[7]The algorithm of [1] is described with a default leaf capacity of B, but one can replace that with any $\beta \in [B, \sqrt{MB}]$ without affecting the algorithm's correctness. In fact, since what we need here is only the leaf level, the algorithm of [1] can be simplified considerably by ignoring all of its details on producing the non-leaf levels of a persistent B-tree.

of the segments' intersections with any vertical line. Specifically, consider an arbitrary vertical line $\ell : x = t$, and two segments $s_1, s_2 \in \mathcal{S}$ intersecting ℓ. If s_1 intersects ℓ at a point lower than that of s_2, then it must hold that $\mathit{rank}(s_1) < \mathit{rank}(s_2)$. Topological segment sorting can be performed in $O(\mathit{sort}(n))$ I/Os [9].

Let s be a segment in $\mathcal{S}$ with endpoints (x_1, y_1) and (x_2, y_2). We denote by $H(s)$ the horizontal segment $s' = [x_1, x_2) \times \mathit{rank}(s)$. Conversely, we denote by $H^{-1}(s')$ the original segment s. Define

$$\mathcal{S}^H = \{H(s) \mid s \in \mathcal{S}\}.$$

For convenience, we refer to the $\mathbb{R}^2$ space containing the segments in $\mathcal{S}$ as the *original space*, while the $\mathbb{R}^2$ containing the segments of $\mathcal{S}^H$ as the *rank space*.

Set $\beta = \sqrt{MB}$. We use Lemma 2 to compute a β-subdivision R of $\mathcal{S}^H$ in $O(\mathit{sort}(n))$ I/Os. Consider an arbitrary rectangle $r \in R$. Let us focus on $\mathcal{S}^H(r)$; as mentioned previously, for each segment $s' \in \mathcal{S}^H$ intersecting r, $\mathcal{S}^H(r)$ contains the portion of s' in r. We obtain a set $\mathcal{S}(r)$ of segments by converting the segments in $\mathcal{S}^H(r)$ back to the original space: for every such s', let $s = H^{-1}(s')$; and we add to $\mathcal{S}(r)$ the portion of s during the time interval $L(r)$ (i.e., the x-projection of r).

After obtaining the $\mathcal{S}^H(r)$ of all $r \in R$ in $O(\mathit{sort}(n))$ I/Os (Lemma 2), we can produce all the $\mathcal{S}(r)$ easily in the same I/O complexity by pairing each segment $s' \in \mathcal{S}^H(r)$ with its corresponding segment $s \in \mathcal{S}$ using their ids.

As an example, let the input $\mathcal{S}$ consist of the 7 segments in Figure 2a. The segments in Figure 3 constitute one possible set $\mathcal{S}^H$. Consider, for instance, rectangle r_1 in Figure 3, where $\mathcal{S}^H(r_1)$ contains the portions of s'_1, s'_2, s'_3 inside r_1. The corresponding $\mathcal{S}(r_1)$ contains the portions of s_1, s_2, s_3 inside z_1 in Figure 2a.

It follows from the definition of β-subdivision that $|\mathcal{S}(r)| = |\mathcal{S}^H(r)| \le \beta$. We create a persistent B-tree $\mathcal{T}(r)$ with leaf capacity B and branching parameter B on each $\mathcal{S}(r)$ (i.e., one separate tree for each r). Since $\beta < M$, the entire $\mathcal{S}(r)$ can be loaded into memory, so that we can build $\mathcal{T}(r)$ in memory using $O(1 + |\mathcal{S}(r)|/B)$ I/Os. As R (by definition) has $O(n/\beta)$ rectangles in total, overall we spend $O(n/B)$ I/Os building all the $\mathcal{T}(r)$.

We will answer a VRS query with query point q using the $\mathcal{T}(r)$ of exactly one $r \in R$. More specifically, we will do so by identifying the rectangle $r \in R$ such that solving q on $\mathcal{S}(r)$ gives precisely the same answer as solving q on $\mathcal{S}$. In turn, this purpose is achieved by building a search structure on top of the β-subdivision, as explained in the next subsections.

