Cache-Oblivious Data Structures for Orthogonal Range Searching

Pankaj K. Agarwal* Lars Arge† Andrew Danner‡ Bryan Holland-Minkley§

Department of Computer Science
Duke University

{pankaj,large,adanner,bhm}@cs.duke.edu

ABSTRACT

We develop cache-oblivious data structures for orthogonal range searching, the problem of finding all T points in a set of N points in $\mathbb{R}^d$ lying in a query hyper-rectangle. Cache-oblivious data structures are designed to be efficient in arbitrary memory hierarchies.

We describe a dynamic linear-size data structure that answers d-dimensional queries in $O((N/B)^{1-1/d} + T/B)$ memory transfers, where B is the block size of any two levels of a multilevel memory hierarchy. A point can be inserted into or deleted from this data structure in $O(\log_B^2 N)$ memory transfers. We also develop a static structure for the two-dimensional case that answers queries in $O(\log_B N + T/B)$ memory transfers using $O(N \log_2^2 N)$ space. The analysis of the latter structure requires that $B = 2^{2^c}$ for some non-negative integer constant c.

Categories and Subject Descriptors

F.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems

General Terms

Algorithms

Keywords

Cache-oblivious, Orthogonal range searching

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

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

‡Supported in part by the National Science Foundation through grant CCR–9984099.

§Supported in part by the National Science Foundation through grant EIA–0112849.

SoCG'03, June 8–10, 2003, San Diego, California, USA.

1. INTRODUCTION

The memory systems of modern computers are becoming increasingly complex; they consist of a hierarchy of several levels of cache, main memory, and disk. The access times of different levels of memory often vary by orders of magnitude, and to amortize the large access times of memory levels far away from the processor, data is normally transfered between levels in large blocks. Thus, it is important to design algorithms that are sensitive to the architecture of the memory system and have a high degree of locality in their memory-access patterns.

The traditional RAM model of computation assumes a flat memory-system with uniform access time; therefore, RAM model algorithms often exhibit low memory-access locality and are thus inefficient in a hierarchical memory system. Although a lot of work has recently been done on algorithms for a two-level memory model, introduced to model the large difference in the access times of main memory and disks, relatively little work has been done in models of multilevel memory. One reason for this is the many parameters in such models. Very recently the *cache-oblivious* model was introduced as a way of achieving algorithms that are efficient in arbitrary memory hierarchies without the use of complicated multilevel memory models.

In this paper we develop cache-oblivious data structures for orthogonal range searching, the problem of finding all points T in a set of N points in $\mathbb{R}^d$ lying in a query hyper-rectangle.

1.1 Model of Computation

In the two-level *I/O-model* (or *external-memory model*), introduced by Aggarwal and Vitter [4], the memory hierarchy consists of an internal memory (or cache) of size M and an arbitrarily large external memory partitioned into blocks of size B. An *I/O*, or *memory transfer*, transfers one block between the internal and external memory. Computation can only occur on data present in internal memory. The complexity of an algorithm in this model (an external memory algorithm) is measured in terms of the number of memory transfers it performs, as well as the amount of external memory it uses.

In the *cache-oblivious model*, introduced by Frigo et al. [22], algorithms are developed and analyzed in the two-level I/O-model, but they cannot make explicit use of M and B. It is assumed that $M > B^2$ (the *tall cache* assumption) and that

when an algorithm accesses an element that is not stored in cache, the relevant block is automatically transfered into the cache. If the cache is full, an *optimal paging strategy* replaces the *ideal* block in cache based on the future accesses of the algorithm. Because an analysis of a cache-oblivious algorithm in the two-level model holds for any block and main memory size, it holds for *any* level of an arbitrary memory hierarchy [22]. As a consequence, an algorithm that is optimal in the two-level model is optimal on *all* levels of an arbitrary multilevel hierarchy.

1.2 Previous Results

Range searching has been studied extensively in the RAM model. In the planar case, for example, some of the best known structures answer queries in $O(\log N + T\log^{\varepsilon}(2N/T))$ time using linear space and in $O(\log N + T)$ time using $O(N\log^{\varepsilon} N)$ space, respectively [18, 19]. Refer to a recent survey for further results [3].

In the I/O-model, the B-tree [21, 9] supports one-dimensional range queries in $O(\log_B N + T/B)$ memory transfers using linear space. In two dimensions, one has to use $\Theta(N\frac{\log_B N}{\log_B \log_B N})$ space to obtain an $O(\log_B N + T/B)$ query bound [8, 20]. The external range-tree structure obtains these bounds and supports updates in $O(\frac{\log_B^2 N}{\log_B \log_B N})$ memory transfers [8]. If only linear space is used, $O(\sqrt{N/B} + T/B)$ is a lower bound on the number of transfers needed to answer a query, and these bounds are obtained by the kd-B tree [25]. The kd-B tree can be constructed in $O(Sort(N)) = O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os [1, 2], the number of I/Os needed to sort N elements, and using the logarithmic method [15] this leads to an $O(\frac{\log_2 N}{B}\log_{M/B}\frac{N}{B}) = O(\log_B^2 N)$ update algorithm. Refer to recent surveys for further I/O-model and hierarchical memory model results [6, 26].

Frigo et al. [22] developed cache-oblivious algorithms for sorting, Fast Fourier Transform, and matrix multiplication. Subsequently, a number of other results have been obtained in the cache-oblivious model [7, 10, 11, 12, 13, 16, 17, 24], among them several cache-oblivious B-tree structure with $O(\log_B N)$ search and update bounds [11, 12, 13, 17, 24]. Several of these structures can also support one-dimensional range queries in $O(\log_B N + T/B)$ memory transfers [12, 13, 17], but at an increased amortized update cost of $O(\log_B N + \frac{\log_2 N}{B}) = O(\log_B^2 N)$ memory transfers. To our knowledge, no cache-oblivious structures for higher-dimensional orthogonal range searching have been developed.

1.3 Our Results

In this paper we develop cache-oblivious data structures for multidimensional orthogonal range searching. In Section 2 we develop a cache-oblivious version of a kd-tree. This structure answers queries in $O(\sqrt{N/B} + T/B)$ memory transfers using linear space. It supports updates in $O(\frac{\log_2 N}{B}\cdot\log_{M/B} N) = O(\log_B^2 N)$ transfers. The structure can be extended to support d-dimensional range queries in $O((N/B)^{1-1/d} + T/B)$ memory transfers with the same update bound. To develop the structure, we use several new ideas as well as several previously developed cache-oblivious techniques, e.g. the van Emde Boas layout [12] and exponential search trees [11, 5]. To make the structure dynamic, we use the logarithmic method [15] and a new algorithm for cache-obliviously constructing a kd-tree in $O(Sort(N))$ memory transfers. The construction algorithm uses ideas from a recent I/O-model algorithm [1].

