Dynamic Ray Stabbing*

Yufei Tao

Department of Computer Science and Engineering
Chinese University of Hong Kong
Sha Tin, New Territories, Hong Kong
taoyf@cse.cuhk.edu.hk

Abstract

We consider maintaining a dynamic set $\mathcal{S}$ of N horizontal segments in $\mathbb{R}^2$ such that, given a vertical ray Q in $\mathbb{R}^2$, the segments in $\mathcal{S}$ intersecting Q can be reported efficiently. In the external memory model, we give a structure that consumes $O(N/B)$ space, answers a query in $O(\log_B N + K/B)$ time (where K is the number of reported segments), and can be updated in $O(\log_B N)$ amortized time per insertion and deletion. With B set to a constant, the structure also works in internal memory, consuming space $O(N)$, answering a query in $O(\log N + K)$ time, and supporting an update in $O(\log N)$ amortized time.

To appear in ACM Transactions on Algorithms.
A preliminary version appeared in SoCG'12.

*This work was supported by grants GRF 4164/12 and GRF 4165/11 from HKRGC.

1 Introduction

In the *orthogonal ray stabbing problem* (henceforth, the ray stabbing problem), we want to store a set $\mathcal{S}$ of N horizontal segments in $\mathbb{R}^2$ such that, given a vertical ray $Q = x \times (-\infty, y]$, all the segments in $\mathcal{S}$ intersecting Q can be reported efficiently. See Figure 1 for an example. We consider the problem in a fully dynamic setting, where segments can be inserted and deleted in S. Applications of this problem have been described in databases [1, 10, 14], GIS [6], networking [12], and so on. Throughout the paper, every logarithm is assumed to be at least 1. In other words, $\log_x y$ should be understood as $\max\{1, \log_x y\}$.

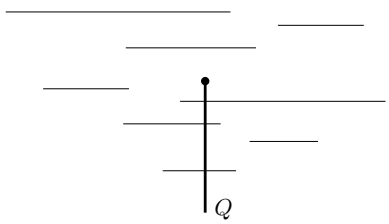

Figure 1: A ray stabbing query that reports 3 segments

Previous Results. The ray stabbing problem is a special instance of the *segment intersection problem*, where a query reports the segments of $\mathcal{S}$ intersecting a vertical segment (instead of a ray). In internal memory, when $\mathcal{S}$ is static, the segment intersection problem can be solved with a persistent binary search tree that uses $O(N)$ space and answers a query in $O(\log N + K)$ time, where K is the number of reported segments. For dynamic data, Cheng and Janardan [9] gave a linear-size structure that solves a query in $O(\log^2 N + K)$ time, and supports an insertion and a deletion in $O(\log N)$ time. Mortensen [13] proposed a structure that has space complexity $O(N \log N / \log\log N)$, $O(\log N + K)$ query time and $O(\log N)$ update time. Blelloch [7] presented a linear-size structure that supports an update in $O(\log N)$ amortized time and a query in $O(\log N + K \log N / \log\log N)$ time.

In the *external memory* (EM) model [2], a machine has a disk formatted into *blocks* of B words, and memory of M words satisfying $M \geq 2B$. Time complexity is measured as the number of I/Os performed, whereas space complexity is measured as the number of disk blocks occupied. *Linear* complexity on an input of size N is interpreted as $O(1 + N/B)$. In this model, the static version of the ray stabbing problem can be settled with the persistent B-tree [5] that consumes linear space and solves a query in $O(\log_B N + K/B)$ I/Os. For the dynamic version, we are not aware of any published result. Perhaps as a folklore, using standard techniques [3, 4, 5], one can obtain a linear-size structure that has query time $O(\log_B^2 N + K/B)$ and supports an update in $O(\log_B N)$ amortized I/Os.

Our Results. We present a structure for solving the ray stabbing problem:

Theorem 1. *For the ray stabbing problem, there is a structure in external memory that uses linear space, answers a query in $O(\log_B N + K/B)$ I/Os, and can be updated in $O(\log_B N)$ amortized I/Os per insertion and deletion.*

The theorem also holds in internal memory by setting B to an appropriate constant.

Technique Overview. Our solution is based on the *external interval tree* [4]. To improve over standard techniques, however, we adopt a different approach to handle the so-called "left segments". Conventionally, a query reports the qualifying left segments of a node when the node is reached, which necessitates $\Omega(\log_B^2 N)$ query time. To break this barrier, we take a delayed approach which does not report a left segment until its left endpoint has "departed" from the query, namely, the endpoint is no longer on the

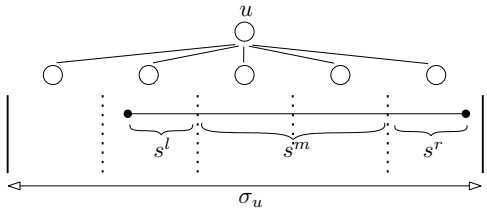

Figure 2: A segment s in the stabbing set of node u

search path. With a filtering-search argument, we can charge most of the cost incurred at a node of the search path on the reported segments, which is the key to achieving logarithmic query time.

Central to our techniques is a new notion called *bottom set*. Concerning the interval tree (or its external version), intuitively, a bottom set includes some low segments (by y-coordinate) among those that are stored at an internal node. Each node can have a large number of bottom sets, each of which is defined with respect to a distinct descendant of the node. A side message delivered by this paper is that bottom sets may equip us with an extra weapon in solving problems (e.g., ray stabbing) for which the interval tree appears to be the right structure. Because of this, the mechanism presented in this paper for maintaining bottom sets may be of independent interest, especially given the fact that it handles an update in logarithmic amortized time.

2 A Static Structure

This section describes a linear-size, static, structure that answers a query in $O(\log_B N + K/B)$ I/Os. Although the same performance can also be achieved by a persistent B-tree, our structure manages segments in an alternative manner. We will make the structure fully dynamic in the later sections, but its static version illustrates some central ideas behind our techniques without the distraction from update handling.

2.1 Structure

Given horizontal segments s_1 and s_2, we say that s_1 is *lower* if it has a smaller y-coordinate than s_2; otherwise, s_1 is *higher*. Also, we will refer to the x-coordinate of a segment's left endpoint simply as its *left x-coordinate*. As before, let $\mathcal{S}$ be the input set of data segments, and $N = |\mathcal{S}|$.

Base Tree. In our structure, the base tree is a B-tree $\mathcal{T}$ indexing the x-coordinates of the segments in $\mathcal{S}$. All the x-coordinates are stored at the leaf level. A leaf node contains $\Theta(b)$ coordinates where $b = B \log_B^3 N$, while an internal node has $\Theta(f)$ child nodes where $f = \max\{B^{1/3}, (\log_B N)^{1/3}\}$. A leaf node is at *level* 0; and in general, the parent of a level-l node is at level $l + 1$.

Naturally, a total order exists on the leaf nodes. The *slab* of a leaf node z is an interval $[x, x')$ where x and x' are the smallest x-coordinates stored in z and the leaf node succeeding z, respectively ($x' = \infty$ if no leaf succeeds z). We denote the slab as σ_z. The slab σ_u of an internal node u, in general, is the union of the slabs of its child nodes. In $\mathbb{R}^2$, a slab σ corresponds to a vertical strip, enclosing all the points (of the data space) with x-coordinates in σ. Henceforth, we will interpret a slab as a strip.

As in the external interval tree [4], each segment $s \in \mathcal{S}$ is assigned to the lowest node u whose slab covers s. The set of segments assigned to u is the *stabbing set* of u, denoted as $stab(u)$.