3.3 Persistent B-tree in the Rank Space

Recall that R—the β-subdivision computed in Section 3.2—is a set of rectangles. For each rectangle $r = [x_1, x_2) \times [y_1, y_2)$, define the horizontal segment $[x_1, x_2) \times y_1$ as the *bottom segment* of r, and denote it as $\mathit{bottom}(r)$. Define $E = \{\mathit{bottom}(r) \mid r \in R\}$.

We use Lemma 1 to create a persistent B-tree $\mathcal{T}^H$ with leaf capacity B and branching parameter $F = \sqrt{MB}$ on E (in the rank space). Since E has only $O(n/\beta)$ segments, the cost of constructing $\mathcal{T}^H$ is

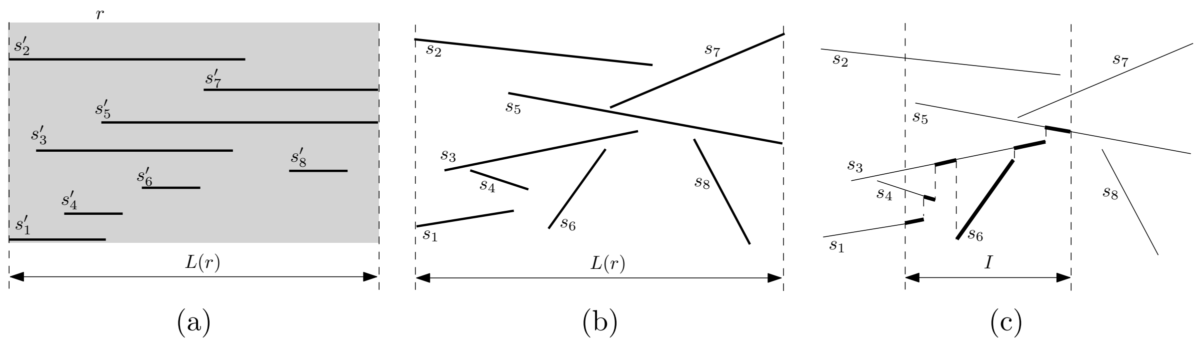


Figure 4: Lower envelope example. Figure (a) shows the segments in $\mathcal{S}^H(r)$ of a rectangle $r \in R$, while Figure (b) shows the segments in $\mathcal{S}(r)$. The bold segments in Figure (c) constitute *envelope*(r, I) for the demonstrated interval I.

$$\begin{aligned} O\left(\frac{n}{\beta}\frac{F}{B}\log_F \frac{n}{\beta}\right) &= O\left(\frac{n}{B}\log_{MB}\frac{n}{\sqrt{MB}}\right) \\ (\text{by } \sqrt{MB} > B) &= O\left(\frac{n}{B}\log_{MB}\frac{n}{B}\right) \\ &= O\left(\frac{n}{B}\log_{M/B}\frac{n}{B}\right). \end{aligned}$$

At first glance, it may appear that $\mathcal{T}^H$ is already ready for query processing. Given a VRS with point q, shouldn't we be able to use $\mathcal{T}^H$ to identify highest edge $\eta \in E$ below q? After that, can't we jump to the rectangle r such that $\eta = bottom(r)$, and solve the query within r? With another moment's thought, one would realize what is wrong: q and E are in different spaces (i.e., original and rank spaces, respectively)! In other words, q cannot be directly compared to the edges in E.

We nevertheless will pursue this seemingly unpromising direction. Our idea is to associate each node u in $\mathcal{T}^H$ with a secondary structure that will help us resolve the comparisons between q and the elements in u. This is done by carrying out the comparisons in the original space, and then "mapping" the results back to the rank space, where the query can proceed to an appropriate child node of u in $\mathcal{T}^H$. In the next subsection, we will introduce a crucial notion that makes this strategy work.