In Section 3 we develop a cache-oblivious version of a two-dimensional range tree. The structure answers queries in the optimal $O(\log_B N + T/B)$ memory transfers but uses $O(N\log_2^2 N)$ space. The central part of the structure is an $O(N\log_2 N)$ space structure for answering two-dimensional three-sided range queries in $O(\log_B N + T/B)$ memory transfers. The analysis of these structures requires that $B = 2^{2^c}$ for some nonnegative integer constant c.

2. LINEAR SIZE STRUCTURE

In this section we describe a linear size structure for answering orthogonal range queries. As mentioned, the structure can be viewed as a cache-oblivious version of a kd-tree. In Section 2.1 we describe the kd-tree and a cache-oblivious memory layout (i.e. an assignment of nodes to memory locations) that allows queries to be answered efficiently. In Section 2.2 we show how to construct the structure efficiently. Finally, in Section 2.3 we show how to make the structure dynamic. In this paper we concentrate on the planar case; in the full version of the paper we discuss how the structure can be extended to $\mathbb{R}^d$.

2.1 Cache-Oblivious kd-tree

In this section we assume, for the sake of simplifying the analysis of our structure, that N is of the form 2^{2^c}, for some non-negative integer c. Our structure generalizes to arbitrary N though the notation and analysis become slightly more cumbersome. Details will appear in the full paper.

The kd-tree proposed by Bentley [14] is a binary tree of height $O(\log_2 N)$ with the N points stored in the leaves of the tree. The internal nodes represent a recursive decomposition of the plane by means of axis-orthogonal lines that partition the set of points into two subsets of equal size. On even levels of the tree the dividing lines are horizontal, and on odd levels they are vertical. In this way a rectangular region R_v is naturally associated with each node v, and the nodes on any particular level of the tree partition the plane into disjoint regions. In particular, the regions associated with the leaves represent a partition of the plane into rectangular regions containing one point each. Refer to Figure 1.

2.1.1 Memory layout

The van Emde Boas layout is the standard way of laying out a balanced tree in memory such that a root-leaf path can be traversed efficiently in the cache-obliviously model.

LEMMA 1 (VAN EMDE BOAS LAYOUT [23, 12]). *A binary tree $\mathcal{T}$ of height $O(\log_2 N)$ with N leaves can be laid out in $\Theta(N)$ contiguous memory locations, such that any root-leaf path can be traversed cache-obliviously in $O(\log_B N)$ memory transfers.*

Using the van Emde Boas layout, we define an *exponential* layout similar to a layout described in [11]. In this layout, a balanced binary tree $\mathcal{T}$ with N leaves is recursively decomposed into a set of *components*, which are each laid out using the van Emde Boas layout. More precisely, we define component $\mathcal{C}_0$ to consist of the first $\frac{1}{2}\log_2 N$ levels of $\mathcal{T}$. $\mathcal{C}_0$ contains $\Theta(\sqrt{N})$ nodes and is called an N-component because its root is the root of a tree ($\mathcal{T}$) with N leaves. To

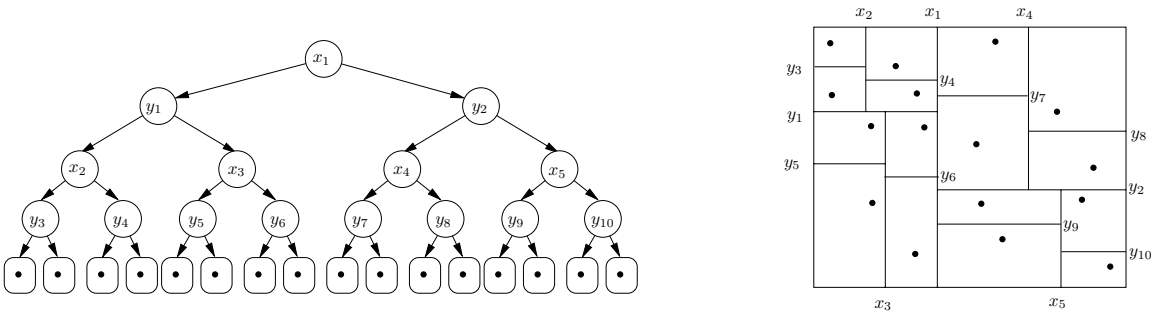


Figure 1: kd-tree and the corresponding partitioning.

obtain the exponential layout of $\mathcal{T}$, we first store $\mathcal{C}_0$ using the van Emde Boas layout (Lemma 1), followed immediately by the recursive layout of the $\sqrt{N}$ subtrees, $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_{\sqrt{N}}$, of size $\sqrt{N}$, beneath $\mathcal{C}_0$ in $\mathcal{T}$, ordered from left to right. The recursion stops when a subtree has 2 leaves; such a 2-component is laid out in 3 consecutive memory locations.

Note how the definition of the exponential layout naturally defines a decomposition of $\mathcal{T}$ into $\log_2 \log_2 N + 2$ *layers*, with layer i consisting of a number of $N^{1/2^{i-1}}$-components. An X-component is of size $\Theta(\sqrt{X})$ and its $\sqrt{X}/2$ leaves are connected to $\sqrt{X}$ $\sqrt{X}$-components. Thus, the root of an X-component is the root of a kd-tree containing X points. Refer to Figure 2.

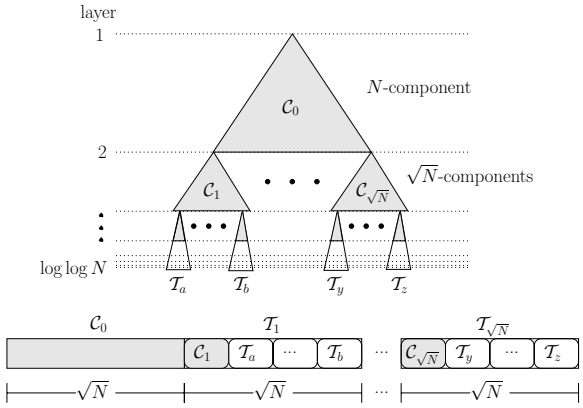


Figure 2: Components and exponential layout.

LEMMA 2. *An exponential layout of a balanced binary tree $\mathcal{T}$ with N leaves uses $\Theta(N)$ contiguous memory locations, and any root-leaf path in $\mathcal{T}$ can be traversed cache-obliviously in $O(\log_B N)$ memory transfers.*

PROOF. The first part of the lemma follows directly from Lemma 1, as the components are laid out contiguously using the van Emde Boas layout.