Now, consider u as an internal node. We refer to σ_v as a *child slab* of u if v is a child node of u. As in the external interval tree, every segment $s \in stab(u)$ defines a *left segment* s^l, a *right segment* s^r, and perhaps a *middle segment* s^m. Specifically, s^l (s^r) is the part of s in the only child slab of u covering the left (right) endpoint of s. On the other hand, s^m is the remainder of s after trimming s^l and s^r. See Figure 2. Let $\mathcal{L}_u, \mathcal{R}_u$ and $\mathcal{M}_u$ be the sets of left, right and middle segments at u, respectively. $\mathcal{L}_u, \mathcal{R}_u$ and $\mathcal{M}_u$ are

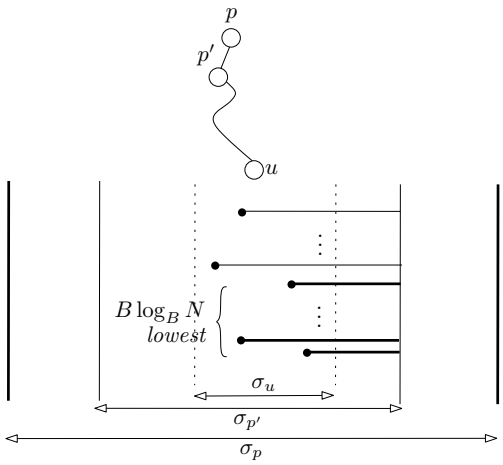

Figure 3: Illustration of $bottom_u(p)$: The segments shown are the left segments at p whose left endpoints are in σ_u. The $B \log_B N$ lowest ones constitute $bottom_u(p)$. All these segments are grounded on the right boundary of $\sigma_{p'}$, where p' is a child node of p and a proper ascenstor of u.

managed by secondary structures. We will focus on $\mathcal{L}_u$ in this section. Symmetric structures exist for $\mathcal{R}_u$, whereas the management of $\mathcal{M}_u$ is left to Section 3.

Bottom Set. Let u be a leaf/internal node in $\mathcal{T}$, and $parent(u)$ be its parent. Consider a proper ancestor p of $parent(u)$. Clearly, σ_u is within σ_p. Some left segments at p may have their left endpoints covered by σ_u. Among them, we are interested in the few *lowest* ones. Let $bottom_u(p)$ be a set consisting of the $B \log_B N$ lowest segments in $\mathcal{L}_p$ whose left endpoints are in σ_u. If less than $B \log_B N$ such segments exist, $bottom_u(p)$ includes all of them. We will refer to $bottom_u(p)$ as the *bottom set of p at u*. See Figure 3. Note that the definition of $bottom_u(p)$ requires that the levels of p and u differ by at least 2.

Now, consider u as an internal node with child nodes $v_1, ..., v_f$. For each $i \in [1, f]$, define:

$$\pi_u(i) \quad = \quad \bigcup_{\forall \text{ proper ancestor } p \text{ of } u} bottom_{v_i}(p). \tag{1}$$

Let $\psi_u(i)$ be the set of B lowest segments in $\pi_u(i)$. If $|\pi_u(i)| < B$, $\psi_u(i)$ includes all the segments in $\pi_u(i)$.

Secondary Structures. An internal node u with children $v_1, ..., v_f$ is associated with:

- for each v_i, a B-tree $\Pi_u(i)$ on the y-coordinates of the segments in $\pi_u(i)$. We call $\Pi_u(i)$ a *bottom structure*.
- (only if the level of u is at least 2) for each v_i, a *priority search tree* [3] $F_u(i)$ on the left endpoints in $\mathcal{L}_u$ that appear in σ_{v_i}. $F_u(i)$ answers *lower-open* 3-sided range queries efficiently[1]. Such a query reports all the endpoints in a rectangle of the form $[x_1, x_2] \times (-\infty, y_2]$, i.e., the lower edge of the rectangle is grounded on the bottom of $\mathbb{R}^2$. We call $F_u(i)$ a *flank structure*.
- a priority search tree P_u on the left endpoints in $\psi_u(1) \cup ... \cup \psi_u(f)$. P_u is also for answering lower-open 3-sided range queries. We call P_u a *pilot structure*.

[1]A priority search tree on N 2d points consumes $O(N/B)$ space. Given a 3-sided axis-parallel rectangle, it can be used to report all the points in the rectangle in $O(\log_B N + K/B)$ I/Os, where K is the number of such points. It supports an update in $O(\log_B N)$ time per insertion and deletion.

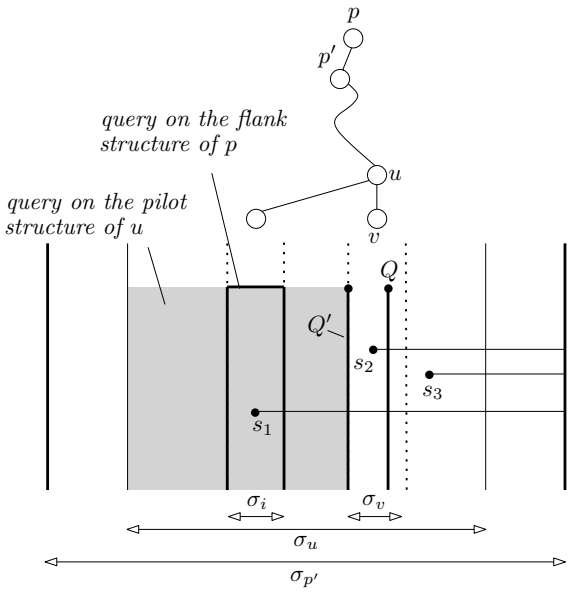

Figure 4: Query processing at an internal node u: s_1, s_2 and s_3 are left segments at p. They are in the spanning, pending, and disposable groups of p, respectively.

Consider a leaf node z of $\mathcal{T}$. If a segment $s \in \mathcal{S}$ has its left x-coordinate in z, we store s at z. Note that z stores at most b segments.

Space. $\mathcal{T}$ itself obviously occupies $O(N/B)$ space. It has height $h = O(\log_f N)$. A bottom structure indexes at most $hB \log_B N$ segments in $O(h \log_B N)$ blocks. An internal node has f bottom structures using $O(fh \log_B N)$ space in total. The node's pilot structure stores at most fB points in $O(f)$ blocks. Since there are $O(N/(fb))$ internal nodes, all the bottom and pilot structures require $O(\frac{N}{fb} fh \log_B N) = O((N/b) \log_B^2 N) = o(N/B)$ space in total. A flank structure consumes linear space. The left endpoint of a data segment is stored in at most one flank structure. Hence, all the flank structures use $O(N/B)$ space. Finally, each segment is stored constant times at the leaf level. Overall, our structure occupies linear space.

2.2 Query

Given a query $Q = x \times (-\infty, y]$, we visit a root-to-leaf path of $\mathcal{T}$, formed by the nodes whose slabs contain Q. Refer to the path as the *query path*. We will report segments only from the secondary structures of the nodes on this path. For a data segment, only one of its left, right and middle segments can intersect Q. Hence, we concentrate on reporting the left, right and middle segments, respectively, without worrying that a data segment may be reported twice. Next, we discuss first left segments.

Internal Node. We examine the nodes of the query path in top-down order. Suppose that we are currently at an internal node u, which has f child slabs $\sigma_1, ..., \sigma_f$. Let v be the child node of u on the query path. Hence, Q is contained in σ_v (which is one of $\sigma_1, ...\sigma_f$).

Consider any proper ancestor p of u. Let S be the set of segments in $\mathcal{L}_p$ whose left endpoints are in σ_u. Based on left endpoints, S can be divided into three disjoint groups as follows. The *spanning group* includes the segments of S whose left endpoints are to the left of σ_v. Every such segment must span σ_v, and hence the group's name. The *pending group*, on the other hand, includes the segments whose left endpoints are in σ_v. In the final *disposable group*, all the segments are to the right of σ_v, and thus can be safely ignored. Figure 4 illustrates this with s_1, s_2 and s_3.

When the processing finishes at u, we ensure an invariant that *all* the qualifying segments in the spanning group of p have been reported. There is no such guarantee for the pending group. Nevertheless, as we descend further along the query path, the qualifying segments in the pending group either will appear in a spanning group later, or are stored in the path's leaf node. Our invariant implies that we will find those segments in the former situation, whereas in the latter we will also find them by inspecting the leaf node. No left segment can be reported twice, noticing that the spanning group of p is completely different if u changes. Specifically, as we move from u to its child node on the query path, the pending group at u will unfold into the spanning, pending and disposable groups at its child node.

Next, we explain how to report segments from the spanning groups of the proper ancestors of u. Obtain a ray Q' by snapping Q onto the left boundary of σ_v. Namely, the resulting ray Q' is $x' \times (-\infty, y]$, where x' is the x-coordinate of the left boundary of σ_v. No segment in the pending group of any proper ancestor of u can intersect Q'. On the other hand, each qualifying segment from a spanning group must intersect Q'; its left endpoint must lie in rectangle $(-\infty, x'] \times (-\infty, y]$. See the gray area in Figure 4.

