

# Cache-Oblivious R-Trees

Lars Arge<sup>\*</sup>  
BRICS, Department of  
Computer Science, University  
of Aarhus  
IT-Parken, Aabogade 34,  
DK-8200 Aarhus N, Denmark.  
large@daimi.au.dk

Mark de Berg<sup>†</sup>  
Department of Computing  
Science, TU Eindhoven  
P.O. Box 513, 5600 MB  
Eindhoven, the Netherlands  
mdberg@win.tue.nl

Herman Haverkort<sup>‡</sup>  
BRICS, Department of  
Computer Science, University  
of Aarhus  
IT-Parken, Aabogade 34,  
DK-8200 Aarhus N, Denmark.  
cs.herman@haverkort.net

## ABSTRACT

We develop a cache-oblivious data structure for storing a set  $S$  of  $N$  axis-aligned rectangles in the plane, such that all rectangles in  $S$  intersecting a query rectangle or point can be found efficiently. Our structure is an axis-aligned bounding-box hierarchy and as such it is the first cache-oblivious R-tree with provable performance guarantees. If no point in the plane is contained in  $B$  or more rectangles in  $S$ , the structure answers a rectangle query using  $O(\sqrt{N/B} + T/B)$  memory transfers and a point query using  $O((N/B)^\epsilon)$  memory transfers for any  $\epsilon > 0$ , where  $B$  is the block size of memory transfers between any two levels of a multilevel memory hierarchy. We also develop a variant of our structure that achieves the same performance on input sets with arbitrary overlap among the rectangles. The rectangle query bound matches the bound of the best known linear-space cache-aware structure.

## Categories and Subject Descriptors

E.1 [Data]: Data structures; F.2 [Theory of computation]: Analysis of algorithms and problem complexity

<sup>\*</sup>Supported in part by the US National Science Foundation through RI grant EIA-9972879, CAREER grant CCR-9984099, ITR grant EIA-0112849, and U.S.-Germany Cooperative Research Program grant INT-0129182, by the US Army Research Office through grant W911NF-04-1-0278, and by a Ole Rømer Scholarship from the Danish National Science Research Council. Part of this work was done while the author was at Duke University.

<sup>†</sup>Supported by the Netherlands' Organisation for Scientific Research (NWO) under project no. 639.023.301.

<sup>‡</sup>Supported by a grant from the Danish National Science Research Council. Part of this work was done while the author was at Karlsruhe University, supported by the European Commission, FET open project DELIS (IST-001907).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SCG'05, June 6-8, 2005, Pisa, Italy.

Copyright 2005 ACM 1-58113-991-8/05/0006 ...\$5.00.

## General Terms

Theory, Algorithms.

## Keywords

I/O-efficiency. Cache-oblivious data structures. Geometric data structures. R-trees.

## 1. INTRODUCTION

Traditional algorithms research assumes that a computer system consists of a processor (CPU) with an unbounded amount of memory. This is motivated by the fact that most programming languages are based on a memory model with one uniform and very large address space. Referencing memory (to read the value of a variable, or to write a new value of a variable) is considered an elementary operation, just like CPU operations such as adding two numbers; the complexity of an algorithm is then measured in terms of the number of elementary operations performed by the algorithm.

In reality, however, the memory systems of modern computers 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; for example, accessing disk can be 100,000 times slower than accessing main memory. To amortize the large access times of memory levels far away from the processor, data is normally transferred 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. Moreover, the complexity analysis should take the number of blocks transfers between the memory levels into account, as these often have much more impact on the actual running time than the CPU time.

In the past decade, a lot of work has been done on algorithms for a two-level memory model, which was introduced to model the large difference in the access times of main memory and disks. However, relatively little work has been done for multilevel memory models. One reason for this is the many parameters in such models. Recently, Frigo et al. [19] introduced the so-called *cache-oblivious* model. Algorithms that are efficient in this elegant two-level model are, under some mild assumptions, automatically efficient in arbitrary memory hierarchies.

In this paper we develop cache-oblivious data structures for storing a set  $S$  of  $N$  axis-aligned rectangles in the plane, such that all rectangles in  $S$  intersecting a query rectangle  $Q$  can be found efficiently. We refer to such queries as *rect-*

*angle queries* and to the special type of query in which  $Q$  is a point as *point queries*. Both types of queries are central to many applications and have been widely studied in several areas, including computational geometry, computer graphics, spatial databases, and GIS. Previously, no cache-oblivious structures were known for this problem.

**Model of computation.** In their seminal paper, Aggarwal and Vitter [5] introduced the *external-memory model* (or *I/O-model*). The memory in this model has two levels: an internal memory, or *cache*, of size  $M$  and an arbitrarily large external memory. In external memory, data is stored in *blocks* of size  $B$ . Whenever an algorithm wants to compute on data not present in internal memory, the block(s) containing those data have to be read from external memory. Writing data to external memory is also done in blocks. The complexity of an algorithm in this model is measured in terms of the number of I/O's—blocks read from and written to external memory—it performs, as well as the amount of external memory it uses.

In the *cache-oblivious model*, introduced by Frigo et al. [19], algorithms are developed and analyzed in the two-level I/O-model, but they cannot make explicit use of  $M$  and  $B$ . Thus, contrary to the I/O-model, one cannot specify that certain data has to be placed together in one block, since the block size is unknown to the algorithm. The only control the algorithm has over the data placement is the following: data written to external memory in a certain order will be grouped into blocks in the same order. It is assumed that  $M \geq B^2$ —the *tall-cache* assumption—and that when an algorithm accesses a data element not stored in cache, the relevant block is automatically transferred into the cache in a *memory transfer*. If the cache is full, and another block has to be evicted and written back to external memory, it is assumed that this is done by an *optimal paging strategy* based on all future accesses.<sup>1</sup> The analysis of a cache-oblivious algorithm is done using the parameters  $B$  and  $M$ . Since the algorithm itself does not use these parameters, however, the analysis holds for any  $B$  and  $M$ . Hence, it holds for *any* level of an arbitrary memory hierarchy [19], and a cache-oblivious algorithm that is optimal in the two-level model is optimal on *all* levels of an arbitrary multilevel hierarchy.

