Computational Geometry 46 (2013) 358–370

Contents lists available at SciVerse ScienceDirect

Computational Geometry: Theory and Applications

www.elsevier.com/locate/comgeo

Efficient external memory structures for range-aggregate queries ☆

Pankaj K. Agarwal [a], Lars Arge [b], Sathish Govindarajan [c], Jun Yang [a], Ke Yi [d,*]

[a] Department of Computer Science, Duke University, Box 90129, Durham, NC 27708-0129, USA
[b] MADALGO, [1] Department of Computer Science, Aarhus University, Aarhus, Denmark
[c] CSA Department, Indian Institute of Science, Bangalore, India
[d] Department of Computer Science and Engineering, HKUST, Hong Kong

ARTICLE INFO

Article history:
Received 8 March 2011
Accepted 4 October 2012
Available online 8 October 2012
Communicated by M. de Berg

Keywords:
External memory algorithms
Range-aggregation
Data structures

ABSTRACT

We present external memory data structures for efficiently answering range-aggregate queries. The range-aggregate problem is defined as follows: Given a set of weighted points in $\mathbb{R}^d$, compute the aggregate of the weights of the points that lie inside a d-dimensional orthogonal query rectangle. The aggregates we consider in this paper include COUNT, SUM, and MAX. First, we develop a structure for answering two-dimensional range-COUNT queries that uses $O(N/B)$ disk blocks and answers a query in $O(\log_B N)$ I/Os, where N is the number of input points and B is the disk block size. The structure can be extended to obtain a near-linear-size structure for answering range-SUM queries using $O(\log_B N)$ I/Os, and a linear-size structure for answering range-MAX queries in $O(\log_B^2 N)$ I/Os. Our structures can be made dynamic and extended to higher dimensions.

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, SUM and MAX, 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 external memory (or I/O-efficient) structures for fundamental range-searching problems. In this paper, we develop an external memory data structure for answering orthogonal range-COUNT, SUM and MAX queries. Note that from these we automatically get some other aggregates like AVE and MIN.

Problem statement. In the *range-aggregate* problem, we are given a set P of N points in d dimensions where each point p is assigned a weight $w(p)$, and we wish to build a data structure so that for an axis parallel query hyper-rectangle Q in d dimensions, we can compute $\text{AGGR}\{w(p) \mid p \in Q\}$ efficiently, where AGGR is some aggregate function, for example COUNT,[2] SUM, or MAX, and we call the corresponding queries range-COUNT, range-SUM, and range-MAX queries, respectively. We assume that coordinates of the points, as well as their weights, are integers in the range between 0 and $N^{O(1)}$.

We work in the external memory model [2]. 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 $M \geqslant B^2$ and each machine word has $\Theta(\log_2 N)$ bits. Our structures will rely on these assumptions so that we can pack things compactly in order to reduce the space cost. We measure the efficiency of a data structure in terms of the amount of disk space it uses

☆ The results in this paper have appeared in preliminary forms in the papers "CRB-tree: An efficient indexing scheme for range-aggregate queries" in ICDT'03 and "I/O-efficient structures for orthogonal range-max and stabbing-max queries" in ESA'03.

* Corresponding author.
E-mail addresses: pankaj@cs.duke.edu (P.K. Agarwal), large@madalgo.au.dk (L. Arge), junyang@cs.duke.edu (J. Yang), yike@cse.ust.hk (K. Yi).

[1] Center for Massive Data Algorithmics, a center of the Danish National Research Foundation.

[2] Note that COUNT is a special case of SUM where $w(p) = 1$ for all p.

0925-7721/$ – see front matter
http://dx.doi.org/10.1016/j.comgeo.2012.10.003

Table 1
Asymptotic bounds of the two-dimensional CRB-tree in the static case.

Problem	Space	Query	Construction
range-COUNT	n	$\log_B n$	$n \log_B n$
range-SUM	$n \log_B((W \log_2 W)/N)$	$\log_B n$	$n \log_B((W \log_2 W)/N) \log_B n$
range-MAX	n	$\log_B^2 n$	$n \log_B n$
	n	$\log_B^{1+\varepsilon} n$	$N \log_2 N$

(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. There has been a lot of work in both the computational geometry and the spatial database community on designing data structures for answering range-aggregate queries. In internal memory, a well known data structure is the *range tree* [5], which uses $O(N \log_2 N)$ space and can answer a range-aggregate query in $O(\log_2 N)$ time. Chazelle [7] exploited that fact a RAM word can store $\log_2 N$ bits, and reduced the space of the range tree to $O(N)$ words by packing bits into words. His basic structure, called the *compressed range tree*, answers a range-COUNT query in $O(\log_2 N)$ time. A key building block in the compressed range-tree is a data structure that answers a *rank* query on an array A of N bits in $O(1)$ time, where the query specifies an index i and asks for the number of 1's in $A[1..i]$. Chazelle's structure for this problem uses $O(N)$ bits, which is all he needed to reduce the size of the overall structure to $O(N)$ words. Jacobson further reduced the space cost to $N + o(N)$ bits [15,14], which generated a lot of interests in the design of *succinct data structures* [9,8,20]. The data structures presented in this paper are external versions of Chazelle's structure [7]. In doing so we design an external version of his $O(N)$-bit rank structure on an alphabet of size B, namely, each $A[i]$ stores a number between 1 and B, the query specifies an i and requires the number of j's in $A[1..i]$ be computed for all $1 \leqslant j \leqslant B$ simultaneously. It is an interesting open question whether we can make our structure both I/O-efficient and succinct. Recently, Bose et al. [6] improved the query time to $O(\log N / \log\log N)$ while still using linear space, but it is unclear how to externalize their structure; please also see the discussion at the end of the paper.

Chazelle's structure can be augmented to support range-MAX queries in $O(\log_2^{1+\varepsilon} N)$ time, for any constant $\varepsilon > 0$. In the dynamic case where points can be inserted or deleted, the query time and update time are both $O(\log_2^2 N)$ for the range-COUNT structure, and $O(\log_2^3 N \log_2 \log_2 N)$ for the range-MAX structure. For the range-COUNT problem, Nekrich [17] presented an alternative dynamic structure that has an $O((\log N / \log\log N)^2)$ query time with linear space, but the update time is $O(\log^{4+\varepsilon} N)$.

Please refer to [7] and the survey by Agarwal and Erickson [1] for additional results on range-aggregation in internal memory.

In the external setting, one-dimensional range-aggregate queries can be answered in $O(\log_B n)$ I/Os using a standard B-tree [10]. 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 [21], cross-tree [12], and O-tree [16], designed for general range searching, can be modified to answer range-aggregate queries in $O(\sqrt{n})$ I/Os. All of them use linear space. The cross-tree [12] and the O-tree [16] can also be updated in $O(\log_B n)$ I/Os. The R-tree [13], designed for storing both points and rectangles, is often used in practice, but no worst-case bound on range-aggregate queries is known for R-trees. There are several super-liner sized structures, for example the aP-tree of Tao et al. [23] and the MVSB-tree of Zhang et al. [27], both of which use $O(n \log_B n)$ disk blocks and answer a range-aggregate query in $O(\log_B n)$ I/Os. Refer to surveys [3,11,18] for additional results in external memory.

Our results. The structures developed in this paper can be seen as the external versions of the data structures of Chazelle [7].

In Section 2 we describe the basic structure of the *Compressed Range B-tree* (or *CRB-tree*), which can be used for answering two-dimensional range-COUNT queries. It uses $O(n)$ disk blocks, answers a query in $O(\log_B n)$ I/Os, and can be built in $O(n \log_B n)$ I/Os. In Sections 3 and 4, we show how to augment the basic CRB-tree to support range-SUM and range-MAX queries. Specifically, our range-SUM structure answers queries in $O(\log_B n)$ I/Os using $O(n \log_B((W \log_2 W)/N))$ disk blocks, where $W = \sum_{p \in S} w(p)$ is the total weight of the input points. This improves upon the result in [23] when the total weights of the points is $O(N)$. Note that W can be as large as $O(N^c)$ for constant $c > 1$, in which case our solution achieves the same bounds as the earlier result. Our range-MAX structure uses $O(n)$ space and answers range-MAX queries in $O(\log_B^2 n)$ I/Os, and these bounds do not depend on W. There is another version of the range-MAX structure that answers a query in $O(\log_B^{1+\varepsilon})$ I/Os, but it cannot be constructed I/O-efficiently. The results are summarized in Table 1. Note that the $M > B^2$ requirement is only needed for the construction and update bounds; the query algorithms only need the internal memory to hold $O(1)$ blocks.