We search the pilot structure of u to retrieve all the left endpoints covered by $(-\infty, x'] \times (-\infty, y]$. Recall that every such endpoint belongs to a left segment in $\psi_u(i)$ for some $i \in [1, f]$. The segments whose endpoints are retrieved definitely satisfy the query, and hence, are reported. If less than B segments from $\psi_u(i)$ are found, no other segment in $\pi_u(i)$ can intersect Q' because $\psi_u(i)$ collects the lowest segments of $\pi_u(i)$. In this case, $\pi_u(i)$ is pruned from further consideration.

If, on the other hand, B segments from $\psi_u(i)$ have been reported, we have earned ourselves a *free I/O*, namely, an extra I/O that we can charge on those B segments without worrying that the output cost can become super-linear—the standard idea of *filtering search* [8]. We spend this I/O jumping to the first leaf node of the bottom structure $\Pi_u(i)$. From there, scan the segments of $\pi_u(i)$ in ascending order of y-coordinate. Keep reporting the segments[2] until encountering the first one that does not intersect Q', i.e., the first segment whose y-coordinate is greater than y.

Every segment in $\pi_u(i)$ is a left segment at some proper ancestor p of u. If less than $B \log_B N$ segments from $\mathcal{L}_p$ have been reported from $\pi_u(i)$, we have found all the segments from $\mathcal{L}_p$ whose left endpoints are in σ_i, because the lowest $B \log_B N$ of those segments are in $\pi_u(i)$. Otherwise, $B \log_B N$ segments from $\mathcal{L}_p$ have been reported. In this case, we have earned ourselves $O(\log_B N)$ free I/Os, which are utilized to launch a 3-sided range query to report the other segments of $\mathcal{L}_p$ with left endpoints in σ_i, as explained next.

Recall that p has f flank structures, each corresponding to a different child node of p. Let F_p be the flank structure for the child node of p on the query path. We formulate a 3-sided rectangle r that has the same x-range as σ_i, and has the same y-range as Q'. See Figure 4. A segment in $\mathcal{L}(p)$ intersects Q' and has its left endpoint in σ_i, if and only if r covers its left endpoint. The cost of answering the 3-sided range query on F_p is $O(\log_B N)$ plus the linear output time.

Leaf Node. At the leaf node z of the query path, we simply report all the qualifying segments stored in z.

A Sentinel Trick. As mentioned before, in searching an internal node u, we need to know whether B segments from a $\psi_u(i)$ have been reported in the pilot structure, for each $i \in [1, f]$. Since f may be greater than $2B$ (the smallest amount of memory that may be available), we cannot simply keep a counter for each i in memory. A similar situation occurs when we are scanning through $\pi_u(i)$ for some i—we need to know whether $B \log_B N$ segments from the $\mathcal{L}_p$ of a proper ancestor p of u have been found. Again, as there can be $O(h) = O(\log_f N)$ different p, we cannot keep a memory-resident counter for each p if h is far greater than $2B$. Maintaining the counters in external memory would lead to cost penalty we cannot afford.

We resolve this issue by marking some segments as *sentinels* such that enough segments (from a certain set) have been reported if and only if a sentinel is reported. Specifically, in the pilot structure, the sentinel

[2]Of course, skip the first B segments in $\pi_u(i)$ because they have been reported from $\psi_u(i)$. Henceforth, we will not discuss trivial duplicate removal like this.

from $\psi_u(i)$ is the B-th lowest segment in $\psi_u(i)$; if $\psi_u(i)$ has less than B segments, no sentinel is marked. Likewise, in a $\pi_u(i)$, the sentinel from an $\mathcal{L}_p$ is the $(B \log_B N)$-th lowest segment in $\mathcal{L}_p$, or undefined if there are less than $B \log_B N$ segments in $\mathcal{L}_p$.

Cost. The query path has $O(\log_f N)$ nodes. At each internal node u of the path, searching its pilot structure requires $O(\log_B f)$ I/Os (recall that the pilot structure indexes at most fB points). The other cost at u can be charged on the output cost, as discussed earlier. Hence, overall, only $O(h \log_B f) = O(\log_B N)$ I/Os has not been charged on the output. At the leaf node of the query path, we spend $O(b/B)$ I/Os reading its contents. Therefore, the total query cost is $O(b/B + \log_B N)$ plus the linear output time. A symmetric algorithm can be used to retrieve the qualifying right segments. We have not discussed the reporting of middle segments. As will be shown later, this can be done in $O(\log_B N)$ I/Os plus the linear output time, by augmenting our current structure with extra information occupying $O(N/B)$ space.

2.3 Bootstrapping

We have obtained a linear-space structure whose query complexity is $O(b/B+\log_B N) = O(\log_B^3 N)$, plus the linear output time. The term b/B arises from finding the qualifying segments in a leaf node z. Currently, no specialized structure exists on the segments in z. Now, let us bootstrap by indexing those segments with the structure described above. As z stores at most b segments, those satisfying a query can be reported in $O(\log_B^3 b) = O((\log_B \log_B N)^3)$ I/Os plus the linear output time. Therefore, the overall query cost is improved to $O((\log_B \log_B N)^3 + \log_B N + K/B) = O(\log_B N + K/B)$. The space consumption of the bootstrapped structure remains linear.

3 The Middle-Indexing Problem

As before, let $\mathcal{S}$ be a set of N horizontal segments. Let L be a set of vertical lines $\{\ell_1, ..., \ell_f\}$ where $f = \Theta(\max\{B^{1/3}, (\log_B N)^{1/3}\})$. ℓ_i is to the left of ℓ_j for any i, j with $1 \leq i < j \leq f$. For each $i \in [1, f-1]$, refer to the vertical strip between ℓ_i and ℓ_{i+1} as a *slab*. A *multi-slab* is the union of a non-empty set of *consecutive* slabs. There are $f-1$ slabs, and hence, $f(f-1)/2$ multi-slabs (note that a slab is also a multi-slab).

All segments of $\mathcal{S}$ lie between ℓ_1 and ℓ_f. A segment $s \in \mathcal{S}$ is *long* if it spans at least one multi-slab, i.e., its x-range covers that of the multi-slab. Otherwise, s is *short*. Every short segment is required to intersect one of $\ell_2, ..., \ell_{f-1}$. A long segment s defines a middle segment, which is the part of s in the widest multi-slab it spans. Denote by $\mathcal{M}$ the set of middle segments thus defined. The goal is to store $\mathcal{S}$ in a structure so that, given a vertical ray Q, all the segments in $\mathcal{M}$ intersecting Q can be reported efficiently (i.e., we are interested in retrieving only middle segments). We refer to the above problem as the *middle-indexing problem*.

Lemma 1. *In the middle-indexing problem, there is a structure of $O(f + N/B)$ space such that a query can be answered in $O(\log_B f + K/B)$ I/Os. The structure can be updated in $O(\log_B N)$ amortized I/Os per insertion and deletion.*

We have encapsulated some standard details into lemmas (such as the above one), whose proofs can be found in the appendix.

Completing the Static Structure. Refer to the structure of Lemma 1 as a *middle structure*. In the static structure of Section 2, each internal node u in the base tree has a stabbing set $stab(u)$. We use a middle structure to manage the segments of $stab(u)$, with L being the set of boundary lines of the child slabs of u. In answering a query Q, we report the qualifying middle segments by searching the middle structures of the internal nodes along the query path. As there are $O(\log_f N)$ such nodes, by Lemma 1, the total cost is $O(\log_f N \cdot \log_B f) = O(\log_B N)$ plus the linear output time.

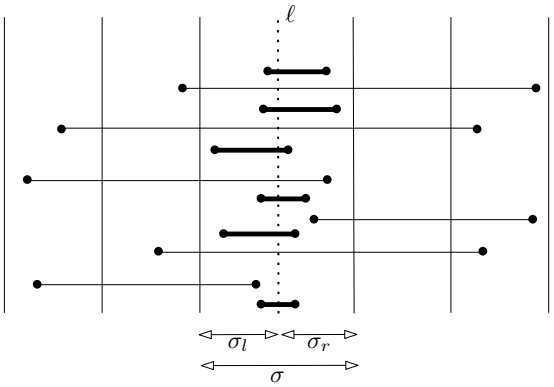

Figure 5: A refine operation: The solid vertical lines constitute the current L. S_{new} consists of the thick segments. The other segments are in the current $\mathcal{S}$.

