# I/O-Efficient Structures for Orthogonal Range Max and Stabbing Max Queries

Pankaj K. Agarwal Lars Arge Jun Yang Ke Yi

Department of Computer Science
Duke University
Durham, NC 27708

**Abstract**

Due to their important applications in e.g. temporal and spatial database systems, range searching and its variants have been studied extensively in both the computational geometry and database communities. Since many database applications involve massive amounts of data stored in external memory, it is important to consider I/O-efficient structures for fundamental range searching problems.

In this paper we develop several new linear or near linear space and I/O-efficient dynamic data structures for orthogonal range max queries and stabbing max queries. These variants of range searching e.g. have important applications in database query optimization. Given a set of $N$ weighted points in $\mathbb{R}^d$, the range max problem asks for the maximum-weight point in a query hyper-rectangle. In the dual stabbing max problem, we are given $N$ weighted hyper-rectangles and want to find the maximum-weight rectangle containing a query point. Our structures improve on previous structures in several important ways.

## 1 Introduction

Range searching and its variants have been studied extensively in the computational geometry and database communities because of its importance in many applications, including temporal and spatial database systems. Range-aggregate queries, such as range-count, range-sum, and range-max queries, are some of the most commonly used versions of range searching in database applications. Since many such applications involve massive amounts of data stored in external memory, it is important to consider I/O-efficient structures for fundamental range-searching problems. In this paper, we develop I/O-efficient data structures for answering range-max queries, as well as for the dual problem of answering stabbing-max queries.

**Problem statement.** In the *orthogonal range max* problem, we are given a set $S$ of $N$ points in $\mathbb{R}^d$ where each point $p$ is assigned a weight $w(p)$, and we want to build a data structure so that for a query hyper-rectangle $Q$ in $\mathbb{R}^d$, we can compute $\max\{w(p) \mid p \in Q\}$ efficiently. The two-dimensional case is illustrated in Figure 1. In the dual *orthogonal stabbing-max* problem, we are given a set $S$ of $N$ hyper-rectangles in $\mathbb{R}^d$ where each rectangle $\gamma$ is assigned a weight $w(\gamma)$, and we want to build a data structure such that for a query point $q$ in $\mathbb{R}^d$, we can compute $\max\{w(\gamma) \mid q \in \gamma\}$ efficiently. The two-dimensional case is illustrated in Figure 2. We will also consider the dynamic version of the two problems, where points or hyper-rectangles can be inserted or deleted dynamically. In the following we drop 'orthogonal', when clear from the context and also 'max', when referring to the two problems.

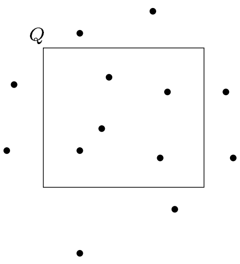


Figure 1: Two-dimensional range queries.

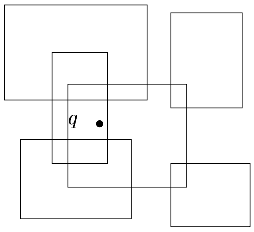


Figure 2: Two-dimensional stabbing queries.

We will work in the standard external memory model [4]. In this model, the main memory holds $M$ words and each disk access (or I/O) transmits a continuous block of $B$ words between main memory and disk. We assume that $M > B^2$ and that any integer less than $N$, as well as any point or weight, can be stored in a single word. The efficiency of a data structure is measured in terms of the amount of disk space it uses (measured in number of disk blocks) and the number of I/Os required to answer a query or perform an update. We will focus on data structures that use linear or almost linear space, that is, use close to $n = \lceil N/B \rceil$ disk blocks.

**Related work.** Range query data structures have been studied extensively in the internal memory RAM model of computation. In two dimensions, the best known linear space structure for the range max problem is due to Chazelle [12]. It answers a query in time $O(\log^{1+\epsilon} n)$ time. In the dynamic case, the structure supports queries and updates in $O(\log^3 n \log\log n)$ time. For the stabbing-max problem, see the paper by Eppstein and Muthukrishnan [15]. Refer to the recent survey by Agarwal and Erickson [3] for a complete overview of results.

In the external setting, one-dimensional range max queries can be answered in $O(\log_B n)$ I/Os using a standard linear space B-tree [13, 9]. The structure can easily be updated using $O(\log_B n)$ I/Os. For two or higher dimensions, however, no efficient linear-space structure is known for the problem. The linear space kdB-tree [20], as well as the cross-tree [17] and O-tree [18], can used to solve the reporting version of the two-dimensional problem, where we want to report all $T$ points in a query rectangle, in $O(\sqrt{n} + T/B)$ I/Os. These structures can relatively easily be modified to answer range max queries in $O(\sqrt{n})$ I/Os. The cross-tree [17] and O-tree [18] can also be updated in $O(\log_B n)$ I/Os. The linear space CRB-tree [2] can be used to count the number of points in a query rectangle in $O(\log_B n)$ I/Os. This structure can relatively easily be modified to support range max queries in $O(\log_B^2 n)$ I/Os but using $O(n \log_B n)$ space.

For the one-dimensional stabbing max problem, the linear space SB-tree [23] can be used to answer queries in $O(\log_B n)$ I/Os. Intervals can be inserted in the structure in $O(\log_B n)$ I/Os. However, the SB-tree does not support deletions. Intuitively, supporting deletions is much harder than supporting insertions. No worst case efficient structures are known for higher-dimensional stabbing max queries. The reporting version of the one-dimensional problem, where we want to report all $T$ intervals containing a query point, can be solved efficiently using the external interval tree [8]. This structure uses linear space and answers queries in $O(\log_B n + T/B)$ I/Os. Insertions and deletions can be handled in $O(\log_B n)$ I/Os. Refer to recent surveys [6, 21, 16] for a complete overview of external memory data structures.

**Our results.** In this paper we obtain three main results. Our first result is a linear-size structure for answering two-dimensional range-max queries in $O(\log_B^2 n)$ I/Os. This is the first linear-size external memory data structure that can answer such queries in polylogarithmic number of I/Os. Using $O(n \log_B \log_B n)$ space, the structure can be made dynamic such that insertions and deletions can be performed in $O(\log_B^2 n \log_{M/B} \log_B n)$ and $O(\log_B^2 n)$ I/Os amortized, respectively. Refer to Table 1 for a brief comparison with previous results.

| Problem | Space | Query | Insertion | Deletion | Source |
|---|---|---|---|---|---|
| 2D range max queries (static) | $n \log_B n$ | $\log_B^2 n$ | | | [2] |
| | $n$ | $\log_B^2 n$ | | | New |
| 2D range max queries (dynamic) | $n$ | $\sqrt{n}$ | $\log_B n$ | $\log_B n$ | [20, 17, 18] |
| | $n \log_B \log_B n$ | $\log_B^3 n$ | $\log_B^2 n \cdot \log_{M/B} \log_B n$ | $\log_B^2 n$ | New |

Table 1: Two-dimensional range max query results.

Our second result is a linear-size dynamic structure for answering one-dimensional stabbing-max queries in $O(\log_B^2 n)$ I/Os. The structure supports both insertions and deletions in $O(\log_B n)$ I/Os. As mentioned above, the previously known structure, the SB-tree [23], only supported insertions.

Our third result is a linear-size structure for answering two-dimensional stabbing max queries in $O(\log_B^4 n)$ I/Os. The structure is an extension of our one-dimensional structure, which also uses our two-dimensional range-max query structure. The structure can be made dynamic with a $O(\log_B^5 n)$ query bound at the cost of a factor of $O(\log_B \log_B n)$ in its size. Insertions and deletions can be performed in $O(\log_B^2 n \log_{M/B} \log_B n)$ and $O(\log_B^2 n)$ I/Os amortized, respectively. Refer to Table 2 for a brief comparison with previous results.

| Problem | Space | Query | Insertion | Deletion | Source |
|---|---|---|---|---|---|
| 1D stabbing max queries (dynamic) | $n$ | $\log_B n$ | $\log_B n$ | | [23] |
| | $n$ | $\log_B^2 n$ | $\log_B n$ | $\log_B n$ | New |
| 2D stabbing max queries (static) | $n$ | $\log_B^4 n$ | | | New |
| 2D stabbing max queries (dynamic) | $n \log_B \log_B n$ | $\log_B^5 n$ | $\log_B^2 n \cdot \log_{M/B} \log_B n$ | $\log_B^2 n$ | New |

Table 2: Two-dimensional stabbing max query results.

Finally, using classical techniques [2, 10, 14], both our range and stabbing structures can be extended to $d$-dimensions for $d > 2$, at the cost of increasing the space, query, and update bounds by a factor of $O(\log_B^{d-2} n)$. Our structures can also be extended and improved in several other ways. For example, our one-dimensional stabbing structure can be modified to support general semigroup stabbing queries, and the ideas utilized in the development of the structure can be used to obtain an improved RAM model structure.

## 2 2D Range Queries

In this section we describe our structure for the two-dimensional range max problem. The structure is an external version of a structure due to Chazelle [12].

**The overall structure.** Let $S$ be a set of $N$ points in $\mathbb{R}^2$. Our structure consists of two parts. The first is simply a B-tree $\Phi$ on the $y$-coordinates of $S$. It uses $O(n)$ blocks and can be constructed in $O(n \log_B n)$ I/Os. To construct the second part, we first build a base B-tree $T$ with fanout $\sqrt{B}$

on the $x$-coordinates of $S$. For each node $v$ of $T$, let $P_v$ be the sequence of points stored in the subtree rooted at $v$, sorted by their $y$-coordinates. Set $N_v = |P_v|$ and $n_v = N_v/B$. With each node $v$ we associate a *slab* $\sigma_v$ containing $P_v$. If $v_1, v_2, \ldots, v_{\sqrt{B}}$ are the children of $v$, then $\sigma_{v_1}, \ldots, \sigma_{v_{\sqrt{B}}}$ partition $\sigma_v$ into $\sqrt{B}$ slabs. We call a wide slab $\sigma_v[i:j]$ formed by the union of a set of contiguous slabs $\sigma_{v_i}, \ldots, \sigma_{v_j}$ a *multislab* at $v$; there are $O(B)$ multislabs at each node $v$ of $T$.

