CRB-Tree: An Efficient Indexing Scheme for Range Aggregate Queries *

Sathish Govindarajan Pankaj K. Agarwal Lars Arge

Center for Geometric Computing,
Department of Computer Science,
Box 90129, Duke University, Durham, NC 27708-0129, USA.
E-mail: {gsat,pankaj,large}@cs.duke.edu

Abstract

We propose a new indexing scheme, called the CRB-tree, 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 weights of points that lie inside a d-dimensional query rectangle. In this paper we focus on COUNT, SUM, AVG aggregates. First, we develop an indexing scheme 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. This is the first optimal index structure for the 2D range-COUNT problem. The index can be extended to obtain a near-linear-size indexing structure for answering range-SUM queries using $O(\log_B N)$ I/Os. We also obtain similar bounds for rectangle-intersection aggregate queries, in which the input is a set of weighted rectangles and a query asks to compute the aggregate of the weights of those input rectangles that overlap with the query rectangle. This result immediately improves a recent result on temporal-aggregate queries. Our indexing scheme can be dynamized and extended to higher dimensions. Finally, we demonstrate the practical efficiency of our index by comparing its performance against kdB-tree. For a dataset of around 100 million points, the CRB-tree query time is 8–10 times faster than the kdB-tree query time. Furthermore, unlike other indexing schemes, the query performance of CRB-tree is oblivious to the distribution of the input points and placement, shape and size of the query rectangle.

*The first two authors are supported by Army Research Office MURI grant DAAH04-96-1-0013, by NSF grants ITR–333–1050, EIA–9870724, EIA–997287, and CCR–9732787, and by a grant from the U.S.-Israeli Binational Science Foundation. The third author is supported by the National Science Foundation through ESS grant EIA–9870734, RI grant EIA-9972879, CAREER grant CCR-9984099, and ITR grant EIA-0112849

1 Introduction

In order to be successful, any data model in a large database requires efficient external memory (secondary storage) support for its language features. Range searching and its variants are problems that often need to be solved efficiently [19]. In on-line analytical processing (OLAP), spatial databases such as geographic information systems (GIS), and several other applications, range-aggregate queries (e.g., range-`COUNT`, range-`SUM`, etc) play an extremely important role, and a large number of algorithms and indexing structures have been proposed to answer such queries; see e.g. [11] and references therein. With the rapid increase in the use of data warehouses to collect historical information, temporal aggregate queries have also received much attention in the last few years [28, 29]. In general, answering temporal (or bi-temporal) aggregate queries is harder than traditional aggregate queries because each data point is associated with a time interval during which its attribute values are valid and a query calls for computing an aggregate over only those points that are valid during the query time interval.

In this paper we present an optimal indexing scheme for answering 2D range-`COUNT` queries. Our index can be extended to efficiently answer multidimensional range-aggregate queries. We can also efficiently answer several important temporal aggregate queries using our index.

Model. Let P be a set of N points in $\mathbb{R}^d$, and let $w : P \to \mathbb{Z}$ be a *weight* function. We wish to build an index on P so that we can efficiently answer *range-aggregate* queries, i.e. compute the aggregate of weights of points in P that lie inside a given d-dimensional query rectangle R. In this paper, we focus on *range-aggregate* queries, such as range-`COUNT`, `SUM`, `AVG`. These three examples call for computing $|P \cap R|$, $\sum_{p \in P \cap R} w(p)$, and $\sum_{p \in P \cap R} w(p)/|P \cap R|$, respectively; see Figure 1 (i). We also study a more general problem in which P is a set of rectangles in $\mathbb{R}^d$. A query is also a d-dimensional rectangle and the goal is to compute an aggregation over the set of input rectangles that overlap with the query rectangle; see Figure 1 (ii). We refer to this problem as the *rectangle-intersection-aggregate* problem. This problem arises naturally when a massive set of points is replaced by a set of rectangles (the weight of a rectangle being the average weight of the data points inside that rectangle) or when an input is a set of complex objects and each object is replaced by its minimum bounding box. In the *temporal range-aggregate* problem, studied in [28, 29], we are given a set P of (time) intervals in $\mathbb{R}^d$ and the goal is to compute an aggregation over the set of intervals that intersect a query rectangle; see Figure 1 (iii). The range-aggregate problem is a special case of the temporal range-aggregate problem, which in turn is a special case of the rectangle-intersection-aggregate problem.

As our main interest is minimizing the number of disk blocks used by the index and the number of disk accesses used to answer a query, we will consider the problem in the standard external memory model [2]. This model assumes that each disk access transmits a contiguous block of B units (words) of data in a single *input/output operation* (or *I/O*). The efficiency of an index is measured in terms of the amount of disk space it uses (measured in units of disk blocks) and the number of I/Os required to answer a query, to bulkload the index, and to update the index. The minimum number of disk blocks we need to store N points is $\lceil N/B \rceil$, which we refer to as "linear" size. We introduce the notation $n = \lceil N/B \rceil$. We also assume that the size of internal memory M is at least B^2 and that the integer N can be stored in a single word, i.e. that a word can

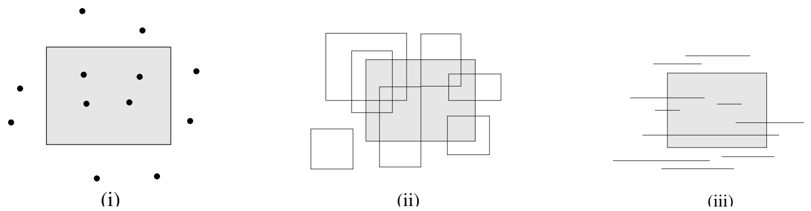


Figure 1. (i) Range-aggregate query, (ii) Rectangle-intersection-aggregate query, (iii) Temporal range-aggregate query.

store $\lceil \log N \rceil$ bits and each bit can be accessed individually. The latter assumption is made in any reasonable computational model and is important for our space- and query-efficient index.