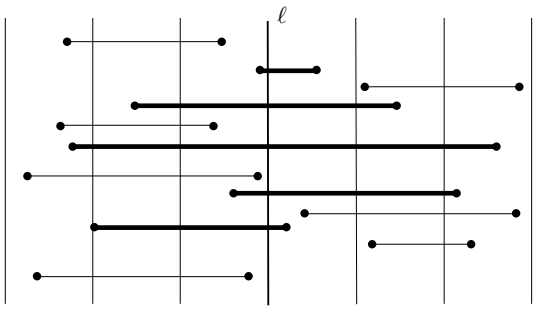

Figure 6: A split operation: S_{cross} consists of the thick segments.

Each middle structure uses $O(f)$ extra blocks, on top of space linear to the number of segments indexed. There are $o(N/B)$ extra blocks for all middle structures because the base tree has $O(N/(bf))$ internal nodes. Hence, the overall space consumption is still linear.

Refinement. Next, we describe a *refine* operation which will be useful later. In this operation, a slab σ is divided into two slabs σ_l and σ_r by a vertical line ℓ, where σ_l is on the left of ℓ. Accordingly, ℓ is added to L. Furthermore, ℓ is associated with a set S_{new} of horizontal segments, all of which are inside σ. See Figure 5. S_{new} is to be incorporated into the dataset $\mathcal{S}$. S_{new} is given in two copies where its segments are sorted by x- and y-coordinate, respectively. The objective is to update the middle structure to index the resulting $\mathcal{S}$ and L.

Let α be the total number of segments in the final $\mathcal{S}$ having an endpoint in σ. In other words, α equals the sum of $|S_{new}|$ and the number of such segments in the original $\mathcal{S}$. The lemma below explains the cost of refinement:

Lemma 2. *A refine operation can be performed in $O(f^2 \log_2 f + \alpha/B)$ I/Os.*

Split. Let ℓ be a line in L. A *split* operation cuts S and L into S_1, S_2 and L_1, L_2, respectively. Specifically, S_1 (S_2) includes all the segments in S entirely on the left (right) of ℓ, whereas L_1 (L_2) includes all the lines of L on the left (right) of ℓ. Let S_{cross} be the segments of S intersecting ℓ. See Figure 6. The operation has two goals. First, output S_{cross} in two copies, with its segments sorted by x- and y-coordinate, respectively. Second, create middle structures for the two middle-indexing problems defined on (S_1, L_1) and (S_2, L_2), respectively. As for the efficiency of the operation, we have:

Lemma 3. *A split operation can be performed in $O((N/B) \log_2 f + f^2 \log_2 f)$ I/Os.*

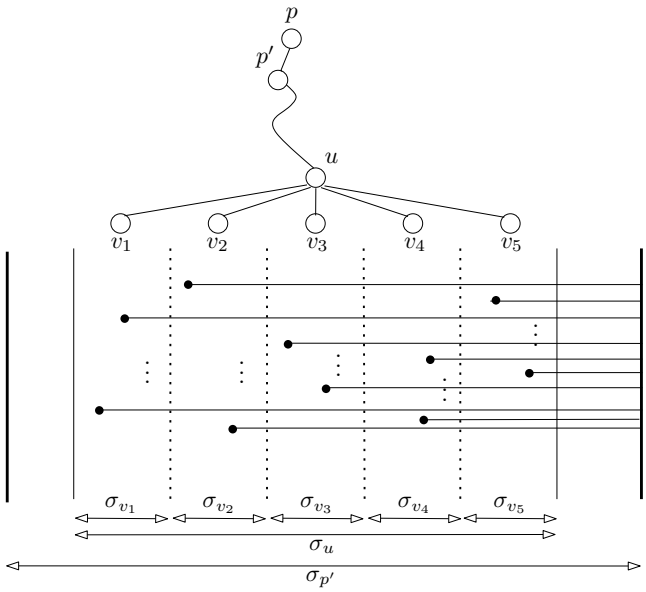

Figure 7: Illustration of $\gamma_u(p)$: For each $bottom_{v_i}(p)$, only the highest and lowest segments are shown, whereas the other segments are omitted. All these segments together constitute $\gamma_u(p)$.

4 A Dynamic Structure

In this section, we make our structure in Section 2 dynamic. The main challenge is to efficiently maintain bottom sets. In general, changes to bottom sets may occur for two reasons. First, segments are moved from a stabbing set to another during tree rebalancing. Second, a newly inserted segment enters a bottom set, or an existing segment there is deleted. We will describe techniques to handle these situations.

4.1 Structure

We now implement the base tree $\mathcal{T}$ as a *weight-balanced B-tree* [4], where a leaf node stores $\Theta(b)$ x-coordinates and an internal node has $\Theta(f)$ child nodes, with b and f defined as in Section 2. All the secondary structures in Sections 2 and 3 are inherited by the nodes of $\mathcal{T}$. Next, we describe the additional structures required on left segments. Symmetric structures exist for right segments, whereas the middle structure has been completely presented in the previous section. Let $h = O(\log_f N)$ be the height of $\mathcal{T}$.

Let u be an internal/leaf node of $\mathcal{T}$, and p a proper ancestor of $parent(u)$. Recall that $bottom_u(p)$, the bottom set of p at u, is the set of the $B \log_B N$ lowest left segments at p whose left endpoints are in σ_u. We maintain a B-tree indexing the segments of $bottom_u(p)$ by y-coordinate. $\mathcal{T}$ has $O(N/b)$ nodes, each of which has $O(h)$ proper ancestors. Hence, the B-trees on all the bottom sets occupy $O(\frac{N}{b} h \log_B N) = o(N/B)$ space in total.

Now, consider u instead as an internal node with child nodes $v_1, ..., v_f$. Define:

$$\gamma_u(p) = \bigcup_{i=1}^{f} bottom_{v_i}(p). \tag{2}$$

See Figure 7. Notice that, if $bottom_u(p)$ is defined (i.e., the levels of p and u differ by at least 2), the segments of $bottom_u(p)$ are in fact the $B \log_B N$ lowest in $\gamma_u(p)$.

We associate u with a (slightly augmented) B-tree $\Gamma_u(p)$, which indexes the y-coordinates of the segments in $\gamma_u(p)$. Given an integer k, $\Gamma_u(p)$ allows us to find efficiently the k-th lowest segment in

$\gamma_u(p)$. Refer to $\Gamma_u(p)$ as a *rank B-tree*. As $\gamma_u(p)$ has at most $fB\log_B N$ segments, a rank B-tree occupies $O(f\log_B N)$ blocks. There are $O(N/(bf))$ internal nodes, each of which has $h = O(\log_f N)$ proper ancestors. Hence, the total space of all rank B-trees is $O(\frac{Nh}{bf} f\log_B N) = o(N/B)$.

Let z be a leaf node in $\mathcal{T}$, and p a proper ancestor of z. Define:

$$\phi_z(p) = \text{the set of left segments at } p \text{ whose left endpoints are in } \sigma_z.$$

If $bottom_z(p)$ is defined, the $B\log_B N$ lowest segments of $\phi_z(p)$ are exactly the segments in $bottom_z(p)$. We maintain a B-tree on the y-coordinates of the segments in $\phi_z(p)$. The B-trees on the $\phi_z(p)$ of all p together occupy $O(h + b/B) = O(b/B)$ space, because only b endpoints are in σ_z and z has $O(h)$ proper ancestors. Thus, all these B-trees require $O(N/B)$ blocks in total. Finally, there is a B-tree on the y-coordinates of the segments stored in z—recall that every such segment has at least one x-coordinate in z. The space of our overall structure still remains linear.

The query algorithm remains the same as in the static structure. The cost is $O(b/B + \log_B N + K/B)$, where the b/B term stems from reading the segments stored in the leaf node of the query path. We will eliminate the term by bootstrapping later.

4.2 Updates

In the sequel, whenever we insert or delete a segment in some $bottom_u(p)$, the B-tree on $bottom_u(p)$ is updated accordingly. The same convention applies to $\phi_z(p)$, and the segment set stored in a leaf node. Recall that our structure uses some sentinels to facilitate query processing. We, however, will not elaborate on the maintenance of sentinels because the relevant details are straightforward extensions to the update procedures presented below.

Insertion. To insert a segment s, first update the base tree $\mathcal{T}$ by inserting the x-coordinates of s. This may trigger one or more node splits in $\mathcal{T}$. Deferring node split handling to Section 4.3, we now proceed assuming that all the splits have been taken care of. Let p be the node whose stabbing set $stab(p)$ contains s. If p is a leaf, we finish by storing s there. Next, we consider that p is an internal node.