**Previous results.** As mentioned, many data structures have been proposed for answering rectangle queries on a set  $S$  of  $N$  rectangles. See e.g. [4, 20] for surveys. In this paper we are interested in structures that use linear space and in particular in (axis-aligned) bounding-box hierarchies, or *box-trees* for short. Box-trees are trees with  $N$  leaves, each storing a distinct rectangle from  $S$ , where each internal node  $\nu$  stores the bounding box of all rectangles stored in the leaves of the subtree rooted in  $\nu$ . A rectangle (or point) query  $Q$  is answered on such a tree by starting at the root and recursively visiting all nodes whose bounding boxes intersect  $Q$ .

In the internal-memory model, Agarwal et al. [3] developed a box-tree structure, called the *kd-interval tree*, that answers a query  $Q$  in  $O(\sqrt{N} + T)$  time, where  $T$  is the number of reported rectangles [22]. They also proved that

<sup>1</sup>The assumption of an optimal paging strategy is not as unrealistic as it seems, because the well known LRU paging strategy for cache size  $M$  is within a factor two from the optimal paging strategy for a cache size  $M/2$ .

this is optimal for rectangle queries. Furthermore, if  $\sigma$  is the *stabbing number* of  $S$ , that is, the maximum number of rectangles in  $S$  containing any query point  $p$ , the kd-interval tree answers point queries in  $O(\log^2 N)$  time for all values  $\sigma = O(\log N)$ ; as  $\sigma$  increases to  $\Theta(N)$ , the point query time gradually degrades to  $O(\sqrt{N} + T)$ . See [3, 22] for surveys of earlier internal-memory box-tree results.

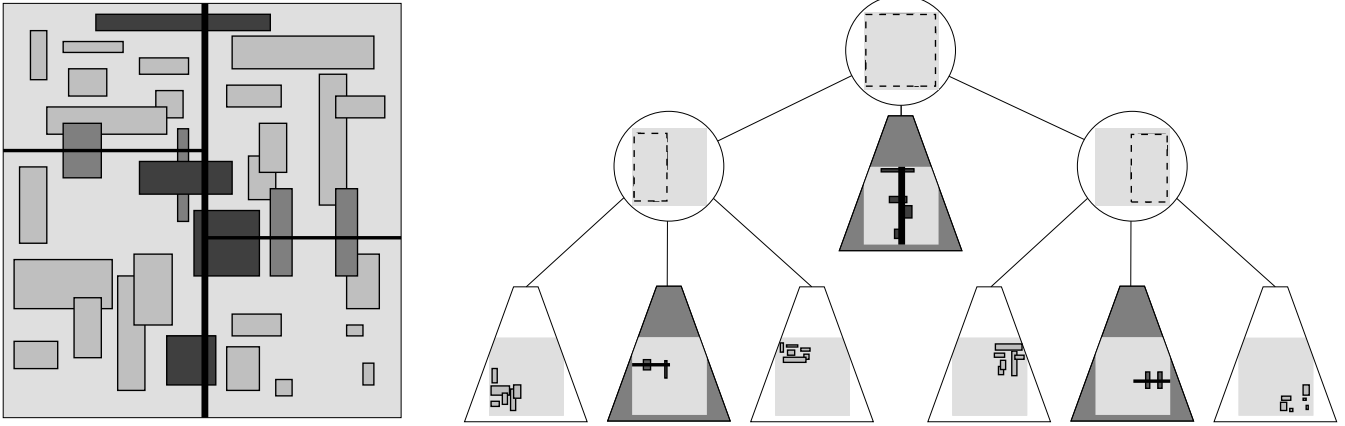
In the external-memory model, trees with fan-out  $\Theta(B)$  (rather than two or constant) are often used to obtain efficient data structures. For example, the ubiquitous B-tree [18, 10] is such a tree; a search (traversal of a root-leaf path) in the B-tree can be performed in  $O(\log_B N)$  memory transfers, since each node can be stored in a constant number of blocks on disk and thus loaded into main memory in a constant number of transfers. Refer e.g. to [6, 29] for general surveys of results in the external-memory model. Box-trees in external memory have been studied extensively, mainly in the database community, under the name of *R-trees* [21]: an R-tree is simply a box-tree with fan-out  $\Theta(B)$  where each node stores the bounding boxes for each of its children. While many R-tree variants have been proposed—see for example [11, 23, 28] or refer to the surveys in [20, 25]—it is only recently that a worst-case-efficient R-tree has been developed. Agarwal et al. [3] described how to modify their kd-interval tree structure to obtain an external-memory structure that answers queries using  $O(\sqrt{N/B} + T)$  memory transfers. Very recently, Arge et al. [8] developed the so-called *priority R-tree* (or *PR-tree*) that improves the bound to  $O(\sqrt{N/B} + T/B)$ , which is optimal for rectangle queries and for point queries on data with high stabbing number [3, 24]. For point queries on data with low stabbing number, no better bounds than the internal-memory results are known.

In the cache-oblivious model, a number of cache-oblivious B-tree structures with  $O(\log_B N)$  search and update bounds have been developed [19, 12, 13, 14, 16, 27]. This may be surprising since the B-tree definition seems to rely crucially on knowledge of  $B$ . The structures all utilize the so-called *Van Emde Boas layout* [19, 26], which is a recursively defined layout of the nodes in a balanced binary tree that allows for a root-leaf path traversal in  $O(\log_B N)$  memory transfers. Other efficient fundamental cache-oblivious data structures such as priority queues [7, 17] and kd- and range-trees [1] have also been developed. However, no efficient structure for rectangle or point queries on a set of rectangles is known. In particular, no cache-oblivious R-tree (box-tree) is known; a Van Emde Boas layout of the kd-interval tree or the PR-tree seems to give a worst-case number of memory transfers for a rectangle query of  $O(\sqrt{N/B} + T)$  at best.