Any root-leaf path passes through exactly one $N^{1/2^{i-1}}$-component for $1 \leq i \leq \log_2 \log_2 N + 2$. Each X-component is stored in a van Emde Boas layout of size $\sqrt{X}$ and therefore by Lemma 1 a component can be traversed in $\Theta(\log_B \sqrt{X})$ memory transfers. We do not, however, actually use a memory transfer for each of the $O(\log \log N)$ components: consider the traversed X-component with $B \leq X \leq B^2$. This component is of size $O(B)$ and is therefore loaded in $O(1)$ memory transfers. All smaller traversed components are of total size $O(B)$ and stored consecutively in memory. Thus, they can also be traversed in $O(1)$ memory transfers. Therefore, only $O(1)$ memory transfers are used to traverse the last $\log_2 \log_2 B$ components. Thus, the total cost of traversing a root-leaf path is $\sum_{i=1}^{\log_2 \log_B N} O(\log_B N^{1/2^i}) = O(\log_B N)$ memory transfers. □

Remark. Readers familiar with the van Emde Boas layout will have noticed that the exponential layout is indeed identical to that layout. The exponential layout really serves to introduce the notion of X-components, which in Section 2.3 will be the key to making the structure dynamic.

2.1.2 Query

Before describing the query algorithm, we prove a lemma about the exponential layout, that will be crucial in the query analysis.

LEMMA 3. *Consider an exponential layout of a balanced binary tree $\mathcal{T}$ with N leaves, and let v be the root in a subtree $\mathcal{T}_v$ of $\mathcal{T}$ containing B leaves. Any traversal of $\mathcal{T}_v$ can be performed in $O(1)$ memory transfers.*

PROOF. The node v is contained in an X-component with $B \leq X \leq B^2$. Refer to Figure 3. The X-component is of size $O(B)$ and is therefore stored in $O(1)$ blocks. Furthermore, the part of T_v that is not included in the X-component is stored consecutively in memory in $O(1)$ blocks. Therefore, the optimal paging strategy can ensure that any traversal of $\mathcal{T}$ is performed in $O(1)$ memory transfers, simply by loading the $O(1)$ relevant blocks. □

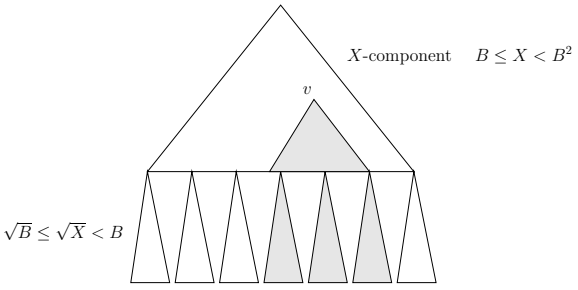


Figure 3: Traversing tree $\mathcal{T}_v$ with B leaves.

Exactly as in the RAM model, we recursively answer a range query Q on a cache-oblivious kd-tree $\mathcal{T}$ starting at the root: at a node v we advance the query to a child v_c of v if Q intersects the region R_{v_c} associated with v_c. At a leaf w we return the point in w if it is contained in Q.

The standard way to bound the number of nodes in $\mathcal{T}$ visited when answering a query Q, or equivalently, the number of nodes v where R_v intersects Q, is to first bound the number of nodes v where R_v intersects a vertical line l. The region R_r associated with the root r is obviously intersected by l, but as the regions associated with its two children represent a subdivision of R_r with a vertical line, only the region R_{r_c} associated with one of these children r_c is intersected. Because the region R_{r_c} is subdivided by a horizontal line, the regions associated with both children of r_c are intersected. As each of these children contain $N/4$ points, the recurrence for the number of regions intersected by l is $Q(N) = 2 + 2Q(N/4) = O(\sqrt{N})$. Similarly, we can show that the number of regions intersected by a horizontal line is $O(\sqrt{N})$. This means that the number of regions intersected by the boundary of Q is $O(\sqrt{N})$. The number of additional nodes visited when answering Q is bounded by $O(T)$, as their corresponding regions are completely contained in Q. Thus, in total $O(\sqrt{N} + T)$ nodes are visited.

If the kd-tree $\mathcal{T}$ is laid out using the exponential layout, we can bound the number of memory transfers used to answer a query by considering the nodes $\log_2 B$ levels above the leaves of $\mathcal{T}$. There are $O(N/B)$ such nodes as the tree $\mathcal{T}_v$ rooted in one such node v contains B leaves. By the same argument as above, the number of these nodes visited by a query is $O(\sqrt{N/B} + T/B)$. Thus, the number of memory transfers used to visit nodes more than $\log_2 B$ levels above the leaves is $O(\sqrt{N/B} + T/B)$. By Lemma 3, $\mathcal{T}_v$ can be traversed in $O(1)$ memory transfers and we obtain the following:

LEMMA 4. *A kd-tree on a set of N points in the plane can be laid out in a linear number of memory cells using the exponential layout, such that a orthogonal range query can be answered cache-obliviously in $O(\sqrt{N/B} + T/B)$ memory transfers, where T is the number of reported points. The leaf containing a given point can be found in $O(\log_B N)$ memory transfers.*

2.2 Kd-tree Construction

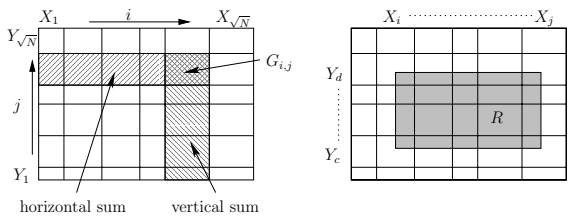
In the RAM model, a kd-tree on N points can be constructed recursively in $O(N \log_2 N)$ time; the root dividing line is found using an $O(N)$ time median algorithm, the points are distributed into two sets according to this line in $O(N)$ time, and the two subtrees are constructed recursively. It is easy to see that the median finding and distribution is performed cache-obliviously in $O(N/B)$ memory transfers [23, 22], and therefore this algorithm is also an $O(\frac{N}{B} \log_2 N)$ cache-oblivious algorithm.

Our algorithm for constructing a kd-tree in $O(Sort(N)) = O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ memory transfers is based on the ideas used in an I/O-model construction algorithm by Agarwal et al. [1]. The algorithm works recursively like the RAM model, but rather than constructing one node in a recursive step, we construct a subtree of height $\log_2 \sqrt{N} = \frac{1}{2} \log_2 N$. In order to construct such a subtree efficiently, we use a number of auxiliary structures. In Section 2.2.1 we describe these structures. In Section 2.2.2 we describe how to construct one node in the tree efficiently using the auxiliary structures. Finally, in Section 2.2.3 we present the details of our algorithm.

2.2.1 Auxiliary structures

To build the top $\log_2 \sqrt{N}$ levels of a kd-tree on a set S of N points, we use four auxiliary structures. Consider dividing the rectangular region containing S into $\sqrt{N}$ vertical slabs $X_1, X_2, \ldots, X_{\sqrt{N}}$ containing $\sqrt{N}$ points each. Our first auxiliary data structure P_v is a list of the points in each of the slabs stored consecutively, with the points in a given slab sorted by y-coordinates. Similarly, to define our second auxiliary structure P_h, we consider dividing S into $\sqrt{N}$ horizontal slabs $Y_1, Y_2, \ldots, Y_{\sqrt{N}}$ and storing the points in each slab consecutively, with the points in a given slab sorted by x-coordinates. The two structures allow us to scan the point in any slab X_i (Y_i) in y-order (x-order) in $O(\sqrt{N}/B)$ memory transfers. We construct P_v and P_h in $O(Sort(N))$ memory transfers using a few sorting steps [22].