Section 5 presents several extensions. We show how to apply the external logarithmic method [4] in our setting and make the structures dynamic. This brings an extra $O(\log_B n)$ factor to the query bound and maintains the space bound (or increases by an $O(\log_B \log_B n)$ factor in the range-MAX case). The detailed results are listed in Table 2. Finally, using standard techniques our results can be extended to d dimensions, by paying an extra $O(\log_B n)$ factor to the space, query and update bounds for each dimension above two.

Table 2
Asymptotic bounds of the two-dimensional CRB-tree in the dynamic case.

Problem	Space	Query	Update
range-COUNT	n	$\log_B^2 n$	$\log_B^2 n$
range-SUM	$n \log_B((W \log_2 W)/N)$	$\log_B^2 n$	$\log_B((W \log_2 W)/N) \log_B^2 n$
range-MAX	$n \log_B \log_B n$	$\log_B^3 n$	Insert: $\log_B^2 n \log_{M/B} \log_B n$
			Delete: $\log_B^2 n$

We have implemented the CRB-tree for COUNT queries in two dimensions, and in Section 6 we report the results of an experimental evaluation of its efficiency. Since we are mainly interested in linear-size structures, we compare the performance of the CRB-tree with that of the *kdB*-tree. We have evaluated the performance of these structures using synthetic and TIGER/Line data. Our experiments show that the query performance of the CRB-tree is significantly better than that of the *kdB*-tree. For a data set with around 100 million points, the CRB-tree query time is 8–10 times faster than the *kdB*-tree query time. Furthermore, the query performance of the CRB-tree does not depend on the distribution of the input, or the size and shape of the query rectangle, both of which can lead to performance fluctuations for the *kdB*-tree.

2. Range-COUNT queries

In this section, we describe the basic structure of the CRB-tree, which can be used to answer two-dimensional range-COUNT queries.

The overall structure. Let P denote the set of N points in the plane. Without loss of generality, we assume that the coordinates of the points of P are all distinct. A CRB-tree consists of two parts. The first is simply a B-tree Φ on the y-coordinates of the N points in P. 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 $\mathcal{T}$ on the x-coordinates of P. For each node v of $\mathcal{T}$, let $P_v = \{p_1, p_2, \ldots\}$ 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 $v_1, v_2, \ldots, v_B$ are the children of v, then $\sigma_{v_1}, \ldots, \sigma_{v_B}$ partition σ_v into B slabs (Fig. 1(i)). For $1 \leqslant i \leqslant j \leqslant B$, we refer to the slab $\sigma_v[i:j] = \bigcup_{l=i}^{j} \sigma_{v_l}$ as a *multi-slab*. Each leaf z of $\mathcal{T}$ stores $\Theta(B)$ points in P_z using one disk block. Each internal node v stores a secondary structure $\mathcal{C}_v$ requiring $O(n_v/\log_B n)$ blocks. Since the height of $\mathcal{T}$ is $O(\log_B n)$, we have that $\Gamma_{v \in \mathcal{T}} n_v = O(n \log_B n)$, therefore the overall structure will use a total of $O(n)$ blocks. For any y-value y_0, let $\rho_v(y_0)$ denote the number of points in P_v whose y-coordinates are not larger than y_0. We also write $\rho_v(p)$ as a shorthand for ρ_v (p's y-coordinate), and $\rho_v(p)$ is often called the *rank* of p in P_v. We will construct $\mathcal{C}_v$ such that, given $\rho_v(y_0)$ for any y_0, it can be used to determine $\rho_{v_i}(y_0)$ for all children v_i of v using a constant number of I/Os. Below we first describe how to use the $\mathcal{C}_v$'s to answer a range-COUNT query in $O(\log_B n)$ I/Os, then we describe their implementation.

*Answering a range-*COUNT *query.* Let $Q = [x_1, x_2] \times [y_1, y_2]$ be a query rectangle. For ease of presentation we assume that no points in P has y-coordinate y_1 or y_2. We wish to compute $|P \cap Q|$. Let z_1 (resp. z_2) be the leaf of $\mathcal{T}$ such that σ_{z_1} (resp. σ_{z_2}) contains (x_1, y_1) (resp. (x_2, y_2)). Let ξ be the lowest common ancestor of z_1 and z_2. Then $P \cap Q = P_\xi \cap Q$, and therefore it suffices to compute $|P_\xi \cap Q|$.

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 internal 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 Γ_v be the widest multi-slab at v whose x-span is contained in $[x_1, x_2]$. Note that $\Gamma_v = \sigma_v[l_v + 1 : r_v - 1]$ when $v = \xi$ (Fig. 1(i)), and that for any other node v on the path from ξ to z_1 (resp. z_2), $\Gamma_v = \sigma_v[l_v + 1 : B]$ (resp. $\Gamma_v = \sigma_v[1 : r_v - 1]$). At each such node v, we compute

$$\left|P_v \cap \Gamma_v \cap Q\right| = \sum_{\sigma_{v_i} \subset \Gamma_v} \left(\rho_{v_i}(y_2) - \rho_{v_i}(y_1)\right). \tag{1}$$

If v is the leaf z_1 or z_2, we simply scan all the points of P_v and compute $|P_v \cap Q|$. The answer to Q is then the sum of all these quantities. When v is the root of $\mathcal{T}$, $\rho_v(y_1)$ and $\rho_v(y_2)$ can be obtained from the B-tree Φ in $O(\log_B n)$ I/Os. Once we have $\rho_v(y_1)$ and $\rho_v(y_2)$, using $\mathcal{C}_v$ we can compute in $O(1)$ I/Os $\rho_{v_i}(y_1)$ and $\rho_{v_i}(y_2)$ for all the children v_i of v, which in turn can be used to obtain the partial count of (1). Thus, since we use $O(\log_B n)$ I/Os in the root of $\mathcal{T}$ and $O(1)$ I/Os in each node along the two paths of $\mathcal{T}$ from the root to z_1 and z_2, the overall cost to answer a query is $O(\log_B n)$ I/Os.

The secondary structure $\mathcal{C}_v$. The secondary structure $\mathcal{C}_v$ in a node v of $\mathcal{T}$ is used to compute $\rho_{v_i}(y_0)$ from $\rho_v(y_0)$ for all the children v_i of v. A naive way to implement $\mathcal{C}_v$ is to maintain an array that stores the rank $\rho_{v_i}(p_j)$, for each point $p_j \in P_v$ and each of v's children v_i, i.e., how many points among the first j points of P_v belong to the slab σ_{v_i}. Using these "prefix counts", we can determine in one I/O the ranks $\rho_{v_i}(y_0)$ given $\rho_v(y_0)$. However, storing these prefix counts requires $O(N_v)$ disk blocks, which is far more than the allowed bound of $O(n_v/(\log_B n))$. Therefore we adopt the following two-level

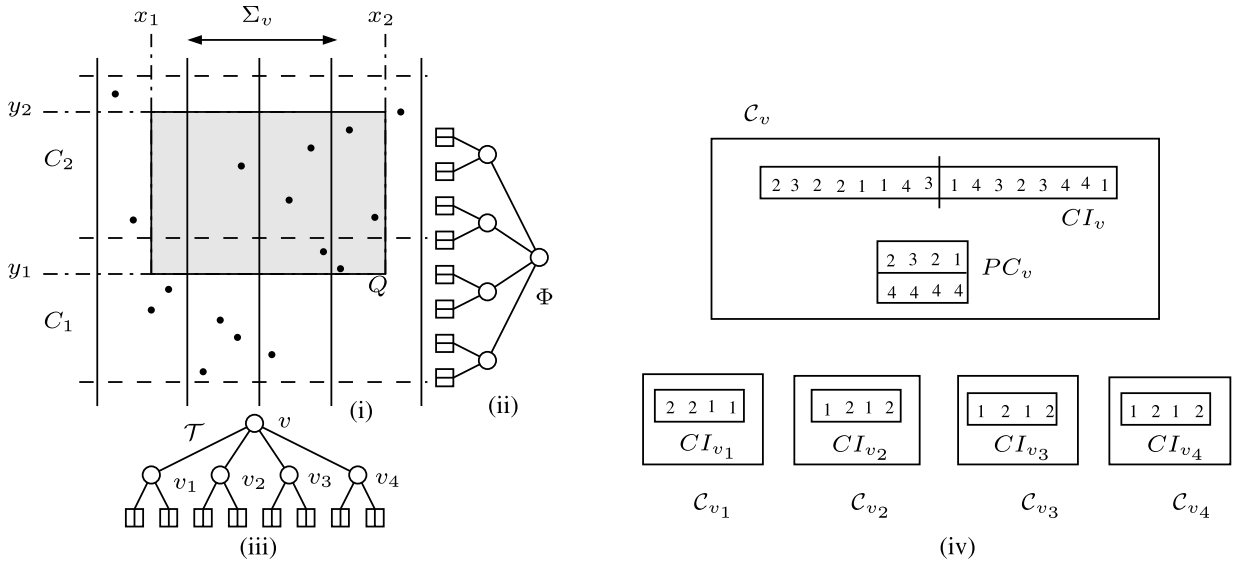

Fig. 1. CRB-tree for a set of $N = 16$ points. (i) The point set P is partitioned into four slabs and two chunks. (ii) B-tree Φ on the y-coordinates of P. (iii) B-tree $\mathcal{T}$ on the x-coordinates of P; $B = 4$ and each leaf stores 2 points of P. (iv) Secondary structure stored at each internal node of $\mathcal{T}$ with $\mu = 8$; since each child of the root stores 4 (< 8) points, no entries are stored in the PC array at those nodes. $PC_v[1, j]$ (resp. $PC_v[2, j]$) is k if k points among the first 8 (resp. 16) points of P are stored in the slab σ_{v_j}.