**Our results.** We develop the first cache-oblivious data structures for storing a set  $S$  of  $N$  axis-aligned rectangles in the plane, such that rectangle and point queries can be answered efficiently.

In Section 2 we develop a cache-oblivious R-tree. If  $S$  has stabbing number  $\sigma < B$ , this structure answers a rectangle query using  $O(\sqrt{N/B} + T/B)$  memory transfers and a point query using  $O((N/B)^\epsilon)$  memory transfers.<sup>2</sup> Note that the

<sup>2</sup>Whenever we write bounds of the form  $O((N/B)^\epsilon)$ , we mean that for any  $\epsilon > 0$  we can construct a structure such that the bound is  $O((N/B)^\epsilon)$ . The constant factor in the bound will depend on the choice of  $\epsilon$ .



**Figure 1: The top of a cache-oblivious R-tree.** To the left we see the rectangles stored in this tree; to the right we see the kd-nodes on the top two levels of the tree (represented by circles) and their subtrees (represented by trapezoids). Inside each circle or trapezoid, we see a simplified copy of the input to the left, indicating what is stored in that node or subtree, respectively. Each node stores the bounding box (dashed rectangle) of the rectangles stored below it. Each subtree stores a number of rectangles. The dark trapezoids represent recursively constructed subtrees rooted at kd-nodes. The white trapezoids represent line-based subtrees—for such subtrees, the figure also shows the base line  $\ell$  which intersects all rectangles stored in that subtree and is also the line that is used to split the parent. Note that the base lines are not actually stored in the data structure.

PR-tree, the only cache-aware R-tree that achieves optimal  $\Theta(\sqrt{N/B} + T/B)$  memory transfers for rectangle queries, needs  $\Theta(\sqrt{N/B})$  memory transfers for point queries in the worst case even when the input rectangles are disjoint.

In Section 3 we turn our attention to input sets whose stabbing number can be arbitrarily high. Our goal is to achieve the same performance as for stabbing number less than  $B$ , that is,  $O(\sqrt{N/B} + T/B)$  memory transfers for rectangle queries, and  $O((N/B)^\epsilon + T/B)$  memory transfers for point queries. Unfortunately, the lower bounds of Agarwal et al. [3] show that any R-tree must do  $\Omega(\sqrt{N/B} + T/B)$  memory transfers in the worst case for point queries if the stabbing number can be arbitrarily high. Therefore we slightly relax the definition of our bounding-box hierarchy: we allow a rectangle to be stored at most twice instead of exactly once, and at some nodes we store some additional (constant amount of) information. We show that this small change in the definition makes it possible to obtain  $O(\sqrt{N/B} + T/B)$  and  $O((N/B)^\epsilon + T/B)$  bounds on the number of memory transfers for rectangle and point queries, respectively.

## 2. A CACHE-OBLIVIOUS R-TREE

In this section we describe our cache-oblivious R-tree. We show that if the set of input rectangles  $S$  has stabbing number  $\sigma < B$ , the structure answers rectangle and point queries in  $O(\sqrt{N/B} + T/B)$  and  $O((N/B)^\epsilon)$  memory transfers, respectively. Our structure is inspired by the kd-interval tree of Agarwal et al. [3]. For convenience, we assume that a point lies in a rectangle if it lies in its interior, and that two rectangles intersect if their interiors intersect.

### 2.1 The structure

Our cache-oblivious R-tree is a box-tree whose nodes have degree at most four. We distinguish two main types of nodes,

*kd-nodes* and *line-based nodes*. This distinction only plays a role during preprocessing—it determines how nodes are handled—and during the analysis; they store exactly the same information (namely a bounding box and pointers to their children) and the query algorithm treats them in the same way. The kd-nodes, which include the root of the tree, form a binary tree that resembles a kd-tree. Besides its two kd-children, each kd-node may have a third child which is the root of a *line-based subtree*, that is, a subtree that consists of line-based nodes and stores rectangles that intersect the axis-parallel line that separates the two kd-children.

The structure is defined recursively as follows. Suppose the recursive call is to create the subtree that is rooted at a node  $\nu$ . Parameters for this recursive call will be the set  $S_\nu$  of rectangles to be stored below  $\nu$ , and the type of the node  $\nu$  (a kd-node or a line-based node). If the type is line-based, a *base line*  $\ell$  that intersects all rectangles in  $S_\nu$  is also given. The recursive construction is started with the construction of a tree on the full input set  $S$  rooted at a kd-node.

*The construction of a tree rooted at a kd-node.* We create a node  $\nu$  that stores the bounding box  $R_\nu$  of the current set of rectangles  $S_\nu$ . Next, we partition  $R_\nu$  into two subrectangles  $R_\nu^-$  and  $R_\nu^+$ , thus partitioning  $S_\nu$  into three subsets: a set  $S_\nu^-$  of rectangles lying completely in  $R_\nu^-$ , a set  $S_\nu^+$  of rectangles lying completely in  $R_\nu^+$ , and a set  $S_\nu^\times$  of rectangles intersecting the boundary between  $R_\nu^-$  and  $R_\nu^+$ . Splitting is done using vertical and horizontal splitting lines in a round-robin fashion, like in a kd-tree, so that nodes at even levels at the structure have vertical splitting lines, and nodes at odd levels have horizontal splitting lines. The splitting line  $\ell$  is chosen such that  $|S_\nu^-|, |S_\nu^+| \leq |S_\nu|/2$ . For the subsets  $S_\nu^-$  and  $S_\nu^+$  we recursively construct two subtrees rooted at kd-nodes, which are made children of  $\nu$ . For the subset  $S_\nu^\times$ , we

recursively construct a line-based subtree rooted at a line-based node with base line  $\ell$ , which is also made a child of  $\nu$ . Refer to Figure 1 for an illustration.