3.4 Lower Envelope

Fix a rectangle r in the β-subdivision R. Consider the set $\mathcal{S}(r)$ of segments (see Section 3.2 for the definition of $\mathcal{S}(r)$). Suppose that we are given a time interval $I = [x_1, x_2)$ such that $I \subseteq L(r)$ (recall that $L(r)$ is the x-projection of r). For every timestamp $t \in I$, define $y_{min}(t)$ as the smallest $s(t)$ of all the segments $s \in \mathcal{S}(r)$ alive at time t. If we increase t from x_1 infinitesimally close to x_2, the value of $y_{min}(t)$ traces out a sequence of segments, which we refer to as the *lower envelope of* r *during* I, and denote it as *envelope*(r, I). See Figure 4 for an example.

Lemma 4. *Both the following are true:*

- *For any $t \in I$, the vertical line $x = t$ intersects envelope(r, I) at only one point.*
- $|envelope(r, I)| \leq 2\beta$.

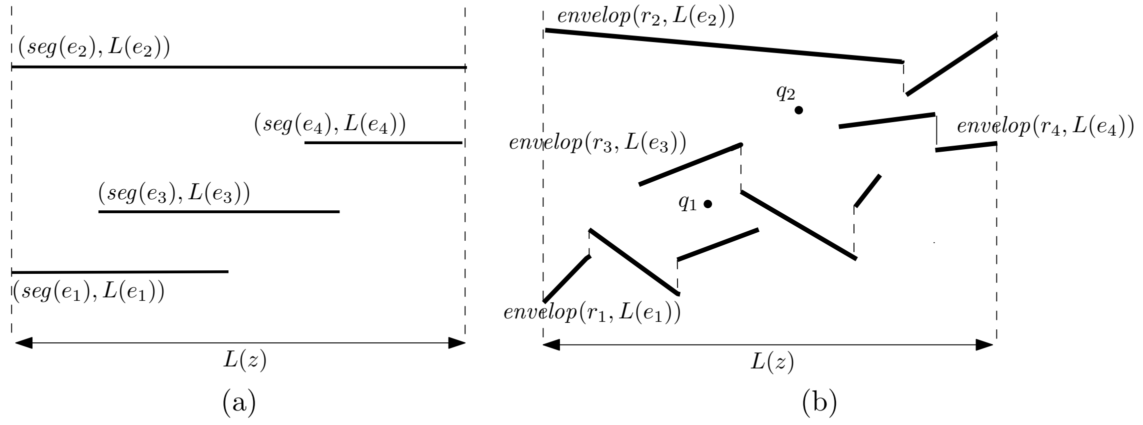


Figure 5: Illustration of *envelope*(z). Figure (a) shows the leaf elements of a leaf node in $\mathcal{T}^H$, and Figure (b) shows the segments in *envelope*(z).

Proof. The first bullet follows immediately from the definition of *envelope*(r, I). To prove the second, note that $\mathcal{S}(r)$ has at most β segments, which have at most 2β distinct endpoints in total—let X be the set of these endpoints' x-coordinates. The second bullet thus follows from the fact that the segments in *envelope*(r, I) have distinct x-coordinates, each of which must come from X. □

3.5 Augmenting $\mathcal{T}^H$ with Secondary Persistent Trees

We are now ready to explain the secondary structures for $\mathcal{T}^H$.

Leaf Node. Let z be a leaf node in $\mathcal{T}^H$. Suppose that it stores g elements $e_1, e_2, ..., e_g$. Recall that each element e_i ($1 \le i \le g$) in z has the form $(seg(e_i), L(e_i))$, where $seg(e_i)$ here is a bottom segment in E, which we denote as η_i. Also recall that η_i comes from a rectangle in R. Let $r_1, r_2, ..., r_g$ be the rectangles in R such that $\eta_i = bottom(r_i)$ for each $i \in [1, g]$.