structure,[3] which is the external version of the one in [7]. Roughly speaking, we partition P_v into consecutive *chunks* and compute the prefix counts only at the "chunk level", i.e., for the lth chunk and for each child v_i of v, we store the number of points in the first l chunks that belong to σ_{v_i}. Let the predecessor of y_0 in P_v belong to the kth chunk of P_v. The desired rank $\rho_{v_i}(y_0)$ is then the sum of the "prefix count" till the $(k-1)$th chunk and the number of points within the kth chunk that belong to σ_{v_i} and whose y-coordinates are less than y_0. We preprocess each chunk separately so that we can compute the latter count using $O(1)$ I/Os.

More precisely, $\mathcal{C}_v$ consists of two arrays—a *child index* array CI_v and a *prefix count* array PC_v. We regard the child index array CI_v as a one-dimensional array with a $(\log_2 B)$-bit entry for each of the N_v points in P_v. $CI_v[i]$ simply stores the index b_i of the slab of v in which the ith point p_i of P_v stays. (For example, in Fig. 1(iv), $CI_v[3] = 1$ since the third point in P_v belongs to the slab σ_{v_1}.) Since $1 \leqslant b_i \leqslant B$, $\log_2 B$ bits are enough to represent b_i. CI_v requires $N_v \log_2 B$ bits and thus can be stored using $\frac{N_v}{B} \log_2 B/(\log_2 N) = n_v/\log_B N$ blocks. Next, let $\mu = B \log_B N$ and $r = N_v/\mu$. (Note that μ entries of CI_v fit in one block.) 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, $C_i = \{p \in P_v \mid \rho_v(p) \in [(i-1)\mu + 1, i\mu]\}$. (For example the 16 points in Fig. 1(i) are partitioned into 2 chunks of size 8.) We store the prefix sum at the chunk level in the prefix count array PC_v. We regard PC_v as a two-dimensional array with r rows and B columns, with each entry storing one word (that is, $\log_2 N$ bits). For $1 \leqslant i \leqslant r$ and $1 \leqslant j \leqslant B$, $PC_v[i, j]$ stores $\rho_{v_j}(p_{i\mu})$, i.e., the number of points in σ_{v_j} that are below (or the same as) the highest point in chunk C_i. (For example, in Fig. 1(iv), $\mu = 8$ and $PC_v[1, 1] = 2$ since among the first 8 points of P_v, only 2 points belong to σ_{v_1}.) PC_v requires $rB = N_v/\log_B N$ words, or $O(n_v/\log_B n)$ blocks. Hence $\mathcal{C}_v$ can be stored using $O(n_v/\log_B n)$ disk blocks, which means that the overall space of the CRB-tree is $O(n)$.

All that remains is to describe the procedure for computing $\rho_{v_1}(y_0), \ldots, \rho_{v_B}(y_0)$ given $\rho_v(y_0)$ in $O(1)$ I/Os. Suppose $\rho_v(y_0) = \mu a + c$ for $a \geqslant 0$ and $0 \leqslant c < \mu$. Then

$$\begin{aligned}\rho_{v_j}(y_0) &= \left|\left\{k \mid k \leqslant \rho_v(y_0) \text{ and } CI_v[k] = j\right\}\right| \\ &= PC_v[a, j] + \left|\left\{k \mid \mu a < k \leqslant \rho_v(y_0) \text{ and } CI_v[k] = j\right\}\right|, \quad \text{for } j = 1, \ldots, B.\end{aligned}$$

Thus the only two blocks we need to access are the two storing $PC_v[a, 1], \ldots, PC_v[a, B]$ and $CI_v[\mu a + 1], \ldots, CI_v[\rho_v(y_0)]$, respectively, and this completes the description of $\mathcal{C}_v$.

Constructing a CRB-tree. A CRB-tree can be constructed efficiently bottom-up, level by level. We construct the leaves of $\mathcal{T}$ using $O(n \log_{M/B} n)$ I/Os by sorting the points in P by their x-coordinates [2]. We then sort the points stored at each leaf v by their y-coordinates to get P_v. Below we describe how we construct all nodes and secondary structures at level i of $\mathcal{T}$, given that we have already constructed the nodes at level $i + 1$.

[3] This two-level directory idea first appeared in [7], and was later also used in designing succinct rank/select data structures [8,15].

We construct a level i node v and its secondary structure $\mathcal{C}_v$ as follows: We first compute P_v (sorted by the y-coordinates) by merging $P_{v_1}, \ldots, P_{v_B}$ together. Since $M > B^2$, the internal memory can hold a block from each P_{v_i} at the same time. This means that we can compute P_v using a single scan through the P_{v_i}'s, i.e., in $O(n_v)$ I/Os. Recall that $CI_v[i]$ contains the index of the slab containing p_i. These indices can be easily obtained during the same scan as we merge $P_{v_1}, \ldots, P_{v_B}$ together to get P_v. Since $PC_v[i, j]$ contains the number of points among the first $i\mu$ points in P_v that belong to σ_{v_j}, we can also construct PC_v in a single scan through CI_v as follows: We compute $PC_v[i, j]$ while scanning the points $p_{(i-1)\mu+1}, \ldots, p_{i\mu}$. We maintain $PC_v[i, 1], \ldots, PC_v[i, B]$ in internal memory as i goes from 1 to N_v/μ. If $i = 1$, we initialize $PC_v[i, j] = 0$, otherwise we initially set $PC_v[i, j] = PC_v[i-1, j]$. If we are currently scanning p_k and $CI_v[k] = j$ we increment $PC_v[i, j]$. After we have scanned $p_{i\mu}$, we store $PC_v[i, 1], \ldots, PC_v[i, B-1]$ in one block on disk.

Since the number of I/Os used to construct a level i node v and its secondary structure $\mathcal{C}_v$ is $O(n_v)$, the total number of I/Os used to build level i of $\mathcal{T}$ is $O(n)$. Thus we can construct the CRB-Tree using $O(n \log_B n)$ I/Os in total.

Theorem 1. *A set of N points in 2D can be stored in a linear-size structure using $O(n \log_B n)$ I/Os such that a range-*COUNT *query can be answered in $O(\log_B n)$ I/Os.*

Remark 1. Note that the fanout of the B-tree $\mathcal{T}$ does not have to be B, it can be B^c for any constant $0 < c < 1$ and Theorem 1 still holds. In Section 4 we will make use of this property and choose the fanout to be $\sqrt{B}$ for our range-MAX structure.

3. Range-SUM queries

In this section we discuss how to extend the CRB-tree to answer range-SUM queries in the plane. Let P be a set of N points in 2D, and let $w(p)$ be the weight of $p \in P$. If the weight of each point can be stored using $O(1)$ bits, then we can easily modify the CRB-tree by storing the weights in an additional array similar to CI and storing the prefix sum of the weights in another array similar to PC. So we focus on the case in which the weights of the points vary considerably. Set $W = \sum_{p \in P} w(p)$. We extend the CRB-tree by storing four arrays W_v, CW_v, L_v, and CL_v in addition to the secondary structure $\mathcal{C}_v$ at each internal node v of the base B-tree $\mathcal{T}$. We store $w(p_i)$ for all $p_i \in P_v$ in the array W_v, using $\log_2(w(p_i))$ bits each, as a continuous sequence of bits. W_v then plays the role of CI_v. All the W_v arrays on one level of the base tree $\mathcal{T}$ require at most $\sum_{p \in P} \log_2(w(p)) = \log_2(\prod_{p \in P} w(p)) \leqslant \log_2(W/N)^N = N \log_2(W/N)$ bits, so the space occupied by all the W_v arrays is $O(N \log_2(W/B) \log_B n/(B \log_2 N)) = O(n \log_B(W/N))$ disk blocks.