The vertical and horizontal slabs divide the rectangular region containing S into a grid of N *basic rectangles*, each of which contains between 0 and $\sqrt{N}$ points. Let $G_{i,j}$ be the basic rectangle defined by the intersection of X_i and Y_j. Let $g_{i,j} = |G_{i,j} \cap S|$ refer to the number of points within $G_{i,j}$. Refer to Figure 4(a). For each basic rectangle $G_{i,j}$, we are interested in knowing the number of points in basic rectangles in the j'th horizontal slab to the left of and including $G_{i,j}$, that is, $\sigma_h(i,j) = \sum_{k=1}^{i} g_{k,j}$ (*horizontal sum*). Our third auxiliary structure, σ_h, is a list of these partial sums for all basic rectangles, ordered such that sums for the same vertical slab are stored consecutively, with the sums for the basic rectangles in a given slab stored in y-order. Similarly, we are interested in knowing the number of points in basic rectangles in the i'th vertical slab below and including $G_{i,j}$, that is, $\sigma_v(i,j) = \sum_{k=1}^{j} g_{i,k}$ (*vertical sum*). Our final auxiliary structure, σ_v, is a list of these partial sums, ordered such that the sums for the same horizontal slab are stored consecutively, with the sums for a given slab ordered by x-order. Refer to Figure 4(a). We can construct σ_h in $O(Sort(N))$ memory transfers by scanning through P_h, computing $\sigma_h(i,j)$ for each basic rectangle, and then sorting all the partial sums first by vertical slab and then by horizontal slab within each vertical slab. We can construct σ_v in a similar way using P_v.



(a) (b)

Figure 4: (a) Slabs, basic rectangles, and horizontal/vertical sums (b) Counting points in a rectangle.

2.2.2 Finding a dividing line

Each node of a kd-tree corresponds to a subdivision of a rectangular region using a vertical or horizontal line. We now describe how to efficiently compute a vertical line l that divides the $K = |R \cap S|$ points in a given rectangular region R in two equal sized sets, assuming that K is already known. We can compute a horizontal dividing line in a similar way.

Let $X_i, X_{i+1}, \ldots, X_j$ and $Y_c, Y_{c+1}, \ldots, Y_d$ be the vertical and horizontal slabs intersected by R, respectively. Refer to Figure 4(b). We start by computing the number of points in $R \cap X_i$ by scanning the contiguous portion of P_v corresponding to X_i in $O(\sqrt{N}/B)$ memory transfers. If we find more than $K/2$ points in R, we know that l is in X_i. In this case we can compute l by selecting the point with rank $K/2$ element among the points in $R \cap X_i$ in $O(\sqrt{N}/B)$ memory transfers using the standard linear time selection algorithm [22, 23]. Otherwise, we continue and compute the number of points in $R \cap X_{i+1}$. We do so by first computing the number of points $\sum_{k=c+1}^{d-1} g_{(i+1),k} = \sigma_v(i+1, d-1) - \sigma_v(i+1, c)$ between (but not including) Y_c and Y_d in X_i by accessing the relevant two partial sums in σ_v. Then we compute and add the number of points from basic rectangles $G_{i+1,c}$ and $G_{i+1,d}$ in R by accessing the two relevant positions of P_h. As above, we determine if l is in X_{i+1}, and if so, compute it in $O(\sqrt{N}/B)$ memory transfers by using selection on $R \cap X_{i+1}$. If l is not in X_{i+1} we continue to expand our search to the right until we find the correct vertical slab.

The only memory transfers we have not already accounted for in the above algorithm are the ones used to access σ_v and P_h when computing the number of points in R within vertical slabs completely spanned by R. We only access the parts of these auxiliary structure corresponding to four horizontal slabs; Y_c and Y_{d-1} for σ_v and Y_c and Y_d for P_h. As the structures are laid out in memory according to horizontal slabs, and from left to right within each such slab, the optimal paging strategy can keep the relevant parts of them in cache between transfers. Since the four slabs contain $O(\sqrt{N})$ elements, we therefore use at most $O(\sqrt{N}/B)$ memory transfers to access σ_v and P_h. Thus, we can compute a single dividing line in $O(\sqrt{N}/B)$ transfers in total.

2.2.3 Construction algorithm

To construct a kd-tree on the set S of N points, we first construct the auxiliary structures defined in Section 2.2.1 using $O(Sort(N))$ memory transfers. Then we construct the top $\frac{1}{2}\log_2 N$ levels of the kd-tree by first constructing the root using the algorithm described in Section 2.2.2 above, and then recursively subdividing the two resulting regions using the same algorithm (and auxiliary structures), until each constructed region contains $\sqrt{N}$ points. We use $O(\sqrt{N}/B)$ memory transfers to construct each of the $\sqrt{N}$ dividing lines, for a total of $\sqrt{N} \cdot O(\sqrt{N}/B) = O(N/B)$ transfers. After constructing the top subtree, we can distribute the N points to the $\sqrt{N}$ regions defined by the leaves of the subtree in $O(Sort(N))$ memory transfers. Details will appear in the full version of the paper. Finally, we recursively construct a kd-tree on the points in each of these regions.

The total cost of building all subtrees with size larger than B^2 is $T(N) = \sqrt{N}T(\sqrt{N}) + Sort(N) = O(Sort(N))$ memory transfers [22]. All subtrees of size smaller than B^2 are constructed in $O(N/B)$ memory transfers in total, since as soon as the number of points in a region becomes smaller than B^2 the optimal paging strategy can hold these points in cache during all recursive calls needed to construct a kd-tree on the points. Thus, in total, we construct the kd-tree on S in $O(Sort(N))$ memory transfers.

Finally, it is easy to lay out the kd-tree in memory according to the exponential layout in $O(Sort(N))$ memory transfers. Details will again appear in the full version of this paper.

LEMMA 5. *A kd-tree on a set of N points in the plane can be constructed and laid out according to the exponential layout cache-obliviously in $O(Sort(N))$ memory transfers.*

2.3 Dynamic Structure

In this section we show how to design a dynamic structure based on the static cache-oblivious kd-tree described in Section 2.1. In Section 2.3.1 we discuss how to support deletions on our cache-oblivious kd-tree using a relaxed version of the exponential layout and partial rebuilding. In Section 2.3.2 we discuss how to support insertions using the logarithmic method.

