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

Pankaj K. Agarwal[⋆], Lars Arge[⋆⋆], Jun Yang, and Ke Yi

Department of Computer Science
Duke University, Durham, NC 27708, USA
{pankaj,large,junyang,yike}@cs.duke.edu

Abstract. We develop several linear or near-linear space and I/O-efficient dynamic data structures for orthogonal range-max queries and stabbing-max queries. 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 we wish 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 their many important applications. 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 orthogonal 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 wish 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(a). In the dual *orthogonal stabbing-max* problem, we are given a set S of N hyper-rectangles in $\mathbb{R}^d$ where each rectangle γ is assigned a weight $w(\gamma)$, and want to build a data structure such that for a query point q in $\mathbb{R}^d$, we can

[⋆] Supported in part by the National Science Foundation through grants CCR-00-86013, EIA–9972879, EIA-98-70724, EIA-01-31905, and CCR-02-04118, and by a grant from the U.S.–Israel Binational Science Foundation.

[⋆⋆] Supported in part by the National Science Foundation through ESS grant EIA–9870734, RI grant EIA–9972879, CAREER grant CCR–9984099, ITR grant EIA–0112849, and U.S.–Germany Cooperative Research Program grant INT–0129182.

Fig. 1. (a) Two-dimensional range queries. (b) Two-dimensional stabbing queries.

compute $\max\{w(\gamma) \mid q \in \gamma\}$ efficiently. The two-dimensional case is illustrated in Figure 1(b). We also consider the dynamic version of the two problems, in which points or hyper-rectangles can be inserted or deleted dynamically. In the following we drop "orthogonal" and often even "max" when referring to the two problems.

We 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 $M \geq B^2$ and that any integer less than N, as well as any point or weight, can be stored in a single word. We measure the efficiency of a data structure 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 near linear space, that is, use close to $n = \lceil N/B \rceil$ disk blocks.

Related work. Range searching 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 by Chazelle [10]. It answers a query in $O(\log^{1+\epsilon} n)$ time in the static case. In the dynamic case, the structure supports queries and updates in $O(\log^3 n \log\log n)$ time. The best known structure for the one-dimensional stabbing-max problem is by Kaplan et al [16]. It uses linear space and supports queries and insertions in $O(\log n)$ time and deletions in $O(\log n \log\log n)$ time. They also discuss how their structure can be extended to higher dimensions. Refer to [10,16] and the survey by Agarwal and Erickson [3] for additional results.

In the external setting, one-dimensional range-max queries can be answered in $O(\log_B n)$ I/Os using a standard B-tree [11,8]. The structure can easily be updated using $O(\log_B n)$ I/Os. For two or higher dimensions, however, no efficient linear-size structure is known; In the two-dimensional case, the kdB-tree [18], the cross-tree [14], and the O-tree [15], designed for general range searching, can be modified to answer range-max queries in $O(\sqrt{n})$ I/Os. All of them use linear space. The cross-tree [14] and the O-tree [15] can also be updated in $O(\log_B n)$ I/Os. The CRB-tree [2] designed for range-counting can be modified to support range-max queries in $O(\log_B^2 n)$ I/Os using $O(n \log_B n)$ space.

For the one-dimensional stabbing-max problem, the SB-tree [20] can be used to answer queries in $O(\log_B n)$ I/Os using linear space. Intervals can be inserted into the structure in $O(\log_B n)$ I/Os. However, the SB-tree does not support deletions. No worst-case efficient structures are known for higher-dimensional stabbing max queries. Refer to recent surveys [5,13] for additional results.

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 so 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 comparison with previous results.

Table 1. Two-dimensional range max query 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	$\sqrt{n}$	$\log_B n$	$\log_B n$	[14,15]
	$n \log_B \log_B n$	$\log_B^3 n$	$\log_B^2 n \cdot \log_{M/B} \log_B n$	$\log_B^2 n$	New

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, the previously known structure only supported insertions [20].

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 an $O(\log_B^5 n)$ 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 comparison with previous results.

Table 2. Two-dimensional stabbing max query results.

Problem	Space	Query	Insertion	Deletion	Source
1D stabbing max queries (dynamic)	n	$\log_B n$	$\log_B n$		[20]
	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

Finally, using standard techniques [2,9,12], both our range and stabbing structures can be extended to higher dimensions at the cost of increasing each of the space, query, and update bounds by an $O(\log_B n)$ factor per dimension. Our structures can also be extended and improved in several other ways. For example, our one-dimensional stabbing-max structure can be modified to support general semigroup stabbing queries.