Insert s into the middle structure of p in $O(\log_B N)$ I/Os (Lemma 1). Now, consider the left segment s^l of s (a similar algorithm applies to its right segment). Let φ be the path from p to the leaf node of $\mathcal{T}$ that stores the left x-coordinate of s^l. Recall that p has a flank structure for each child. In $O(\log_B N)$ I/Os, we insert the left endpoint of s^l in the flank structure of p for its child on φ.

Let u be an internal node that is a proper descendant of p on φ. Let v be the child of u on φ. Recall that u has a bottom structure for v; suppose that the structure is $\Pi_u(i)$ for some $i \in [1, f]$. Whether $\Pi_u(i)$ needs to be updated depends on whether s^l enters $bottom_v(p)$. This is also true for the rank B-tree $\Gamma_u(p)$. There are two situations where s^l should be added to $bottom_v(p)$:

- first, $bottom_v(p)$ has less than $B\log_B N$ segments;
- second, s^l is lower than the highest segment, say s', in $bottom_v(p)$.

Using the B-tree on $bottom_v(p)$, which (if any) situation occurs can be decided in one I/O: simply read the last leaf node of the B-tree. This also allows us to retrieve the segment s'.

If neither of the aforementioned situation happens, there is no more processing on v. Otherwise, insert s^l in $bottom_v(p)$, and if $bottom_v(p)$ has more than $B\log_B N$ segments, also remove s' from $bottom_v(p)$. In any case, the cost is $O(\log_B \log_B N)$. Accordingly, s^l and s' (if applicable) are inserted in and deleted from $\Pi_u(i)$, respectively. As $\Pi_u(i)$ indexes $O(hB\log_B N)$ segments, this can be done in $O(\log_B(hB\log_B N)) = O(\log_B \log_B N)$ I/Os. Likewise, insert s^l and delete s' (again, if applicable) in the rank B-tree $\Gamma_u(p)$, which takes $O(\log_B(fB\log_B N)) = O(\log_B f)$ I/Os.

Changes to $\Pi_u(i)$ may necessitate updating the pilot structure P_u, because only the lowest B segments in $\Pi_u(i)$ should appear in P_u. Those B segments can be obtained in one I/O from $\Pi_u(i)$. At most one insertion and one deletion are then needed on P_u. Their cost is $O(\log_B f)$ as P_u stores at most fB segments.

In summary, the processing at u incurs $O(\log_B \log_B N + \log_B f) = O(\log_B f)$ I/Os. We need to do the same to $O(h)$ nodes on φ, the cost of which adds up to $O(h \log_B f) = O(\log_B N)$. Finally, at the leaf z of φ, insert s^l to $\phi_z(p)$ and the segment set stored in z with $O(\log_B b) = o(\log_B N)$ I/Os.

Deletion. To delete a segment s, we first find the node p such that $s \in stab(p)$. If p is a leaf, finish by deleting s from the segments stored at p. Now, consider that p is an internal node. Remove s from the middle structure of p in $O(\log_B N)$ I/Os. Next we explain the rest of processing on only the left segment s^l. Let φ be the path from p to the leaf node of $\mathcal{T}$ storing the left x-coordinate of s^l. Delete in $O(\log_B N)$ I/Os the left endpoint of s^l from the relevant flank structure at p.

Let u be an internal node that is a proper descendant of p on φ. Let v be the child of u on φ. In one I/O, we check whether s^l is in $bottom_v(p)$. If so, delete s^l from $bottom_v(p)$, the rank B-tree $\Gamma_u(p)$, and the bottom structure of u for v. Update the pilot structure P_u accordingly. The cost is $O(\log_B f)$, as should have been clear from the earlier analysis on insertion. Currently, u is *dirty* because $bottom_v(p)$ may have one less segment than required. At the leaf z of φ, we remove s^l from $\phi_z(p)$ and the segment set stored in z with $o(\log_B N)$ I/Os.

Next, we fix the dirty nodes in bottom-up order. Let u be the lowest dirty node, and v its child on φ. If v is a leaf node, we check whether $\phi_v(p)$ has at least $B \log_B N$ segments. If not, $bottom_v(p)$ is already accurate. Otherwise, retrieve the $(B \log_B N)$-th lowest segment s' from $\phi_v(p)$ using $O(\log_B N)$ I/Os, by simply reading the entire $\phi_v(p)$. Insert s' into $bottom_v(p)$, $\Gamma_u(p)$, the bottom structure of u for v, and update P_u in totally $O(\log_B f)$ I/Os. At this time, $bottom_v(p)$ has been brought up-to-date, and so have the secondary structures of u.

On the other hand, if v is an internal node, we check whether $\gamma_v(p)$ (see (2)) has at least $B \log_B N$ segments. If not, $bottom_v(p)$ is accurate. Otherwise, retrieve the $(B \log_B N)$-th lowest segment s' from $\gamma_v(p)$. Using the rank B-tree $\Gamma_v(p)$, this can be done in $O(\log_B f)$ I/Os. Then, insert s' into the relevant structures (same as the case where v is a leaf) in $O(\log_B f)$ I/Os.

In summary, to delete a segment s, we spend $O(\log_B f)$ I/Os per internal level, and $O(\log_B N)$ I/Os at the leaf level. Hence, the total cost is $O(\log_B N)$. Note that we keep the x-coordinates of the endpoints of s in the base tree $\mathcal{T}$. This does not affect the correctness of our query algorithm since it reports segments only from secondary structures, which are always properly maintained. On the other hand, it brings the benefit of not having to deal with node underflows. With global rebuilding, we ensure that the height of $\mathcal{T}$ should remain $O(\log_f N)$.

4.3 Handling Node Splits

We have explained that each insertion and deletion can be performed in $O(\log_B N)$ I/Os, excluding the cost of handling node splits in the base tree $\mathcal{T}$. This subsection completes the description of our update procedure by elaborating the split algorithms. Our discussion concentrates on the secondary structures for managing left and middle segments.

We start by giving some basic facts. Recall that, a lower-open 3-sided range query specifies a rectangle of the form $[x_1, x_2] \times (-\infty, y]$. The flank structures of our index are external priority search trees for answering such queries efficiently. The lemma below explains how fast a flank structure can be constructed.

Lemma 4. *An external priority search tree for answering lower-open 3-sided range queries on N points, can be built in $O(N/B)$ I/Os, provided that all the points have been sorted by x-coordinate.*

The next lemma indicates that, once the bottom sets at the child nodes of a node u are ready, we can build several secondary structures of u efficiently. Recall that $h = O(\log_f N)$ is the height of $\mathcal{T}$.

Lemma 5. *Let u be an internal node with child nodes $v_1, ..., v_f$. Suppose that the segments of $bottom_{v_i}(p)$ have been sorted by y-coordinate, for each v_i ($1 \leq i \leq f$) and each proper ancestor p of u. In $O(hf \log_B N \cdot \log_2 f)$ I/Os, we can build the following secondary structures of u: the bottom structure $\Pi_u(i)$ of each i, the pilot structure P_u, the B-tree on $bottom_u(p)$ of each relevant p, and the rank B-tree $\Gamma_u(p)$ of each p.*

Leaf Split. Next, we discuss the split of a leaf node z. Let u be the parent of z, and z_1, z_2 be the leaf nodes that z splits into. Let ℓ be the vertical line that divides σ_{z_1} and σ_{z_2}.

The stabbing set $stab(z)$ can be divided into three groups. The first (second) group includes those segments on the left (right) of ℓ, while the last group, denoted as S_{cross}, includes those intersecting ℓ. The first (second) group becomes $stab(z_1)$ ($stab(z_2)$), whereas S_{cross} will be merged into $stab(u)$.

Let S_z be the set of (at most b) segments stored at z, and similarly, S_{z_1} (S_{z_2}) the set of segments to be stored at z_1 (z_2) after the split. Recall that $stab(z)$ is a subset of S_z (because a segment in S_z has at least an endpoint in σ_z, whereas σ_z covers both endpoints of a segment in $stab(z)$). Each segment in S_z should be added to S_{z_1} if its left endpoint is on the left of ℓ, or to S_{z_2} otherwise. Utilizing the B-tree on S_z, in $O(b/B)$ I/Os, we can generate S_{z_1}, S_{z_2} and S_{cross} such that the segments of each set are sorted by y-coordinate. Then, the B-trees on S_{z_1} and S_{z_2} can be built in $O(b/B)$ I/Os.