2.3.1 Semi-dynamic structure

In the RAM model a kd-tree $\mathcal{T}$ can relatively easily be modified to support deletions efficiently using global rebuilding. To delete a point from $\mathcal{T}$ we simply find the relevant leaf w in $O(\log_2 N)$ time and remove it. We then remove w's parent and connect w's grandparent to w's sibling. The resulting tree is no longer a kd-tree but it still answers queries in $O(\sqrt{N} + T)$ time, as the number of nodes corresponding to a rectangle intersected by a vertical line remains $\sqrt{N}$, and as the number of nodes corresponding to a rectangle completely contained in a query Q remains $O(T)$ (because the tree is still binary). To insure that N is proportional to the actual number of points in $\mathcal{T}$, the structure is completely rebuilt after $N/2$ deletions. This adds only an extra $O(\log_2 N)$ time amortized to the deletion bound.

Relaxed exponential layout. The cache-oblivious algorithm for deleting a point from $\mathcal{T}$ is similar to the RAM model algorithm. To preserve data locality, however, we need to carefully maintain the layout of $\mathcal{T}$ in memory.

Recall that, to obtain a static cache-oblivious version of a kd-tree $\mathcal{T}$, we laid out $\mathcal{T}$ in memory using the exponential layout: we decomposed $\mathcal{T}$ into $\log_2 \log_2 N + 2$ layers, with layer i consisting of a number of $N^{1/2^{i-1}}$-components, and laid out each component using the van Emde Boas layout. An X-component was of size $\Theta(\sqrt{X})$ and its root was the root in a kd-tree containing X points (Figure 2). In order to support deletions efficiently, we slightly modify this layout. In the *relaxed exponential layout*, we still decompose $\mathcal{T}$ into $\log_2 \log_2 N + 2$ layers, with layer i consisting of $N^{1/2^{i-1}}$-components. We allow some flexibility, however, in the size and space use of components. More precisely, we impose the following two invariants on the components.

INVARIANT 1. *An X-component in $\mathcal{T}$ is of size $O(\sqrt{X})$ and its root is the root in a tree with between $X/2$ and $2X$ leaves.*

INVARIANT 2. *Let v be a node of $\mathcal{T}$ in an X-component, let $\mathcal{T}_v$ be the subtree rooted at v, and let K be the number of leaves in $\mathcal{T}_v$. The $O(\sqrt{X})$ subtrees of $\mathcal{T}_v$ below the X-component containing v are stored within $4K$ contiguous memory cells.*

The exponential layout obviously fulfills the invariants of the relaxed exponential layout. Furthermore, Lemma 2 and 3 hold for the relaxed exponential layout, because Invariant 1 guarantees that the asymptotic size of the X-components is unchanged, and Invariant 2 guarantees that a subtree with B leaves is still stored in $O(1)$ blocks. Because we can still argue that a query visits $O(\sqrt{N/B}+T/B)$ nodes that are roots in subtrees containing B points, we obtain the following.

LEMMA 6. *A kd-tree on a set of N points in the plane can be laid out in a linear number of memory cells using the relaxed exponential layout, such that an orthogonal range query can be answered cache-obliviously in $O(\sqrt{N/B}+T/B)$ memory transfers, where T is the number of reported points. The leaf containing a given point can be found in $O(\log_B N)$ memory transfers.*

Delete algorithm. To delete a point from a kd-tree laid out using the relaxed exponential layout, we, as in the RAM model algorithm, first find the relevant leaf w and remove it and its parent. This may result in the two invariants being violated. Below we show how to restore them.

First consider Invariant 1. The removal of w can result in this invariant being violated for each of the $O(\log\log N)$ components along the path from the root of $\mathcal{T}$ to w. Let $\mathcal{C}$ be the topmost component where Invariant 1 is violated. Let v be the root of $\mathcal{C}$, and let $\mathcal{T}_v$ be the subtree rooted at v. If $\mathcal{T}_v$ is an X-component then it contains $X/2-1$ points. To restore the invariant, we first collect the $X/2-1$ points in $\mathcal{T}_v$ as well as the $X/2 \leq X' \leq 2X$ points in the subtree $\mathcal{T}_{v'}$, rooted at the sibling v' of v, and destroy $\mathcal{T}_v$, $\mathcal{T}_{v'}$, and their parent y. We then construct a kd-tree $\mathcal{T}'$ on the collected K points. If $X-1 \leq K \leq 2X$, we lay out $\mathcal{T}'$ in the space previously occupied by $\mathcal{T}_v$ and $\mathcal{T}_{v'}$ using the exponential layout, and connect the grandparent of v to the root of $\mathcal{T}'$. In effect we *merge* $\mathcal{T}_v$ and $\mathcal{T}_{v'}$ into $\mathcal{T}'$. Refer to Figure 5. If $K > 2X$ we replace y with the root r of $\mathcal{T}'$, and lay out the two subtrees $\mathcal{T}_u$ and $\mathcal{T}_{u'}$, rooted in the children of r, in the space previously occupied by $\mathcal{T}_v$ and $\mathcal{T}_{v'}$ using the exponential layout. In effect we *share* points between $\mathcal{T}_v$ and $\mathcal{T}_{v'}$. Refer to Figure 6. Since $\mathcal{T}'$ contains between $X-1$ and $2X$ points in the first case, and $\mathcal{T}_u$ and $\mathcal{T}_{u'}$ each contain between X and $5X/4$ points in the second case, we can in both of the above cases lay out the constructed trees such that their roots are roots of an X-component. Thus, Invariant 1 is restored.

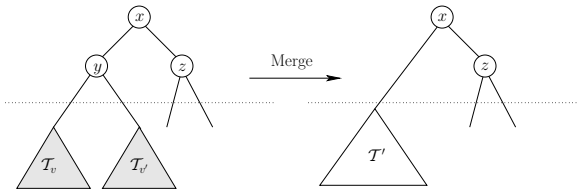


Figure 5: A merge of two trees.

Next consider Invariant 2. The removal of w can result in this invariant being violated in nodes on the path from the root of $\mathcal{T}$ to l. Let v be the topmost node where Invariant 2 is violated, and let K be the number of points in the subtree $\mathcal{T}_v$ rooted at v. If v is in an X-component, the $O(\sqrt{X})$ subtrees $\mathcal{T}_1, \ldots \mathcal{T}_{O(\sqrt{X})}$ of $\mathcal{T}_v$ below the X-component containing v use more than $4K$ contiguous memory cells. To restore the invariant we *compress* all these subtrees. We compress a subtree $\mathcal{T}_i$ containing $|\mathcal{T}_i|$ nodes by traversing $\mathcal{T}_i$ and rewriting the nodes in $|\mathcal{T}_i|$ contiguous memory cells. The compressed layout of $\mathcal{T}_i$ is followed immediately in memory by the compressed layout of $\mathcal{T}_{i+1}$ (effectively "pushing" all the unused space in each subtree past the end of the last subtree). This way the subtrees now use less than $2K$ contiguous memory cells and Invariant 2 is restored.