### The construction of a tree rooted at a line-based node.

Assume, for the sake of simplicity, that  $\ell$  is a vertical line; a horizontal line can be handled similarly. To obtain the  $O((N/B)^\epsilon)$  bound for point queries, define a parameter  $\delta := (1 - 1/2^\epsilon)^{1/\epsilon}$ . We create a node  $\nu$  that stores the bounding box  $R_\nu$  of  $S_\nu$ . Let  $S_\nu^l$  be the  $(\delta/2)|S_\nu|$  rectangles in  $S_\nu$  that extend farthest to the left and let  $S_\nu^r$  be the  $(\delta/2)|S_\nu|$  rectangles in  $S_\nu \setminus S_\nu^l$  that extend farthest to the right. We partition the remaining set of rectangles  $S_\nu \setminus (S_\nu^l \cup S_\nu^r)$  into three subsets  $S_\nu^-, S_\nu^+$ , and  $S_\nu^\times$ , in the same way a set  $S_\nu$  of a kd-node is partitioned, but always splitting with a horizontal line (that is, a line orthogonal to  $\ell$ ). Note that all rectangles in  $S_\nu^\times$  overlap in the intersection of the splitting line and  $\ell$ . For each of the four sets  $S_\nu^-, S_\nu^+, S_\nu^\times$  and  $S_\nu^{lr} := S_\nu^l \cup S_\nu^r$ , if not empty, we construct a line-based subtree rooted at a line-based node with base line  $\ell$  recursively, making the subtrees' roots, denoted  $\nu^-, \nu^+, \nu^\times$  and  $\nu^{lr}$ , respectively, children of  $\nu$ : refer to Figure 2.

In the analysis, we will call  $\nu^-, \nu^+, \nu^\times$  and  $\nu^{lr}$ , the *lower (regular) child*, the *upper (regular) child*, the *separator child* and the *priority child*, respectively. Correspondingly, the trees rooted at these nodes are called the *lower (regular) subtree*, the *upper (regular) subtree*, the *separator subtree* and the *priority subtree*. The input rectangles stored in these trees—that is, the sets  $S_\nu^-, S_\nu^+, S_\nu^\times$  and  $S_\nu^{lr}$ —are called the *lower rectangles*, the *upper rectangles*, the *separator rectangles*, and the *priority rectangles* of  $\nu$ .

Observe that when constructing a node  $\nu$  in a separator tree, we will never be able to find a horizontal splitting line such that both the upper and the lower regular subtree of  $\nu$  are non-empty. Nevertheless, each node  $\nu$  in a separator subtree will have a priority subtree containing a fraction  $\delta$  of the rectangles stored below  $\nu$ , and each of the other subtrees below  $\nu$  will have at most a fraction  $1 - \delta$  of the rectangles stored below  $\nu$ . Thus, the recursive construction of a separator subtree on a set  $S'$  will not go deeper than  $O(\log |S'|)$  levels.

**Memory layout.** Even though our cache-oblivious R-tree is defined in terms of kd-nodes and line-based nodes, it is simply a tree of bounded height with nodes of degree at most four. We need to specify how to lay this tree out in memory in order to obtain an efficient query algorithm. Unlike most previous cache-oblivious structures we do not use a Van Emde Boas layout, but simply lay the tree out in depth-first order: to lay-out a tree  $T_\nu$  rooted in a node  $\nu$ , we define an ordering of the children  $\nu^1, \nu^2, \dots, \nu^c$  of  $\nu$  and lay out the tree such that  $\nu$  is followed by a recursive layout of the tree  $T_{\nu^1}$  rooted in  $\nu^1$ , followed by a recursive layout of  $T_{\nu^2}$ , and so on.

## 2.2 Query algorithm

Like in any other box-tree, a query with a rectangle or a point  $Q$  is answered by recursively visiting all nodes in the tree whose bounding boxes intersect  $Q$ . To obtain a good query bound, we visit these nodes in depth-first order, always following the same order on the children of a node that was used when we laid the structure out in memory. To make sure the depth-first search will not incur too many

memory transfers, we implement it using a stack that contains the nodes still to be visited. We initialize the stack by pushing a pointer to the root of the tree onto the stack. We then do the following:

1. **repeat**
2.   pop a node  $\nu$  from the stack
3.   **if**  $Q$  intersects the bounding box of  $\nu$
4.    **then if**  $\nu$  is a leaf
5.     **then** report it as an answer;
6.     **else** push the children of  $\nu$  onto the stack in the order  $\nu^4, \nu^3, \nu^2, \nu^1$ , with  $\nu^1$  on top
7. **until** the stack is empty.

## 2.3 Analysis

We will now analyze the number of memory transfers needed to answer a rectangle query. Below, we will distinguish three types of nodes that are visited by a query. We will show that the number of memory transfers needed to answer a query is basically determined by the number of nodes of only one type, and then analyze the number of such nodes, first for a single line-based subtree, and then for the full structure.

Let the weight of a node  $\nu$  be the number of rectangles stored in the subtree  $T_\nu$  rooted at that node. A node is called *heavy* if its weight is at least  $B$ . Denote by  $S_\nu$  the set of rectangles stored in  $T_\nu$ . We distinguish the following three types of visited nodes:

*Hot nodes:* heavy nodes  $\nu$  such that at least  $\delta|S_\nu|/2$  rectangles of  $S_\nu$  are reported, and all descendants of such nodes  $\nu$ ; notice that a descendant  $\mu$  of a hot node  $\nu$  is called hot even if  $\mu$  itself is not heavy or does not have any of its rectangles reported.

*Heavy cold nodes:* visited heavy nodes that are not hot.