Since W_v is stored as a packed array, we use two additional arrays L_v and CL_v to determine the position in W_v that stores the leftmost bit of $w(p_i)$, for any given $i \leqslant N_v$. L_v is an array of length N_v, each entry composed of $\log_2 \log_2 W$ bits. $L_v[i]$ stores the value of $\log_2(w(p_i))$, which needs at most $\log_2 \log_2 W$ bits. The size of L_v is thus $O(n_v \log_2 \log_2 W / \log_2 N)$ disk blocks. CL_v stores the prefix sums of L_v at the "block level", i.e., it has N_v/β entries where $\beta = B \log_2 N / \log_2 \log_2 W$, and each entry is one word long. $L_v[k]$ stores the prefix sum $\sum_{i \leqslant k\beta} \log_2 w(p_i)$, which can always be represented with at most $\log_2(\sum_{p \in P} \log_2(w_p)) = O(\log_2 N)$ bits. Thus, using L_v and CL_v, the position of the leftmost bit of w_i in W_v, for any $i \leqslant N_v$, can be computed in $O(1)$ I/Os. The size of CL_v is $O((N_v \log_2 N)/(\beta B \log_2 N)) = O(\frac{n_v}{B} \log_2 \log_2 W / \log_2 N)$ blocks. So the overall size of all the L_v and CL_v arrays is $O(n \log_2 \log_2 W / \log_2 N \cdot \log_B n) = O(n \log_B \log_2 W)$ disk blocks.

Finally, CW_v stores prefix sum of the weights in W_v at the block level for each slab σ_{v_j}, similar to PC_v, such that for any $i \leqslant N_v$, one can compute the sum of the weights of the points in $\{p_1, \ldots, p_i\} \cap \Gamma_v$ in $O(1)$ I/Os. Since any prefix sum of the weights takes $O(\log_2 W) = O(\log_2 N)$ bits, the size of CW_v is $O(n \log_B(W/N)) \cdot B \log_2 N/(B \log_2 N) = O(n \log_B(W/N))$ blocks. Therefore, the total size of all the additional arrays that we add to the basic CRB-tree is $O(n(\log_B(W/N) + \log_B \log_2 W)) = O(n \log_B((W \log_2 W)/N))$ blocks.

The construction and query procedures are essentially the same as those of the basic CRB-tree described in Section 2, and we conclude with the following.

Theorem 2. *Let P be a set of weighted N points in 2D, and let $W = \sum_{p \in P} w(p)$. We can build a structure in $O(n \log_B((W \log_2 W)/N) \log_B n)$ I/Os that uses $O(n \log_B((W \log_2 W)/N))$ disk blocks such that a range-*SUM *query can be answered using $O(\log_B n)$ I/Os. If the weight of each point requires $O(1)$ bits, then the space and construction bounds are $O(n)$ and $O(n \log_B n)$, respectively.*

4. Range-MAX queries

Both our range-COUNT and range-SUM structures depend heavily on the ability to perform subtractions on the partial aggregates. However, we do not have this possibility any more when the aggregation is MAX. In this section, we show how to augment our basic CRB-tree of Section 2 to support range-MAX queries. We first decrease the fanout of the B-tree $\mathcal{T}$ from B to $\sqrt{B}$; note that this way we only have $\sqrt{B}$ slabs and $O(B)$ multi-slabs at each node v. Next, we attach a secondary structure $\mathcal{M}_v$ of size $O(n_v/\log_B n)$ to each internal node v of $\mathcal{T}$, in addition to $\mathcal{C}_v$. For any rank range $[\rho_1, \rho_2]$, the new secondary structure $\mathcal{M}_v$ can be used to compute $\max\{w(p) \mid p \in \sigma_v[i:j], \rho_1 \leqslant \rho_v(p) \leqslant \rho_2\}$ for any multi-slab $\sigma_v[i:j]$ of v in $O(\log_B n)$ I/Os. Let $Q = [x_1, x_2] \times [y_1, y_2]$ be a query rectangle. To answer a range-MAX query, we follow the same procedure as a range-COUNT query, and visit $O(\log_B n)$ nodes along two root-to-leaf paths of $\mathcal{T}$. Since we now have $\rho_v(y_1)$

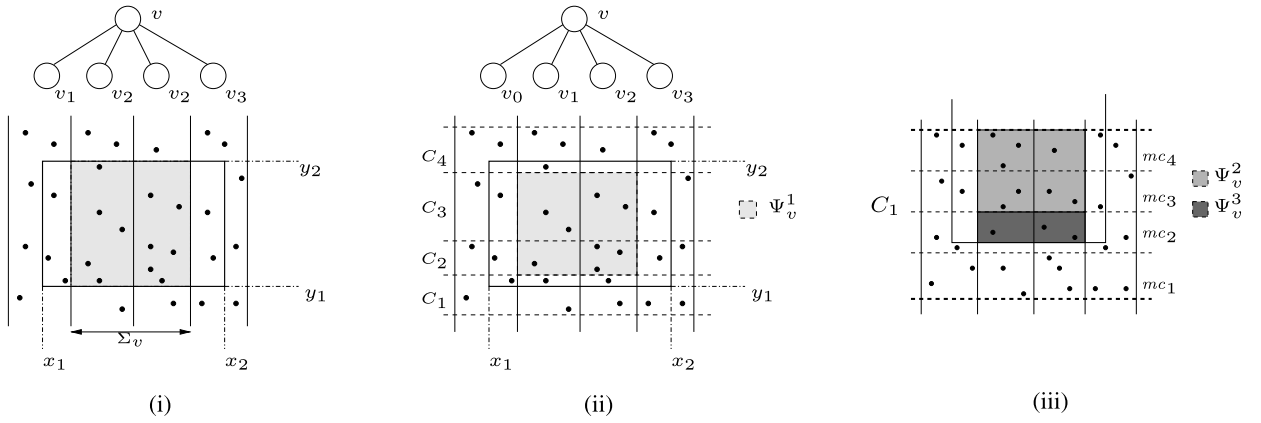

Fig. 2. (i) Answering a query. (ii) Finding the max at the chunk level (using Ψ_v^1). (iii) Finding the max at the mini-chunk level (using Ψ_v^2) and within a mini-chunk (using Ψ_v^3).

and $\rho_v(y_2)$ for every node v visited, we can query $\mathcal{M}_v$ with the rank range $[\rho_v(y_1)+1, \rho_v(y_2)]$ and the multi-slab Γ_v (recall that Γ_v is the widest multi-slab whose x-span is contained in $[x_1, x_2]$; refer to Fig. 2(i)). The answer to Q is then the maximum of these $O(\log_B n)$ partial maximums. Thus the overall query time will be $O(\log_B^2 n)$ I/Os.

Because we need to store $\mathcal{M}_v$ compactly, $\mathcal{M}_v$ will only return the rank $\rho_v(p)$ of the point p with the maximum weight, instead of its real weight. In this case we need an extra step to find its actual weight in another $O(\log_B n)$ I/Os using $\mathcal{C}_v$ as follows: If v is a leaf, we examine all the points of P_v and return the weight of the point whose rank is $\rho_v(p)$. Otherwise, we use $\mathcal{C}_v$ to determine the slab σ_{v_j} in which p resides (using the CI_v array), compute $\rho_{v_j}(p)$, the rank of p in the set P_{v_j}, and continue the search recursively in v_j. We call this step the *identification* process.

The secondary structure $\mathcal{M}_v$. As before, we first partition P_v into $s = \lceil N_v/\mu \rceil$ chunks $C_1, \ldots, C_s$, each of size $\mu = B\log_B N$. Next, we further partition each chunk C_i into *mini-chunks* of size B: C_i is partitioned into $mc_1, \ldots, mc_{v_i}$, where $v_i = \lceil |C_i|/B \rceil$ and $mc_j \subseteq C_i$ is the sequence of points with 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 mini-chunk) X if for all $p \in X$, $\rho_v(p) \in [\rho_1, \rho_2]$, and that X *crosses* a rank ρ if there are points $p, q \in X$ such that $\rho_v(p) < \rho < \rho_v(q)$.