Each leaf $z$ of $T$ stores the points in $P_z$ and their weights using $O(1)$ disk blocks. Each internal node $v$ stores two secondary structures $\mathcal{C}_v$ and $\mathcal{M}_v$, each of which requires $O(n_v/\log_B n)$ blocks. Below we describe the functionality of these structures. After describing how to answer a query, we then describe their implementation.

For a point $p \in \mathbb{R}^2$, let $rk_v(p)$ denote the rank of $p$ in $P_v$, i.e., the number of points in $P_v$ whose $y$-coordinates are not larger than the $y$-coordinate of $p$. Given $rk_v(p)$ of a point $p$, $\mathcal{C}_v$ can be used to determine $rk_{v_i}(p)$ for all children $v_i$ of $v$ using $O(1)$ I/Os. Moreover, if $p \in P_v$, it also returns the index of the child of $v$ storing $p$. Suppose we know the rank $\rho = rk_v(p)$ of a point $p \in P_v$, we can find the weight of $p$ in $O(\log_B n)$ I/Os using $\mathcal{C}_v$: If $v$ is a leaf, then we examine all the points of $P_v$ and return the weight of the point whose rank is $\rho$. If $v$ is an interior node, we use $\mathcal{C}_v$ to find the child $v_j$ of $v$ that stores $p$ as well as the rank of $p$ in $P_{v_j}$, and then recursively search at $v_j$. We call this an *identification process*.

The other secondary structure $\mathcal{M}_v$ will enable us to compute the maximum weight among the points in a multislab whose $y$-coordinates lie in a given range. More precisely, given $1 \le i \le j \le \sqrt{B}$ and $1 \le \rho_1 \le \rho_2 \le N_v$, $\mathcal{M}_v$ can be used to determine in $O(\log_B n)$ I/Os the maximum value in the set $\{w(p) \mid p \in P_v \cap \sigma_v[i:j] \text{ and } rk_v(p) \in [\rho_1, \rho_2]\}$.

Since the secondary structures on each level of $T$ use $O(n/\log_B n)$ blocks, the total size of the structure is $O(n)$.

**Answering a query.** Let $Q = [x_1, x_2] \times [y_1, y_2]$ be a query rectangle. We wish to compute $\max\{w(p) \mid p \in S \cap Q\}$. The overall query procedure is the same as for the CRB-tree [2]. Let $z_1$ (resp. $z_2$) be the leaf of $T$ such that $\sigma_{z_1}$ (resp. $\sigma_{z_2}$) contains $(x_1, y_1)$ (resp. $(x_2, y_2)$). Let $\xi$ be the nearest common ancestor of $z_1$ and $z_2$. Then $S \cap Q = P_\xi \cap Q$, and therefore it suffices to compute $\max\{w(p) \mid p \in P_\xi \cap Q\}$.

We visit the nodes on the paths from the root to $z_1$ and $z_2$ in a top-down manner. For any node $v$ on the path from $\xi$ to $z_1$ (resp. $z_2$), let $l_v$ (resp. $r_v$) be the index of the child of $v$ such that $(x_1, y_1) \in \sigma_{l_v}$ (resp. $(x_2, y_2) \in \sigma_{r_v}$), and let $\Sigma_v$ be the widest multislab at $v$ whose $x$-span is contained in $[x_1, x_2]$. Note that $\Sigma_v = \sigma_v[l_v + 1, r_v - 1]$ when $v = \xi$ (Figure 3(a)), and that for any other node $v$ on the path from $\xi$ to $z_1$ (resp. $z_2$), $\Sigma_v = \sigma_v[l_v + 1, \sqrt{B}]$ (resp. $\Sigma_v = \sigma_v[1 : r_v - 1]$). At each such node $v$, we compute the maximum weight of a point in the set $P_v \cap \Sigma_v \cap Q$ in $O(\log_B n)$ I/Os using the secondary structure $\mathcal{C}_v$ and $\mathcal{M}_v$. The answer to $Q$ is then the maximum of the $O(\log_B n)$ obtained weights. We compute the maximum weight in $P_v \cap \Sigma_v \cap Q$ as follows: For a node $v$ on the path to $z_1$ and $z_2$, let $\rho_v^- = rk_v((x_1, y_1))$ and $\rho_v^+ = rk_v((x_2, y_2))$. If $v$ is the root of $T$, we compute $\rho_v^-, \rho_v^+$ in $O(\log_B n)$ I/Os using the B-tree $\Phi$. Otherwise, if we know $\rho_{p(v)}^-, \rho_{p(v)}^+$ at the parent of $v$, then we can compute $\rho_v^-, \rho_v^+$ in $O(1)$ I/Os using the secondary structure $\mathcal{C}_{p(v)}$ stored at $p(v)$. Once we know $\rho_v^-, \rho_v^+$, we query $\mathcal{M}_v$ with the multislab $\Sigma_v$ and the rank interval $[\rho_v^-, \rho_v^+]$. It returns in $O(\log_B n)$ I/Os the maximum weight of a point in the set $P_v \cap \Sigma_v \cap Q$, as desired.

Overall the query procedure uses $O(\log_B n)$ I/Os in $O(\log_B n)$ nodes, for a total of $O(\log_B^2 n)$ I/Os.

**Secondary structures.** We now describe the secondary structures stored at a node $v$ of $T$. The structure $\mathcal{C}_v$ is the same as a structure used in the CRB-tree [2], and we therefore only describe $\mathcal{M}_v$. Recall that $\mathcal{M}_v$ is a data structure of size $O(n_v/\log_B n)$, and for a multislab $\sigma_v[i:j]$ and a rank range

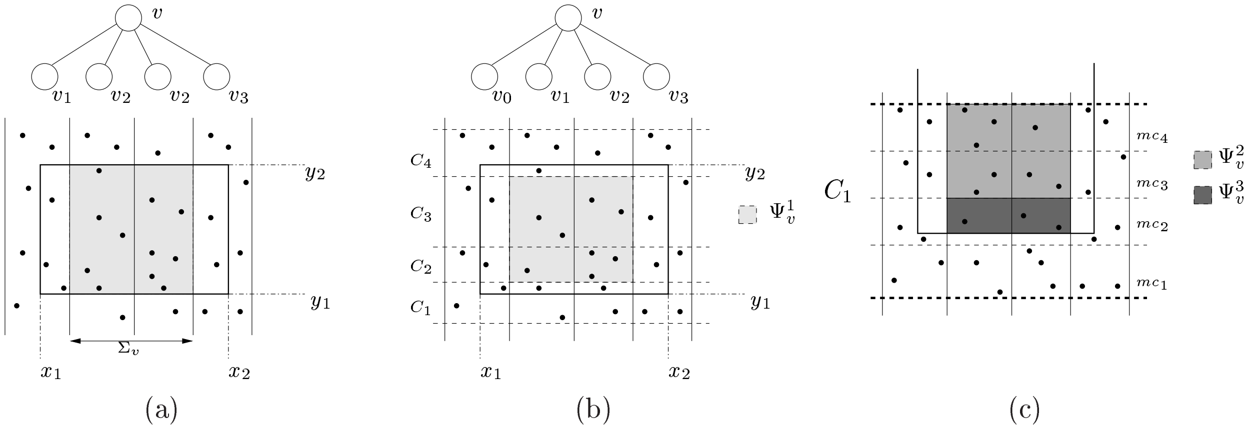


Figure 3: (a) Answering a query. (b) Finding the max at the chunk level (using $\Psi_v^1$). (c) Finding the max at the minichunk level (using $\Psi_v^2$) and inside a minichunk (using $\Psi_v^3$).

$[\rho_1, \rho_2]$, it returns the maximum weight of the points in the set $\{p \in \sigma_v[i:j] \cap P_v \mid rk_v(p) \in [\rho_1, \rho_2]\}$. Since the size of $\mathcal{M}_v$ is only $O(n_v/\log_B n)$, it cannot store all the coordinates and weights of the points in $P_v$ explicitly. Instead, we store them in a compressed manner.

Let $\mu = B\log_B n$. We partition $P_v$ into $s = \lceil N_v/\mu \rceil$ *chunks* $C_1, \ldots, C_s$, each of size (except possibly the last one) $\mu$. More precisely, $C_i = \{p \in P_v \mid rk_v(p) \in [(i-1)\mu + 1, i\mu]\}$. Next, we partition each chunk $C_i$ further into *minichunks* of size $B$. Namely, $C_i$ is partitioned into $mc_1, \ldots, mc_{\nu_i}$, where $\nu_i = \lceil |C_i|/B \rceil$ and $mc_j \subseteq C_i$ is the sequence of points whose $y$-coordinates have ranks (within $C_i$) between $(j-1)B + 1$ and $jB$. We say that a rank range $[\rho_1, \rho_2]$ *spans* a chunk (or a minichunk) $X$ if for all $p \in X$, $rk_v(p) \in [\rho_1, \rho_2]$, and that $X$ *crosses* a rank $\rho$ if there are points $p, q \in X$ such that $rk_v(p) < \rho < rk_v(q)$.