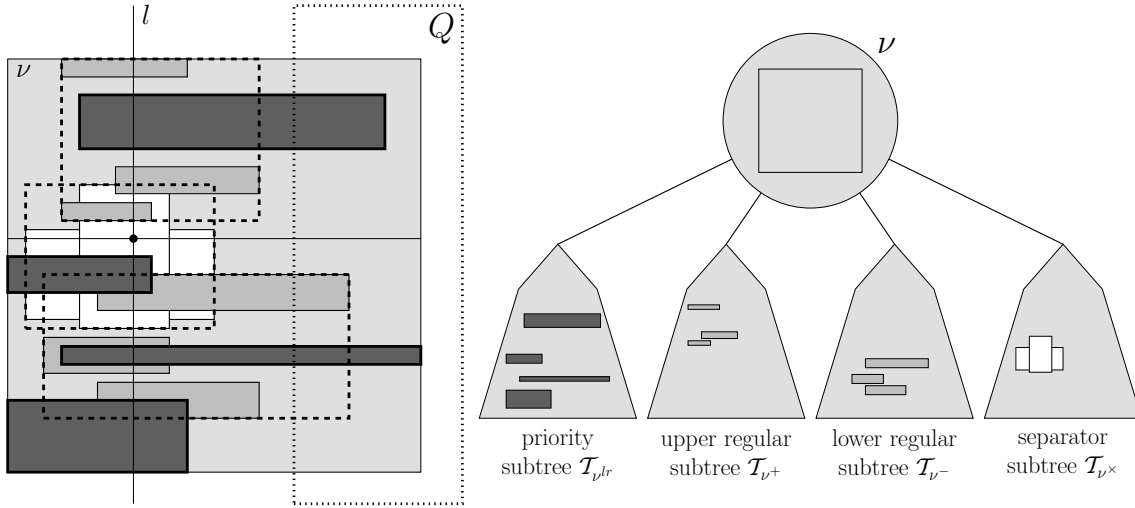
*Light cold nodes:* the remaining nodes, that is, visited nodes of weight less than  $B$  that are not hot.

**LEMMA 1.** *Let  $\mathcal{T}$  be a cache-oblivious R-tree storing  $N$  rectangles. The number of memory transfers needed to answer a query on  $\mathcal{T}$  with a rectangle  $Q$  is  $O(C + T/B)$ , where  $C$  is the number of heavy cold nodes visited and  $T$  is the number of answers reported.*

**Proof:** For each of the three types of nodes, we analyze the number of memory transfers needed to visit them.

*Hot nodes.* Let  $\nu^1, \dots, \nu^t$  be all hot nodes that have no hot node as ancestor. Observe that the query algorithm searches the subtree  $T_{\nu^i}$  rooted at  $\nu^i$  using at most  $O(|S_{\nu^i}|)$  stack operations and reading each node in  $T_{\nu^i}$  at most once. Moreover, the nodes are read in the same depth-first order as the order in which they are stored in memory (possibly skipping children of nodes whose bounding boxes do not intersect  $Q$ ), so that  $O(|S_{\nu^i}|/B)$  memory transfers suffice to read  $T_{\nu^i}$ . The stack operations take at most  $O(|S_{\nu^i}|/B)$  memory transfers as well [7]. Since the trees  $T_{\nu^1}, \dots, T_{\nu^t}$  are disjoint and have at least a fraction  $\delta/2$  of their contents reported as answers, we have  $\sum_{i=1}^t |S_{\nu^i}| \leq 2T/\delta$ , so the total number of memory transfers needed to traverse these trees is  $O(\sum_{i=1}^t (1 + |S_{\nu^i}|/B)) = O(t + T/B)$ . Since each tree  $T_{\nu^i}$  contains at least  $\delta B/2$  answers, we have  $t = O(T/B)$ , and it follows that the total number of memory transfers needed to visit hot nodes is  $O(T/B)$ .

*Heavy cold nodes.* Clearly, at most  $C$  memory transfers are needed to visit these nodes.



**Figure 2:** A line-based node  $\nu$ , with  $|S_\nu| = 12$  rectangles stored below it. To the left we see the bounding box of  $\nu$  (the big square) with the twelve rectangles and the base line  $\ell$  that intersects them all. For the sake of illustration, we set  $\delta := 1/3$ , so that  $\delta|S_\nu| = 4$  rectangles go into the priority subtree: the two rectangles extending farthest to the left and the two rectangles extending farthest to the right. The remaining rectangles are partitioned into three sets with a horizontal line: the upper regular subtree stores the rectangles above the line, the lower regular subtree the rectangles below the line, and the separator subtree the rectangles intersecting the line. — In the proof of Lemma 3, node  $\nu$  is considered to be a middle node, because it lies between the top and the bottom edge of the query range  $Q$ . Since its lower regular child extends far enough to the right to intersect  $Q$ , we know that at least half of its (dark-shaded) priority rectangles intersect  $Q$  as well.

*Light cold nodes.* Let  $\nu$  be a light cold node. Imagine walking back up the tree until we reach a heavy node, that is a node  $\mu$  of weight at least  $B$ . By the definition of hot nodes, if  $\mu$  would be hot,  $\nu$  would have been hot as well. So  $\mu$  must be a heavy cold node.

Let  $\mu'$  be the child of  $\mu$  whose subtree contains  $\nu$ —see Figure 3. Observe that the weight of  $\mu'$  is less than  $B$ , and since the complete structure is stored in depth-first order,  $\mathcal{T}_{\mu'}$  must be stored in  $O(B)$  consecutive memory locations. Thus it can be retrieved in  $O(1)$  memory accesses, which we charge to  $\mu$ . Since there are  $C$  different heavy cold nodes  $\mu$  and each such node can be charged at most four times (once for each child  $\mu'$ ), the number of transfers needed to visit light cold nodes is  $O(C)$ . □

It remains to bound the number of heavy cold nodes. In the analysis we will often use the following lemma.