Let p be an ancestor of u (note: p can be u). After the split, a segment $s \in \phi_z(p)$ appears in $\phi_{z_1}(p)$ if its left endpoint is to the left of ℓ; otherwise, s appears in $\phi_{z_2}(p)$. Moreover, if $p = u$, $\phi_{z_1}(p)$ also includes those segments of S_{cross} whose left endpoints are to the left of ℓ. Using the B-tree on $\phi_z(p)$ (and S_{cross}, if $p = u$), we create the B-trees on $\phi_{z_1}(p)$ and $\phi_{z_2}(p)$ in $O(1 + |\phi_z(p)|/B)$ I/Os (and plus $O(b/B)$ if $p = u$). As there are $O(h)$ different p, and the sum of $|\phi_z(p)|$ for all p is at most b, the B-trees on the $\phi_{z_1}(p)$ and $\phi_{z_2}(p)$ of all p can be built in $O(h + b/B)$ I/Os. If p is at level 2 or above, $bottom_{z_1}(p)$ is simply a prefix of $\phi_{z_1}(p)$. Hence, the B-trees on the $bottom_{z_1}(p)$ of all p can also be built in $O(h + b/B)$ I/Os. The same is true for $bottom_{z_2}(p)$.

At this point, the secondary structures of z_1 and z_2 are ready, so we proceed to fix those of u. Create an additional copy of S_{cross} where the segments are sorted by x-coordinate. This can be achieved in $O(\frac{b}{B} \log_2 b)$ I/Os. Perform a refine operation (Section 3) on the middle structure of u, with ℓ and S_{cross} as the inputs. By Lemma 2, the cost is $O(f^2 \log_2 f + b/B)$, noticing that the value of α in the lemma is at most b. The other secondary structures of u mentioned in Lemma 5 can be built in $O(hf \log_B N \cdot \log_2 f)$ I/Os.

In summary, a leaf split can be performed in $O(f^2 \log_2 f + \frac{b}{B} \log_2 b + hf \log_B N \cdot \log_2 f)$ I/Os.

Internal Split. Consider that an internal node u is split into u_1 and u_2 along a vertical line ℓ that divides σ_{u_1} and σ_{u_2}. Define $w(u)$ as the number of x-coordinates stored at the leaf nodes in the subtree of u, and $w(u_1), w(u_2)$ similarly. We start with a split operation on the middle structure of u with input ℓ. This operation creates the middle structures of u_1 and u_2, and generates a segment set S_{cross} in two copies: one sorted by y-coordinate, and the other by x-coordinate. S_{cross} includes all the segments from $stab(u)$ that intersect ℓ, and will be added to the stabbing set of $parent(u)$. By Lemma 3, the operation requires $O(\frac{w(u)}{B} \log_2 f + f^2 \log_2 f)$ I/Os, noticing that the middle structure of u indexes at most $w(u)$ segments.

If u is at level 2 or above, we need to build the flank structures of u_1 and u_2. Let v be a child node of u that becomes a child node of u_1 after the split (the case of u_2 is analogous). Let $F_{u_1}(i)$ be the flank structure of u_1 for v, and F_{old} the flank structure of u for v before the split. $F_{u_1}(i)$ should store those endpoints in F_{old} that do not belong to the segments of S_{cross}. As the endpoints indexed by F_{old} are sorted by x-coordinate, in linear I/Os, we can generate the list of endpoints to be stored in $F_{u_1}(i)$, preserving their ordering. After this, $F_{u_1}(i)$ can be built in linear I/Os (Lemma 4). Hence, the flank structures of u_1 and u_2 can be constructed using $O(f + w(u)/B)$ I/Os in total, because they together manage at most $w(u)$ endpoints.

At $parent(u)$, we update its middle structure with a refine operation using $O(w(u)/B + f^2 \log_2 f)$ (Lemma 2), feeding ℓ and the two copies of S_{cross} obtained earlier. Next, we discuss how to build the flank structures of $parent(u)$ for u_1 and u_2—denote them, respectively, as $F_{parent(u)}(i)$ and $F_{parent(u)}(i+1)$ for

some $i \in [1, f-1]$. Also, let F'_{old} be the flank structure of $parent(u)$ for u before the split. $F_u(i+1)$ should store the endpoints in F'_{old} that are on the right of ℓ. $F_u(i)$ should store (i) the other endpoints of F'_{old}, and (ii) the left endpoints of the segments in S_{cross}. In F'_{old}, the at most $w(u)$ endpoints are sorted in ascending order of x-coordinate. Given that a copy of S_{cross} is sorted in the same way, in $O(w(u)/B)$ I/Os, we generate the lists of endpoints managed by $F_u(i)$ and $F_u(i+1)$ respectively, where each list is sorted by x-coordinate. Then, $F_u(i)$ and $F_u(i+1)$ are created in $O(w(u)/B)$ I/Os with Lemma 4.

Recall that u may have bottom sets at its proper descendants. After the splits, those bottom sets no longer exist, but instead we should build the bottom sets of u_1 and u_2. Furthermore, as new segments have been added to the stabbing set of $parent(u)$, some bottom sets of $parent(u)$ may contain errors. Also affected are the secondary structures dependent on the aforementioned bottom sets. Next, we explain a *bottom overhaul* to fix all these issues, by processing the descendants of u in bottom-up manner.

Let z be a leaf descendant of u that is now a descendant of u_1 (the case of u_2 is similar). Create the B-tree of $\phi_z(u_1)$ by scanning the B-tree on $\phi_z(u)$ in $O(b/B)$ I/Os (ignoring segments intersecting ℓ). Regarding $\phi_z(parent(u))$, the segments in S_{cross} whose left endpoints are in σ_z should now be added to $\phi_z(parent(u))$. As those segment must be in $\phi_z(u)$ before the split, in $O(b/B)$ I/Os, we can extract them into a list sorted by y-coordinate, and then, merge the list into the B-tree of $\phi_z(parent(u))$. After this, $bottom_z(parent(u))$ and if applicable, also $bottom_z(u_1)$ and $bottom_z(u_2)$, can be built in $O(b/B)$ I/Os. Note that the set of segments stored at z is not affected; hence, its B-tree before the split can be used directly. As u has $O(w(u)/b)$ leaf descendants, processing all of them takes $O(\frac{w(u)}{b}\frac{b}{B}) = O(w(u)/B)$ I/Os in total.

For every proper internal descendant of u, we rebuild its secondary structures mentioned in Lemma 5 in $O(hf \log_B N \cdot \log_2 f)$ time. Since there are $O(w(u)/(bf))$ such internal nodes, the total cost is

$$\begin{aligned} O\left(\frac{w(u)}{bf} hf \log_B N \cdot \log_2 f\right) &= O\left(\frac{w(u)}{B \log_B N} \log_2 f\right) \\ &= o(w(u)). \end{aligned}$$

We complete the bottom overhaul by applying Lemma 5 to u_1, u_2 and p, necessitating another $O(hf \log_B N \cdot \log_2 f)$ I/Os.

In summary, an internal split takes $O(\frac{w(u)}{B} \log_2 f + w(u) + f^2 \log_2 f + hf \log_B N \cdot \log_2 f)$ time, which is $O(\frac{w(u)}{B} \log_2 f + w(u))$ given the fact $w(u) = \Omega(bf)$.

Amortized Cost. The WBB-tree guarantees that when a node u is split, $\Omega(w(u))$ insertions must have been performed in its subtree since the creation of u. Hence, if u is a leaf, we charge the split cost over those $\Omega(b)$ insertions, so that each insertion accounts for

$$\begin{aligned} & O\left(\frac{f^2 \log_2 f}{b} + \frac{b \log_2 b}{Bb} + \frac{hf \log_B N \cdot \log_2 f}{b}\right) \\ = \; & O\left(1 + \frac{\log_2 b}{B} + \frac{f \cdot \log_2 f}{B \log_B N}\right) \\ = \; & o(\log_2 \log_B N) \end{aligned}$$

I/Os. If u is an internal node, after amortization, each insertion bears $O\left(1 + (1/B) \log_2 f\right)$ I/Os incurred from splitting u. As an insertion bears such cost for $O(h)$ nodes, its amortized cost increases by

$$\begin{aligned} O\left(h + \frac{h \log_2 f}{B}\right) &= O\left(h + \frac{\log_f N \cdot \log_2 f}{B}\right) \\ &= O\left(h + \frac{\log_2 N}{B}\right) \end{aligned}$$

which is $O(\log_B N)$.