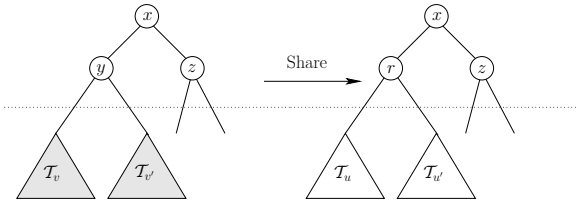


Figure 6: A share between two trees.

Delete analysis. The search for leaf w requires $O(\log_B N)$ memory transfers (Lemma 6).

The cost of restoring Invariant 1 at a node v (using merging or sharing) is dominated by the $O(Sort(X))$ memory transfers (Lemma 5) needed to construct the new trees and X-components. Because the roots of the new X-components are the roots of trees containing at least X points, at least $X/2 = \Theta(X)$ deletes have to be performed below them before the invariant can be violated again. Since a single delete decreases the number of points in the subtrees rooted in the roots of $O(\log\log N)$ components (in $O(\log\log N)$ layers), the total amortized cost of restoring Invariant 1 is $\sum_{i=0}^{\log\log N} O(\frac{1}{B}\log_{M/B} N^{1/2^i}) = O(\frac{1}{B}\log_B N) = O(\log_B N)$ memory transfers.

Restoring Invariant 2 at a node v (using compression) requires $O(K/B)$ memory transfers if v is the root in a subtree $\mathcal{T}_v$ containing K points (Lemma 3). Because $\mathcal{T}_v$ is stored in less than $2K$ contiguous memory locations after the compression, at least $K/2$ deletes have to be performed below v before the invariant can be violated at v again. Since a single delete decreases the number of points below $O(\log N)$ nodes (on each level of the tree), the amortized cost of restoring Invariant 2 is $O(\frac{1}{B}\log_2 N) = O(\log_B N)$ memory transfers.

THEOREM 1. *A semi-dynamic kd-tree on a set of N points in the plane can be laid out in a linear number of memory cells using the relaxed exponential layout, such that a range query can be answered cache-obliviously in $O(\sqrt{N/B}+T/B)$ memory transfers, where T is the number of reported points. Deletes can be supported cache-obliviously in $O(\log_B N)$ memory transfers amortized.*

2.3.2 Fully dynamic structure

We use the logarithmic method [15] to obtain a fully dynamic structure from the semi-dynamic structure.

First consider the case where we only want to support insertions. We maintain a set of $O(\log_2 N)$ static kd-trees (Lemma 4) $\mathcal{D}_0, \mathcal{D}_1, \ldots$ such that $\mathcal{D}_i$ is either empty or has size 2^i. An insertion is performed by finding the first empty structure $\mathcal{D}_i$, discarding all structures $\mathcal{D}_j, j < i$, and building $\mathcal{D}_i$ from the new point and the $\sum_{l=0}^{i-1} 2^l = 2^i - 1$ points

in the discarded structures using $O(Sort(2^i))$ memory transfers (Lemma 5). If we divide this cost between the 2^i points, each of them is charged $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ transfers. Because points never move from higher to lower indexed structures, we charge each point $O(\log_2 N)$ times. Thus the amortized cost of an insertion is $O(\frac{\log_2 N}{B}\log_{M/B}\frac{N}{B}) = O(\log_B^2 N)$ memory transfers.

To answer a query we simply query each of the $O(\log_2 N)$ structures. Querying $\mathcal{D}_i$ normally takes $O(1+\sqrt{2^i/B}+T_i/B)$ transfers, where T_i is the number of reported points. The optimal paging strategy can keep the first $\Theta(\log_2 B^2)$ structures in memory at all times, so the number of transfers used to query these structures is actually $O(1+T/B)$ in total. Thus $O(1+T/B)+\sum_{i=\log_2 B^2}^{\log_2 N} O(\sqrt{2^i/B}+T_i/B) = O(\sqrt{N/B}+T/B)$ is the total query cost.

To handle deletions we use the semi-dynamic cache-oblivious kd-tree (Theorem 1) instead of the static cache-oblivious structures $\mathcal{D}_i$. To delete a point we simply delete it from the relevant $\mathcal{D}_i$ using $O(\log_B N)$ memory transfers. We globally rebuild the entire structure after every $N/2$ deletes, such that the number of structures remains $O(\log_2 N)$. This ensures a range query can still be answered in $O(\sqrt{N/B}+T/B)$ memory transfers. In terms of the logarithmic method, we ignore deletions, that is, we destroy and reconstruct structures $\mathcal{D}_i$ as if no deletes were taking place. This way, points still only move from lower to higher index structures, which ensures that the amortized insertion cost remains the same.

Regarding deleted points as still being present in terms of the logarithmic method also lets us efficiently find the structure $\mathcal{D}_i$ containing a point to be deleted. We maintain a separate cache-oblivious B-tree on the points in the structure. For point p it stores how many points had been inserted since the last global rebuild when p was inserted. Maintenance of this structure adds $O(\log_B N)$ memory transfers to the insertion bound [12]. To find the structure $\mathcal{D}_i$ containing a given point, we query the B-tree using $O(\log_B N)$ transfers. A simple calculation, based on the obtained information and the current number of elements inserted since the last global rebuilding then determines i.

THEOREM 2. *There exists a cache-oblivious data structure for storing a set of N points in the plane using linear space, such that a orthogonal range query can be answered in $O(\sqrt{N/B}+T/B)$ memory transfers, where T is the number of reported points. The structure can be constructed cache-obliviously in $O(Sort(N))$ memory transfers and supports updates in $O(\frac{\log N}{B}\log_{M/B} N) = O(\log_B^2 N)$ memory transfers.*

The results and data structures presented in this section can be extended to higher dimensions. We provide details in the full version of the paper.

THEOREM 3. *There exists a cache-oblivious data structure for storing a set of N points in $\mathbb{R}^d$ using linear space, such that a d-dimensional orthogonal range query can be answered in $O((N/B)^{1-1/d}+T/B)$ memory transfers, where T is the number of reported points. The structure can be constructed cache-obliviously in $O(Sort(N))$ memory transfers and supports updates in $O(\frac{\log N}{B}\log_{M/B} N) = O(\log_B^2 N)$ memory transfers.*

3. QUERY-EFFICIENT STRUCTURE

In this section, we describe our static cache-oblivious range tree structure for answering two-dimensional range queries in $O(\log_B N+T/B)$ memory transfers. The main part of this structure is a cache-oblivious structure for answering three-sided queries presented in Section 3.1. We describe how this structure is used to obtain our cache-oblivious range tree in Section 3.2

3.1 Three-Sided Queries

A three-sided query $Q = [x_l, x_r] \times [y_b, \infty)$ on a set S of N points in the plane asks for all T points in the set $Q \cap S$. In this section we develop a structure for answering such queries in $O(\log_B N + T/B)$ memory transfers using $O(N \log_2 N)$ space. The structure is inspired by the external priority search tree of Arge et al. [8].