**LEMMA 2.** *Let  $\varepsilon$  be a constant such that  $\varepsilon > 0$ . Define  $\delta := (1 - 1/2^\varepsilon)^{1/\varepsilon}$ . Suppose  $g(N)$  is a function of  $N$  such that  $g(N) \leq c_g(N/B)^\varepsilon$ , for some constant  $c_g \geq 0$  and all  $N \geq \delta B$ . Let  $f(N)$  be a function of  $N$  such that:*

$$f(N) \leq \max \left\{ f \left( \frac{1-\delta}{2} N \right), g \left( \frac{1-\delta}{2} N \right) \right\} + g \left( \frac{1-\delta}{2} N \right) + \max \{ f(\delta N), g(\delta N) \}$$

for  $N \geq B$ , where  $f(N) = 0$  for  $N < B$ ; then there is a constant  $c_f > c_g$  such that  $f(N) \leq c_f(N/B)^\varepsilon$  for all  $N \geq 0$ .

**Proof:** Let  $c := ((1 - \delta)/2)^\varepsilon + \delta^\varepsilon$ . Note that  $c > 0$  and  $c < 1/2^\varepsilon + \delta^\varepsilon = 1$ . We claim that for  $c_f := c_g/(1 - c) > c_g$ , we have  $f(N) \leq c_f(N/B)^\varepsilon$ . We prove this by induction. The base case  $f(N) = 0 \leq c_f(N/B)^\varepsilon$  for  $0 \leq N < B$  is obviously true. Now suppose that  $f(N') \leq c_f(N'/B)^\varepsilon$  for all  $0 \leq N' < N$ , where  $N \geq B$ . Then we have:

$$\begin{aligned} f(N) &\leq \max \{c_f, c_g\} \left( \frac{(1-\delta)N}{2B} \right)^\varepsilon + c_g \left( \frac{(1-\delta)N}{2B} \right)^\varepsilon + \max \{c_f, c_g\} \left( \frac{\delta N}{B} \right)^\varepsilon \\ &< (c_f c + c_g)(N/B)^\varepsilon \\ &= c_f(N/B)^\varepsilon, \end{aligned}$$

which proves our claim. □

We will now proceed to bound the number of heavy cold nodes. Observe that the nodes we visit are the nodes whose bounding boxes intersect  $Q$ , and the children of such nodes. For convenience, from now on, we consider a node *visited* only if its own bounding box intersects  $Q$ . To include the children of such nodes in the count, we would have to multiply all counts by a constant.

We start by analyzing the number of heavy cold nodes visited when querying a line-based subtree.

LEMMA 3. Let  $\mathcal{T}$  be a line-based subtree storing  $N$  rectangles, such that no point is contained in  $B$  or more rectangles. Then, for any  $\varepsilon > 0$ , setting the parameter  $\delta$  in the construction to  $(1 - 1/2^\varepsilon)^{1/\varepsilon}$  ensures that a query with a rectangle  $Q$  on  $\mathcal{T}$  visits at most  $O((N/B)^\varepsilon)$  heavy cold nodes.

**Proof:** We assume that the query rectangle  $Q$  lies at least partially to the right of the line  $\ell$  intersecting all rectangles; the case where  $Q$  lies to the left is symmetric. Throughout this proof, we ignore all nodes with weight less than  $B$ , since they cannot have any heavy cold descendants. In particular, we ignore the separator subtrees. Such subtrees contain boxes that overlap in a single point. By assumption, no point is contained in  $B$  or more rectangles, so all nodes in separator subtrees must have weight less than  $B$ .

We distinguish four classes of heavy cold nodes, as illustrated by Figure 4, and analyze them one by one.

*Top nodes:* heavy cold nodes whose bounding boxes intersect (or contain) the top but not the bottom edge of  $Q$ . Let  $V_t(N)$  be the worst-case number of such nodes in a subtree of size at most  $N$ . Since the bounding boxes of the regular children of a node are disjoint, a node can only have one regular subtree, and possibly a priority subtree, whose bounding boxes intersect the top edge. The regular subtree would have size at most  $\frac{1}{2}(1 - \delta)N$ , and the priority subtree at most  $\delta N$ . Therefore we have:

$$V_t(N) \leq V_t\left(\frac{1 - \delta}{2}N\right) + V_t(\delta N) + 1,$$

for  $N \geq B$ , and  $V_t(N) = 0$  for  $N < B$ . By Lemma 2 (with  $f(N) = V_t(N)$  and  $g(N) = 1$ ), this recurrence solves to  $V_t(N) = O((N/B)^\varepsilon)$ .

*Bottom nodes:* heavy cold nodes whose bounding boxes intersect (or contain) the bottom edge, but not the top edge of  $Q$ . The analysis for this class is analogous to the analysis of top nodes.

*Spanning nodes:* heavy cold nodes whose bounding boxes intersect (or contain) both the top and the bottom edge of  $Q$ . Observe that the analysis for the number of top nodes, given above, only considered whether or not bounding boxes of nodes intersect the top edge; possible intersections with the bottom edge were not considered. Hence, nodes that intersect both the top edge and the bottom edge were in fact included in the count. The number of such nodes is therefore  $O((N/B)^\varepsilon)$  as well.

*Middle nodes:* heavy cold nodes whose bounding boxes lie between the lines containing the top and bottom edges of  $Q$ . To bound the number of middle nodes, we will first bound the maximum number  $V_m^m(N)$  of middle nodes that can occur in any subtree of size at most  $N$ , rooted at a middle node. After that, we will use the result to bound the num-

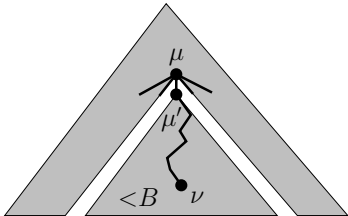


Figure 3: Walking from a light cold node  $\nu$  to a heavy cold node  $\mu$ .

ber of middle nodes  $V_m^t(N)$ ,  $V_m^b(N)$  and  $V_m^s(N)$  in subtrees rooted at top, bottom, and spanning nodes, respectively.