4.4 Bootstrapping

Currently, our ray-stabbing structure handles an update in $O(\log_B N)$ I/Os amortized, but its query time is $O(b/B + \log_B N + K/B) = O(\log_B^3 N + K/B)$. Similar to the bootstrapping in Section 2, we use the above structure to index the (at most b) segments stored in a leaf node, thus improving the query cost to $O(\log_B N + K/B)$. In a leaf split, this structure can be rebuilt in $O(b \log_B b) = O(b \log_B \log_B N)$ time by doing at most b insertions. Note that the structure does not need to be touched in a bottom overhaul (which does not affect the segments stored in a leaf, as mentioned earlier). Hence, each update (on the bootstrapped structure) is charged only additional $O(\log_B \log_B N)$ I/Os. This completes the proof of Theorem 1.

5 Conclusions and Future Work

In the *orthogonal ray stabbing problem*, the input is a set $\mathcal{S}$ of N horizontal segments in $\mathbb{R}^2$. Given a vertical ray Q, a query reports all the segments in $\mathcal{S}$ intersecting Q. This paper has described a dynamic structure in the external memory model that uses $O(N/B)$ space, answers a query in $O(\log_B N + K/B)$ I/Os, and supports an insertion/deletion in $O(\log_B N)$ I/Os amortized, where B is the block size and K the number of reported segments. Setting B to an appropriate constant, the structure also works in internal memory, retaining exactly the same efficiency.

An obvious direction for future work is to study whether the above result is helpful in attacking the *orthogonal segment intersection problem*, where the input $\mathcal{S}$ is the same as before, but a query is given a vertical *segment* Q, and reports all the segments in $\mathcal{S}$ intersecting Q. It would be a breakthrough to solve the problem by matching the performance of ray stabbing, or argue against the existence of such a result. In internal memory, our discussion has essentially focused on the pointer machine model, whereas it would be interesting to improve the query and/or update efficiency using RAM features.

References

[1] P. K. Agarwal, S.-W. Cheng, Y. Tao, and K. Yi. Indexing uncertain data. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 137–146, 2009.

[2] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM (CACM)*, 31(9):1116–1127, 1988.

[3] L. Arge, V. Samoladas, and J. S. Vitter. On two-dimensional indexability and optimal range search indexing. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 346–357, 1999.

[4] L. Arge and J. S. Vitter. Optimal external memory interval management. *SIAM Journal of Computing*, 32(6):1488–1508, 2003.

[5] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion B-tree. *The VLDB Journal*, 5(4):264–275, 1996.

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

[7] G. E. Blelloch. Space-efficient dynamic orthogonal point location, segment intersection, and range reporting. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 894–903, 2008.

[8] B. Chazelle. Filtering search: A new approach to query-answering. *SIAM Journal of Computing*, 15(3):703–724, 1986.

[9] S.-W. Cheng and R. Janardan. Efficient dynamic algorithms for some geometric intersection problems. *Information Processing Letters (IPL)*, 36(5):251–258, 1990.

[10] J. Enderle, N. Schneider, and T. Seidl. Efficiently processing queries on interval-and-value tuples in relational databases. In *Proceedings of Very Large Data Bases (VLDB)*, pages 385–396, 2005.

[11] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *Proceedings of Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 714–723, 1993.

[12] H. Kaplan, E. Molad, and R. E. Tarjan. Dynamic rectangular intersection with priorities. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 639–648, 2003.

[13] C. W. Mortensen. Fully-dynamic two dimensional orthogonal range and line segment intersection reporting in logarithmic time. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 618–627, 2003.

[14] B. Salzberg and V. J. Tsotras. Comparison of access methods for time-evolving data. *ACM Computing Surveys*, 31(2):158–221, 1999.

Appendix

Proof of Lemma 1. We assign each long segment $s \in \mathcal{S}$ to the widest multi-slab it spans. A multi-slab is *large* if it has at least B segments (assigned to it), and is *small* otherwise. Let S_{low} be the union of the B lowest segments in each multi-slab. All the segments of a small multi-slab are in S_{low}. We index the $O(f^2B)$ segments in S_{low} with a persistent B-tree [5] so that the segments intersecting a vertical ray can be reported efficiently. For each large multi-slab X, create a B-tree on the y-coordinates of its segments. Call the B-tree a *multi-slab B-tree*.

To answer a query with a ray Q, first search the persistent B-tree to report all the segments intersecting Q. For each large multi-slab X, if B segments have been reported (as can be detected using the sentinel trick in Section 2.2, namely, by marking some segments in the persistent tree as sentinels), deploy the B-tree of X to report the other qualifying segments of X in ascending order of y-coordinate. The query cost is $O(\log_B f)$ plus the linear output cost, where the logarithmic term comes from querying the persistent B-tree.

To support updates, we maintain a B-tree $\mathcal{T}_{low}$ that indexes the segments in $\mathcal{S}_{low}$ by the ids of their origin multi-slabs. Given any multi-slab X, $\mathcal{T}_{low}$ allows us to retrieve the lowest B segments of X in $O(\log_B f^2) = o(\log_B N)$ I/Os. Furthermore, additional structures are needed in order to facilitate some operations required later in Lemmas 2 and 3. For each ℓ_i ($2 \leq i \leq f-1$), we create a B-tree on the y-coordinates of the short segments intersecting ℓ_i. Call the B-tree a *line B-tree*. Furthermore, for each slab σ, create a B-tree on the x-coordinates of all (long and short) the segments whose endpoints are covered by σ. Call this B-tree a *slab B-tree*.

All structures use linear space, except that a line/slab B-tree occupies at least one block in any case. As each segment exists in constant structures, the space consumption is $O(f + N/B)$.

Next, we describe updates. First note that the persistent B-tree can be constructed in $O(f^2 \log_2 f)$ I/Os with the algorithm in [11]. Using a block to buffer updates, we apply global rebuilding to update the structure in $O(\frac{f^2}{B} \log_2 f) = o(\log_B N)$ I/Os per insertion and deletion.

Inserting/deleting a short segment s only requires updating a line B-tree and a slab B-tree. Consider inserting a long segment s to $\mathcal{S}$. First, update the relevant slab and multi-slab B-trees. Assume that s belongs to multi-slab X. Using $\mathcal{T}_{low}$, check in $o(\log_B N)$ I/O whether s is among the B lowest segments in X. If no, the insertion is complete. Otherwise, insert s in S_{low} (updating the relevant structures), and if S_{low} now has $B+1$ segments from X, delete the highest segment of X in S_{low}. Accordingly, the sentinel segment of X can be maintained, if necessary, with an insertion and perhaps also a deletion in the persistent B-tree.

Deleting a long segment from $\mathcal{S}$ can be handled in a reverse manner with the same overhead.

Proof of Lemma 2. We will utilize the structure described in the proof of Lemma 1.

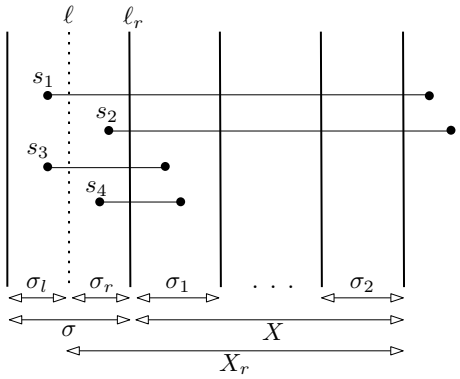

Figure 8: Proof of Lemma 2

Let σ_1 be the slab next to σ on the right. Let X be a multi-slab that starts from σ_1 and ends at some slab σ_2 (it is possible that $\sigma_2 = \sigma_1$). Denote by X_r the multi-slab after refinement that starts from σ_r and ends at σ_2. See Figure 8. Consider a segment s_{long} of X. After refinement, s_{long} belongs to either X_r or X, depending on whether the left endpoint of s_{long} is in σ_l or σ_r. For example, in Figure 8, s_1 (s_2) is a long segment that will appear in X_r (X). In $O(\log_B f)$ I/Os, we can obtain a list of segments (currently) in X sorted by y-coordinate: if X is large, such a list is already availabe from its multi-slab B-tree; otherwise, we retrieve all the (less than B) segments of X from $\mathcal{T}_{low}$. Then, we can generate two lists of segments that should belong to X_r and X after refinement, respectively. In both lists, segments are sorted by y-coordinate. The cost of the generation is linear to the number of segments in X before refinement.