Related work. There has been a lot of work in the spatial database community on indexing a set of points for answering range queries. Indexing schemes such as numerous variants of kdB-trees and R-trees, external priority search trees, etc. have been proposed; see [3, 19] for recent surveys. A kdB-tree uses $O(n)$ disk blocks and can be used to answer a range-aggregate query in $O(\sqrt{n})$ I/Os. No worst-case bound on the query performance of the R-tree is known. Recently, Tao et al [24] proposed a aP-tree that uses $O(n \log_B n)$ disk blocks and answers a range aggregate query using $O(\log_B n)$ I/Os. Their main idea is to reduce the range aggregate query to a special case of temporal aggregate query. This temporal aggregate query is answered using a modified multi-version B-tree [6]

As temporal aggregation is being included in most of the temporal languages, including TQUEL [23] and TSQL2 [22], there has been a flurry of activity on answering temporal aggregate queries. After several early results, including [16, 17, 15], Yang and Widom [28] proposed an indexing scheme called the SB-tree that stores a set of N "time" intervals in $\mathbb{R}^2$ using $O(n \log_B n)$ disk blocks so that for a time interval Δ, the aggregate over all keys that are valid at some time in the interval Δ can be computed using $O(\log_B n)$ I/Os. Recently, Zhang *et al.* [29] proposed a *multiversion* SB-tree (MVSB-tree) that uses $O(n \log_B n)$ disk blocks in the worst case and that can answer a temporal range-aggregate query in $O(\log_B n)$ I/Os. Their index can also answer range-aggregate queries such as range-`COUNT` and range-`SUM` in $O(\log_B n)$ I/Os. The SB-tree and MVSB-tree also support insertion/deletion in $O(\log_B n)$ I/Os.

There is also a vast literature on answering range-aggregate queries in OLAP systems, in which the data is typically modeled as a multidimensional cube and queries typically involve aggregation across various cube dimensions [7, 12, 13, 14, 18, 21, 27]. Although the data cube is a good data model for OLAP systems, it only works well when the data is dense in the sense that the data in a d-dimensional data cube with a total of N cells has more than $N^{1/d}$ distinct keys (on average) in each dimension. In many applications such as spatial databases, data tends to be sparse and data cube model does not work well. For example, consider a set of N points in the plane with distinct x- and y-coordinates. Modeling such a set by a data cube will require a $N \times N$ two-dimensional cube with a total of N^2 cells, out of which only N cells contain entries.

Several range-searching data structures in the internal memory model have been developed in computational geometry; see [1] for a survey. The best-known data structure in $\mathbb{R}^2$ is the *range tree*, which uses $O(N \log_2 N)$ space and can answer a range-aggregate query such as range-`SUM` in $O(\log_2 N)$ time [1]. Chazelle [8] developed the *compressed range-tree* data structure that uses $O(N)$ space under the so-called bit model (in which a word can store $\log_2 N$ bits and each bit can be manipulated individually). This structure can be used to answer a range-`COUNT` query in $O(\log_2 N)$ time. Both of these structures use subtraction to answer range-`COUNT` queries. The compressed range trees can be extended to other range-aggregate queries by paying a polylogarithmic overhead in the query time. If we do not allow subtraction, then the lower-bound results by Chazelle [9] in the semigroup model (see e.g. [1, 9] for a precise definition of this model) suggest that a structure that answers a 2D range-`COUNT` query in $O(\log_2 N)$ time has to use super-linear storage.

Our results. Our main result, described in Section 2, is a new indexing scheme, called the *Compressed Range B-tree* (or *CRB-tree*), for answering two-dimensional range-`COUNT` queries. This structure is an external version of the compressed range-tree [8]. It uses $O(n)$ disk blocks, answers a query in $O(\log_B n)$ I/Os, and can be bulk-loaded using $O(n \log_B n)$ I/Os. This is the first optimal indexing scheme for the 2D range-`COUNT` problem in the I/O model. Using a partial-rebuilding scheme [5], a point can be inserted/deleted in $O(\log_B^2 n)$ I/Os.

Section 3 presents several extensions of our basic structure. We first adapt the structure for answering 2D range-`SUM` 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. It can also answer other range-aggregate queries such as range-MAX, MIN in $O(\log_B^2 n)$ I/Os. Next we extend our index to higher dimensions. In $\mathbb{R}^d$, the structure uses $O(n \log^{d-2} n)$ disk blocks, answers a range-COUNT query in $O(\log_B^{d-1} n)$ I/Os, and can be bulk loaded in $O(n \log_B^{d-1} n)$ I/Os. Similar bounds can be derived for other range-aggregate queries. Using a result by Edelsbrunner and Overmars [10], our index can also be used to answer rectangle-intersection-COUNT(resp. SUM) queries without affecting the asymptotic bound. Since temporal range-aggregate queries are a special case of rectangle-intersection aggregate queries, our structure improves upon the recent results by Zhang et al. [29] when the total weights of the points is $O(N)$. More precisely, we can answer a temporal range-aggregate query in $O(\log_B n)$ I/Os using an $O(n \log_B(\log_2 N))$-size index.

We have implemented the two-dimensional CRB-trees and in Section 4 we report the results of an extensive experimental evaluation of its efficiency. Since we are mainly interested in linear space indexes, we compare the performance of the CRB-tree with that of the kdB-tree. We have evaluated the performance of these index structures using synthetic and real life data. Our first set of experiments study the query and bulk loading performance on datasets of size ranging from 20 to 140 million points. Our experiments show that the query performance of CRB-trees is significantly better than that of kdB-trees. 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. Our second set of experiments compare the query performance by varying the size and shape of the query rectangle. The query performance of CRB-trees is independent of the shape and size of the query rectangle while the query time of kdB-tree increases rapidly with the size of the query rectangle.