$\mathcal{M}_v$ 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 "mini-chunk level", and Ψ_v^3 answers MAX queries within a mini-chunk. 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 $C_a, \ldots, C_b$, then we use Ψ_v^1 to report the maximum weight of the points in $\bigcup_{l=a}^{b} C_l \cap \sigma_v[i:j]$ (Fig. 2(ii)). We use Ψ_v^2, Ψ_v^3 to report the maximum weight of a point in $C_{a-1} \cap \sigma_v[i:j]$, as follows. If $mc_\alpha, \ldots, mc_\beta$ are the mini-chunks 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 mini-chunks that cross ρ_1 (Fig. 2(iii)). The maximum weight of a point 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 Ψ_v^3. Ψ_v^3 consists of a small structure $\Psi_v^3[l]$ for each mini-chunk mc_l, $1 \leqslant l \leqslant N_v/B = n_v$. 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 $0 \leqslant \xi_p, \omega_p \leqslant B$, so we need $O(\log_2 B)$ 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 $\rho_v(p)$'s (their ranks in P_v). $\Psi_b^3[l]$ needs a total of $O(B\log_2 B)$ bits. Therefore $\log_B N$ small structures use $O(B\log_2 B \log_B N) = O(B\log_2 N)$ 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 $l \leqslant n_v$, we wish to return the maximum weight of a point in the set $\{p \in mc_l \mid p \in \sigma_v[i:j], \rho_v(p) \in [\rho_1, \rho_2]\}$. We first load the whole $\Psi_v^3[l]$ structure into memory using $O(1)$ I/Os. Since we know the $\rho_v(a)$ of the first point $a \in mc_l$, we can compute in $O(1)$ I/Os the contiguous subsequence of pairs (ξ_p, ω_p) in $\Psi_v^3[l]$ such that $\rho_v(p) \in [\rho_1, \rho_2]$. Among these pairs we select the point q for which $i \leqslant \xi_q \leqslant j$ (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 $\rho_v[q]$, 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 Ψ_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 $\nu_k \leqslant \log_B N$ mini-chunks $mc_1, \ldots, mc_{\nu_k}$. For each multi-slab $\sigma_v[i:j]$, we do the following. For each $l \leqslant \nu_k$, 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* [26] 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 mini-chunks $mc_\alpha, \ldots, mc_\beta$ in C_k, the maximum weight in these mini-chunks 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 maximum weight in a range of mini-chunks, but only the index of the mini-chunk containing the maximum 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 maximum weight of the points in the mini-chunks 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 mini-chunk 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 Ψ_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 maximum 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 maximum weight in chunks $C_a, \ldots, C_b$ 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 $C_a, \ldots, C_b$, and select the maximum weight point. This takes $O(1)$ I/Os. Finally, we select the maximum of the $O(\log_B N)$ weights.

Construction of $\mathcal{M}_v$. The new secondary structures $\mathcal{M}_v$ that we attach to the basic CRB-tree can also be built efficiently. Recall that if we can show that $\mathcal{M}_v$ can be built in $O(n_v)$ I/Os, then the whole structure can be built in $O(n \log_B n)$ I/Os. First, Ψ_v^1 is a B-tree built on top of the maximum weights of each chunk and each slab. These weights can be computed by a scan of P_v and thus Ψ_v^1 can be constructed in $O(n_v)$ I/Os. Since each mini-chunk has B points, we can easily compute the ranks of their weights for each mini-chunk by a scan of P_v, so Ψ_v^3 can also be constructed in $O(n_v)$ I/Os.

Now we focus on the construction of Ψ_v^2, i.e., how to build and encode the Cartesian trees on the mini-chunk level for each multi-slab inside each $\Psi_v^2[k]$. We are now facing the following problem: given $a = O(B)$ sequences of $b = \log_B N$ weights each, $w_{1,1}, \ldots, w_{1,b}$; $w_{2,1}, \ldots, w_{2,b}$; …; $w_{a,1}, \ldots, w_{a,b}$, where $w_{i,j}$ is the maximum weight of multi-slab 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 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 built on $\{w_{i,1}, \ldots, w_{i,j-1}\}$, 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 Ψ_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 construction cost of our range-MAX structure is $O(n \log_B n)$ I/Os.

Theorem 3. *A set of N points in 2D can be stored in a linear-size structure such that a 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.*

The query time of our range-MAX structure can be improved to $O(\log_B^{1+\varepsilon} n)$ for any constant $\varepsilon > 0$, matching the internal memory bound of $O(\log_2^{1+\varepsilon} N)$ [7], by adding the following two ingredients. First, we also use Cartesian trees to implement Ψ_v^1 such that any range-MAX query for any multi-slab on the chunk level can be answered in $O(1)$ I/Os. Secondly, we can use the more complex identification process in [7] to speed it up from $O(\log_B n)$ to $O(\log_B^\varepsilon n)$ I/Os, while maintaining linear space. However, we cannot construct this structure efficiently and therefore cannot make it dynamic either (Section 5).

Theorem 4. *A set of N points in 2D can be stored in a linear-size structure such that a range-MAX query can be answered in $O(\log_B^{1+\varepsilon} n)$ I/Os.*

5. Extensions

In this section we discuss various extensions of the CRB-trees.

Dynamization. Since we can construct our static structures I/O-efficiently, it is rather straightforward to use the *external logarithmic method* [4] to support insertions.

Lemma 1 *(External logarithmic method). (See [4].) Let $\mathcal{D}$ be a static structure of size $O(S(n))$, which can be constructed in $O(nC(n))$ I/Os, such that queries can be answered in $O(Q(n))$ I/Os. Then there exists a data structure $\mathcal{D}'$ of size $O(S(n))$ that answers queries in $O(Q(n)\log_B n)$ I/Os, and supports insertions in $O(C(n)\log_B n)$ I/Os amortized.*

It is also fairly easy to support deletions for the range-COUNT and range-SUM structures, where subtraction is allowed. When a point is deleted we insert it into another insert-only structure $\mathcal{Z}$, which stores all the deleted points so far. A range-COUNT/SUM query can be answered by subtracting the result of the same query in $\mathcal{Z}$ from the answer obtained in the original structure. When we have collected $N/2$ deletions we rebuild the whole structure. Therefore we have the following.

Theorem 5. *A set of N points in $\mathbb{R}^2$ can be stored in a structure using $O(n)$ disk blocks (resp. $O(n\log_B((W\log_2 W)/N)\log_B n)$ disk blocks), such that a range-COUNT (resp. range-SUM) query can be answered in $O(\log_B^2 n)$ I/Os, and a point can be inserted or deleted in $O(\log_B^2 n)$ amortized I/Os.*

We need to do slightly more in order to support deletions for our range-MAX structure. We will only delete the point from the $\mathcal{M}_v$ structures while leave the rest of the CRB-tree untouched. The deleted point will remain in the tree as a "ghost" element (with a weight of $-\infty$), until we perform a global rebuilding when there are $N/2$ of them.

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 mini-chunks (that is, we remove Ψ_v^2). Instead we construct $\Psi_v^3[k]$ directly on the points in C_k. Note that this modification does not increase the query bound. Now we can 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 $-\infty$, and then update Ψ_v^1 if necessary. Recall that Ψ_v^1 is a B-tree built on top of the maximums of each chunk for each slab. It does not include all the points in P_v, so if the point deleted is not the one with the maximum weight in its chunk-slab, then we are done with $\mathcal{M}_v$. Otherwise, we scan through $\Psi_v^3[k]$ to find out the new maximum weight point, identify its weight, and then replace p in Ψ_v^1 with this new point and update the maximums stored in the internal nodes of Ψ_v^1 bottom-up. This process takes a total of $O(\log_B n)$ I/Os. Since we totally update $O(\log_B N)$ secondary structures, the overall cost of a deletion is $O(\log_B^2 N)$ I/Os. However, since $|C_k| \leqslant B\log_B N$, $\Psi_v^3[k]$ 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.

Theorem 6. *A set of N points in $\mathbb{R}^2$ 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.*

Extending to higher dimensions. All our static and dynamic structures can be extended to d dimensions with an $O(\log_B^{d-2} n)$ factor increase in the space, query and update bounds. We do so by constructing multi-level tree structures as follows. 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 $\mathcal{T}^d$. It is a B-tree with fanout b, built on the x_d-coordinates of P. Each internal node v of $\mathcal{T}^d$ is associated naturally with a subset P_v of P and stores a secondary structure $\mathcal{T}^{d-1}$, 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 $\mathcal{T}^2$ (with fanout b) in the x_1x_2-plane.

With each internal node v of $\mathcal{T}^2$, 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 $\mathcal{T}^i$ to which $\mathcal{T}^2$ is attached. For each point $p \in P_v$, and for $2 \leqslant i \leqslant d$, let $w^i_{a_i}$ be the child of w^i (in $\mathcal{T}^i$) 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-multi-slab*. Since all child indexes are less than b, there are $b^{d-1} = \sqrt{B}$ hyper-slabs and $O(B)$ hyper-multi-slabs in each $\mathcal{T}^2$. By using hyper-slabs and hyper-multi-slabs instead of slabs and multi-slabs, we can build the same structures as we described earlier.