$\mathcal{M}_v$ consists of three data structures $\Psi_v^1, \Psi_v^2$, and $\Psi_v^3$; $\Psi_v^1$ answers max queries at the "chunk level", $\Psi_v^2$ answers max queries at the "minichunk level", and $\Psi_v^3$ answers max queries within a minichunk. Namely, let $\sigma_v[i:j]$ be a multislab and $[\rho_1, \rho_2]$ be a rank range, if the chunks that are spanned by $[\rho_1, \rho_2]$ are $C_a, \ldots, C_b$, then we use $\Psi_v^1$ to report the maximum weight of the points in $\bigcup_{l=a}^{b} C_l \cap \sigma_v[i:j]$ (Figure 3(b)). Let $mc_\alpha, \cdots, mc_\beta$ be the minichunks of $C_{a-1}$ that are spanned by $[\rho_1, \rho_2]$, then we use $\Psi_v^2$ to report the maximum weight of the points in $\bigcup_{l=\alpha}^{\beta} mc_l \cap \sigma_v[i:j]$. Finally, we use $\Psi_v^3$ to report the maximum weight of the points that lie in the minichunks that cross $\rho_1$ or $\rho_2$ (Figure 3(c)). We will see that the query in each of these structures takes $O(\log_B n)$ I/Os, for a total query cost at $v$ of $O(\log_B n)$ I/Os. Below we describe $\Psi_v^1, \Psi_v^2$ and $\Psi_v^3$ in details.

*Structure* $\boldsymbol{\Psi_v^3}$. $\Psi_v^3$ consists of a small structure $\Psi_v^3[l]$ for each minichunk $l$. Since we can only use $O(n_v/\log_B n)$ space to store the $N_v/B = n_v$ minichunks at $v$, we have to store $\log_B n$ small structures in $O(1)$ blocks. Thus, instead of storing the weights of the $B$ points in $l$, we store for each point $p \in l$ the index of the slab containing $p$, along with the rank of the weight of $p$ within $l$. The (slab index, weight rank) pairs for the points in $l$ are stored in $\Psi_v^3[l]$ sorted by their rank in $P_v$. For each point, we use $O(\log B)$ bits to encode the (slab index, weight rank) pair, for a total of $O(B\log B)$ bits for each $\Psi_v^3[l]$. This way $\log_B n$ small structures use $O(B\log B\log_B n) = O(B\log n)$ bits, which fit in $O(1)$ disk blocks.

To find the maximum weight of the points in minichunk $l$ that are inside multislab $\sigma_v[i:j]$ and have ranks in the range $[\rho_1, \rho_2]$, we first load the whole $\Psi_v^3[l]$ structure into memory using $O(1)$ I/Os. Then we consider the points within $\sigma_v[i:j]$ (determined using the stored slab indexes) and with ranks in $[\rho_1, \rho_2]$, and select the point $p$ with the largest weight rank (determined using the stored weight ranks). Finally, from the rank of $p$ in $P_v$, we use the identification process (the $\mathcal{C}_v$ structures) to identify the actual weight of $p$ in $O(\log_B n)$ I/Os.

*Structure* $\boldsymbol{\Psi_v^2}$. Similar to $\Psi_v^3$, $\Psi_v^2$ consists of a small structure $\Psi_v^2[k]$ for each chunk $C_k$. Since there are $N_v/\mu = n_v/\log_B n$ chunks at $v$, we can use $O(1)$ blocks for each $\Psi_v^2[k]$.

Chunk $C_k$ has $\nu_k \leq \log_B n$ minichunks $mc_1, \ldots, mc_{\nu_k}$. Consider the points in multislab $\sigma_v[i:j]$, and imagine collecting the maximum weight in each minichunk. Next consider a *Cartesian tree* [22] on these $\nu_k$ weights. A Cartesian tree on a sequence of weights $w_1, \ldots, w_{\nu_k}$ is a binary tree with the maximum weight, say $w_k$, in the root and with $w_1, \ldots, w_{k-1}$ and $w_{k+1}, \ldots, w_{\nu_k}$ stored recursively in the left and right subtree, respectively. This way, given a range of minichunks $mc_\alpha, \cdots, mc_\beta$ in $C_k$, the maximal weight in these minichunks is stored in the nearest common ancestor of $w_\alpha$ and $w_\beta$. Conceptually, $\Psi_v^2[k]$ consists of such a Cartesian tree for each of the $O(B)$ multislabs. However, we do not actually store the weights in a tree, but only an encoding of its structure. This way, we can not use it to find the maximal weight in a range of minichunks, but only the minichunk containing the maximal weight. It is well known that the structure of a binary tree of size $\nu_k$ can be encoded in $O(\nu_k)$ bits. Thus, we use $O(\log_B n)$ bits to encode the Cartesian tree of each of the $O(B)$ multislabs, for a total of $O(B \log_B n)$ bit, which again fit in $O(1)$ blocks.

Consider a multislab $\sigma_v[i:j]$. To find the maximal weight of the points in the minichunks of a chunk $C_k$ spanned by a rank range $[\rho_1, \rho_2]$, that is, to answer a minichunk level query, we first load the relevant Cartesian tree using $O(1)$ I/Os. We use it to identify the minichunk $l$ containing the maximum-weight point $p$. Then we use $\Psi_v^3[l]$ to find the rank of $p$ in $O(1)$ I/Os. Finally, we as previously use the identification process to identify the actual weight of $p$ in $O(\log_B n)$ I/Os.

*Structure* $\boldsymbol{\Psi_v^1}$. $\Psi_v^1$ is a B-tree with fanout $\sqrt{B}$ conceptually built on the $s = N_v/\mu = n_v/\log_B n$ chunks $C_1, \ldots, C_s$. Each leaf of $\Psi_v^1$ corresponds to $\sqrt{B}$ contiguous chunks, and stores for each of the $\sqrt{B}$ slabs in $v$, the point with the maximum weight in each of the $\sqrt{B}$ chunks. Thus a leaf stores $O(B)$ points and fit in $O(1)$ blocks. Similarly, an internal node of $\Psi_v^1$ stores for each of the $\sqrt{B}$ slabs the point with the maximal weight in each of the subtrees rooted in its $\sqrt{B}$ children. Therefore an internal node also fit in $O(1)$ blocks, and $\Psi_v^1$ use $O(n_v/(\log_B n\sqrt{B})) = O(n_v/(\log_B n)$ blocks in total.

Consider a multislab $\sigma_v[i:j]$. To find the the maximum weight in chunks $C_a, \cdots, C_b$ spanned by a rank range $[\rho_1, \rho_2]$, we visit the nodes on the paths from the root of $\Psi_v^1$ to the leaves corresponding to $C_a$ and $C_b$. In each of these $O(\log_B n)$ nodes we consider the points contained in both multislab $\sigma_v[i:j]$ and one of the chunks $C_a, \cdots, C_b$, and select the maximal weight point. This takes $O(1)$ I/Os. The answer to the query is then the maximal weight of the $O(\log_B n)$ selected points.

**Bulk loading.** Our structure can be bulk-loaded efficiently bottom-up, level by level. The ordinary B-tree $\Phi$ can be constructed easily in $O(n \log_{M/B} N)$ I/Os, so we concentrate on the construction of $T$. We construct the leaves of $T$ using $O(n \log_{M/B} N) = O(n \log_B N)$ I/Os by sorting the points in a non-decreasing order of their $x$-coordinates. In each leaf $v$, the points are sorted in their $y$-coordinates to form $P_v$. Below we describe how we construct all nodes and secondary structures at level $i$ of $T$, given that we have already constructed the nodes at level $i+1$.

We construct a level $i$ node $v$ and it secondary structures as follows. We first compute $P_v$ by merging $P_{v_1}, \ldots, P_{v_{\sqrt{B}}}$ together. Since $M > B^2$, the memory can hold a block from each $P_{v_i}$ at the same time, and this merging process can be done in a single scan, which takes $O(n_v)$ I/Os. The secondary structure $\mathcal{C}_v$ can be constructed easily during this scan [2]. The structure $\Psi_v^1$ is a B-tree built on top of the maximum weights of each chunk and each slab. These weights can also be computed by a scan of $P_v$ and thus $\Psi_v^1$ can be constructed in $O(n_v)$ I/Os. Since each minichunk has $B$ points, we can easily compute the ranks of their weights for each minichunk by a scan of $P_v$, so $\Psi_v^3$ can be constructed in $O(n_v)$ I/Os.

Now we focus on the construction of $\Psi_v^2$, i.e., how to build and encode the Cartesian trees on the minichunk level for each multislab inside each $\Psi_v^2[k]$. We are now facing the following problem: given $a = O(B)$ sequences of $b = \log_B N$ weights $w_{1,1}, \ldots, w_{1,b}$; $w_{2,1}, \ldots, w_{2,b}$; $\ldots$; $w_{a,1}, \ldots, w_{a,b}$, where $w_{i,j}$ is the maximum weight of multislab $i$ in mini-chunk $j$, we want to build and encode a Cartesian tree for each of the sequence. These $w_{i,j}$'s can be easily computed with a scan of $P_v$ and we insert them as they become available into the $a$ partially constructed Cartesian trees. To insert $w_{i,j}$ into the $i$-th Cartesian tree made of $\{w_{i,1}, \ldots, w_{i,j-1}\}$, we only need to compare $w_{i,j}$ with the rightmost path bottom-up, until we reach a node $u$ whose weight is no less than $w_{i,j}$. Then we make the right child of $u$ the left child of $w_{i,j}$, which now becomes the right child of $u$. From this procedure we see that only the weights on the rightmost path need to be maintained explicitly, while other nodes can be encoded as they leave the rightmost path. Since the entire encoding of a Cartesian tree is $O(\log_B N)$ bits, we can always maintain all the encoded parts of the trees in memory, while each of the rightmost paths is implemented as a stack, which might be partially stored on disk. Since $M > B^2$, we can have $\Omega(B)$ words for each stack to act as a buffer such that the $O(b)$ push and pop operations take $O(b/B)$ I/Os for each stack, which amounts to a total of $O(b)$ I/Os. Therefore, we spend $O(\log_B N)$ I/Os to construct $\Psi_v^2[k]$, and the total number of I/Os spent for constructing $\Psi_v^2$ is $O((n_v/\log_B N)\log_B N) = O(n_v)$.