Define

$$envelope(z) \quad = \quad \bigcup_{i=1}^{g} envelope(r_i, L(e_i)).$$

See Figure 5 for an example.

The following is an important observation:

Lemma 5. *envelope*(z) *is a shallow set of* $O(\beta B)$ *segments. In* $O(sort(|envelope(z)|) + \beta)$ *I/Os, we can construct* *envelope*(z) *and have the endpoints of the segments therein sorted by x-coordinate.*

Proof. $\mathcal{T}^H$ has leaf capacity B. Hence, at any time $t \in L(z)$, there are at most B alive elements in z. Each of these elements produces exactly one alive segment in *envelope*(z). The size bound on $|envelope(z)|$ follows from Lemma 4.

Recall that we have already computed the $\mathcal{S}(r)$ of every $r \in R$. To obtain *envelope*(z), identify the relevant $g \le B$ rectangles $r_1, r_2, ..., r_g$. For each r_i, compute *envelope*$(r_i, L(e_i))$ by loading the whole $\mathcal{S}(r_i)$ in memory—recall that $|\mathcal{S}(r_i)| \le \beta < M$—in

$$O(1 + |\mathcal{S}(r_i)|/B) = O(1 + \beta/B)$$

I/Os. This results in total I/O cost $O(\beta)$. Sorting the endpoints in $envelope(z)$ adds another $O(sort(|envelope(z)|)$ I/Os. □

We can then apply Lemma 3 to construct a persistent B-tree $\mathcal{T}(z)$ with leaf parameter B and internal fanout B on $envelope(z)$ using

$$O(1 + |envelope(z)|/B) = O(\beta)$$

I/Os.

For each leaf element e of $\mathcal{T}(z)$, let $s = seg(e)$; note that s is a segment in $envelope(z)$. Suppose that s comes from $envelope(r_i, L(e_i))$ for some $i \in [1, g]$, we record the index value i along with e. This allows us to jump to r_i in answering a VRS query. More specifically, suppose that, given a query point q, we have already reached $\mathcal{T}(z)$, from which we find s to be the highest segment in $envelope(z)$ below q. The index value i permits us to locate the persistent B-tree $\mathcal{T}(r_i)$ (built in Section 3.2), where the query answer will be found. For instance, in Figure 5b, query point q_1 will lead to the access of $\mathcal{T}(r_1)$, while q_2 will lead to $\mathcal{T}(r_3)$.

Internal Node. Consider an internal node u of $\mathcal{T}^H$. Suppose that u has g elements $e_1, e_2, ..., e_g$. Recall that e_i $(1 \leq i \leq g)$ has the form $(seg(e_i), L(e_i), ptr(e_i))$. Further recall that $seg(e_i)$ is

- either a dummy placeholder $\triangle$,
- or a bottom segment η_i in E (recall that E is the set of bottom segments of the rectangles in R); let r_i be the rectangle in R such that $\eta_i = bottom(r_i)$.

For each $i \in [1, g]$, let v_i be the child node of u referenced by $ptr(e_i)$.

Define $envelope(u)$ as the union of the following for all $i \in [1, g]$:

- a dummy segment $L(e_i) \times -\infty$, if $seg(e_i)$ is dummy;
- $envelope(r_i, L(e_i))$, otherwise.

The following lemma gives a property on u analogous to Lemma 5:

Lemma 6. *$envelope(u)$ is a shallow set of $O(\beta F)$ segments. In $O(sort(|envelope(u)|)+F\beta/B)$ I/Os, we can construct $envelope(u)$ and have the endpoints of the segments therein sorted by x-coordinate.*

Proof. Similar to the previous proof, except that u can have up to F elements. □

We apply Lemma 3 to construct a persistent B-tree $\mathcal{T}(u)$ with leaf parameter B and internal fanout B on $envelope(u)$ using

$$O(1 + |envelope(u)|/B) = O(\beta F/B)$$

I/Os.