2 A 2D Indexing Scheme

In this section, we describe the *Compressed Range B-tree* (CRB-tree), an indexing scheme for answering two-dimensional range-COUNT queries. The structure is an external version of an internal memory structure due to Chazelle [8]. We first describe the CRB-tree and how to bulkload it, and then present the query procedure.

CRB-tree. Let P denote the set of N points in the plane. A CRB-tree consists of two structures: a B^+-tree T constructed on the x-coordinates of P, with each internal node v storing a secondary structure Σ_v, and a normal B^+-tree Ψ constructed on the y-coordinates of P. Figure 2 shows an example of a CRB-tree T on $N = 16$ points. Let $P_v = \{p_1, p_2, \dots\}$ denote the sequence of points contained in the subtree of T rooted at v, sorted in a non-decreasing order of their y-coordinates. Set $N_v = |P_v|$ and $n_v = N_v/B$.

The secondary structure Σ_v will be used to count the number of points of P_v that "belong" to a given child of v and whose y-coordinates are at most y_0, for a given y-value y_0. Intuitively, for each child v_i of v, Σ_v should maintain an array whose jth entry, for $j \le N_v$, stores the count, i.e. how many points among the first j points of P_v belong to P_{v_i}. Using these "prefix sums", we can determine in one I/O the number of points of P_v that belong to P_{v_i} and whose y-coordinates are less than y_0. However, storing these prefix sums requires $O(n_v)$ disk blocks, which would lead to an overall space of $O(n \log_B n)$ blocks. We therefore store the "prefix sum" array compactly, using only $O(n_v/(\log_B n))$ blocks. Roughly speaking, we partition P_v into consecutive *"chunks"* and compute the prefix-sums 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 P_{v_i}. Let the predecessor of y_0 in P_v belong to the kth *chunk* of P_v. The desired count is the sum of the "prefix sum" till the $(k-1)$th *chunk* and the number of points within the kth *chunk* that belong to P_{v_i} and whose y-coordinates are less than y_0. We also preprocess each *chunk* separately so that we can compute the latter count using $O(1)$ I/Os.

More precisely, Σ_v consists of two arrays — a *child index* array CI_v and a *prefix count* array PC_v. Let $v_0, v_1, \dots, v_{B-1}$ be the children of 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 subtree of v storing the ith point p_i of P_v, i.e. p_i is stored at a leaf in the subtree rooted at v_{b_i}. (For example, in Figure 2 (iv), $CI_v[3] = 1$ since the third point in P_v, $(7, 3)$, belongs to the subtree of v_1). Since $b_i < B$, the $\log_2 B$ bits of the ith entry 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 the CI_v array into *"chunks"* of length μ and store the prefix sum at the *chunk* level in prefix count array. We regard the prefix count array 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 $1 \le i \le r$ and $0 \le j < B$, $PC_v[i,j]$ stores the number of points among the first $i\mu$ points of P_v that belong to P_{v_j}, $PC_v[i,j] = |\{p_1, \ldots, p_{i\mu}\} \cap P_{v_j}|$. (For example, in Figure 2 (iv), $\mu = 8$ and $PC[1,0] = 2$ since among the first 8 points of P_v, only 2 points belong to v_0). PC_v requires $rB = N_v/\log_B N$ words. Hence Σ_v can be stored using $O(n_v/\log_B n)$ disk blocks. Since the height of T is $O(\log_B n)$ and a point of P is stored only at one leaf of T, we have that $\sum_{v \in T} n_v = O(n \log_B n)$. The total space used by T is thus $\sum_{v \in T} n_v/\log_B n = O(n)$ disk blocks.

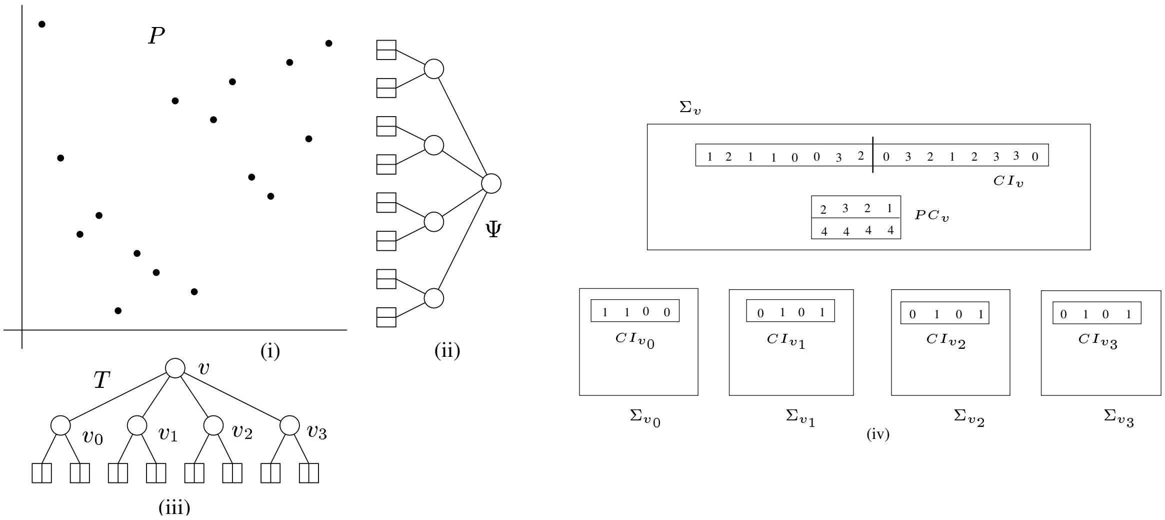


Figure 2. CRB-tree for a set of $N = 16$ points.
(i) $P = \{(5,1), (9,2), (7,3), (6,4), (3,5), (4,6), (13,7), (12,8), (2,9), (15,10), (10,11), (8,12), (11,13), (14,14), (16,15), (1,16)\}$ (ii) B-tree Ψ on the y-coordinates of P;B=4 and each leaf stores 2 points of P. (iii) B-tree T on the x-coordinates of P;B=4 and each leaf stores 2 points of P. $P_v = P$, $P_{v_0} = \{(3,5), (4,6), (2,9), (1,16)\}$, $P_{v_1} = \{(5,1), (7,3), (6,4), (8,12)\}$, $P_{v_2} = \{(9,2), (12,8), (10,11), (11,13)\}$, $P_{v_3} = \{(13,7), (15,10), (14,14), (16,15)\}$ (iv) Secondary structure stored at each internal node of T. $\mu = 8$; 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 at the child v_j of the root.