A query $Q = [\alpha_1, \beta_1] \times \cdots \times [\alpha_d, \beta_d]$ in $\mathbb{R}^d$ is now answered as follows: The query procedure in $\mathcal{T}^d$ follows two paths to the leaves corresponding to α_d and β_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 $\mathcal{T}^2$, we carry out the same query procedure as before. Only this time, at each internal node of $\mathcal{T}^2$, we answer a range-aggregate query on the points whose x_1-coordinate falls between $[\alpha_1, \beta_1]$ and $(x_2, \ldots, x_d)$ falls in a hyper-multi-slab. This can be easily handled by our 2D structure.

Theorem 7. *The results of Theorems 1–6 can be extended to d-dimension with an $O(\log_B^{d-2} n)$ factor increase in the space, query and update bounds, for any $d > 2$.*

6. Experiments

In this section we report the experimental results on the CRB-tree for 2D range-COUNT queries. The emphasis of our experiments is on the size and query time of the index. Since we are mainly interested in linear-size data structures, we chose to compare the performance of the CRB-tree with that of the *kdB*-tree [21], and not with for example the MVSB-tree [27] and the aP-tree [22] which use $O(n \log_B n)$ space.

6.1. Implementation

We implemented the CRB-tree using the TPIE system developed at Duke. TPIE is designed to facilitate easy and portable implementations of I/O-efficient algorithms and indexing structures, and consists of a set of templated C++ classes and functions. The TPIE system consists of a stream and a block oriented part [25]. In the stream oriented part, user programs are fed a continuous stream of elements in an I/O-efficient manner. In the block oriented part, the external memory is viewed as a collection of blocks and primitives for manipulating such blocks are provided. Both the CRB-tree and *kdB*-tree implementation use both parts of TPIE. The nodes of the B-tree and *kdB*-tree are implemented using blocks. The stream oriented part is used for efficiently implementing the bulk loading algorithm of both the indexes.

For the CRB-tree, the block size of $8K$ bytes allowed for a fanout of 500 and a maximum leaf size of 681. The precise number of blocks used for the CRB-tree, can be roughly estimated to $4n$; n blocks for each of the B-tree Φ, base B-tree T, and the secondary structure arrays *CI* and *PC*. The arrays *CI* and *PC* were also implemented using blocks. We implemented the arrays such that the count $|P_v \cap \Gamma_v \cap Q|$ at any node v of T can be computed using only four I/Os (loading two blocks of *CI* and *PC* array). Thus the query process uses 5 I/Os at each node of T (4 I/Os to access the secondary structure and 1 I/O to access the node). The total number of nodes of T accessed by the query procedure is at most $2\log_B n - 1$, since the query search corresponds to 2 root-to-leaf paths in T. The same is true for the number of nodes of Φ accessed by the query. Thus the total number of I/Os performed by the query procedure is at most $5(2\log_B n - 1) + (2\log_B n - 1) = 6(2\log_B n - 1)$.

Since each node v of our *kdB*-tree stores a balanced binary tree of height 8 whose leaves are the children of v, the $8K$ block size allowed for a fanout of 255 and a maximum leaf size of 681. The number of blocks used for the *kdB*-tree can be roughly estimated to be n. We bulk-load the *kdB*-tree using a top-down approach. At each node of the *kdB*-tree, we store the count of the number of points contained in the subtree of each of its children. The query process traverses the *kdB*-tree, starting from the root. At each node v, it checks which regions corresponding to v's children intersects the query region. The query process recurses on those childrens whose region is intersected by the query region and accumulates the count for those children whose region is contained in the query region.

6.2. Experiments

We evaluated the performance of the CRB-tree using both synthetic and TIGER/Line data. Below we report both the number of (TPIE) I/Os performed and the wall-clock running time of a set of bulk loading and query experiments. Query bounds are averages over 100 queries with the buffer cache being flushed between queries. All our experiments were performed on a Dell PowerEdge 2400 workstation with a 500 MHZ PIII processor and 128 MB of main memory, running FREEBSD 4.3. Physically the machine had 1 GB of main memory, but to simulate a real multi-user database environment we restricted the main memory usage to 128 MB. Furthermore, TPIE was configured to use a maximum of 80 MB, leaving the rest of the memory to the operating system. The external memory consisted of a RAID0 disk array of four 36 GB SCSI disks (IBM DDYS T36950M).

Uniformly distributed points. Our first set of experiments were performed on uniformly distributed points in the range $[0, 10^9] \times [0, 10^9]$ (points were generated by independently choosing a random value for the x and y coordinates). The experiments were performed using data sets sizes ranging from 20 to 140 million points. For each query, we choose a random square with an area equal to 1% of the area of the bounding box of the data set.

Fig. 3 shows the number of I/Os and time taken by the bulk-loading algorithm. The *bulk-loading* time for the CRB-tree is 1.5–2.5 times slower than the *kdB*-tree. This is mainly because the number of blocks in the CRB-tree is 3–4 times larger than that of *kdB*-tree. Hence the CRB-tree algorithm performs more I/Os than the *kdB*-tree algorithm.

Fig. 4 shows the number of I/Os and time taken by the query process. The *query time* of the CRB-tree is almost independent of the dataset size and significantly lower than the query time of the *kdB*-tree. For the datasets sizes used in the experiments, the height of the CRB-tree (T and Φ) is either 2 or 3. Thus the total number of I/Os performed by the query is at most $6(2 * 3 - 1) = 30$. This explains the fact that the query time remains almost constant in these experiments. Since the number of nodes visited by the *kdB*-tree query algorithm increases with increase in data size (it varies as $\sqrt{n}$ in worst case), the query time (I/Os performed) increases significantly as N varies from 20 to 140 million.

From Fig. 4, we can see that the speedup ratio between CRB-tree and *kdB*-tree is significantly higher for number of I/Os (Fig. 4(ii)) compared to that of the of execution time (Fig. 4(i)). The reason for this is as follows: The total execution time is composed of three components: (1) user CPU time, (2) I/O time (time spent in performing I/Os) and (3) kernel CPU time. The CRB-tree query spends a significant time in CPU calculations (because of lots of bit operations) compared to *kdB*-tree query. Fig. 5(i) shows the percentage of time spent in CPU calculations for both the CRB-tree and *kdB*-tree query. Fig. 5(ii)

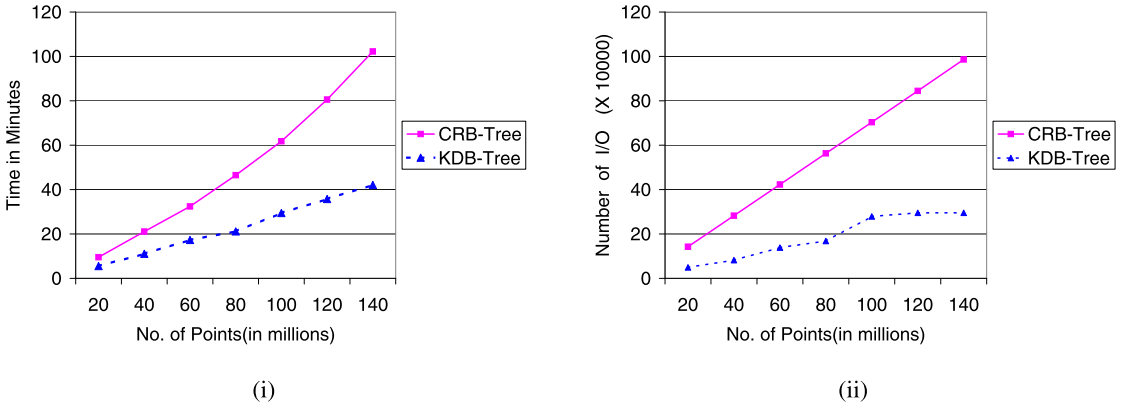

Fig. 3. Comparison of (i) running time and (ii) number of I/Os performed when bulk loading the CRB-tree and *kdB*-tree.