Let ℓ_r be the right boundary of σ. Consider a short segment s_{short} intersecting ℓ_r before refinement. After refinement, s_{short} may or may not become long. It will, if its left endpoint is in σ_l, in which case it will belong to multi-slab σ_r. Otherwise, it will not. For example, in Figure 8, s_3 (s_4) is a short segment that will become long (remain short). Hence, from the line B-tree of ℓ_r, we can generate a list of segments that have become long or remain short, respectively. In both lists, segments are sorted by y-coordinate. The cost is linear to the number of short segments intersecting ℓ_r before refinement.

So far we have obtained a sorted list for each new/affected multi-slab on the right of ℓ, and a sorted list of the short segments intersecting ℓ_r after refinement. Symmetrically, we can do the same with respect to each new/affected multi-slab on the left of ℓ, and the left boundary ℓ_l of σ. The total cost is $O(f \log_B f + \alpha/B)$. After this, all the corresponding multi-slab/line B-trees can be constructed in $O(\alpha/B + f)$ I/Os. $\mathcal{T}_{low}$ and the persistent B-tree can be rebuilt in $O(f^2 \log_2 f)$ I/Os.

Finally, we rebuild the line B-tree of ℓ, and the slab B-trees of σ_l and σ_r in $O(\alpha/B)$ I/Os. In total, a refine operation takes $O(f \log_B f + f^2 \log_2 f + \alpha/B) = O(f^2 \log_2 f + \alpha/B)$ I/Os.

Proof of Lemma 3. Again, our discussion is based on the structure described in the proof of Lemma 1.

Refer to the middle structure for the problem defined on (S_1, L_1) as the *first middle structure*, and to the one on (S_2, L_2) as the *second middle structure*. As mentioned in the proof of Lemma 2, in $O(\log_B f)$ I/Os, we can obtain a list of all the segments in a multi-slab, sorted by y-coordinate. Hence, the lists of all f^2 multi-slabs can be generated in $O(f^2 \log_B f)$ I/Os, after which they can be merged on y-coordinates in $O((N/B) \log_2 f)$ I/Os. By scanning this merged list and the line B-tree of ℓ, in $O(N/B)$ I/Os, we create S_{cross} with its segments sorted by y-coordinate. Another copy of S_{cross} where its segments are sorted by x-coordinate can be obtained easily from the slab B-trees in $O(f + N/B)$ I/Os.

If a multi-slab does not intersect ℓ, its B-tree can be re-used directly in either the first or second middle structure. Likewise, every line B-tree (except, obviously, the one on ℓ) can also be re-used. $\mathcal{T}_{low}$ and the persistent B-trees of the two new middle structures can be constructed in $O(f^2 \log_2 f)$ I/Os from the

multi-slab lists obtained earlier. Finally, the slab B-trees of the two middle structures are built in $O(f+N/B)$ I/Os from the original slab B-trees.

Proof of Lemma 4. First, build the base tree, namely, a B-tree on the x-coordinates of all the data points. Each leaf node stores between $B/4$ and B coordinates, while an internal node node has between $B/4$ and B child nodes. If a data point's x-coordinate is in a leaf, we store the point in the leaf as well. Assume that the leaf nodes are at level 0, and that the parent of a level-l node is at level $l+1$.

As discussed in [3], each internal node u is associated with at most B^2 *prior points* defined as follows. Let $v_1, ..., v_B$ be the child nodes of u. If u is the root of the base tree, for each child v_i, u stores the B highest points from the subtree of v_i. Otherwise, for each v_i, u stores the B highest points from the subtree of v_i that are not a prior point at any proper ancestor of u. In any case, denote the set of those B points as $prior_u(i)$. Let $prior_u = prior_u(1) \cup ... \cup prior_u(B)$. The (at most) B^2 points in $prior_u$ are indexed by a structure called a B^2-*structure*. Consuming $O(B)$ space, the B^2-structure answers a lower-open 3-sided range query in $O(1)$ I/Os plus the linear output time. If the B^2 points indexed have been sorted by x-coordinate, the structure can be built in $O(B)$ I/Os [3], and supports an update in $O(1)$ I/Os per insertion and deletion. In addition, create a B-tree on the y-coordinates of the points in $prior_u$.

We build the B^2-structures of internal nodes in bottom-up order, i.e., finishing with all the nodes at a level before attending to the parent level. Let u be a node at level 1, with child nodes $v_1, ..., v_B$ (which are leaves). For each $i \in [1, B]$, obtain the B points stored in v_i in one I/O. These points form $prior_u(i)$. After this, the B^2-structure of u is constructed in $O(B)$ I/Os. The B-tree on $prior_u$ be easily built in $O(B)$ I/Os.

Now, consider u instead as a node at level $l \geq 2$, with $v_1, ..., v_B$ as its child nodes. For each i, with the B-tree on $prior(v_i)$, we obtain the B highest points from $prior(v_i)$ in one I/O; and they constitute $prior_u(i)$. The B^2-structure of u is then created in $O(B)$ I/Os. Some additional processing is necessary at v_i. Suppose that a point o from $prior(v_i)$ has been included in $prior_u(i)$. Accordingly, o is deleted from the B^2-structures at v_i with $O(1)$ I/Os. We also need to *promote* a point, say o', by inserting it in $prior(v_i)$. Specifically, o' is the highest of the prior points at the child nodes of v_i. Recursively, the promotion necessitates similar promotions along a path from v_i to a leaf. As shown in [3], all these promotions can be performed with a *bubble-up* procedure in $O(l)$ I/Os. In total, at most B^2 bubble-up procedures are performed due to u, entailing $O(lB^2)$ I/Os.

In summary, the secondary structures of each node at level 1 can be built in $O(B)$ I/Os, whereas those of a node at level $l \geq 2$ in $O(lB^2)$ I/Os. As there are at most $N/(B/4)^{l+1}$ nodes at level $l \geq 1$, the total cost of creating all the secondary structures is

$$O\left(\frac{N}{B^2} \cdot B + \sum_{l=2}^{h} \frac{N}{(B/4)^{l+1}} lB^2\right) = O(N/B)$$

where h is the level of the base tree's root.

Proof of Lemma 5. For each p, we first merge the segments of $bottom_{v_1}(p)$, ..., $bottom_{v_f}(p)$ into a list where the segments are sorted by y-coordinate. As each $bottom_{v_f}(p)$ has at most $B \log_B N$ segments, the merge can be done in $O(\frac{1}{B} fB \log_B N \cdot \log_2 f) = O(f \log_B N \cdot \log_2 f)$ I/Os. Then, $bottom_u(p)$ can be acquired by reading the first $B \log_B N$ segments of the merged list, after which the B-tree on $bottom_u(p)$ can be created in $O(\log_B N)$ I/Os. Hence, the B-trees of $bottom_u(p)$ for all p can be constructed with cost $O(hf \log_B N \cdot \log_2 f)$.

To build the bottom structure $\Pi_u(i)$, first generate $\pi_u(i)$—as defined in (1)—by merging the $bottom_{v_i}(p)$ of all p. As u has at most h proper ancestors, the merge takes $O(\frac{1}{B} hB \log_B N \cdot \log_2 h) = O(h \log_B N \cdot \log_2 h)$ time. Then, $\Pi_u(i)$ can be created in $O(h \log_B N)$ I/Os. Hence, all the bottom structures can be constructed with cost $O(hf \log_B N \cdot \log_2 h)$. To build the pilot structure P_u, collect the lowest B

segments from each $\pi_u(i)$, and sort the at most fB segments collected by x-coordinate. This requires $O((fB/B)\log_2 f) = O(f \log_2 f)$ time. P_u can then be constructed in $O(f)$ I/Os using Lemma 4.

To build the rank B-tree $\Gamma_u(p)$, first obtain $\gamma_u(p)$ by merging the $bottom_{v_i}(p)$ of all $i \in [1, f]$ in $O(f \log_B N \cdot \log_2 f)$ I/Os, and then create $\Gamma_u(p)$ from $\gamma_u(p)$ in $O(f \log_B N)$ I/Os. Hence, all the rank B-trees can be constructed with cost $O(hf \log_B N \cdot \log_2 f)$.

We have built all the secondary structures stated in the lemma using $O(hf \log_B N \cdot (\log_2 f + \log_2 h))$ I/Os. Note that $\log_2 h = O(\log_2 \log_B N)$. If $\log_B N \leq B$, then $\log_2 h = O(\log_2 B) = O(\log_2 f)$ because $f = B^{1/3}$. Otherwise, $f = (\log_B N)^{1/3}$ and $\log_2 h$ is clearly still $O(\log_2 f)$. The lemma thus follows.