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

We construct a level i node v and its secondary secondary structure Σ_v as follows: We first, compute P_v (sorted by the y-coordinates) by merging $P_{v_0}, P_{v_1}, \ldots, P_{v_u}$, where $v_0, \ldots, v_u$ are the children of v. 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, that is, in $O(n_v)$ I/Os. Recall that $CI_v[i]$ contains the index of the child containing p_i. In order to construct $CI_v[i]$ we record the index of the child of v from which each point of P_v came from. This way we can construct CI_v using a single scan through P_v. Since $PC_v[i,j]$ contains the number of points among the first $i\mu$ points in P_v that belong to P_{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 $p_{(i-1)\mu+1}, \ldots, p_{i\mu}$. Since $0 \le j \le B-1$, we maintain $PC_v[i,0], PC_v[i,1], \ldots, PC_v[i,B-1]$ in internal

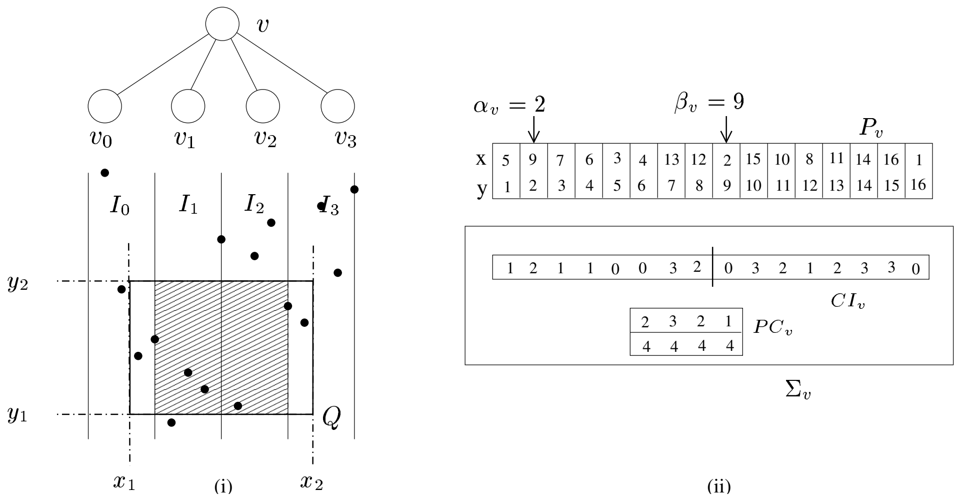


Figure 3. (i) The query rectangle $Q = [2.5, 13.5] \times [1.5, 9.5]$ along with the input points P drawn in Figure 2. I_j is the interval associated with the child v_j of the root. $\lambda = 0, \rho = 3$. $|P_{v_0} \cap Q|$, $|P_{v_3} \cap Q|$ are computed recursively and $|P_{v_1} \cap Q|$, $|P_{v_2} \cap Q|$ are computed at the root. (ii) P_v and the secondary structure Σ_v at the root.

memory. 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 $PC_v[i, 0], PC_v[i, 1], \ldots, PC_v[i, B-1]$ to disk.

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

Answering queries. Let $Q = [x_1, x_2] \times [y_1, y_2]$ be a query rectangle. We compute $|P \cap Q|$ by traversing T in a top-down manner, visiting $O(\log_B n)$ nodes. The query procedure traverses T along two paths , namely the paths from the root to the leaves w and z containing x_1 and x_2, respectively. We use the secondary structures along the two paths to answer the query. To explain how, imagine associating an x-interval I_v with each node v of T. At the root u of T, I_u is the entire x-axis and the interior of the node v partition I_v into B x-intervals. We associate the ith interval with the ith child of v. Consider the topmost node v such that I_v contains $[x_1, x_2]$ but none of the x-intervals I_{v_j} associated with a child of v contains $[x_1, x_2]$; v is the nearest common ancestor of w and z. Let v_λ and v_ρ be the two children of v such that x_1 and x_2 lie in the intervals I_λ and I_ρ respectively. (Figure 3 (i) shows the intervals for the children of the root node of T. For the given Q, $\lambda = 0, \rho = 3$ at the root). Obviously $|P \cap Q| = \sum_{\lambda \le j \le \rho} |P_{v_j} \cap Q|$. To answer the query we compute the count $C = \sum_{\lambda < j < \rho} |P_{v_j} \cap Q|$ at v using the secondary structure Σ_v and recursively visit the children v_λ and v_ρ to compute $|P_{v_\lambda} \cap Q|$ and $|P_{v_\rho} \cap Q|$. (As shown in Figure 3 (i), we compute the counts $|P_{v_1} \cap Q|$ and $|P_{v_2} \cap Q|$ at v, and recursively compute $|P_{v_0} \cap Q|$ and $|P_{v_3} \cap Q|$). Below we show how we can compute the count $|P_{v_i} \cap Q|$ at one node in $O(1)$ I/Os. When we reach a leaf w, we compute $|P_w \cap Q|$ in $O(1)$ I/Os by scanning all the (at most B) points in P_v. Since we visit $O(\log_B n)$ nodes we need $O(\log_B n)$ I/Os to answer a query.