In summary, we spend $O(n_v)$ I/Os to construct each secondary structure at node $v$, so the total cost of the bulk loading is $O(n \log_B n)$ I/Os. This completes the description of our static two-dimensional range max structure.

**Theorem 1** *A set of $N$ points in the plane can be stored in a linear-size structure such that an orthogonal range max query can be answered in $O(\log_B^2 n)$ I/Os. The structure can be constructed in $O(n \log_B n)$ I/Os.*

**Dynamization.** We first describe how to update the structure when we delete a point $p$ from $S$. In fact, we do not actually delete it from the base tree, instead we only mark it as deleted and update the $\mathcal{M}_v$ structures. To ensure that the height of the base tree is $O(\log_B n)$, we rebuild the whole structure once we have collected $N/2$ deletions. Recall that $\mathcal{M}_v$ consists of three separate structures $\Psi_v^1, \Psi_v^2$, and $\Psi_v^3$. Since we do not know how to handle deletions in a Cartesian tree efficiently, which is the building block of $\Psi_v^2$, we no longer partition each chunk $C_k$ of $P_v$ into minichunks. Instead we construct $\Psi_v^3[k]$ directly on the points in $C_k$. Since $|C_k| \le B \log_B N$, $\Psi_v^3[k]$ now uses $O(\log_B \log_B n)$ blocks, instead of $O(1)$ blocks as in the static structure. Hence, the overall size of the structure now becomes $O(n \log_B \log_B n)$ blocks, and the construction cost becomes $O(n \log_B n \log_{M/B} \log_B n)$ I/Os accordingly.

If we know $rk_v(p)$, we can determine the chunk $C_k$ that contains $p$ and can update $\Psi_v^3[k]$ in $O(\log_B \log_B n)$ I/Os. Suppose $p$ is stored at the $i$th child of $v$. Let $r = rk_v(q')$ and $q'$ is the maximum-weight point in $C_k \cap P_{v_i}$ after deleting $p$ from $C_k$. As described above, we can identify, in $O(\log_B n)$ I/Os, the weight $w$ of $q'$ whose rank in $P_v$ is $r$. Once we have $w$, we can update $\Psi_v^1$ in $O(\log_B n)$ I/Os in a straightforward manner—we first update the information corresponding to the $i$th slab at the leaf $z$ that stores $C_k$, and then update the information at the ancestors of $z$. Since $p$ is stored at $O(\log_B n)$ nodes of $T$, the total amortized cost in deleting $p$ is $O(\log_B^2 n)$ I/Os.

Finally, we can use the external logarithmic method [7] to handle insertions in amortized $O(\log_B^2 n \log_{M/B} \log_B n)$ I/Os by paying an extra $O(\log_B n)$ factor to the query cost. Putting everything together, we obtain the following.

**Theorem 2** *A set of $N$ points in the plane can be stored in a structure that uses $O(n \log_B \log_B n)$ disk blocks so that a range max query can be answered in $O(\log_B^3 n)$ I/Os. A point can be inserted or deleted in $O(\log_B^2 n \log_{M/B} \log_B n)$ and $O(\log_B^2 n)$ I/Os amortized, respectively.*

**Remarks.** In the appendix we describe the various extensions and improvements.
*(i)* By using Cartesian trees to implement $\Psi_v^1$ and a technique to speed up the identification process due to Chazelle [12], we can improve the query bound of our linear-size static structure to $O(\log_B^{1+\epsilon} n)$ I/Os. However, we can not construct this structure efficiently and therefore cannot make it dynamic. The details appear in Appendix A.
*(ii)* Using standard techniques [2, 10], both our static and dynamic structures can be extended to $d$-dimension with an $O(\log_B^{d-2} n)$ factor increase in the space, query and update bounds. The details appear in Appendix B.

# 3 Stabbing Queries

In this section we describe our dynamic structure for stabbing queries. In Section 3.1 we describe our structure for the one-dimensional case, and in Section 3.2 we then sketch how to extend it to two and higher dimensions.

## 3.1 1D structure

Given a set $S$ of $N$ intervals, where each interval $\gamma \in S$ is assigned a weight $w(\gamma)$, we want to compute the maximum-weight interval in $S$ containing a query point. Our structure for this problem is based on the external interval tree of Arge and Vitter [8], as well as on the ideas utilized in the point location structure of Agarwal et al [1]. We are mainly interested in the dynamic case, since the static version of the problem is easily solved.

**Overall structure.** Our structure consists of a fanout $\sqrt{B}$ base B-tree $T$ on the endpoints of the intervals in $S$, with the intervals stored in secondary structures associated with the internal nodes of $T$. Each leaf represents $B$ consecutive points and the tree has height $O(\log_B n)$.

As in Section 2, a canonical interval $\sigma_v$ is associated with each node $v$; $\sigma_v$ is partitioned into $k \leq \sqrt{B}$ *slabs* by the ranges $\sigma_{v_1}, \ldots, \sigma_{v_k}$, associated with the children $v_1, v_2, \ldots, v_k$ of $v$. An input interval $\gamma$ is assigned to $v$ if $\gamma \subseteq \sigma_v$ but $\gamma \nsubseteq \sigma_{v_i}$ for any $1 \leq i \leq k$. Let $S_v$ be the set of intervals assigned to $v$. A leaf $z$ stores intervals whose both endpoints lie in $\sigma_z$. The $O(B)$ intervals $S_z$ assigned to $z$ are stored using $O(1)$ blocks. At each internal node $v$, $\Theta(\sqrt{B})$ secondary structures are used to store the set of intervals $S_v$ assigned to $v$. A *left slab structure* $\mathcal{L}_v[i]$ and a *right slab structure* $\mathcal{R}_v[i]$, for each of the $\sqrt{B}$ slabs, as well as a *multislab structure* $\mathcal{M}_v$. The left (right) slab structure $\mathcal{L}_v[i]$ contains intervals from $S_v$ whose left (right) endpoints lie in $\sigma_{v_i}$. They use linear space and supports stabbing queries for points in $\sigma_{v_i}$ in $O(\log_B n)$ I/Os. The multislab structure $\mathcal{M}_v$ stores all intervals that span at least one slab. It uses linear space, and for any query point $q \in \sigma_{v_i}$, it returns in $O(1)$ I/Os the maximum-weight interval that completely spans $\sigma_{v_i}$. We describe the slab and multislab structures below. Overall, an interval is stored in at most three secondary structures (two slab structures and possibly one multislab structure). Refer to Figure 4(a). Since all secondary structures use linear space, the structure uses linear space.

**Answering a query.** To answer a query $q$, we search down the base tree $T$ for the leaf containing $q$. At each of the $O(\log_B n)$ nodes $v$ on the path, we compute the maximum-weight interval of $S_v$ containing $q$. The maximum-weight interval in $S$ containing $q$ is the maximum-weight interval of these $O(\log_B n)$ intervals. To answer a query at an internal node $v$ with $q \in \sigma_{v_i}$, we simply query the left slab structure $\mathcal{L}_v[i]$ and right slab structure $\mathcal{R}_v[i]$ to compute the maximum-weight interval whose one endpoint lies in $\sigma_{v_i}$ and that contains $q$. We then query the multislab structure $\mathcal{M}_v$ to compute the maximum-weight interval spanning $\sigma_{v_i}$. Refer to Figure 4(b). At the leaf $z$, we simply scan the $O(B)$ intervals stored at $z$ to find the maximum. Since we spend $O(\log_B n)$ I/Os in each node, we in total answer a query in $O(\log_B^2 n)$ I/Os.

**Secondary structures.** We now describe the secondary structures stored at a node $v$ of $T$.

*Left/right slab structure.* Let $R_v^i \subseteq S_v$ be the set of intervals whose right endpoints lie in $\sigma_{v_i}$. These intervals are stored in the right slab structure $\mathcal{R}_v[i]$. Answering a stabbing query on $R_v^i$ with a point $q \in \sigma_{v_i}$ is equivalent to answering a one-dimensional range max query $[q, \infty]$ on the right endpoints of $R_v^i$. Refer to Figure 4(c). This can easily be implemented using a B-tree. The left slab structure is implemented in a similar way. The slab structures use linear space, and support queries and updates in $O(\log_B n)$ I/Os. Given the intervals in $R_v^i$ sorted by left and right endpoints, the structures can be constructed in a linear number of I/Os.

*Multislab structure.* A multislab structure $\mathcal{M}_v$ stores intervals $S'_v$ from $S_v$ that span at least one slab. $\mathcal{M}_v$ is a fan-out $\sqrt{B}$ B-tree on $S'_v$, and we use interval id's as keys to construct the $B$-tree. Each leaf of $\mathcal{M}_v$ corresponds to $B$ intervals, and $\mathcal{M}_v$ has height $O(\log_B n)$. For a node $u \in \mathcal{M}_v$, let $\gamma_{ij}$ be the maximum-weight interval that spans $\sigma_{v_i}$ and that is stored in the subtree of $\mathcal{M}_v$ rooted at the $j$-th child of $u$. For $1 \le i, j \le \sqrt{B}$, we store $\gamma_{ij}$ at $u$. In particular, the root of $\mathcal{M}_v$ stores the maximum-weight interval spanning each of the $\sqrt{B}$ slabs. $\mathcal{M}_v$ uses linear space, and a stabbing query $q$ in any slab $\sigma_{v_i}$ can be answered in $O(1)$ I/Os. (Intuitively, $\mathcal{M}_v$ corresponds to combining $\sqrt{B}$ B-trees with fan-out $\sqrt{B}$, one for each slab, into a single B-tree). As the slab structures, $\mathcal{M}_v$ can easily be constructed in a linear number of I/Os if the intervals are given sorted by left endpoints.