For each leaf element e of $\mathcal{T}(u)$, let $s = seg(e)$. We associate with e a pointer determined based on s:

- If s comes from $envelope(r_i, L(e_i))$ for some $i \in [1, g]$. Store $ptr(e_i)$ with e.
- Otherwise, s comes from the dummy segment $L(e_i) \times -\infty$ for some $i \in [1, g]$. Store $ptr(e_i)$ with e.

These pointers allow us to jump to the appropriate child node v_i of u using 1 I/O in answering a VRS query, once s is identified.

Space. We have completed the description of our construction algorithm. Let us now analyze the space consumption of the final structure. It suffices to focus on the secondary structures of $\mathcal{T}^H$ because the rest parts of the structure obviously occupy $O(n/B)$ space. The persistent B-tree on a leaf node of $\mathcal{T}^H$ consumes $O(\beta)$ space, while that of an internal node consumes $O(\beta F/B)$ space. Recall that $\mathcal{T}^H$ is built on E, which has $O(n/\beta)$ segments. By Lemma 1, $\mathcal{T}^H$ has $O(n/(\beta B))$ leaf nodes and $O(n/(\beta F))$ internal nodes. Hence, overall all the secondary structures use $O(n/B)$ space.

Bounding the Construction Cost. From Lemmas 5, 6 and the numbers of leaf/internal nodes in $\mathcal{T}^H$, we know that the total construction cost is bounded by

$$\begin{aligned} & O\left(\frac{n}{\beta B}\beta + \frac{n}{\beta F}\frac{\beta F}{B} + \sum_{\text{each level } i \text{ of } \mathcal{T}^H} \sum_{\text{level-}i \text{ nodes } u} sort(|envelope(u)|)\right) \\ = \; & O\left(n/B + \sum_{\text{each level } i \text{ of } \mathcal{T}^H} \sum_{\text{level-}i \text{ nodes } u} sort(|envelope(u)|)\right). \qquad (1) \end{aligned}$$

Lemma 7.

$$\sum_{\textit{leaf nodes } z \textit{ of } \mathcal{T}^H} |envelope(z)| = O(n) \qquad (2)$$

$$\sum_{\textit{internal nodes } u \textit{ of } \mathcal{T}^H} |envelope(u)| = O(n). \qquad (3)$$

Proof. For each rectangle r in the β-subdivision R, define $envelope^*(r) = envelope(r, L(r))$; recall that $L(r)$ is the x-projection of r. Clearly, $envelope^*(r)$ has at most β segments. Since $|R| = O(n/\beta)$, we know that $\sum_{r\in R} |envelope^*(r)| = O(n)$.

Fix a rectangle $r \in R$. Denote by η the bottom segment of r. Let $e_1, e_2, ..., e_{\lambda(r)}$ be all the elements at the leaf level of $\mathcal{T}^H$ satisfying $seg(e_i) = \eta$ $(1 \le i \le \lambda(r))$, where $\lambda(r)$ is the number of such elements. Note that these elements must have disjoint lifespans. This implies that the segments in $envelope^*(r)$ can contribute at most $|envelope^*(r)| + \lambda(r)$ to the summation in (2). Therefore,

$$\begin{aligned} \sum_{\text{leaf nodes } z} |envelope(z)| & \le \sum_{r \in R} (|envelope^*(r)| + \lambda(r)) \\ & = O(n) \end{aligned}$$

where the last equality used the fact that the total number of leaf elements in $\mathcal{T}^H$ is $O(n/\beta)$.