3.1.1 Structure

Our structure is recursively defined. It consists of $3\sqrt{N}$ structures of size $O(\sqrt{N})$ and $2\sqrt{N}-1$ recursive structures on $\sqrt{N}$ points each. Thus the structure uses $S(N) \leq 2\sqrt{N}S(\sqrt{N})+O(N) = O(N\log_2 N)$ space.

To define the structure, we first consider dividing the plane into $\sqrt{N}$ vertical *slabs* $X_1, X_2, \ldots, X_{\sqrt{N}}$ containing $\sqrt{N}$ points each. Using these slabs we then define $2\sqrt{N}-1$ *buckets*. A bucket is a rectangular region of the plane that completely spans one or more consecutive slabs and is unbounded in the positive y-direction, like a three-sided query. Each bucket contains $\sqrt{N}$ points and is defined as follows: we start with $\sqrt{N}$ *active* buckets $b_1, b_2, \ldots, b_{\sqrt{N}}$ corresponding to the $\sqrt{N}$ slabs. The x-range of the slabs define a natural linear ordering on these buckets. We then imagine sweeping a horizontal sweep line from $y = -\infty$ to $y = \infty$. Every time the total number of points above the sweep line in two adjacent active buckets, b_i and b_j, in the linear order falls to $\sqrt{N}$, we mark b_i and b_j as *inactive*. Then we construct a new active bucket spanning the slabs spanned by b_i and b_j with a bottom y-boundary equal to the current position of the sweep line. This bucket replaces b_i and b_j in the linear ordering of active buckets. The total number of buckets defined in this way is $2\sqrt{N}-1$, since we start with $\sqrt{N}$ buckets and the number of active buckets decreases by one every time a new bucket is constructed. Note that the procedure defines an *active* y-interval for each bucket in a natural way. Buckets overlap but the set of buckets with active y-intervals containing a given y-value (the buckets active when the sweep line was at that value) are non-overlapping and span all the slabs. This means that the active y-intervals of buckets spanning a given slab are non-overlapping. Refer to Figure 7.

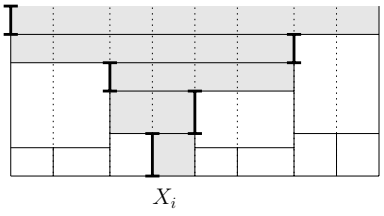


Figure 7: Active intervals of buckets spanning slab X_i.

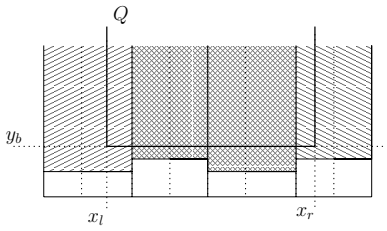


Figure 8: Buckets active at y_b.

After defining the $2\sqrt{N}-1$ buckets, we are now ready to present the three-sided query data structure. It consists of a cache-oblivious B-tree $\mathcal{T}$ on the $\sqrt{N}$ boundaries defining the $\sqrt{N}$ slabs, as well as a cache-oblivious B-tree for each of the $\sqrt{N}$ slabs. The tree $\mathcal{T}_i$ for slab i contains the bottom endpoint of the active y-intervals of the $O(\sqrt{N})$ buckets spanning the slab. For each bucket b_i we also store the $\sqrt{N}$ points in b_i in a list $\mathcal{B}_i$ sorted by y-coordinate. Finally, recursive structures $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_{2\sqrt{N}-1}$ are built on the $\sqrt{N}$ points in each of the $2\sqrt{N}-1$ buckets. The layout of the structure in memory consists of $O(N)$ memory locations containing $\mathcal{T}$, then $\mathcal{T}_1, \ldots, \mathcal{T}_{\sqrt{N}}$, and $\mathcal{B}_1, \ldots, \mathcal{B}_{2\sqrt{N}-1}$, followed by the recursive structures $\mathcal{S}_1, \ldots, \mathcal{S}_{2\sqrt{N}-1}$.

3.1.2 Query

To answer a three-sided query $Q = [x_l, x_r] \times [y_b, \infty)$, we consider the buckets whose active y-interval contains y_b. These buckets are non-overlapping and together they contain all points in Q since they span all slabs and have bottom y-boundary below y_b. We report all points that satisfy Q in each of the buckets with x-range completely between x_l and x_r. At most two other buckets b_l and b_r—the ones containing x_l and x_r—can contain points in Q, and we find these points recursively by advancing the query to $\mathcal{S}_l$ and $\mathcal{S}_r$. Refer to Figure 8.

We find the buckets b_l and b_r that need to be queried recursively and report the points in the completely spanned buckets as follows. We first query $\mathcal{T}$ using $O(\log_B \sqrt{N})$ memory transfers to find the slab X_l containing x_l. Then we query $\mathcal{T}_l$ using another $O(\log_B \sqrt{N})$ memory transfers to find the bucket b_l with active y-interval containing y_b. We can similarly find b_r in $O(\log_B \sqrt{N})$ memory transfers. If b_l spans slabs $X_l, X_{l+1}, \ldots, X_m$ we then query $\mathcal{T}_{m+1}$ with y_b in $O(\log_B \sqrt{N})$ memory transfers to find the active bucket b_i to the right of b_l completely spanned by Q (if it exists). We report the relevant points in b_i by scanning $\mathcal{B}_i$ top-down until we encounter a point not contained in Q. If T' is the number or reported points, a scan of $\mathcal{B}_i$ takes $O(1 + T'/B)$ memory transfers. We continue this procedure for each of the k completely spanned active buckets, using $O(k \log_B \sqrt{N} + T_i/B)$ memory transfers in total. The number of points T_i reported in the scans must be larger than $\lfloor \frac{k}{2} \rfloor \sqrt{N}$ since, by construction, every two adjacent active buckets contain at least $\sqrt{N}$ points above y_b. Thus we spend $O(\log_B \sqrt{N} + \frac{T_i}{B} \cdot (1 + \frac{\log_B \sqrt{N}}{\sqrt{N}/B}))$ memory transfers altogether, not counting the recursive queries.