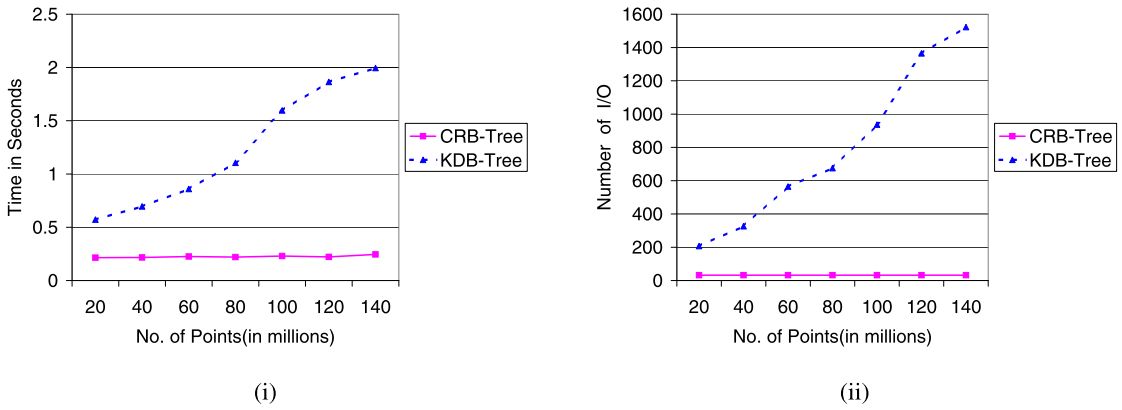

Fig. 4. Comparison of (i) running time and (ii) number of I/Os performed when querying the CRB-tree and *kdB*-tree.

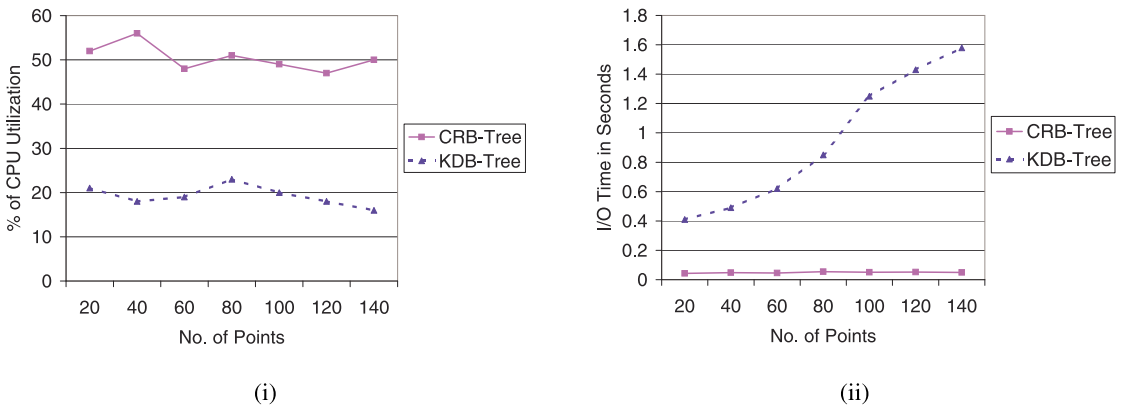

Fig. 5. Comparison of (i) percentage of CPU calculations and (ii) time to perform I/Os (I/O time) of the query algorithm for CRB-tree and *kdB*-tree.

shows the comparison of I/O time for both the CRB-tree and *kdB*-tree query. As we can see, the speedup of I/O time is almost similar to the speedup of the number of I/Os (Fig. 4(ii)).

TIGER/Line data. We used the TIGER/Line data set from the US Bureau of the Census [24], which is one of the standard benchmark datasets used in spatial databases. The TIGER/Line'97 distribution we used consists of six CD-ROMs of data corresponding to six regions of the United States. We performed experiments with six point datasets, corresponding to the data on CD-ROM 1 through i, for $1 \leqslant i \leqslant 6$. The number of points in each of these data sets is shown in Fig. 6.

Fig. 7 shows the result of bulk loading and query experiments with the TIGER/Line datasets. Since the bulk loading time is independent of the characteristics of the data sets, the bulk loading results are similar to the results we obtained with uniformly distributed points. In the query experiments we again used a randomly placed query square with an area equal

	CD1	CD1-2	CD1-3	CD1-4	CD1-5	CD1-6
Number of points(in millions)	22	44	60	81	97	114
Size(in MB)	641	1238	1698	2319	2789	3297

Fig. 6. Points data sets extracted from TIGER/Line'97.

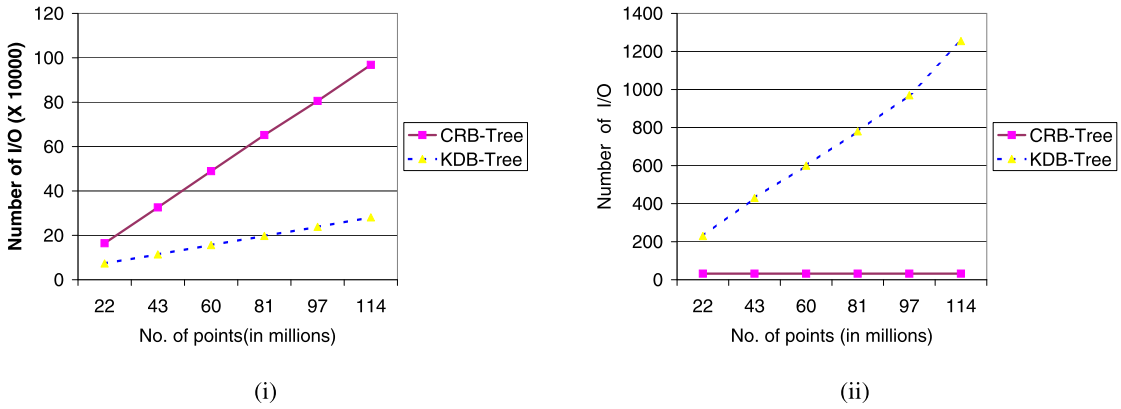

Fig. 7. Comparison of the number of I/Os performed when (i) bulk loading the CRB-tree and kdB-tree and (ii) querying the CRB-tree and kdB-tree using TIGER/Line datasets.

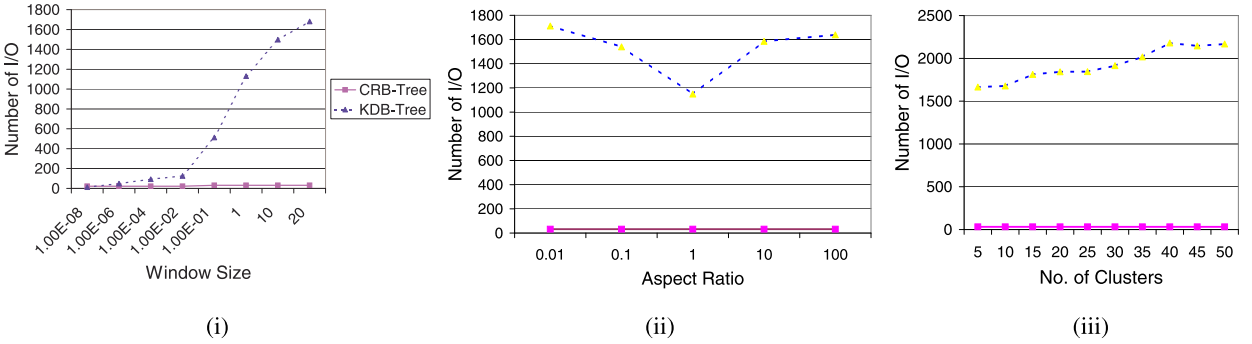

Fig. 8. Comparison of number of I/Os performed when querying the CRB-tree and kdB-tree using the largest TIGER/Line data set and (i) varying the size of query square and (ii) varying the aspect ratio of query rectangle; (iii) shows the comparison of number of I/Os performed during query on synthetic clustered datasets.

to 1% of the area of the bounding box of the data set. The query performance of CRB-tree is similar to that of uniformly distributed points (the base B-tree T is also of height 3 in these experiments). The query time of the kdB-tree on the other hand, increases significantly with increase in dataset size.

Next we investigated the effect of the query rectangle characteristic on query performance. The experiments were performed using the largest data set of TIGER/Line. First we performed query experiments with query squares of different sizes. The results of these experiments are shown in Fig. 8(i), where the size of the query square is characterized by the ratio of its area to the area of the bounding box of the input points. The size of query square is varied from 10^{-8}% to 20%. As it can be seen, the query time of the kdB-tree increases rapidly with increasing window sizes. The query time of the CRB-tree on the other hand is almost constant.

Next we performed query experiments with query rectangles instead of query squares. The results of our experiments with rectangles of varying aspect ratio (the ratio between the length and the breadth of the rectangle) are shown in Fig. 8(ii). The area of the query rectangle is fixed at 1% of the area of the bounding box of the input dataset while the aspect ratio is varied from 0.01 to 100. As it can be seen, the query time of the kdB-tree increases slightly as the query rectangle becomes "skinny" (high aspect ratio or low aspect ratio). The reason for this is that the kdB-tree consists of alternating splits along the x and y dimensions and hence a skinny rectangle intersects more nodes of the kdB-tree than a square of the same area. As expected, the query time for the CRB-tree is almost constant.