To prove (3), redefine $\lambda(r)$ to be the number of elements e_i $(1 \le i \le \lambda(r))$ from all the internal levels of $\mathcal{T}^H$ satisfying $seg(e_i) = \eta$. Again, these elements must have disjoint lifespans because the same element can serve as a router only once in any snapshot B-tree (see our B-tree definition in Section 2). Then, (3) follows from the same argument for (2), noticing that the total number of elements at all internal nodes of $\mathcal{T}^H$ is $O(n/\beta)$. $\square$

We thus conclude from (1) that the overall construction cost of $\mathcal{T}^H$ and its secondary structures is $O(sort(n))$.

3.6 Query

It remains to clarify how to use our structure (as built in Sections 3.2-3.5) to answer a VRS query in $O(\log_B n)$ I/Os. Let $q = (x, y)$ be the query point. As mentioned earlier, we will search $\mathcal{T}^H$ to identify a rectangle r in the β-subdivision, and then, solve the query using the persistent B-tree $\mathcal{T}(r)$ on r.

First, identify the root of the snapshot B-tree $\mathcal{T}^H(x)$ responsible for timestamp x. This can be done in $O(\log_B n)$ I/Os by a B-tree on the lifespans of the roots of $\mathcal{T}^H$. In general, suppose that we are currently in an internal node u of $\mathcal{T}^H(x)$. Our mission is to descend to the correct child of u, among those alive at x. To do so, we perform a VRS query using q on the secondary persistent tree $\mathfrak{T}(u)$. Suppose that this query retrieves a segment s; let e be the leaf element of $\mathfrak{T}(u)$ whose *seg*(e) gives this answer s. We then descend to the child node (of u) that is recorded along with e.

The processing at a leaf node z of $\mathcal{T}^H(x)$ is similar. We perform a VRS query using q on $\mathfrak{T}(z)$. Suppose that the search ends up at a leaf element e of $\mathfrak{T}(z)$. We then jump to the rectangle $r \in R$ that is associated with e. Finally, in r, we perform another VRS query using q on $\mathcal{T}(r)$, and return directly this answer.

To analyze the cost, notice that the VRS query at each level of $\mathcal{T}^H(x)$ costs $O(\log_B(\beta F)) = O(\log_B(MB))$ I/Os. On the other hand, $\mathcal{T}^H(x)$ has $O(\log_F(n/\beta)) = O(\log_{MB} n)$ levels. Therefore, the queries at all levels demand $O(\log_B(MB) \cdot \log_{MB} n) = O(\log_B n)$ I/Os. Finally, the query on $\mathcal{T}(r)$ requires another $O(\log_B \beta) = O(\log_B M)$ I/Os. We have now established our main theorem:

Theorem 1. *We can build an $O(n/B)$-size structure on n non-intersecting segments in $\mathbb{R}^2$ using $O(sort(n))$ I/Os, such that a VRS query can be answered in $O(\log_B n)$ I/Os.*

4 Applications

Next, we explain several new results that are made possible by Theorem 1.

For decomposable problems [13][8], the *external logarithmic method* of [8] can be used to make a static structure constructable in $O(\frac{n}{B} \log^\alpha n)$ I/Os semi-dynamic. The resulting structure handles an insertion in $O(\frac{1}{B^{1-\delta}} \log_B^{\alpha+1} n)$ I/Os amortized for any positive constant $\delta < 1$.[9] The space consumption remains the same (as the static structure), but the query time deteriorates by a factor of $O(\log_B n)$. By applying the method to Theorem 1, we obtain a semi-dynamic structure of $O(n/B)$ space that answers a VRS query in $O(\log_B^2 n)$ I/Os, and supports an insertion in $O(\frac{1}{B^{1-\delta}} \log_B^2 n)$ I/Os amortized.

Vertical Segment Intersection. Let $\mathcal{S}$ be a set of n non-intersecting segments. Our structure in Theorem 1 can be readily deployed to answer a vertical segment intersection query in $O(\log_B n + k/B)$ I/Os, where k is the number of reported segments. Nevertheless, next we describe an alternative solution demanding one more persistent B-tree but permitting a much simpler query algorithm.