Obviously, a middle node  $\nu$  cannot have any top, bottom or spanning nodes as children. Furthermore, a middle node can have at most one middle node among its children, and if it has such a child, it must be the priority child. This can be seen as follows. If any of the regular children of  $\nu$  has its bounding box extend far enough to the right to intersect  $Q$ , then certainly  $Q$  intersects the  $\delta|S_\nu|/2$  rectangles in  $S_\nu$  that were put in the priority subtree of  $\nu$  because they extended farthest to the right—for an example, see the middle node  $\nu$  in Figure 2. But then  $\nu$  and all its descendants are hot or light, not heavy and cold. So if any visited child of  $\nu$  is a heavy cold node, it must be the priority child of  $\nu$  and it must be a middle node. It follows that all middle descendants of a middle node  $\nu$  are found on a single path of priority nodes from  $\nu$  down into the tree. Therefore the maximum number  $V_m^m(N)$  of middle nodes in a subtree of size  $N$ , rooted at a middle node, is at most  $1 + \lfloor \log_{1/\delta}(N/B) \rfloor$ .

Next we bound the maximum number of middle nodes  $V_m^t(N)$  in subtrees of size  $N$  rooted at top nodes. As mentioned above, one regular child of a top node may be a top node as well, with at most  $(1 - \delta)N/2$  rectangles stored under it. The child could also be a middle node, but never a spanning or a bottom node. The second regular child, if it is heavy and cold, can only be a middle node. As observed before, the priority child could be a top node—or a middle node, with at most  $\delta N$  rectangles stored under it. Therefore, the worst-case number of middle nodes in a subtree of size  $N$ , rooted at a top node is:

$$V_m^t(N) \leq \max\left\{V_m^t\left(\frac{1 - \delta}{2}N\right), V_m^m\left(\frac{1 - \delta}{2}N\right)\right\} + V_m^m\left(\frac{1 - \delta}{2}N\right) + \max\{V_m^t(\delta N), V_m^m(\delta N)\},$$

for  $N \geq B$ , and  $V_m^t(N) = 0$  for  $N < B$ . Again, by Lemma 2 this recurrence solves to  $V_m^t(N) = O((N/B)^\varepsilon)$ .

The worst-case number of middle nodes in a subtree rooted at a bottom node can be bounded by  $V_m^b(N) = O((N/B)^\varepsilon)$  in an analogous manner.

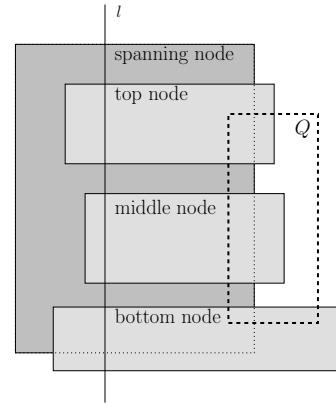


Figure 4: Four classes of heavy cold nodes in a line-based subtree, shown by means of their bounding boxes.

It remains to bound the maximum number of middle nodes  $V_m^s(N)$  in subtrees of size at most  $N$  rooted at spanning nodes. Of the two regular children of a spanning node, we visit neither, one or both of them. If we visit one, the visited child can be a spanning node or any other type of node with at most  $(1 - \delta)N/2$  rectangles stored below it. If we visit both regular children, both must be non-spanning nodes: one of the visited children must be a top or a middle node and the other must be a bottom or a middle node. A spanning node can have any type of priority child. Let us denote the maximum number of middle nodes in subtrees of size  $N$  rooted at *non*-spanning nodes by

$$V_m^n(N) := \max(V_m^t(N), V_m^m(N), V_m^b(N)) = O((N/B)^\varepsilon),$$

The worst-case number of middle nodes in a subtree of size  $N$ , rooted at a spanning node is then at most:

$$\begin{aligned} V_m^s(N) \leq & \max \left\{ V_m^s \left( \frac{1 - \delta}{2} N \right), V_m^n \left( \frac{1 - \delta}{2} N \right) \right\} + \\ & V_m^n \left( \frac{1 - \delta}{2} N \right) + \\ & \max \{ V_m^s(\delta N), V_m^n(\delta N) \}, \end{aligned}$$

for  $N \geq B$ , and  $V_m^s(N) = 0$  for  $N < B$ . Again, by Lemma 2 this solves to  $V_m^s(N) = O((N/B)^\varepsilon)$ .

It follows that the total number of middle nodes visited is  $O(\max(\log_{1/\delta}(N/B), (N/B)^\varepsilon)) = O((N/B)^\varepsilon)$ .

Since no other heavy cold nodes can exist, adding up the bounds for top nodes, bottom nodes, spanning nodes and middle nodes yields the claimed bound of  $O((N/B)^\varepsilon)$ .  $\square$

We can now analyze our total structure.

**THEOREM 4.** *Let  $S$  be a set of  $N$  rectangles in the plane, such that no point is contained in  $B$  or more rectangles. For any  $\varepsilon > 0$ , we can construct a cache-oblivious R-tree on  $S$  such that the number of memory transfers needed to answer a rectangle query is  $O(\sqrt{N/B} + T/B)$ , where  $T$  is the number of reported answers, and the number of memory transfers needed to answer a point query is  $O((N/B)^\varepsilon)$ .*

**Proof:** By Lemma 1, we only need to prove that for rectangle queries we visit  $O(\sqrt{N/B})$  heavy cold nodes, and for point queries we visit  $O((N/B)^\varepsilon)$  heavy cold nodes (note that for point queries,  $T < B$ ).