A multislab structure $\mathcal{M}_v$ can be updated in $O(\log_B n)$ I/Os. To insert or delete an interval $\gamma$, we first search down $\mathcal{M}_v$ to find and update the relevant leaf $z$. After updating $z$, some of the intervals stored at the nodes on the path $P$ from the root of $\mathcal{M}_v$ to $z$ may need to be updated. To maintain a balanced tree, we may also need to perform B-tree rebalancing operations (splits, fuses, and shares) on the nodes on $P$. Both for insertions and deletions we can easily do so in $O(\log_B n)$

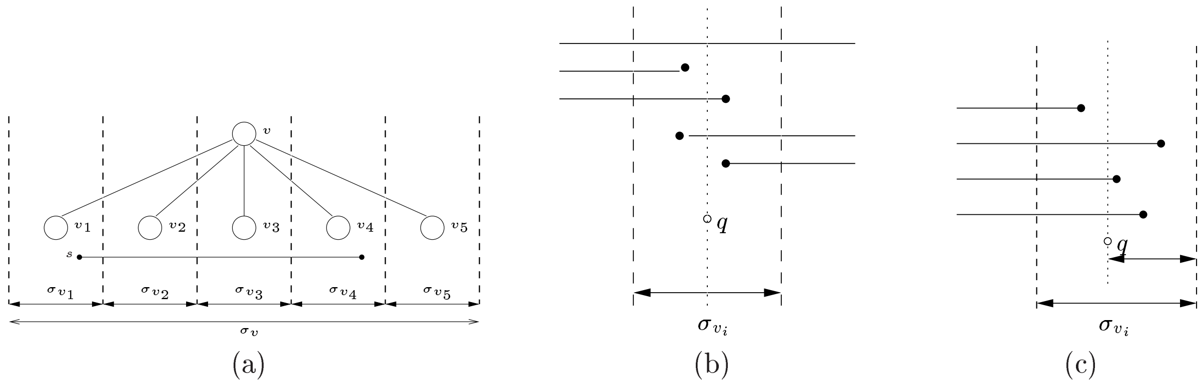


Figure 4: (a) Node $v$ in the base tree. The range $\sigma_v$ associated with $v$ is divided into 5 slabs. Interval $s$ is stored in the left slab structure corresponding to $\sigma_{v_1}$ and the right slab structure corresponding to $\sigma_{v_4}$, as well as in the multislab structure $\mathcal{M}_v$. (b) Querying a node with $q$. (c) Equivalence between a stabbing max query $q$ and a one-dimensional range max query $[q, \infty]$.

I/Os in a traversal of $P$ from $z$ towards the root.

**Updates.** To insert a new interval $\gamma$, we first insert the endpoints of $\gamma$ in $T$. Below we sketch how to do so in $O(\log_B n)$ I/Os. Next we use $O(\log_B n)$ I/Os to search down $T$ for the node $v$ where $\gamma$ needs to be inserted in the secondary structures. Finally, we use another $O(\log_B n)$ I/Os to insert $\gamma$ in a left and right slab structure, as well as in the multislab structure $\mathcal{M}_v$ when it spans at least one slab.

We use techniques similar to the ones used in [8] to insert a new interval into $T$: We implement $T$ using a weight-balanced B-tree, which is similar to the standard B-tree except that balance is maintained using a weight constraint rather than a degree constraint. As for B-trees, rebalancing after an insertion is performed using $O(\log_B n)$ node splits. The key property of a weight-balanced B-tree is that after a split of a node $v$ the number of inserts that has to be performed below $v$ before another split is needed is proportional to the number of points in the subtree $T_v$ rooted in $v$. The number of intervals stored in secondary structures of $v$ is also proportional to the number of points in $T_v$ (they all have endpoints in $T_v$). Since the relevant secondary structures can be reconstructed in a linear number of I/Os after a split of $v$ (since the intervals in $v$ are already sorted), the amortized split cost is $O(1)$ I/Os. Refer to [8] for details. The split bound can even be made worst-case using the lazy rebuilding techniques [8]. Thus in total an insertion requires $O(\log_B n)$ I/Os.

Deletions can be handled using global rebuilding [19]. To delete an interval $\gamma$, we first delete it from the relevant secondary structures using $O(\log_B n)$ I/Os, similar to the way we performed an insertion. Unlike insertions, however, we do not immediately delete the endpoints of $\gamma$ from $T$. Instead we just mark the two endpoints in the leaves of $T$ as deleted. To maintain the $O(\log_B n)$ tree height, we rebuild the structure completely after $N/2$. In the full paper we describe how we can easily rebuild the structure in $O(n \log_B n)$ I/Os, obtaining an $O(\log_B n)$ amortized delete bound. This bounds can again be made worst-case using standard lazy rebuilding techniques.

**Theorem 3** *A set of $N$ intervals can be stored in a linear space data structure such that a stabbing max query can be answered in $O(\log_B^2 n)$ I/Os, and such that updates can be performed in $O(\log_B n)$ I/Os. The structure can be constructed in $O(n \log_B n)$ I/Os.*

**Remarks.**
*(i)* Our structure can easily be modified to handle not only max stabbing queries, but more generally *semigroup stabbing queries*. Let $(\mathbf{S}, +)$ be a commutative semigroup. Given a set of $N$ intervals $S$, where interval $\gamma \in S$ is assigned a weight $w(\gamma) \in \mathbf{S}$, the result of a semigroup stabbing query $q$ is $\sum_{q \in \gamma, \gamma \in S} w(\gamma)$. Max queries is the special case where the semigroup is taken to be $(\mathbb{R}, \max)$. Unlike the structure presented in this section, the 2D range max structure described in Section 2 can not be generalized, since it takes the advantage of fact that in the semigroup $(\mathbb{R}, \max)$ the result of a semigroup operation is one of the operands.
*(ii)* By increasing the fanout of the base tree from 2 to $\sqrt{\log N}$ and using the ideas presented in this section, we can improve the internal memory stabbing max structure. The improved structure uses linear space and supports queries and updates in $O(\log^2 N / \log\log N)$ time. The result holds in the pointer-machine model of computation.
*(iii)* By combining the ideas used in our structure with ideas from the external segment tree of Arge and Vitter [8], we can obtain a space-time tradeoff. More precisely, for any $\epsilon > 0$, a set of $N$ intervals can be stored in a structure that uses $O(n \log_B^{\epsilon} n)$ disk blocks, such that a stabbing max query can be answered in $O(\log_B^{2-\epsilon} n)$ I/Os, and such that updates can be performed in $O(\log_B^{1+\epsilon} n)$ I/Os amortized. Details appear in Appendix C.

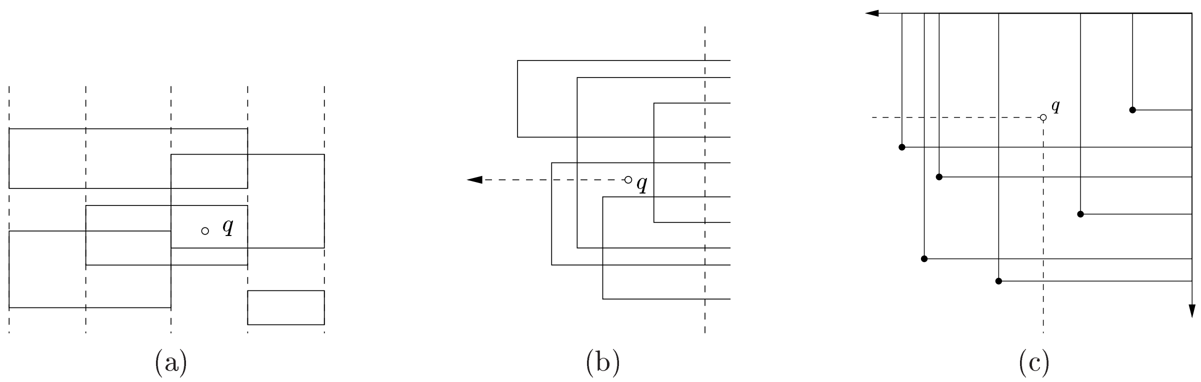


Figure 5: Dividing a 2D stabbing query into parts.

### 3.2 2D structure

We can extend our structure to 2D by using standard techniques, e.g. as in [12], and using our 2D range query structure as a building block. We sketch our structure below and omit the details.

Let $S$ be a set of $N$ rectangles in $\mathbb{R}^2$. Like in our 1D stabbing query structure, we build a base B-tree $T$ with fanout $B^{1/3}$ on the $x$-coordinates of the corners of the rectangles in $S$. Each node $v$ of $T$ is now associated with a slab $\sigma_v$, which is partitioned into $k$ ($\leq B^{1/3}$) vertical slabs by its $k$ children, as before. Let $S_v$ be the set of rectangles stored at $v$. A rectangle $\gamma$ is stored at an internal $v$ of $T$ if $\gamma \subseteq \sigma_v$ but $\gamma \nsubseteq \sigma_{v_i}$ for any child $v_i$ of $v$. As before, each internal node $v$ of $T$ stores a multislab structure and one left (right) slab structure for each of its slab boundaries. Thus, a multislab structure of $T$ stores rectangles that span multislabs, like those shown in Figure 5(a), and a left slab structure of $T$ stores rectangles like those shown in Figure 5(b).