Consider a query segment connecting points $p_1 = (x, y_1)$ and $p_2 = (x, y_2)$ with $y_1 \le y_2$. By issuing two VRS queries, we can find the segment s_1 immediately above p_1, and the segment s_2

[8]Suppose that we have a problem Π on an input set S. Π is *decomposable* if we can partition S into $S_1, S_2, ..., S_\gamma$ such that, once we have the answer on each S_i ($1 \le i \le \gamma$), we can obtain the answer of S using $O(\gamma)$ additional I/Os.

[9]A weaker insertion cost of $O(\log_B^{\alpha+1} n)$ was claimed in [8]. However, it should be folklore that the cost can be easily improved to as we stated here (for readers familiar with the technique: by creating a structure on $B, B^{1+\delta/2}, B^{1+\delta}, B^{1+3\delta/2}, ...$ elements, respectively).

immediately below p_2. Then, we convert the original vertical segment intersection query into the rank space (obtained by topological segment sorting; see Section 3.2). Specifically, let r_1 and r_2 be the ranks of s_1 and s_2, respectively; let q' be the vertical segment connecting (x, r_1) and (x, r_2). Then, we retrieve all the segments $s \in \mathcal{S}$ such that $H(s)$ intersects q'. This can be done easily in $O(\log_B n + k/B)$ I/Os using an additional persistent B-tree on $\mathcal{S}^H$. Applying the external logarithmic method gives a semi-dynamic structure of $O(n/B)$ space that answers a query in $O(\log_B^2 n + k/B)$ I/Os, and supports an insertion in $O(\frac{1}{B^{1-\delta}} \log_B^2 n)$ I/Os amortized.

Nearest Neighbor Search. By building a point-location structure on the *voronoi diagram* (VD) of n points in $\mathbb{R}^2$, one can answer a NN query in $O(\log_B n)$ I/Os. The structure uses $O(n/B)$ space, and can be built in $O(sort(n))$ I/Os by leveraging Theorem 1 and an algorithm in [18] for computing a VD I/O-efficiently. Applying the external logarithmic method (notice that NN search is decomposable) gives a semi-dynamic structure of $O(n/B)$ space that answers a NN query in $O(\log_B^2 n)$ I/Os, and supports an insertion in $O(\frac{1}{B^{1-\delta}} \log_B^2 n)$ I/Os amortized.

Approximate Circular Counting. Let $\mathcal{P}$ be a set of n points in $\mathbb{R}^2$; given a circle C, a circular emptiness query reports *whether* $\mathcal{P} \cap C$ is empty, i.e., a boolean answer is returned. Such a query is essentially NN search in disguise. Specifically, let q be the center of C. We can first retrieve the NN, say point p, of q in $\mathcal{P}$, and then, compare the distance between p and q to the radius of C. Therefore, we have a structure of $O(n/B)$ space that answers a circular emptiness query in $O(\log_B n)$ I/Os, and can be constructed in $O(sort(n))$ I/Os.

Aronov and Har-Peled [10] showed a general reduction from approximate range counting to range emptiness. Applying their technique to the above structure for circular emptiness gives a structure of $O((n/B) \log n)$ space that answers an approximate circular counting query with high probability in $O(\log_B n \cdot \log n)$ I/Os, and can be constructed in $O(sort(n) \cdot \log n)$ I/Os. Combining this with the external logarithmic method yields a structure of $O((n/B) \log n)$ space that answers a query with high probability in $O(\log_B^2 n \cdot \log n)$ I/Os, and supports an insertion in $O(\frac{1}{B^{1-\delta}} \log_B^3 n)$ I/Os amortized.

References

[1] Daniar Achakeev and Bernhard Seeger. Efficient bulk updates on multiversion B-trees. *Proceedings of the VLDB Endowment (PVLDB)*, 6(14):1834–1845, 2013.