In order to compute the above count, we maintain two variables α_v and β_v when visiting a node v during the traversal of the tree: α_v is the rank of the first point in P_v whose y-coordinate is at least y_1, and β_v is the rank of the last point in P_v whose y-coordinate is not larger than y_2. For $0 \le j \le B-1$ and $1 \le r \le N_v$, let $\varphi(j, r)$ denote the number of points in P_v of rank at most r that belong to P_{v_j}, i.e. $\varphi(j, r)$ is the number of entries among the first r entries of CI_v that store the index j. For $\lambda < j < \rho$, since the x-coordinates of P_{v_j} lie in the range $[x_1, x_2]$, $|P_{v_j} \cap Q| = \varphi(j, \beta_v) - \varphi(j, \alpha_v)$. (For example, from the P_v array in Figure 3 (ii), $\varphi(1, 9) = 3$, since 3 points among the first 9 points of P_v belong to P_{v_1}. Similarly, $\varphi(1, 2) = 1$. Thus we have $|P_{v_1} \cap Q| = 3 - 1 = 2$. From Figure 3 (i), we see that $|P_{v_1} \cap Q| = 2$ since there are 2 points of P_v in the shaded region of child v_1). It thus suffices to describe how to maintain variables α_v and β_v and how to compute $\varphi(0,r),\ldots,\varphi(B-1,r)$.

We can compute α_v and β_v, at the root of T in $O(\log_B n)$ I/Os by searching for y_1 and y_2 in the B-tree ψ. α_v and β_v are the ranks of the points at which the search terminates. Assuming we have computed α_v and β_v at a node v of T, we can compute α_{v_j} and β_{v_j} for all children v_j of v as follows. Since α_{v_j} is the rank of the first point in P_v that belongs to P_{v_j} and whose y-coordinate is at least y_1, $\alpha_{v_j} = \varphi(j, \alpha_v)$ and $\beta_{v_j} = \varphi(j, \beta_v)$. Thus the problem of maintaining α_v, β_v reduces to computing $\varphi(j, \alpha_v)$ and $\varphi(j, \beta_v)$ for all $0 \le j \le B-1$.

All that remains is to describe the procedure for computing $\varphi(0,r),\ldots,\varphi(B-1,r)$ for a given r in $O(1)$ I/Os. Suppose $r = \mu a + c$ for $a \ge 0$ and $0 \le c \le \mu$. Then

$$\begin{aligned} \varphi(j,r) &= |\{k \mid k \le r \text{ and } CI_v[k] = j\}| \\ &= PC_v[a,j] + |\{k \mid \mu a < k \le r \text{ and } CI_v[k] = j\}|. \end{aligned} \tag{1}$$

Thus our procedure simply reads the two disk blocks storing $PC_v[a,1],\ldots,PC_v[a,B]$ and $CI_v[\mu a + 1],\ldots,CI_v[r]$, respectively, and then computes $\varphi(0,r),\ldots,\varphi(B-1,r)$ using (1). (For example, let us calculate $\varphi(2,11)$. From the PC and CI array in Figure 3 (ii), $\varphi(2,11) = PC[1,2] + 1 = 3$. From the P_v array in Figure 3 (i), there are 3 points among the first 11 points that belong to P_{v_2}).

Theorem 2.1 *A set of N points in the plane can be stored in a linear-size index structure using $O(n \log_B n)$ I/Os so that a range-*`COUNT` *query can be answered in $O(\log_B n)$ I/Os.*

We can make the CRB-tree structure dynamic by slightly modifying the external-memory logarithmic method of Arge and Vahrenhold [5]. Omitting all the details from this abstract, we obtain the following.

Theorem 2.2 *A set of N points in the plane can be stored in a linear-size index structure so that a range-*`COUNT` *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.*

3 Extensions

In this section we discuss various extensions of CRB-trees. We present the main ideas in these extensions and omit the details.

Range-`SUM` queries. We first discuss how to extend CRB-trees to answer range-`SUM` queries in the plane. Let P be a set of N points in $\mathbb{R}^2$, and let $w : P \to \mathbb{Z}$ be the weight function. 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 $W = \sum_{p \in P} w(p)$ and $\omega = \log_B(\log_2 W) + \log_B(W/N)$. We extend the CRB-tree by storing four additional arrays W_v, CW_v, L_v, and CL_v in the secondary structure Σ_v of each internal node v of the B-tree. Let $w_i = w(p_i)$ be the ith point $p_i \in P_v$. Set $s_i = \max\{\log_2 \log_2 W, \log w_i\}$; $s_i \le \log_2 W$. We store w_i in the array W_v, using s_i bits, as a continuous sequence of bits. W_v requires at most $\mu_v = \sum_i s_i \le N_v \log_2 \log_2 W + N_v \log_2(W/N)$ bits and thus $\mu_v = O(n\omega/\log_B n)$ disk blocks. W_v plays the role of CI_v. Since W_v is stored as a packed array, we need two additional arrays L_v and CL_v to determine the index in W_v that stores the leftmost bit of w_i, for any given $i \le N_v$. 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 s_i. Since $s_i \le \log_2 W$, it needs at most $\log_2 \log_2 W$ bits. The size of L_v is thus $O(\frac{N_v}{B} \log_2(\log_2 W)/(\log_2 N)) = O(n_v \log_B(\log_2 W)/(\log_B n))$ disk blocks. CL_v, an array of length $O(n\omega/\log_B n)$ blocks, stores the prefix sum of L_v, as PC_v in Section 2, so that the leftmost bit of w_i

in W_v, for any $i \leq N_v$, can be computed using $O(1)$ I/Os. Finally, CW_v stores prefix sum of weights in W_v, as PC_v in Section 2, so that for any $i \leq N_v$, one can compute the sum of weights of points in $\{p_1, \ldots, p_i\} \cap P_w$ for all children w of v. Since the details are similar to Section 2, we omit them from this abstract and conclude the following.