To implement the multislab structure associated with a node $v$ of $T$, we use the same "combining" technique again: we build one 1D stabbing structure $T'_v[i]$ for each of the $B^{1/3}$ slabs on the $y$-projections of those rectangles that span the corresponding slab, and then combine them into a single structure $T'_v$. This is possible by lowering both the fanout of the base tree of $T'_v[i]$ and the fanout of all the B-trees used in the secondary structures associated with $T'_v[i]$ to $B^{1/3}$. Thus, $T'_v$ can answer a stabbing query in $O(\log_B^2 n)$ I/Os.

The left (or right) slab structure associated with an internal node $v$ of $T$ stores rectangles like those that are shown in Figure 5(b). These rectangles, still, can be stored in another 1D stabbing structure $T''_v$ built on their projections onto the $y$-axis. Each multislab structure of $T''_v$ can be easily implemented as a B-tree that supports queries in $O(\log_B n)$ I/Os. While each left (right) slab structure of $T''_v$ stores rectangles like those shown in Figure 5(c). A stabbing query for these rectangles is equivalent to a two-sided range query for their left-lower corners, and we use our 2D range query structure to handle these queries.

To answer a stabbing max query, we follow the path in $T$ and visit $O(\log_B n)$ nodes. At each node, we answer one Figure 5(a)-type query in $O(\log_B n)$ I/Os, and two Figure 5(b)-type queries, each of which is further decomposed into $O(\log_B n)$ Figure 5(c)-type queries. We can also make the structure dynamic by the external logarithmic method. Omitting further details, we have the following.

**Theorem 4** *For a set of $N$ rectangles in $\mathbb{R}^2$, there exists a linear-size structure that can answer stabbing max queries in $O(\log_B^4 n)$ I/Os. For the dynamic case, there exists a structure that uses $O(n \log_B \log_B n)$ disk blocks, answers stabbing max queries in $O(\log_B^5 n)$ I/Os, performs insertions and deletions in $O(\log_B^2 n \log_{M/B} \log_B n)$ and $O(\log_B^2 n)$ I/Os amortized, respectively.*

**Remark.** Using standard techniques, the results in Theorem 4 can be extended to $d$-dimension ($d > 2$), at the cost of increasing the space, query, and update bounds by a factor of $O(\log_B^{d-2} n)$.

## 4 Conclusions

We have developed linear or almost linear external memory data structures that can answer range or stabbing max queries in 1D and 2D, either in a static or a dynamic setting. Previous results either use more space, do not provide worst case query performance guarantee, or do not support deletions for certain problems. One disadvantage of our structures, however, is that they are often too complicated to be implemented in practice.

We conclude by mentioning some open questions. 1) Can we get rid of the extra $O(\log_B \log_B n)$ factor in the space bound of our 2D dynamic range max structure? 2) Can we modify our range max structures to handle general semigroup queries? 3) Can we improve the query bound of our 1D and 2D stabbing max structure?

## References


[1] P. K. Agarwal, L. Arge, G. S. Brodal, and J. S. Vitter. I/O-efficient dynamic point location in monotone planar subdivisions. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, pages 1116–1127, 1999.

[2] P. K. Agarwal, L. Arge, and S. Govindarajan. CRB-tree: An optimal indexing scheme for 2d aggregate queries. In *Proc. International Conference on Database Theory*, 2003.

[3] P. K. Agarwal and J. Erickson. Geometric range searching and its relatives. In B. Chazelle, J. E. Goodman, and R. Pollack, editors, *Advances in Discrete and Computational Geometry*, volume 223 of *Contemporary Mathematics*, pages 1–56. American Mathematical Society, Providence, RI, 1999.

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

[5] S. Alstrup, C. Gavoille, H. Kaplan, and T. Rauhe. Nearest common ancestors: a survey and a new distributed algorithm. In *SPAA 2002*, pages 258–264, 2002.

[6] L. Arge. External memory data structures. In J. Abello, P. M. Pardalos, and M. G. C. Resende, editors, *Handbook of Massive Data Sets*, pages 313–358. Kluwer Academic Publishers, 2002.

[7] L. Arge and J. Vahrenhold. I/O-efficient dynamic planar point location. In *Proc. ACM Symposium on Computational Geometry*, pages 191–200, 2000.

[8] L. Arge and J. S. Vitter. Optimal dynamic interval management in external memory. In *Proc. IEEE Symposium on Foundations of Computer Science*, pages 560–569, 1996.

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

[10] J. L. Bentley. Multidimensional divide and conquer. *Communications of the ACM*, 23(6):214–229, 1980.

[11] O. Berkman and U. Vishkin. Recursive star-tree parallel data structures. *SIAM Journal on Computing*, 22(2):221–242, 1993.

[12] B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM J. Comput.*, 17(3):427–462, June 1988.

[13] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[14] H. Edelsbrunner and H. A. Maurer. On the intersection of orthogonal objects. *Information Processing Letters*, 13:177–181, 1981.

[15] D. Eppstein and S. Muthukrishnan. Internet packet filter management and rectangle geometry. In *Proc. 12th ACM-SIAM Symp. Discrete Algorithms*, pages 827–835, 2001.

[16] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[17] R. Grossi and G. F. Italiano. Efficient cross-tree for external memory. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*, pages 87–106. American Mathematical Society, DIMACS series in Discrete Mathematics and Theoretical Computer Science, 1999. Revised version available at ftp://ftp.di.unipi.it/pub/techreports/TR-00-16.ps.Z.

[18] K. V. R. Kanth and A. K. Singh. Optimal dynamic range searching in non-replicating index structures. In *Proc. International Conference on Database Theory, LNCS 1540*, pages 257–276, 1999.

[19] M. H. Overmars. *The Design of Dynamic Data Structures.* Springer-Verlag, LNCS 156, 1983.

[20] J. Robinson. The K-D-B tree: A search structure for large multidimensional dynamic indexes. In *Proc. SIGMOD International Conference on Management of Data*, pages 10–18, 1981.

[21] J. S. Vitter. Online data structures in external memory. In *Proc. International Colloquium on Automata, Languages, and Programming, LNCS 1644*, pages 119–133, 1999.

[22] J. Vuillemin. A unifying look at data structures. *Comm. ACM*, 23:229–239, 1980.

[23] J. Yang and J. Widom. Incremental computation and maintenance of temporal aggregates. In *Proc. IEEE International Conference on Data Engineering*, pages 51–60, 2001.

# A An Alternative Static Structure

## A.1 The $\Psi_v^1$ Structure

The $\Psi_v^1$ structure is used to answer max queries on the chunk level for each multislab. But building a Cartesian tree for each multislab separately is not I/O efficient, so we use a variant of the *discrete range max* algorithm [11, 5] and combine the structures for the $O(B)$ multislabs into a single structure. The resulted structure still allows us to answer any range max query on the chunk level for any multislab in $O(1)$ I/Os. We divide the $N_v/\mu = n_v/\log_B N$ chunks into groups of $\log N$: $\beta_1, \beta_2, \ldots$ and build a complete binary tree on top of these groups with leaves $l_1, l_2, \ldots$. To answer a max query, we first answer the query on the group level and then use (at most) two lookups in the boundary groups. With each internal node $u$ of the binary tree we associate a *left weights* array $lw_u$ that stores the maximum among the weights in its left subtree for multislab $k$ in $lw_u[k]$. The *right weights* array $rw_u$ are defined similarly. At each leaf $l_i$, we store two two-dimensional tables $\mathit{left}_{l_i}$ and $\mathit{right}_{l_i}$. $\mathit{left}_{l_i}[0][k]$ and $\mathit{right}_{l_i}[0][k]$ is defined to be the maximum in the group $\beta_i$ for multislab $k$; for $j > 0$, $\mathit{left}_{l_i}[j][k]$ is defined recursively as $\max\{\mathit{left}_{l_i}[j-1][k], rw_u[k]\}$ where $u$ is the $j$-th closest ancestor of $l_i$ and $l_i$ is in the right subtree of $v$; $\mathit{right}_{l_i}$ is defined similarly. To answer a group-level max query $(\beta_i, \beta_j)$ for a particular multislab $k$, we look at the two leaf nodes $l_i$ and $l_j$ and we can use a simple NCA algorithm for complete binary trees to compute the height $h$ of the NCA of $l_i$ and $l_j$. Then we return the maximum among $\mathit{right}_{l_i}[h-1][k]$ and $\mathit{left}_{l_j}[h-1][k]$ in $O(1)$ I/Os. Each leaf requires $\log(n_v/(\log_B N \log N))$ blocks and there are $n_v/(\log_B N \log N)$ leaves so the binary tree takes $O(n_v/\log_B N)$ blocks (linear size in total). The structure can also be constructed in $O(n_v/\log_B N)$ I/Os once the $\mathit{left}_{l_i}[0][k]$'s and $\mathit{right}_{l_i}[0][k]$'s are known for each group $\beta_i$, which in turn, can be easily computed as a byproduct of the earlier single scan of $P_v$. Note that all the internal nodes of the binary tree can be discarded once the tree is constructed since we do not need them when answering queries.