[2] Peyman Afshani and Timothy M. Chan. On approximate range counting and depth. *Discrete & Computational Geometry*, 42(1):3–21, 2009.

[3] Pankaj K. Agarwal, Lars Arge, Gerth Stølting Brodal, and Jeffrey Scott Vitter. I/O-efficient dynamic point location in monotone planar subdivisions. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 11–20, 1999.

[4] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM (CACM)*, 31(9):1116–1127, 1988.

[5] Lars Arge, Gerth Stølting Brodal, and Loukas Georgiadis. Improved dynamic planar point location. In *Proceedings of Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 305–314, 2006.

[6] Lars Arge, Gerth Stølting Brodal, and S. Srinivasa Rao. External memory planar point location with logarithmic updates. *Algorithmica*, 63(1-2):457–475, 2012.

[7] Lars Arge, Andrew Danner, and Sha-Mayn Teh. I/O-efficient point location using persistent B-trees. *ACM Journal of Experimental Algorithmics*, 8, 2003.

[8] Lars Arge and Jan Vahrenhold. I/O-efficient dynamic planar point location. *Computational Geometry*, 29(2):147–162, 2004.

[9] Lars Arge, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory algorithms for processing line segments in geographic information systems. *Algorithmica*, 47(1):1–25, 2007.

[10] Boris Aronov and Sariel Har-Peled. On approximating the depth and related problems. *SIAM Journal of Computing*, 38(3):899–921, 2008.

[11] Hanna Baumgarten, Hermann Jung, and Kurt Mehlhorn. Dynamic point location in general subdivisions. *J. Algorithms*, 17(3):342–380, 1994.

[12] Michael A. Bender, Richard Cole, and Rajeev Raman. Exponential structures for efficient cache-oblivious algorithms. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 195–207, 2002.

[13] Jon Louis Bentley and James B. Saxe. Decomposable searching problems I: Static-to-dynamic transformation. *Journal of Algorithms*, 1(4):301–358, 1980.

[14] Elisa Bertino, Barbara Catania, and Boris Shidlovsky. Towards optimal indexing for segment databases. In *Proceedings of Extending Database Technology (EDBT)*, pages 39–53, 1998.

[15] Siu-Wing Cheng and Ravi Janardan. New results on dynamic planar point location. *SIAM Journal of Computing*, 21(5):972–999, 1992.

[16] Mark de Berg, Herman J. Haverkort, Shripad Thite, and Laura Toma. I/O-efficient map overlay and point location in low-density subdivisions. In *International Symposium on Algorithms and Computation (ISAAC)*, pages 500–511, 2007.

[17] Jochen Van den Bercken, Bernhard Seeger, and Peter Widmayer. A generic approach to bulk loading multidimensional index structures. In *Proceedings of Very Large Data Bases (VLDB)*, pages 406–415, 1997.

[18] Michael T. Goodrich, Jyh-Jong Tsay, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory computational geometry. In *Proceedings of Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 714–723, 1993.

[19] David Haussler and Emo Welzl. Epsilon-nets and simplex range queries. *Discrete & Computational Geometry*, 2:127–151, 1987.

[20] Anil Maheshwari and Norbert Zeh. I/O-efficient planar separators. *SIAM Journal of Computing*, 38(3):767–801, 2008.

[21] Mark H. Overmars. Range searching in a set of line segments. In *Proceedings of Symposium on Computational Geometry (SoCG)*, pages 177–185, 1985.

[22] Mihai Patrascu and Mikkel Thorup. Time-space trade-offs for predecessor search. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 232–240, 2006.

[23] Neil Sarnak and Robert Endre Tarjan. Planar point location using persistent search trees. *Communications of the ACM (CACM)*, 29(7):669–679, 1986.

[24] Freek van Walderveen, Norbert Zeh, and Lars Arge. Multiway simple cycle separators and i/o-efficient algorithms for planar graphs. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 901–918, 2013.