Clustered data. Finally, in order to further investigate the influence of the input data distribution on the query performance of the two structures, we performed experiments with artificial clustered datasets. The datasets consists of 150 million points distributed evenly among k clusters, where each cluster is generated by uniformly distributing the points on

a randomly oriented ellipse of length 4×10^8 and width 10^4 centered at $(5 \times 10^8, 5 \times 10^8)$. Fig. 8(iii) shows the results of experiments when k is varied from 5 to 50. As previously, the CRB-tree performance is almost constant. Note that the CRB-tree query performance does not depend on whether the input data is uniform or skewed, since the number of I/Os performed by the CRB-tree query procedure, depends only on the height of the tree and not on the distribution of input data. The query time of the *kdB*-tree increases slightly with increase in number of clusters.

Experimental conclusions. The overall conclusions of our experiments is that while the CRB-tree use 3–4 times more space than the *kdB*-tree and takes 1.5–2.5 times longer to bulk load than a *kdB*-tree, the query performance of the CRB-tree is much better than that of the *kdB*-tree. For a data set with around 100 million points, the CRB-tree query time is 8–10 times faster than *kdB*-tree query time. Furthermore, the query time of the CRB-tree depends only on the height of the tree ($\log_B n$). Thus it is independent of the distribution of the input points and query characteristics, and almost constant for the range of data set sizes used in our experimentation. The query time of the *kdB*-tree on the other hand, depends significantly on the size of the input dataset and the size of the query rectangle. To a lesser extent the query time of the *kdB*-tree also depends on the aspect ratio of the query window and the input point distribution.

However, as pointed out in [22], the CRB-tree is not a particularly practical data structure, due to its relatively high implementation complexity, the bit-packing tricks, and the requirement that the whole index (each level of the tree to be more precise) be stored in consecutively addressed disk pages. Thus, we would recommend using the CRB-tree only in situations where query efficiency and space consumption are much more important than other practical considerations, and only when COUNT queries are required. For SUM queries, the CRB-tree uses the same space (when $W \gg N$) and achieves the same query cost as the more practical aP-tree [22], so the latter is recommended for dealing with SUM queries in practice.

7. Final remarks

We have presented a linear-size structure that supports range-COUNT queries in $O(\log_B n)$ I/Os in the external memory model. This basic structure can be augmented to support various other range-aggregate queries such as range-SUM and range-MAX. The $O(\log_B n)$-I/O bound is optimal when comparison is the only allowed operation on the coordinates. In our structure, we essentially convert the coordinates into the rank space $[1, N]$ using a comparison-based B-tree and then solve the problem in the rank space using non-comparison-based techniques. An intriguing question is whether we can improve the query bound with non-comparison-based techniques on the coordinates as well. In the RAM model, it is indeed possible to improve the query time to $O(\log N / \log\log N)$ [6]. Thus one may wonder if we can beat the $O(\log_B n)$ bound of the CRB-tree using similar techniques. The answer to this question will actually depend on the delicate relationship between B and N. In particular when $B = \Omega(N^\varepsilon)$, certainly the CRB-tree is optimal; when $B = \log^{o(1)} N$, the naive implementation of the $O(\log N / \log\log N)$-query structure in external memory that ignores the blocking at all already beats the $O(\log_B n)$ bound. How about the B values in between? Deciding the precise range of B's in which the $O(\log_B n)$ bound is optimal (with near-linear space) is a very interesting open problem. In fact, the problem has been resolved in one dimension by Pătraşcu and Thorup [19], where it is equivalent to the *predecessor problem*, one of the most fundamental data structure problems. They gave a full characterization of the complexity of the (static version of the) problem in terms of B, N, and the universe size, in particular characterizing the parameter ranges where the B-tree is and is not optimal. Determining when the CRB-tree is optimal in two dimensions will be even more challenging.

Acknowledgements

Pankaj Agarwal was supported by NSF under grants IIS-07-13498, CCF-09-40671, CCF-10-12254, and CCF-11-61359, by ARO grants W911NF-07-1-0376 and W911NF-08-1-0452, and by an ARL award W9132V-11-C-0003. Lars Arge was supported by the Danish National Research Foundation.

References

[1] P.K. Agarwal, J. Erickson, Geometric range searching and its relatives, in: Advances in Discrete and Computational Geometry, American Mathematical Society, 1999, pp. 1–56.
[2] A. Aggarwal, J.S. Vitter, The input/output complexity of sorting and related problems, Communications of the ACM 31 (9) (1988) 1116–1127.
[3] L. Arge, External memory data structures, in: J. Abello, P.M. Pardalos, M.G.C. Resende (Eds.), Handbook of Massive Data Sets, Kluwer Academic Publishers, 2002, pp. 313–358.
[4] L. Arge, J. Vahrenhold, I/O-efficient dynamic planar point location, Computational Geometry: Theory and Applications 29 (2) (2004) 147–162.
[5] J.L. Bentley, Multidimensional divide and conquer, Communications of the ACM 23 (6) (1980) 214–229.
[6] P. Bose, M. He, A. Maheshwari, P. Morin, Succinct orthogonal range search structures on a grid with applications to text indexing, in: Proc. Workshop on Algorithms and Data Structures, 2009, pp. 98–109.
[7] B. Chazelle, A functional approach to data structures and its use in multidimensional searching, SIAM Journal on Computing 17 (3) (1988) 427–462.
[8] D. Clark, Compact pat trees, PhD thesis, University of Waterloo, 1996.
[9] D.R. Clark, J.I. Munro, Efficient suffix trees on secondary storage, in: Proc. ACM–SIAM Symposium on Discrete Algorithms, 1996, pp. 383–391.
[10] D. Comer, The ubiquitous B-tree, ACM Computing Surveys 11 (2) (1979) 121–137.
[11] V. Gaede, O. Günther, Multidimensional access methods, ACM Computing Surveys 30 (2) (1998) 170–231.

[12] R. Grossi, G.F. Italiano, Efficient cross-tree for external memory, in: J. Abello, J.S. Vitter (Eds.), External Memory Algorithms and Visualization, American Mathematical Society, 1999, pp. 87–106.
[13] A. Guttman, R-trees: A dynamic index structure for spatial searching, in: Proc. SIGMOD International Conference on Management of Data, 1984, pp. 47–57.
[14] G. Jacobson, Space-efficient static trees and graphs, in: Proc. IEEE Symposium on Foundations of Computer Science, 1989, pp. 549–554.
[15] G. Jacobson, Succinct static data structures, PhD thesis, Carnegie Mellon University, 1989.
[16] K.V.R. Kanth, A.K. Singh, Optimal dynamic range searching in non-replicating index structures, in: Proc. International Conference on Database Theory, 1999, pp. 257–276.
[17] Y. Nekrich, Orthogonal range searching in linear and almost-linear space, Computational Geometry: Theory and Applications 42 (4) (2009) 342–351.
[18] J. Nievergelt, P. Widmayer, Spatial data structures: Concepts and design choices, in: J.-R. Sack, J. Urrutia (Eds.), Handbook of Computational Geometry, Elsevier Science Publishers B.V. North-Holland, Amsterdam, 2000, pp. 725–764.
[19] M. Pătraşcu, M. Thorup, Time–space trade-offs for predecessor search, in: Proc. ACM Symposium on Theory of Computing, 2006, pp. 232–240.
[20] R. Raman, V. Raman, S.S. Rao, Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets, ACM Transactions on Algorithms 3 (4) (2007), Article No. 43.
[21] J. Robinson, The K-D-B tree: A search structure for large multidimensional dynamic indexes, in: Proc. SIGMOD International Conference on Management of Data, 1981, pp. 10–18.
[22] Y. Tao, D. Papadias, Range aggregate processing in spatial databases, IEEE Transactions on Knowledge and Data Engineering 16 (12) (2004) 1555–1570.
[23] Y. Tao, D. Papadias, J. Zhang, Aggregate processing of planar points, in: Proc. Conference on Extending Database Technology, 2002, pp. 682–700.
[24] TIGER/Line™ Files, 1997 Technical Documentation, Washington, DC, September 1998, http://www.census.gov/geo/tiger/TIGER97D.pdf.
[25] D.E. Vengroff, A transparent parallel I/O environment, in: Proc. DAGS Symposium on Parallel Computation, 1994.
[26] J. Vuillemin, A unifying look at data structures, Communications of the ACM 23 (1980) 229–239.
[27] D. Zhang, A. Markowetz, V. Tsotras, D. Gunopulos, B. Seeger, Efficient computation of temporal aggregates with range predicates, in: Proc. ACM Symposium on Principles of Database Systems, 2001, pp. 237–245.