We now need to specify the structure used to find the maximum within a group of $\log N$ chunks for any multislab. The problem at hand is, given $s = O(B)$ sequences of $t = \log N$ weights $x_{1,1}, \ldots, x_{1,t}$; $x_{2,1}, \ldots, x_{2,t}$; $\ldots$; $x_{s,1}, \ldots, x_{s,t}$, and we need to support DRM queries for any of them, where $x_{i,j}$ is the maximum weight of multislab $i$ in chunk $j$. They can also be easily computed by the scan of $P_v$ and are stored in disk in the following order: $x_{1,1}, \ldots, x_{s,1}$; $x_{1,2}, \ldots, x_{s,2}$; $\ldots$. For each $x_{i,j}$, we compute $g(i,j) = \max\{k | k < j \text{ and } x_{i,k} > x_{i,j}\} \cup \{-1\}$. It is not hard to see that we can do so by scanning the sequence $x_{i,1}, \ldots, x_{i,t}$ and maintaining a stack which stores the largest weights in all the suffixes of the prefix of the sequence that has been scanned. We can scan the $s$ sequences "in parallel" in $t$ I/Os. Since the memory can hold $B^2$ words, we can allocate $\Omega(B)$ words in memory to each of the $s$ stacks so that all operations on one stack takes $O(t/B)$ I/Os, making a total of $O(t)$ I/Os. Based on the values of $g(i,j)$ we associate a label, $l(i,j)$, of size $t$ bits with each $x_{i,j}$. The $k$th bit of $l(i,j)$ is set if and only if $k < j$, $x_{i,k} > x_{i,j}$, and for each $k < r < j$, $x_{i,r} < x_{i,k}$. We compute these labels as follows. First set $l(i,1) = 0$ for all $i$. For $j > 1$, $l(i,j)$ is the same as $l(i, g(i,j))$ but with bit $g(i,j)$ also set. Notice that if $g(i,j) = a$, then for any $k > j$, we have either $g(i,k) < a$ or $g(i,k) > j$. So, we can also maintain the already computed $l(i,j)$ values in a stack for each sequence for later lookup and similarly, the operations done on all the stacks sum up to $O(t)$ I/Os. At this point, the group structure is finished and we can throw away the $g(i,j)$'s and keep the $l(i,j)$'s and the the weights in $ts/B = O(\log N)$ blocks. There are $n_v/(\log_B N \log N)$ groups so the storage of this group structure in $\Sigma_v$ is $O(n_v/\log_B N)$ blocks, which again, is linear. The construction cost is also $O(n_v/\log_B N)$ I/Os for each $v$. To answer a DRM query $\max(j,k)$ for the $i$th sequence, we look at $l(i,k)$ and clear all bits with index smaller than $j$, getting a word $w$. Then we return $x_{i,k}$ if $w = 0$, otherwise we return $x_{i,\mathrm{lsb}(w)}$, where $\mathrm{lsb}(w)$ is the index of the least significant bit of $w$. Therefore, all the follow-up work to construct $\Psi_v^1$ is $O(n_v/\log_B N)$ I/Os after a scan of $P_v$.

### A.2 Speed up the Identification Process

We notice that in the algorithm described in Section 2, identifying a point in $P_v$ is the bottleneck of the query process. It turns out that we can shorten this process and get some kind of space-time tradeoff. We basically use the same technique as in [12] but in an I/O setting.

We define the *level* of a node of $T$ as its number of ancestors. Let $w$ be a descendent of $v$, and let $r$ be the rank of $p_v^r$ in $P_v$ which we need to identify. How can we compute the rank $s$ of $p_w^s \in P_w$ in $O(1)$ I/Os such that $p_v^r = p_w^s$? First note that $w$ cannot be just any descendent of $v$, at any given level of $T$, there is a unique node $w$ that $p_v^r \in P_w$. The problem at hand is that of computing a batch of consecutive transitions, and more precisely, the function $\theta(v,r,l) = (w,s)$, where $p_w^s = p_v^r$ and $w$ is at level $l$ of $T$. Whenever $l$ is understood, we will say that $(w,s)$ is the companion of $(v,r)$.

We show how to compute $\theta(v,r,l)$ in $O(1)$ I/Os, adding only a small amount of storage to the secondary structure $\Sigma_v$. Obviously, we only need to do that for $l$ larger than the level of $v$. Let $h$ be the difference between $l$ and the level of $v$, and let $w_0, \ldots, w_{B^{h/2}-1}$ denote the descendents of $v$ at level $l$. This time, $P_v$ is broken up into groups $\beta_0, \beta_1, \ldots$ of size $\lambda = \log_B N$. (Note that $\lambda$ entries of $CI_v$ fit in one word). We cannot afford to attach with each point in $P_v$ the address of its companion, but we still want to do so for one point in each group, say, for the last point in each group $p_v^{\lambda k-1}, k = 1, \ldots, N_v/\lambda$. In order to avoid the situation where all these companions land in the same node at level $l$, we do not link the points to their companions but to *neighboring points* chosen in a round-robin fashion among the nodes at level $l$. We augment each group $\beta_i$ with a pointer $\gamma_i$ and a bit vector $\Gamma_i$, defined as follows.

The pointers $\gamma_i$ allow us to jump from selected places in $v$ to related places at level $l$. For any $j$ such that $0 \le j < B^{h/2}$, we say that $(w_j, s)$ is the *$j$-companion* of $(v,r)$ if the $y$-coordinate of $p_{w_j}^s$ is the smallest in $P_{w_j}$ that is at least as large as the $y$-coordinate of $p_v^r$. Note that there is a unique $j$ for which the $j$-companion of $(v,r)$ is also its companion. We define $\gamma_i$ as the address of the group in $w_{i \bmod B^{h/2}}$ that contains the $(i \bmod B^{h/2})$-companion of the last point in $\beta_i$. If there is no such companion, $\gamma_i$ need not to be defined.

The bit vectors $\Gamma_i$ allow us to correct the errors introduced by the $\gamma_i$'s. Let $\theta(v,r,l) = (w_j, s)$, and let $k$ be the largest integer less than $i$ such that $\gamma_k$ points to a group of $w_j$. Suppose for the time being that $k$ is well defined. Then it is always at least $i - B^{h/2}$ because of the round-robin scheme. The address in $\gamma_k$ allows us to jump from $v$ to $w_j$ within some reasonable accuracy. To complete the computation, we need information to (1) compute $k$, given $(v,r)$, and (2) go from $\gamma_k$ to the actual companion of $(v,r)$. If $p_v^r$ is the $d$th point in the group of $\beta_i$, we define $X_d$ as an $\lceil h/2 \cdot \log B \rceil$-bit vector containing the binary representation of $i - k$. This allows us to complete the first task. For the second one, suppose $p_{w_j}^a$ is the first point in the group pointed by $\gamma_k$, we then encode the binary representation of $s - a$ in another bit vector $Y_d$. By the choice of $k$, there are at most $\lambda(B^{h/2}+1)$ points between $p_{w_j}^a$ and $p_{w_j}^s$, so $\lceil \log(s-a) \rceil < \lceil \log \lambda + h/2 \cdot \log B \rceil + 1$ bits are enough for $Y_d$ to encode $s - a$. If $k$ is not defined then $X_d$ and $Y_d$ store the values of $j$ and $s$, respectively, with a flag indicating so; again, this can be encoded in $O(\log \lambda + h \log B)$ bits. Finally, we define $\Gamma_i$ as the concatenated string $X_1 Y_1 X_2 Y_2 \cdots X_\lambda Y_\lambda$. Note that $\Gamma_i$ is $O(\lambda(\log \lambda + h \log B))$ bits long.

Computing $\theta(v,r,l)$ is now straightforward. To begin with, from the value of $r$ we find the group $\beta_i$ that contains $p_v^r$, and the position $d$ of that point in $\beta_i$, which leads to the two bit vectors $X_d$ and $Y_d$. With $X_d$ at hand, we can retrieve $k$ as well as the address stored in $\gamma_k$. Using $Y_d$ as a relative address, we automatically gain access to the value of $s$ where $(w_j, s)$ is the companion of $(v,r)$. It is also easy to compute the address of the node $w_j$ from $\gamma_k$, since all secondary structures associated with an internal node of $T$ are laid out consecutively on disk. So we have devised a technique for computing repeated transitions in $O(1)$ I/Os. But at what cost in storage? Since all $\gamma_i$ and $\Gamma_i$'s are accessed individually, we can store them continuously for each node of $T$. We will not waste more than $O(n)$ blocks by doing so since there are $O(n)$ internal nodes in the tree $T$. So for convenience, we will compute the storage overhead for the $\gamma_i$'s and $\Gamma_i$'s in terms of bits, the number of blocks is just the number of bits divided by $B \log N$. Since each $\gamma_i$ is $\log N$ bits, each $\Gamma_i$ is $O(\lambda(\log \lambda + h \log B))$ bits and there is one $\gamma_i$ and one $\Gamma_i$ for each group, the total number of bits on each level of $T$ is $O((\log N + \lambda(\log \lambda + h \log B)) \cdot N/\lambda) = O((\log \lambda + h \log B)N)$. To summarize, we have a scheme for computing $h$ consecutive transitions in $O(1)$ I/Os from a given level, using $O((\log \lambda + h \log B)N)$ extra bits.

We are now ready to attack the main question: how to identify $p_v^r$? Let $m$ be the level of $v (m < 2\lambda - 1)$; the idea is to express $m$ in some appropriate number system. We will build several data structures on the same model. Let $\Theta(x, y)$ be a structure that allows us to compute transitions from level $x$ to level $y$ in $O(1)$ I/Os, using $O((\log \lambda + (y-x) \log B)N)$ extra bits. Let $D(x, y)$ denote a data structure for jumping from any level $l_1$ to any level $l_2$, with $x \le l_1 < l_2 \le y$. Identification can be done by using $D(0, 2\lambda)$. We define $D(x, y)$ in a recursive manner. Let $\alpha(y)$ be an integer function of $y$ such that $0 \le \alpha(y) \le y/2$. The data structure $D(0, y)$ is made of (the definition for the case $x \ne 0$ is similar)

$$\Theta(0, y) \cup \{D(0, \alpha(y) - 1), \ldots, D(j\alpha(y), (j+1)\alpha(y) - 1), \ldots, D(\lfloor y/\alpha(y) \rfloor \alpha(y), y)\},$$

while $D(x, y)$ is null when $y - x \le \nu$ for some threshold $\nu$. The storage $S(y)$ occupied by $D(0, y)$ follows the recurrence

$$S(y) \le \left\lceil \frac{y}{\alpha(y)} \right\rceil S(\alpha(y)) + O((\log \lambda + y \log B)N),$$

and $S(y) = 0$, for $y \le \nu$. Let $\alpha^{(0)}(y) = y$ and for any $i > 0$, $\alpha^{(i)}(y) = \alpha(\alpha^{(i-1)}(y))$. Define $\alpha^*(y) = \max\{i | \alpha^{(i)}(y) \ge \nu\} + 1$. We derive the relation

$$S(2\lambda) = O\left(\left(\alpha^*(\lambda) \log N + 1 + \frac{\lambda \log \lambda}{\nu}\right) N\right),$$

so the total number of extra disk blocks is $O((\alpha^*(\lambda) + \log_B \lambda/\nu)n)$.