Theorem 3.1 *Let P be a set of N points in $\mathbb{R}^2$, let $w : P \rightarrow \mathbb{Z}$ be a weight function, and let $W = \sum_{i=1}^{N} w_i$. We can bulk load an index 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 so 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 size and bulk loading bounds are $O(n)$ and $O(n \log_B n)$, respectively.*

Indexing in higher dimensions. The CRB-tree can be extended to answer range-aggregate queries in $\mathbb{R}^d$ by constructing a multi-level tree structure as follows. Again, we focus on range-`COUNT` queries. Let P be a set of N points in $\mathbb{R}^d$, and set $b = B^{1/(d-1)}$. A d-dimensional CRB-tree is a B-tree T^d, with fanout b, built on the x_d-coordinates of P. Each internal node v of T^d is associated with a subset P_v of P and stores a secondary structure, which is a $(d-1)$-dimensional CRB-tree T^{d-1} on the projection of P_v onto the hyperplane $x_d = 0$. The recursion stops when we have built the two-dimensional CRB-tree (with fanout b) in the x_1x_2-plane. As in the 2D case, the secondary structure of each internal node v of T^2 stores two arrays CI_v and PC_v, though each entry now keeps more information.

Let T^2 be a two-dimensional CRB-tree and let v be an internal node of T^2. We associate a $(d-1)$ tuple $(w_d, w_{d-1}, \ldots, w_2)$ with v where $w_2 = v$ and w_i is a node of the i-dimensional CRB-tree to which T^2 is attached. For each point $p \in P_v$, and for $2 \leq i \leq d$, let w_{a_i} be the child of w_i (in the i-dimensional tree) so that $p \in P_{w_{a_i}} \subset P_{w_i}$. We call $(a_d, \ldots, a_2)$ the child-index sequence of p. CI_v is a two-dimensional array with N_v rows and $d-1$ columns. The row corresponding to the point $p \in P_v$ stores the child-index sequence of p. CI_v requires $N_v(d-1)\log b = O(N_v \log B)$ bits. PC_v stores the prefix sum of CI_v, as in Section 2. The total size of T^d is $O(n \log^{d-2} n)$ disk blocks and can be constructed using $O(n \log^{d-1} n)$ I/Os.

Let the query rectangle Q in $\mathbb{R}^d$ be $[\alpha_1, \beta_1] \times \cdots \times [\alpha_d, \beta_d]$. The query procedure in T_d follows two paths to the leaves corresponding to the points α_d and β_d. For each node v on these paths, the procedure recursively visits the $(d-1)$-dimensional CRB-tree stored at v .When we reach a node v of T^2, for all children w of v, we count $|P_w \cap [\alpha_1, \beta_1] \times \cdots [\alpha_d, \beta_d]|$. This can be reduced to answering the following query: given a real value α and a $(d-1)$-tuple $\omega = (\omega_d, \ldots, \omega_2)$, count all the points in P_w whose x_1-coordinates are at most α and whose child-index sequence is ω. We use arrays CI_v and PC_v to answer the above query efficiently. Omitting all the details, we conclude the following.

Theorem 3.2 *Let P be a set of N points in $\mathbb{R}^d$. We can bulk load an index on P using $O(n \log_B^{d-1} n)$ I/Os that uses $O(n \log_B^{d-2} n)$ disk blocks and can answer a d-dimensional range-*`COUNT`* query in $O(\log_B^{d-1} n)$ I/Os.*

Rectangle-intersection aggregate queries. CRB-trees can be extended to answer rectangle-intersection aggregate queries by using the reduction of Edelsbrunner and Overmars [10] to transform rectangle-intersection `COUNT` queries in $\mathbb{R}^d$ to range-`COUNT` queries in $\mathbb{R}^d$. Omitting all the details, we have the following result.

Theorem 3.3 *Let P be a set of N rectangles in $\mathbb{R}^d$. We can bulk load an index on P using $O(n \log_B^{d-1} n)$ I/Os that uses $O(n \log_B^{d-2} n)$ disk blocks and can answer a d-dimensional range-intersection *`COUNT`* query in $O(\log_B^{d-1} n)$ I/Os.*

4 Experimental Results

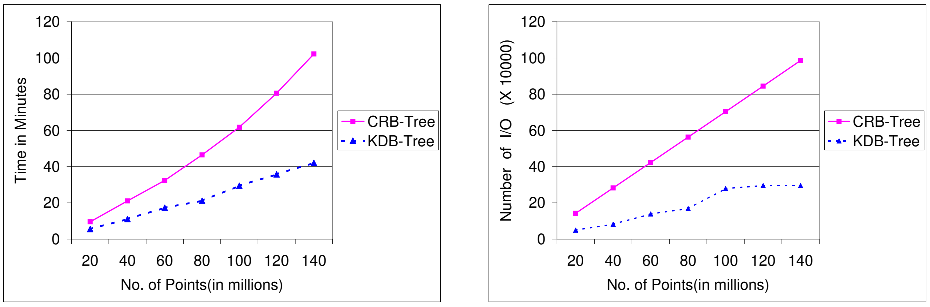
In this section we report the results of an extensive set of experiments with the CRB-tree. The emphasis of our experiments is on the size and query time of the index, and therefore we provide experimental results only for 2D range COUNT queries. Since we are mainly interested in linear space indexes, we chose to compare the performance of the CRB-tree with that of the kdB-tree [20], and not e.g. with the MVSB-tree which uses $O(n \log_B n)$ space [29]. In the full version of this paper, we will provide experimental results on SUM queries and as well as on dynamization.