2 Two-Dimensional Range-Max Queries

In this section we describe our structure for the two-dimensional range-max problem. The structure is an external version of a structure by Chazelle [10].

The overall structure. Our structure consists of two parts. The first is simply a B-tree Φ on the y-coordinates of the N points in 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 vertical *slab* σ_v containing P_v. If $v_1, v_2, \ldots, v_k$, for $k = \Theta(\sqrt{B})$, are the children of v, then $\sigma_{v_1}, \ldots, \sigma_{v_k}$ partition σ_v into k slabs. For $1 \le i \le j \le k$, we refer to the slab $\sigma_v[i:j] = \bigcup_{l=i}^{j} \sigma_{v_l}$ as a *multi-slab*; there are $O(B)$ multi-slabs at each node of T. Each leaf z of T stores $\Theta(B)$ points in P_z and their weights using $O(1)$ disk blocks. Each internal node v stores two secondary structures $\mathcal{C}_v$ and $\mathcal{M}_v$ requiring $O(n_v/\log_B n)$ blocks each, so that the overall structure uses a total of $O(n)$ blocks. We first describe the functionality of these structures. After describing how to answer a query, we 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. 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 ρ. Otherwise, we use $\mathcal{C}_v$ to find the rank of p in the set P_{v_j} associated with the relevant child v_j, and continue the search recursively in v_j. We call this step the *identification process*.

The other secondary structure $\mathcal{M}_v$ enables us to compute the maximum weight among the points in a given multi-slab and rank 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 $\{w(p) \mid p \in P_v \cap \sigma_v[i:j]$ and $rk_v(p) \in [\rho_1, \rho_2]\}$.

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 σ_{z_1} (resp. σ_{z_2}) contains (x_1, y_1) (resp. (x_2, y_2)). Let ξ 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 $\max\{w(p) \mid p \in P_\xi \cap Q\}$.

To answer the query 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 ξ 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 Σ_v be the widest multi-slab 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 2(a)), and that for any other node v on the path from ξ 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: 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 ρ_v^-, ρ_v^+ in $O(\log_B n)$ I/Os using the B-tree Φ. Otherwise, since we know $\rho_{p(v)}^-, \rho_{p(v)}^+$ at the parent of v, we can compute ρ_v^-, ρ_v^+ in $O(1)$ I/Os using the secondary structure $\mathcal{C}_{p(v)}$ stored at the parent $p(v)$ of v. Once we know ρ_v^-, ρ_v^+, we find the maximal weight point in $P_v \cap \Sigma_v \cap Q$ in $O(\log_B n)$ I/Os by querying $\mathcal{M}_v$ with the multi-slab Σ_v and the rank interval $[\rho_v^-, \rho_v^+]$. 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. Since $\mathcal{C}_v$ is the same as a structure used in the CRB-tree [2], we 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 multi-slab $\sigma_v[i : j]$ and a rank range $[\rho_1, \rho_2]$, 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 (except possibly the last one) of size μ. 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

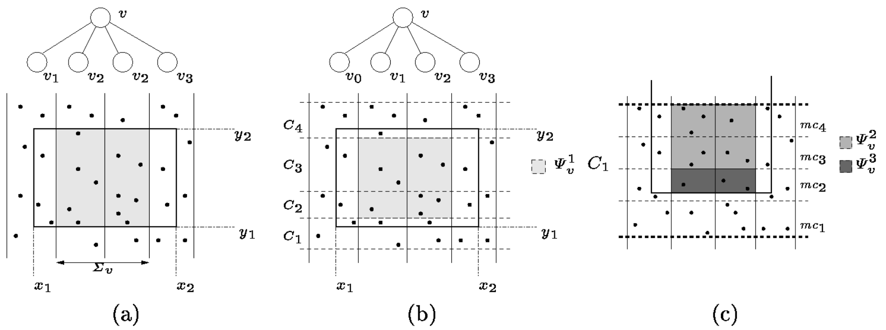

Fig. 2. (a) Answering a query. (b) Finding the max at the chunk level (using Ψ_v^1). (c) Finding the max at the minichunk level (using Ψ_v^2) and within a minichunk (using Ψ_v^3).