To identify $p_v^r$, we begin by checking to see if $m$, the level of $v$, is equal to 0. If this is the case, we can then use $\Theta(0, 2\lambda)$ and conclude in $O(1)$ I/Os. If $m > 0$, we compute the starting interval of the form $[j\alpha(2\lambda), (j+1)\alpha(2\lambda) - 1]$ (or $[\lfloor 2\lambda/\alpha(2\lambda) \rfloor \alpha(2\lambda), 2\lambda]$) that contains $m$. We leap from $v$ to the companion of $p_v^r$ at level $(j+1)\alpha(2\lambda) - 1$ recursively, using $D(j\alpha(2\lambda), (j+1)\alpha(2\lambda) - 1)$. When the recursion passes the cutoff point, we complete the computation by taking at most $\nu$ transitions one level at a time. When this is done, we jump from level $(j+1)\alpha(2\lambda) - 1$ to level $2\lambda$ by taking at most $2 \cdot \lceil 2\lambda/\alpha(2\lambda) \rceil$ steps: for $k = j+1, j+2, \ldots$, we take a regular transition from level $k\alpha(2\lambda) - 1$ to $k\alpha(2\lambda)$, followed by a jump from level $k\alpha(2\lambda)$ to $(k+1)\alpha(2\lambda) - 1$, using $D(k\alpha(2\lambda), (k+1)\alpha(2\lambda) - 1)$. Thus, the total number of I/Os spent in identifying any point is

$$O\left(\nu + \sum_{\nu \le \alpha^{(i)} \le \lambda} \frac{\alpha^{(i)}(\lambda)}{\alpha^{(i+1)}(\lambda)}\right).$$

Now, setting $\alpha(y) = \lfloor y/B \rfloor$ and $\nu = 1$, we obtain $O(n \log_B \log_B N)$ space and $O(\log_B \log_B N)$ query bounds. Setting $\alpha(y) = \lfloor y/\lambda^\epsilon \rfloor$ and $\nu = \log_B \lambda$ will give us $O(n)$ space and $O(\log_B^\epsilon N)$ query bounds. We can even reduce the query bound to $O(1)$ by modifying the definition of $D(0, y)$ such that besides

$$\{D(0, \alpha(y) - 1), \ldots, D(j\alpha(y), (j+1)\alpha(y) - 1), \ldots, D(\lfloor y/\alpha(y) \rfloor \alpha(y), y)\},$$

it also includes

$$\{\Theta(0, y), \Theta(\alpha(y), y), \ldots, \Theta(j\alpha(y), y), \ldots, \Theta(\lfloor y/\alpha(y) \rfloor \alpha(y), y)\}.$$

We set $\alpha(y) = \lfloor y/\lambda^{\epsilon/2} \rfloor$ and $\nu = 1$. The only difference in the identification process now is that at a given recursion level, at most two jumps are needed and the query cost becomes proportional to the number of recursive calls, which is constant. The space cost now follows the relation

$$S(y) \leq \left\lceil \frac{y}{\alpha(y)} \right\rceil S(\alpha(y)) + O\left(\frac{y}{\alpha(y)}(y \log B + \log \lambda)N\right),$$

which yields $S(2\lambda) = O(N \log_B N(\log_B^\epsilon N \log B + \log\log_B N)) = O(N \log_B^{1+\epsilon} N \log B)$, and the number of disk blocks used is then $O(n \log_B^\epsilon N)$. However, bulk loading these structures requires $O(N \log_B N)$ I/Os and we omit the details.

**Theorem 5** *A set of $N$ points in the plane can be stored in a structure so that an orthogonal range max query can be answered in (1) $O(\log_B^{1+\epsilon} N)$ I/Os using $O(n)$ disk blocks; (2) $O(\log_B N \log_B \log_B N)$ I/Os using $O(n \log_B \log_B N)$ disk blocks; or (3) $O(\log_B N)$ I/Os using $O(n \log_B^\epsilon N)$ disk blocks. All these structures can be constructed in $O(N \log_B N)$ I/Os.*

# B Extending to Higher Dimensions

Both our static and dynamic structures can be extended to higher dimensions by constructing multi-level tree structures as follows. We take the structure of Theorem 1 as an example. Let $P$ be a set of $N$ points in $\mathbb{R}^d$, and set $b = B^{1/(2d-2)}$. Let us call our $d$-dimensional structure $T^d$. It is a a B-tree with fanout $b$, built on the $x_d$-coordinates of $P$. Each internal node $v$ of $T^d$ is associated naturally with a subset $P_v$ of $P$ and stores a secondary structure $T^{d-1}$, which is a $(d-1)$-dimensional structure built on the projection of $P_v$ onto the hyperplane $x_d = 0$. The recursion stops when we have built our 2D structure $T^2$ (with fanout b) in the $x_1x_2$-plane.

With each internal node $v$ of $T^2$, we associate a $(d-1)$ tuple $(w^d, w^{d-1}, \ldots, w^2)$ where $w^2 = v$ and $w^i$ is a node of $T^i$ to which $T^2$ is attached. For each point $p \in P_v$, and for $2 \leq i \leq d$, let $w^i_{a_i}$ be the child of $w^i$ (in $T^i$) so that $p \in P_{w^i_{a_i}} \subset P_{w^i}$. We call $(a_d, \ldots, a_2)$ the child-index sequence of $p$. All points that have the same child-index sequence form a *hyper-slab*, and adjacent hyper-slabs whose child-index sequences fall in $[a_d, b_d] \times [a_{d-1}, b_{d-1}] \times \cdots \times [a_2, b_2]$ are called a *hyper-multislab*. Since all child indexes are less than $b$, there are $b^{d-1} = \sqrt{B}$ hyper-slabs and $O(B)$ hyper-multislabs in each $T^2$. By using hyper-slabs and hyper-multislabs instead of slabs and multislabs, we can build the same structures as we described earlier.

Let us now look at how a query $Q = [\alpha_1, \beta_1] \times \cdots \times [\alpha_d, \beta_d]$ in $\mathbb{R}^d$ is answered. The query procedure in $T^d$ follows two paths to the leaves corresponding to $\alpha_d$ and $\beta_d$. For each node $v$ on these paths, the procedure recursively visits the $(d-1)$-dimensional structure attached at $v$. When we reach a 2D structure $T^2$, we carry out the same query procedure as before. Only this time, at each internal node of $T^2$, we report the maximum of the points whose $x_1$ falls between $[\alpha_1, \beta_1]$ and $(x_2, \ldots, x_d)$ falls in a hyper-multislab. This can be easily handled by our 2D structure. Thus, both our static and dynamic structures can be extended to $d$-dimension with an $O(\log_B^{d-2} n)$ factor increase in the space, query and update bounds.

# C Time-Space Tradeoff

In the following, we briefly describe a technique to trade space and update cost for better query performance.

Let $h = 2\log_B N$ be the height of our base tree $T$, and define the level of a node to be the number of its ancestors. The idea is to divide the $h$ levels into groups of size $\alpha(h)$, where $\alpha(h)$ is an integer function of $h$ and $\alpha(h) \leq h$. The "inter-group" structure is an interval tree while the "intra-group" structures are segment trees. We call the last level of each group a *fat* level. As before, intervals and their partial sums are stored in the secondary structures $\Sigma_v$ of each internal node $v$ of $T$. The difference is that any non-fat level node $v$ only have the multislab structure while an interval $\gamma$ is stored recursively at node $v_i$ and $v_j$ where the two endpoints of $\gamma$ are inside $X_{v_i}$ and $X_{v_j}$ respectively, not including the boundaries of $X_{v_i}$ and $X_{v_j}$. The recursion stops at a fat level node where these intervals are stores in the left (right) slab lists. Note that an interval is stored at less than $2\alpha(h)$ nodes, and the whole structure takes $O(n \cdot \alpha(h))$ disk blocks.

The query process remains the same, only that we do not need to look at the slab lists on all the non-fat levels. Since we visit $O(h/\alpha(h))$ fat nodes, the query cost is $O(\log_B^2 N/\alpha(h))$ I/Os. To insert an interval, we may need to insert into $O(\alpha(h))$ multislab lists and at most two slab lists, incurring $O(\log_B N \cdot \alpha(h))$ I/Os. Then we insert the new interval's two endpoints into the base tree. Since in the modified structure, we maintain the property that all the intervals stored in $\Sigma_v$ have at least one endpoint in $X_v$, the amortized cost of a split remains $O(1/\sqrt{B})$. So, the amortized cost of an insertion is $O(\log_B N \cdot \alpha(h))$. As before, we use global rebuilding to perform a deletion.

Thus, we can choose an appropriate $\alpha(h)$ to achieve some kind of space-query time tradeoff. In particular, choosing $\alpha(h) = h^\epsilon$ gives us the following result.

**Theorem 6** *A set of $N$ intervals can be stored in a structure that uses $O(n \log_B^\epsilon N)$ disk blocks such that a stabbing semigroup query can be answered in $O(\log_B^{2-\epsilon} N)$ I/Os, and updates can be performed in $O(\log_B^{1+\epsilon} N)$ I/Os amortized, for any $\epsilon > 0$.*