4.1 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 [26, 4]. 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 base tree T, Ψ, 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 entries needed to compute the count C at any node of T can be loaded 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 almost $2\log_B n - 1$, since the query search corresponds to 2 root-to-leaf paths in T. The same is true for the number of nodes of ψ accessed by the query. Thus the total number of I/Os performed by the query procedure is $5(2\log_B n - 1) + (2\log_B n - 1) = 6(2\log_B n - 1)$.

Since each node of our implementation of the kdB-tree contains a small binary tree of height 8, 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.



(i) (ii)

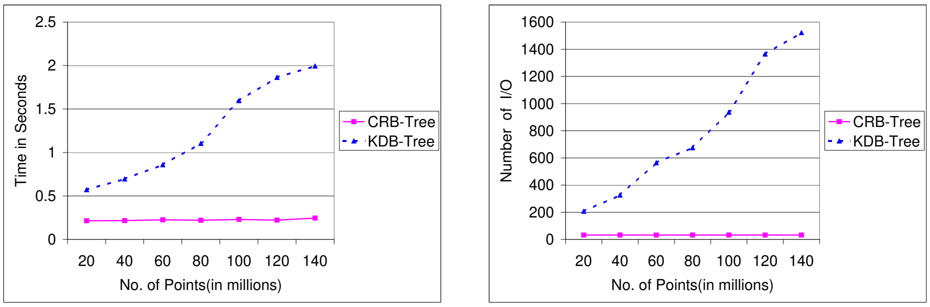
Figure 4. Comparison of (i) running time and (ii) number of I/Os performed when bulk loading the CRB-tree and kdb-tree.

4.2 Experiments

We evaluated the performance of the CRB-tree using both synthetic and real-life 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 (where the buffer cache is flushed between queries). All our experiments were performed on a Dell PowerEdge 2400 workstation with a 500MHZ PIII processor and 128MB of main memory, running FREEBSD 4.3. Physically the machine had 1GB of main memory, but to simulate a real multi-user database environment we restricted the main memory usage to 128MB. Furthermore, TPIE was configured to use a maximum of 80MB, leaving the rest of the memory to the operating system. The external memory consisted of a RAID0 disk array of four 36GB 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 the query, we used a randomly placed query square with an area equal to 1% of the area of the bounding box of the data set.

Figure 4 shows the number of I/Os and time taken by the bulkloading algorithm. The *bulk loading* time for the CRB-tree is 1.5–2.5 times slower than the kdB-tree and as we can see, this is mainly because the number of blocks in the CRB-tree is 3–4 times larger than kdB-tree, hence the CRB-tree algorithm performs more I/Os than the kdB-tree algorithm.



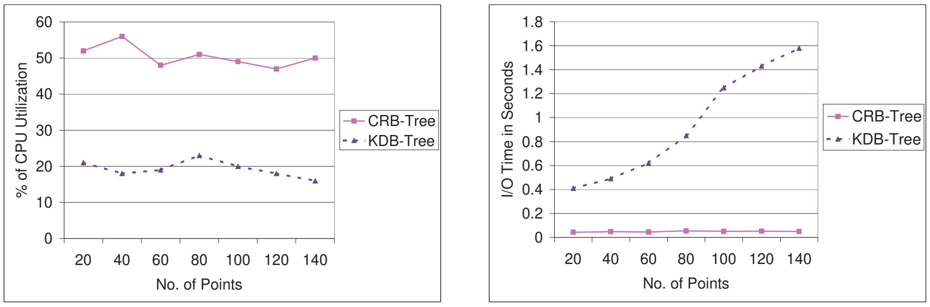
(i) (ii)

Figure 5. Comparison of (i) running time and (ii) number of I/Os performed when querying the CRB-tree and kdB-tree.

Figure 5 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 Ψ) 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 is varied from 20 to 140 million.

From Figure 5, we can see that the speedup ratio between CRB-tree and kdB-tree is significantly higher for number of I/Os(Figure 5 (i)) compared to that of the of execution time(Figure 5 (ii)). 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 process spends a significant time in CPU calculations (because of lots of bit operations) compared to kdB-tree query. Figure 6 (i) shows the percentage of time spent in CPU calculations for both the CRB-tree and kdB-tree query processes. Figure 6 (ii) shows the comparison of I/O time of the query processes. As we can see, the speedup of I/O time is almost similar to the speedup of the number of I/Os of Figure 5 (ii).

Real life data. We used the TIGER/Line data set from the US Bureau of the Census [25], 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 $1 \leq i \leq 6$. The number of points



(i) (ii)

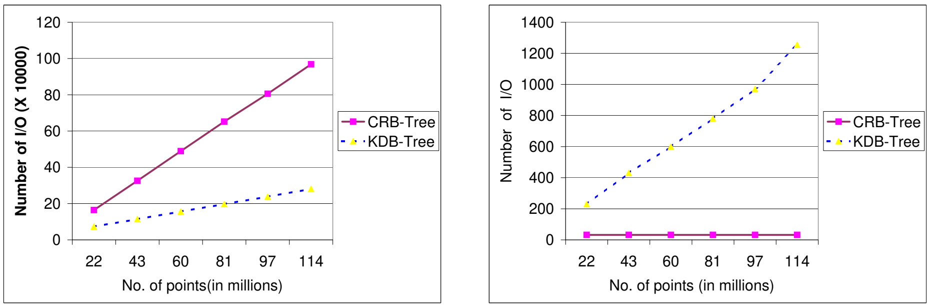
Figure 6. Comparison of (i) percentage of CPU calculations and (ii) time to perform I/Os(I/O time) of the query algorithm for CRB-tree and kdB tree.

in each of these data sets is shown in Figure 7.