size B; C_i is partitioned into $mc_1, \dots, mc_{\nu_i}$, 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 ρ 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 Ψ_v^1, Ψ_v^2, and Ψ_v^3; Ψ_v^1 answers max queries at the "chunk level", Ψ_v^2 answers max queries at the "minichunk level", and Ψ_v^3 answers max queries within a minichunk. More precisely, let $\sigma_v[i:j]$ be a multi-slab and $[\rho_1, \rho_2]$ be a rank range, if the chunks that are spanned by $[\rho_1, \rho_2]$ are $C_a, \dots, C_b$, then we use Ψ_v^1 to report the maximum weight of the points in $\bigcup_{l=a}^{b} C_l \cap \sigma_v[i:j]$ (Figure 2(b)). We use Ψ_v^2, Ψ_v^3 to report the the maximum weight of a point in $C_{a-1} \cap \sigma_v[i:j]$, as follows. If $mc_\alpha, \cdots, mc_\beta$ are the minichunks of C_{a-1} that are spanned by $[\rho_1, \rho_2]$, then we use Ψ_v^2 to report the maximum weight of the points in $\bigcup_{l=\alpha}^{\beta} mc_l \cap \sigma_v[i:j]$. Then we use Ψ_v^3 to report the maximum weight of the points that lie in the minichunks that cross ρ_1 (Figure 2(c)). The maximum weight of a point in in $C_{b+1} \cap \sigma_v[i:j]$ can be found similarly. Below we describe Ψ_v^1, Ψ_v^2 and Ψ_v^3 in detail and show how they can be used to answer the relevant queries in $O(\log_B n)$ I/Os.

Structure $\boldsymbol{\Psi_v^3}$. Ψ_v^3 consists of a small structure $\Psi_v^3[l]$ for each minichunk mc_l, $1 \le l \le N_v/B = n_v$. Since we can only use $O(n_v/\log_B n)$ space, we store $\log_B n$ small structures together in $O(1)$ blocks. For each point p in mc_l we store a pair (ξ_p, ω_p), where ξ_p is the index of the slab containing p, and ω_p is the rank of the weight of p among the points in mc_l (i.e., $\omega_p - 1$ points in mc_l have smaller weights than that of p). Note that $0 \le \xi_p, \omega_p \le B$, so we need $O(\log B)$ bits to store this pair. The set $\{(\xi_p, \omega_p) \mid p \in mc_l\}$ is stored in $\Psi_v^3[l]$, sorted in increasing order of $rk_v(p)$'s (their ranks in P_v). $\Psi_b^3[l]$ needs a total of $O(B \log B)$ bits. Therefore $\log_B n$ small structures use $O(B \log B \log_B n) = O(B \log n)$ bits and fit in $O(1)$ disk blocks.

A query on Ψ_v^3 is of the following form: Given a multi-slab $\sigma_v[i:j]$, an interval $[\rho_1, \rho_2]$, and an integer $l \le n_v$, we wish to return the the maximum weight of a point in the set $\{p \in mc_l \mid p \in \sigma_v[i:j], rk_v(p) \in [\rho_1, \rho_2]\}$. We first load the whole $\Psi_v^3[l]$ structure into memory using $O(1)$ I/Os. Since we know the $rk_v(a)$ of the first point $a \in mc_l$, we can compute in $O(1)$ time the contiguous subsequence of pairs (ξ_p, ω_p) in $\Psi_v^3[l]$ such that $rk_v(p) \in [\rho_1, \rho_2]$. Among these pairs we select the point q for which $i \le \xi_q \le j$ (i.e., q lies in the multi-slab $\sigma_v[i:j]$) and ω_q has the largest value (i.e., q has the maximum weight among these points). Since we know $rk_v[q]$, we use the identification process (the $\mathcal{C}_v$ structures) to determine, in $O(\log_B n)$ I/Os, the actual weight of q.

Structure $\boldsymbol{\Psi_v^2}$. Similar to Ψ_v^3, Ψ_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 $\nu_k \le \log_B n$ minichunks $mc_1, \dots, mc_{\nu_k}$. For each multi-slab $\sigma_v[i:j]$, we do the following. For each $l \le \nu_k$, we choose the point of the maximum weight in $\sigma[i:j] \cap mc_l$. Let Q_{ij}^k denote the resulting set of points. We construct a *Cartesian tree* [19] on Q_{ij}^k with their weights as the key. 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_α and w_β. Conceptually, $\Psi_v^2[k]$ consists of such a Cartesian tree for each of the $O(B)$ multi-slabs. However, we do not actually store the weights in a Cartesian tree, but only an encoding of its structure. Thus we can not use it to find the actual maximal weight in a range of minichunks, but only the index of the minichunk containing the maximal weight. It is well known that the structure of a binary tree of size ν_k can be encoded using $O(\nu_k)$ bits. Thus, we use $O(\log_B n)$ bits to encode the Cartesian tree of each of the $O(B)$ multi-slabs, for a total of $O(B \log_B n)$ bits, which again fit in $O(1)$ blocks.