To analyze the total number of memory transfers used by the query algorithm, note that the algorithm performs at most two queries on each level of the recursion; in the active buckets containing x_l and x_r. Assume that $N = 2^{2^d}$ for some non-negative integer d. We first consider the query cost on all levels containing at least B^2 points. On level $1 \leq i \leq \log_2 \log_B N$ containing $N^{1/2^{i-1}} \geq B^2$ points, the cost is $O(\log_B N^{1/2^i} + \frac{T_i}{B} \cdot (1 + \frac{\log_B N^{1/2^i}}{N^{1/2^i}/B})) = O(\log_B N^{1/2^i} + \frac{T_i}{B})$ transfers. Thus, the total cost over all such levels is $\sum_{i=1}^{\log_2 \log_B N} O(\log_B N^{1/2^i} + \frac{T_i}{B}) = O(\log_B N + T/B)$ transfers. Next consider levels containing less than B^2 points. If we assume that $B = 2^{2^c}$ for some constant non-negative integer c, we know that there will be a level containing precisely $B^2 = 2^{2^{c+1}}$ points, since $N = 2^{2^d}$. The first smaller level thus contains precisely B points, which means that it ($\mathcal{T}$, $\mathcal{T}_1, \ldots, \mathcal{T}_{\sqrt{B}}$, and $\mathcal{B}_1, \ldots, \mathcal{B}_{2\sqrt{B}-1}$) fits in a constant number of memory blocks. Because all structures are stored contiguously in memory, the level can be loaded into cache in $O(1)$ memory transfers. Thus, instead of spending $O(\log_B \sqrt{B} + \frac{T_i}{B} \cdot (1 + \frac{\log_B \sqrt{B}}{\sqrt{B}/B})) = \sqrt{B} \cdot O(T_i/B)$ transfers on this level, the optimal paging strategy ensures that we only spend $O(1)$ transfers. On the next level of recursion a structure contains precisely $\sqrt{B}$ points and thus it and all levels of recursion below it occupies $O(\sqrt{B} \log_2 \sqrt{B}) = O(B)$ space. Thus, the optimal paging strategy can load all relevant lower levels in $O(1)$ memory transfers. In summary, a query uses $O(1)$ transfers on all levels containing less than B^2 points, and the total query costs is therefore $O(\log_B N + T/B)$.

In the above argument, the assumptions on B and N guaranteed that the structure contained recursive levels on exactly B^2 and B points, and no levels with between B^2 and B points. Our complexity argument would not apply if such a level existed, since it would not fit in $O(1)$ blocks and thus the optimal paging strategy could not avoid the $\sqrt{B} \cdot O(T_i/B)$ query cost. In the full version of this paper we show how the assumption on N can easily be removed by adjusting the number of slabs on the highest level of recursion appropriately. Thus we obtain the following:

THEOREM 4. *There exists a cache-oblivious data structure for storing N points in the plane using $O(N \log_2 N)$ space, such that, if $B = 2^{2^c}$ for some nonnegative integer constant c, a three-sided range query can be answered in $O(\log_B N + T/B)$ memory transfers.*

3.2 Cache-Oblivious Range Tree

Using our structure for three-sided queries, we can construct a cache-oblivious range tree structure for general range queries in a standard way. The structure consists of a cache-oblivious B-tree $\mathcal{T}$ on the N points sorted by x-coordinates. With each internal node v we associate two secondary structures for answering three-sided queries on the points stored in the leaves of the subtree rooted at v; one structure for answering queries with the opening to the left and one for answering queries with the opening to the right. The secondary structures on each level of the tree use $O(N \log_2 N)$ space, for a total space usage of $O(N \log_2^2 N)$.

To answer a range query $Q = [x_l, x_r] \times [y_b, y_t]$, we search down $\mathcal{T}$ using $O(\log_B N)$ memory transfers to find the first node v where the left and right x-coordinate of Q are contained in different children of v. Then we query the right opening secondary structure of the left child of v, and the left opening secondary structure of the right child of v, using $O(\log_B N + T/B)$ memory transfers. It is easy to see that this correctly reports all T points in Q.

THEOREM 5. *There exists a cache-oblivious data structure for storing N points in the plane using $O(N \log_2^2 N)$ space, such that, if $B = 2^{2^c}$ for some nonnegative integer constant c, a range query can be answered in $O(\log_B N + T/B)$ memory transfers.*

4. CONCLUSION

In this paper we presented cache-oblivious data structures for multidimensional range searching. While our results represent the first cache-oblivious multidimensional range searching data structures, they also introduce a number of interesting and challenging open problems. These problems include improving the kd-tree update bound, improving the space bound of the range-tree, removing the block size assumption from the range-tree and three-sided query structures, as well as making these structures dynamic. Developing higher-dimensional range-query structures with polylogarithmic query bounds is also a challenging open problem.

References

[1] P. K. Agarwal, L. Arge, O. Procopiuc, and J. S. Vitter. A framework for index bulk loading and dynamization. In *Proc. Annual International Colloquium on Automata, Languages, and Programming*, pages 115–127, 2001.

[2] P. K. Agarwal, L. Arge, O. Procopiuc, and J. S. Vitter. Bkd-tree: A dynamic scalable kd-tree. Manuscript, 2002.

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

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

[5] A. Andersson and M. Thorup. Tight(er) worst-case bounds on dynamic searching and priority queues. In *Proc. ACM Symp. on Theory of Computation*, pages 335–342, 2000.

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

[7] L. Arge, M. Bender, E. Demaine, B. Holland-Minkley, and J. I. Munro. Cache-oblivious priority-queue and graph algorithms. In *Proc. ACM Symp. on Theory of Computation*, pages 268–276, 2002.

[8] L. Arge, V. Samoladas, and J. S. Vitter. On two-dimensional indexability and optimal range search indexing. In *Proc. ACM Symp. Principles of Database Systems*, pages 346–357, 1999.

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

[10] M. Bender, R. Cole, E. Demaine, and M. Farach-Colton. Scanning and traversing: Maintaining data for traversals in memory hierarchy. In *Proc. Annual European Symposium on Algorithms*, pages 152–164, 2002.

[11] M. A. Bender, R. Cole, and R. Raman. Exponential structures for cache-oblivious algorithms. In *Proc. Annual International Colloquium on Automata, Languages, and Programming*, pages 195–207, 2002.

[12] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, pages 339–409, 2000.

[13] M. A. Bender, Z. Duan, J. Iacono, and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 29–38, 2002.

[14] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18:509–517, 1975.

[15] J. L. Bentley. Decomposable searching problems. *Information Processing Letters*, 8(5):244–251, 1979.

[16] G. S. Brodal and R. Fagerberg. Cache oblivious distribution sweeping. In *Proc. Annual International Colloquium on Automata, Languages, and Programming*, pages 426–438, 2002.

[17] G. S. Brodal, R. Fagerberg, and R. Jacob. Cache oblivious search trees via binary trees of small height. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 39–48, 2002.

[18] B. Chazelle. Filtering search: a new approach to query-answering. *SIAM J. Comput.*, 15(3):703–724, 1986.

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

[20] B. Chazelle. Lower bounds for orthogonal range searching: I. the reporting case. *Journal of the ACM*, 37(2):200–212, Apr. 1990.

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

[22] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, pages 285–298, 1999.

[23] H. Prokop. Cache-oblivious algorithms. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, June 1999.

[24] N. Rahman, R. Cole, and R. Raman. Optimized predecessor data structures for internal memory. In *Proc. Workshop on Algorithm Engineering, LNCS 2141*, pages 67–78, 2001.

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

[26] J. S. Vitter. External memory algorithms and data structures: Dealing with MASSIVE data. *ACM Computing Surveys*, 33(2):209–271, 2001.