	CD1	CD1-2	CD1-3	CD1-4	CD1-5	CD1-6
Number of points	22681842	43961741	60133868	81679982	97323656	114293665
Size(in MB)	641	1238	1698	2319	2789	3297

Figure 7. Points data sets extracted from TIGER/Line'97

Figure 8 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 to 1% of the area of the bounding box of the data set. The query performance of CRB-trees 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.

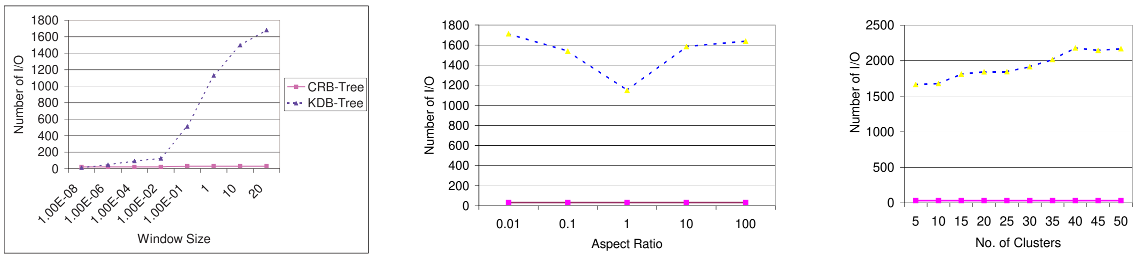


(i) (ii)

Figure 8. 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 (number of points in millions).

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 Figure 9(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



(i) (ii) (iii)

Figure 9. 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

.

our experiments with rectangles of varying aspect ratio (the ratio between the length and the breadth of the rectangle) are shown in Figure 9(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 $(5 \times 10^8, 5 \times 10^8)$. Figure 9(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.

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.

Acknowledgments

The authors thank Sariel Har-Peled for useful discussions and Octavian Procopiuc for answering numerous questions related to the TPIE system.

References

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

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

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

[4] L. Arge, R. Barve, O. Procopiuc, L. Toma, D. E. Vengroff, and R. Wickremesinghe. *TPIE User Manual and Reference (edition 0.9.01a)*. Duke University, 1999. The manual and software distribution are available on the web at `http://www.cs.duke.edu/TPIE/`.

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

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

[7] C. Y. Chan and Y. E. Ioannidis. Hierarchical cubes for range-sum queries. In *Proc. of 25th International Conference on Very Large DataBases*, pages 675–686, 1999.

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

[9] B. Chazelle. Lower bounds for orthogonal range searching, II: The arithmetic model. *J. ACM*, 37:439–463, 1990.

[10] H. Edelsbrunner and M. H. Overmars. On the equivalence of some rectangle problems. *Information Processing Letters*, 14(3):124–128, 1982.

[11] V. Gaede and O. Günther. Multidimensional access methods. *ACM Comput. Surv.*, 30:170–231, 1998.

[12] S. Geffner, D. Agarwal, and A. E. Abbadi. The dynamic datacube. In *Proc of Intl. Conference on Extending Database Technology*, pages 237–253, 2000.

[13] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *Proc. of ACM SIGMOD Intl. conference on Management of Data*, pages 205–216, 1996.

[14] C. Ho, R. Agarwal, N. Megiddo, and R. Srikant. Range queries in olap data cubes. In *Proc. of SIGMOD Conference on Management of Data*, pages 73–88, 1997.

[15] J. Kim, S. Kang, and M. Kim. Effective temporal aggregation using point-based trees. In *Database and Expert Systems Applications*, pages 1018–1030, 1999.

[16] N. Kline and R. T. Snodgrass. Computing temporal aggregates. In *Proc. of Intl conference on Data Engineering*, pages 222–231, 1995.

[17] C. P. Kolovson and M. Stonebraker. Segment indexes: Dynamic indexing techniques for multidimensional interval data. In *Proc. of ACM SIGMOD Intl. conference on Management of Data*, pages 138–147, 1991.

[18] S. Lee, W. Ling, and H. Li. Hierarchical compact cubes for range-max queries. In *Proc of 26th International Conference on Very Large DataBases*, 2000.

[19] J. Nievergelt and P. Widmayer. Spatial data structures: Concepts and design choices. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 725–764. Elsevier Science Publishers B.V. North-Holland, Amsterdam, 2000.

[20] J. Robinson. The k-d-b tree: A search structure for large multidimensional dynamic indices. In *Proc. of SIGMOD Conference on Management of Data*, pages 10–18, 1981.

[21] N. Roussopoulos, Y. Kotidis, and M. Roussopoulos. Cubetree: Organization of and bulk incremental updates on the data cube. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 88–99, 1997.

[22] R. T. Snodgrass. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, Boston, Mass., 1995.

[23] R. T. Snodgrass, S. Gomez, and L. E. McKenzie. Aggregates in the temoral query language tquel. *IEEE Trans. on Knowledge and Data Eng.*, 5:826–842, 1993.

[24] Y. Tao, D. Papadias, and J. Zhang. Aggregate processing of planar points. In *Extending Database Technology*, pages 682–700, 2002.

[25] *TIGER/Line*$^{\mathrm{TM}}$ *Files, 1997 Technical Documentation*. Washington, DC, September 1998.

[26] D. E. Vengroff. A transparent parallel I/O environment. In *Proc. DAGS Symposium on Parallel Computation*, 1994.

[27] J. S. Vitter and M. Wang. Approximate computation of multidimensional aggregates of sparse data using wavelets. In *Proc. of ACM SIGMOD Intl. conference on Management of Data*, pages 193–204, 1999.

[28] J. Yang and J. Widom. Incremental computation and maintenance of temporal aggregates. In *Proceedings of the 17th International Conference on Data Engineering*, 2001.

[29] D. Zhang, A. Markowetz, V. Tsotras, D. Gunopulos, and B. Seeger. Efficient computation of temporal aggregates with range predicates. In *Proc. Principles Of Database Systems*, pages 237–245, 2001.