Consider a multi-slab $\sigma_v[i:j]$. To find the maximal weight of the points in the minichunks of a chunk C_k spanned by a rank range $[\rho_1, \rho_2]$, we load the relevant Cartesian tree using $O(1)$ I/Os, and 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 $\boldsymbol{\Psi_v^1}$. Ψ_v^1 is a B-tree with fanout $\sqrt{B}$ conceptually built on the $s = n_v / \log_B n$ chunks $C_1, \ldots, C_s$. Each leaf of Ψ_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 fits in $O(1)$ blocks. Similarly, an internal node of Ψ_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 fits in $O(1)$ blocks, and Ψ_v^1 uses $O(n_v/(\log_B n\sqrt{B})) = O(n_v/(\log_B n)$ blocks in total.

Consider a multi-slab $\sigma_v[i:j]$. 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 Ψ_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 multi-slab $\sigma_v[i:j]$ and one of the chunks $C_a, \cdots, C_b$, and select the maximal weight point. This takes $O(1)$ I/Os. Finally, we select the maximum of the $O(\log_B n)$ weights.

This completes the description of our static two-dimensional range max structure. In the full version of the paper we describe how it can be constructed in $O(n \log_B n)$ I/Os in a bottom-up, level-by-level manner.

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. Next we sketch how to make our data structure dynamic. Details will appear in the full paper.

To delete a point p from S we delete it from the relevant $O(\log_B n)$ $\mathcal{M}_v$ structures as well as from the base tree. The latter is done in $O(\log_B n)$ I/Os using global rebuilding [17]. To delete p from a $\mathcal{M}_v$ structure we need to delete it from Ψ_v^1, Ψ_v^2, and Ψ_v^3. Since we cannot update a Cartesian tree efficiently, which is the building block of Ψ_v^2, we modify the structure so that we no longer partition each chunk C_k of P_v into minichunks (that is, we remove Ψ_v^2). Instead we construct $\Psi_v^3[k]$ directly on the points in C_k. This allows us to delete p from $\mathcal{M}_v$ in $O(\log_B n)$ I/Os: We first delete p from Ψ_v^3 by marking its weight rank ω_p as ∞, and then update Ψ_v^1 if necessary. However, since $|C_k| \leq B \log_B N$, $\Psi_v^3[k]$ now uses $O(\log_B \log_B n)$ blocks and the overall size of the structure becomes $O(n \log_B \log_B n)$ blocks. The construction cost becomes $O(n \log_B n \log_{M/B} \log_B n)$ I/Os.

To handle insertions we use the external logarithmic method [6]; This way an insertion takes $O(\log_B^2 n \log_{M/B} \log_B n)$ I/Os amortized and the query cost is increased by a factor of $O(\log_B n)$.

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 such 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.*

In the full paper we describe various extensions and improvements. For example, by using Cartesian trees to implement Ψ_v^1 and a technique to speed up the identification process [10], we can improve the query bound of our linear-size static structure to $O(\log_B^{1+\epsilon} n)$ I/Os. However, we cannot construct this structure efficiently and therefore cannot make it dynamic.

3 Stabbing-Max Queries

In Section 3.1 we describe our stabbing-max structure for the one-dimensional case, and in Section 3.2 we sketch how to extend it to two dimensions.

3.1 One-Dimensional 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 [7], 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 σ_v is associated with each node v; σ_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 γ is assigned to v if $\gamma \subseteq \sigma_v$ but $\gamma \nsubseteq \sigma_{v_i}$ for any $1 \leq i \leq k$. A leaf z stores intervals whose both endpoints lie in σ_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, and a *multi-slab structure* $\mathcal{M}_v$. $\mathcal{L}_v[i]$ (resp. $\mathcal{R}_v[i]$) contains intervals from S_v whose left (resp. right) endpoints lie in σ_{v_i}. It supports stabbing queries for points in σ_{v_i} in $O(\log_B n)$ I/Os. The multi-slab structure $\mathcal{M}_v$ stores all intervals that span at least one slab. For any query point $q \in \sigma_{v_i}$, it can be used to find the maximum-weight interval that completely spans σ_{v_i} in $O(1)$ I/Os. We describe the slab and multi-slab structures below. Refer to Figure 3(a). Overall, an interval is stored in at most three secondary structures, and each secondary structure uses linear space, therefore the overall structure also uses linear space.

Answering a query. To report the maximum-weight interval containing a query point q, we search down the base tree T for the leaf z 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 and return 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 σ_{v_i} and that contains q. We then query the multi-slab structure $\mathcal{M}_v$ to compute the maximum-weight interval spanning σ_{v_i}. Refer to Figure 3(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 answer a query in a total of $O(\log_B^2 n)$ I/Os.

Left/right-slab structure. Let $R_v^i \subseteq S_v$ be the set of intervals whose right endpoints lie in σ_{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 3(c). As discussed in Section 2, such a query can easily be answered in $O(\log_B n)$ I/Os using a B-tree. $\mathcal{L}_v[i]$ is implemented in a similar way.

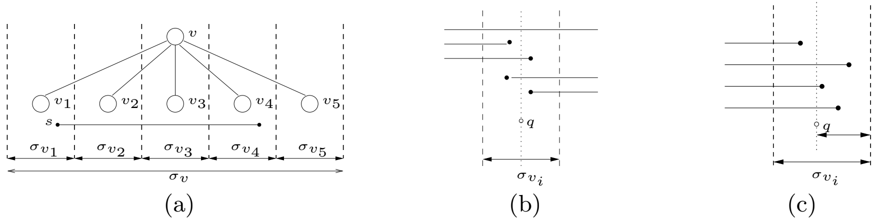

Fig. 3. (a) Node v in the base tree. The range σ_v associated with v is divided into 5 slabs. Interval s is stored in the left slab structure corresponding to σ_{v_1} and the right slab structure corresponding to σ_{v_4}, as well as in the multi-slab structure $\mathcal{M}_v$. (b) Querying a node with q. (c) Equivalence between a stabbing-max query q and a one-dimensional range max query $[q, \infty]$.

Multi-slab structure. A multi-slab 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 ordered by interval id's. For a node $u \in \mathcal{M}_v$, let γ_{ij} be the maximum-weight interval that spans σ_{v_i} and that is stored in the subtree rooted at the j-th child of u. For $1 \leq i, j \leq \sqrt{B}$, we store γ_{ij} at u. In particular, the root of $\mathcal{M}_v$ stores the maximum-weight interval spanning each of the $\sqrt{B}$ slabs, and a stabbing query in any slab σ_{v_i} can therefore be answered in $O(1)$ I/Os. Since each node can be stored in $O(1)$ blocks, $\mathcal{M}_v$ uses linear space. Note how $\mathcal{M}_v$ corresponds to "combining" $\sqrt{B}$ B-trees with fan-out $\sqrt{B}$ in a single B-tree.

To insert or delete an interval γ, we first search down $\mathcal{M}_v$ to find and update the relevant leaf z. After updating z, some of the intervals stored at 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 also perform B-tree rebalancing operations on the nodes on P. Both can easily be done in $O(\log_B n)$ I/Os in a traversal of P from z towards the root, as in [6].

Dynamization. To insert a new interval γ we first insert the endpoints of γ in T. By implementing T as a weight-balanced B-tree we can do so in $O(\log_B n)$ I/Os. Refer to [7] for details. Next, we use $O(\log_B n)$ I/Os to search down T for the node v where γ needs to be inserted in the secondary structures. Finally, we use another $O(\log_B n)$ I/Os to insert γ in a left and right slab structure, as well as in the multi-slab structure $\mathcal{M}_v$ if it spans at least one slab. To delete an interval γ we first delete it from the relevant secondary structures using $O(\log_B n)$ I/Os. Then we delete its endpoints from T using the global-rebuilding technique [17]. Since we can easily rebuild the structure in $O(n \log_B n)$ I/Os, this adds another $O(\log_B n)$ I/Os to the delete bound. Details will appear in the full paper.

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.*

In the full paper we describe various extensions and improvements. For example, we can easily modify our structure to handle *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 cannot be generalized, since it utilizes that in the semigroup $(\mathbb{R}, \max)$ the result of a semigroup operation is one of the operands.

By combining the ideas used in our structure with ideas from the external segment tree of Arge and Vitter [7], we can also 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.

3.2 Two-Dimensional Structure

In the two-dimensional stabbing-max problem we are given a set S of N weighted rectangles in $\mathbb{R}^2$, and want to be able to find the maximal-weight rectangle containing a query point q. We can extend our one-dimensional structure to this case using our one-dimensional stabbing-max and two-dimensional range-max structures. For space reasons we only give a rough sketch of the extension.

The structure consists of a base B-tree T with fanout $B^{1/3}$ on the x-coordinates of the corners of the rectangles in S. As in the 1D case, an interval σ_v is associated with each node v, and this interval is partitioned into $B^{1/3}$ vertical slabs by its children. A rectangle γ is stored at an internal node v of T if $\gamma \subseteq \sigma_v$ but $\gamma \nsubseteq \sigma_{v_i}$ for any child v_i of v. Each internal node v of T stores a multi-slab structure and one left- and right-slab structure for each slab. A multi-slab structure stores rectangles that span slabs and the left-slab (right-slab) structures of the i-th slab σ_{v_i} at v stores rectangles whose left (right) edges lie in σ_{v_i}.

The slab and multi-slab structures are basically one-dimensional stabbing-max structures on the y-projections of those rectangles. For the multi-slab structure we utilize the same "combining" technique as in the one-dimensional case to conceptually build a one-dimensional structure for each slab. The decreased fanout of $B^{1/3}$ allows us to use only linear space while being able to answer a query in $O(\log_B^2 n)$ I/Os. For the slab structures we utilize our two-dimensional range-max structure to be able to answer a query in $O(\log_B^3 n)$ I/Os. Details will appear in the full paper.

We answer a stabbing-max query by visiting $O(\log_B n)$ nodes on a path in T, and querying two slab structures and the multi-slab structure in each node. Overall, a query is answered in $O(\log_B^4 n)$ I/O. As previously, we can also make the structure dynamic using the external logarithmic method. Again details will appear in the full paper.

Theorem 4. *A set of N rectangles in $\mathbb{R}^2$ can be stored in a linear-size structure such that stabbing-max queries can be answered in $O(\log_B^4 n)$ I/Os.*

A set of N rectangles in $\mathbb{R}^2$ can be stored in a structure using $O(n \log_B \log_B n)$ disk blocks such that stabbing-max queries can be answered in $O(\log_B^5 n)$ I/Os, and 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.

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 Symp. 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. Intl. Conf. on Database Theory*, 2003.
3. P. K. Agarwal and J. Erickson. Geometric range searching and its relatives. In *Advances in Discrete and Computational Geometry* (B. Chazelle, J. Goodman, R. Pollack, eds.), 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. *Comm. ACM*, 31(9):1116–1127, 1988.
5. L. Arge. External memory data structures. In *Handbook of Massive Data Sets*, pages 313–358. Kluwer Academic Publishers, 2002.
6. L. Arge and J. Vahrenhold. I/O-efficient dynamic planar point location. In *Proc. ACM Symp. on Computational Geometry*, pages 191–200, 2000.
7. L. Arge and J. S. Vitter. Optimal dynamic interval management in external memory. In *Proc. IEEE Symp. on Foundations of Computer Science*, pages 560–569, 1996.
8. R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.
9. J. L. Bentley. Multidimensional divide and conquer. *Comm. ACM*, 23(6):214–229, 1980.
10. B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM J. Comput.*, 17(3):427–462, June 1988.
11. D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.
12. H. Edelsbrunner and H. A. Maurer. On the intersection of orthogonal objects. *Information Processing Letters*, 13:177–181, 1981.
13. V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.
14. R. Grossi and G. F. Italiano. Efficient cross-tree for external memory. In *External Memory Algorithms and Visualization*, pp. 87–106. AMS, DIMACS series in Discrete Mathematics and Theoretical Computer Science, 1999.
15. K. V. R. Kanth and A. K. Singh. Optimal dynamic range searching in non-replicating index structures. In *Proc. Intl. Conf. on Database Theory, LNCS 1540*, pages 257–276, 1999.
16. H. Kaplan, E. Molad, and R. E. Tarjan. Dynamic rectangular intersection with priorities. In *Proc. ACM Symp. on Theory of Computation*, pages 639-648, 2003.
17. M. H. Overmars. *The Design of Dynamic Data Structures.* Springer-Verlag, LNCS 156, 1983.
18. J. Robinson. The K-D-B tree: A search structure for large multidimensional dynamic indexes. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 10–18, 1981.
19. J. Vuillemin. A unifying look at data structures. *Comm. ACM*, 23:229–239, 1980.
20. J. Yang and J. Widom. Incremental computation and maintenance of temporal aggregates. In *Proc. IEEE Intl. Conf. on Data Engineering*, pages 51–60, 2001.