We will first characterize and analyze the number of heavy cold nodes in the case of *rectangle queries*. Recall that the complete structure is defined in terms of two types of nodes: kd-nodes and line-based nodes. Every kd-node can be associated with a region in the plane, like in a kd-tree. Note that the bounding box of a node  $\nu$  is contained in the region of  $\nu$ . A descendant of a kd-node whose region lies completely inside the query rectangle cannot be heavy and cold: all rectangles in the subtree rooted at that kd-node will be reported as answers, and therefore all nodes in that subtree will be hot or light.

So we only have to count kd-nodes of weight at least  $B$  whose regions contain or intersect an edge of  $Q$ , and heavy cold nodes in line-based subtrees rooted at children of such kd-nodes. We analyze the number of such kd-nodes following the standard kd-tree analysis, as follows. Fix an edge  $e$  of  $Q$ . Let  $E(N)$  be the maximum number of kd-nodes in

a tree storing  $N$  rectangles whose regions intersect  $e$ . Then  $E(N)$  satisfies the recurrence:

$$E(N) \leq 2E(N/4) + 3,$$

for  $N \geq B$ , with  $E(N) = 0$  for  $N < B$ . To account for the number of heavy cold nodes in line-based subtrees rooted at kd-nodes, we extend the recurrence to:

$$E(N) \leq 2E(N/4) + 3 + Q_1(N) + 2Q_1(N/2),$$

where  $Q_1(N)$  is the maximum number of heavy cold nodes visited in a line-based subtree on  $N$  rectangles. We know from Lemma 3 that  $Q_1(N) = O((N/B)^\varepsilon)$ . Hence, if we choose  $\varepsilon < 1/2$ , it follows that  $E(N) = O(\sqrt{N/B})$ .

For *point queries*, we observe that the number of regions  $P(N)$  that contains a certain point satisfies:

$$P(N) \leq P(N/2) + 1,$$

for  $N \geq B$ , with  $P(N) = 0$  for  $N < B$ . To account for the number of heavy cold nodes in line-based subtrees, we extend this recurrence to:

$$P(N) \leq P(N/2) + 1 + Q_1(N),$$

which gives  $P(N) = O((N/B)^\varepsilon)$ .  $\square$

### 3. CACHE-OBVIOUS AUGMENTED R-TREES

We will now describe our cache-oblivious augmented R-tree, and show that it answers rectangle and point queries in the same asymptotic bounds on the number of memory transfers as the cache-oblivious R-tree from the previous section, but regardless of the stabbing number  $\sigma$  of the input. This is achieved by increasing the storage and changing the query algorithm slightly: an input rectangle can now be stored twice instead of exactly once, and some nodes will store a so-called *reference point* to guide the query algorithm.

#### 3.1 The structure

The construction proceeds in exactly the same way as before for kd-nodes, and also for line-based nodes, except that separator subtrees—subtrees storing rectangles containing a common point—are treated in a special way.

*The construction of a tree rooted at a separator node.*

Given a set of rectangles  $S_\nu^\times$  to be stored in a separator subtree below a line-based node  $\nu$ , we construct a *separator node*  $\nu^\times$  that stores a *reference point*. The reference point is the intersection of the base line (which we assume to be vertical) and the splitting line used at  $\nu$  (which is horizontal); this point is known to lie in all rectangles in  $S_\nu^\times$ . We give  $\nu^\times$  two children, which will be the roots of a *lower separator tree* and an *upper separator tree*. Both separator trees will store all rectangles in  $S_\nu^\times$ , but in a different order.

The *upper separator tree* is built exactly like a line-based subtree, except that splitting is done on the basis of only the *top edges* of the rectangles. More precisely, when splitting a set of rectangles  $S_\nu \setminus S_\nu^{lr}$  in a node  $\nu$  of the upper separator tree with a horizontal line, all rectangles with their top edge above the splitting line will be considered to lie above the line and go into the upper regular subtree of  $\nu$ , and all rectangles with their top edge below the splitting line will be

considered to lie below the line and go into the lower regular subtree of  $\nu$ . Rectangles with their top edge on the splitting line are arbitrarily distributed among the lower and the upper regular subtree so that the number of rectangles in the two regular subtrees differs by at most one.

The *lower* separator tree is built in the same way but on the basis of the *bottom edges* rather than the top edges.

Thus the upper and the lower separator tree contain priority subtrees and regular subtrees but no more separator subtrees, as all rectangles go into priority and regular subtrees.

**Memory layout.** The cache-oblivious augmented R-tree is laid out in memory in depth-first order, exactly as the cache-oblivious R-tree from the previous section.

### 3.2 The query algorithm

As with an R-tree, a query with a rectangle or point  $Q$  is answered by recursively visiting all the nodes in the tree whose bounding boxes intersect  $Q$  in depth-first order. When a separator node  $\nu$  is encountered, the query proceeds to only one of the two children. For the result of the query, it does not matter which of the two subtrees is explored, as both subtrees store the same set of rectangles. However, to be able to guarantee good query bounds, we choose the subtree to search as follows: if, in the orthogonal projection on  $\ell$ , the query lies completely below the reference point of  $\nu$ , the query visits the lower separator tree, otherwise it visits the upper separator tree.

### 3.3 Analysis

We take the same approach to analyzing the query time as in the previous section, again distinguishing hot nodes (nodes in subtrees with many answers), heavy cold nodes (other nodes of weight at least  $B$ ) and light cold nodes (the remaining nodes).

LEMMA 5. *Let  $\mathcal{T}$  be a cache-oblivious augmented R-tree that stores  $N$  rectangles. The number of memory transfers needed to answer a rectangle query on  $\mathcal{T}$  is  $O(C + T/B)$ , where  $C$  is the number of heavy cold nodes visited and  $T$  is the number of answers reported.*

**Proof:** The proof is the same as for Lemma 1; the only difference is in the number of memory transfers needed to search a complete subtree. In the proof of Lemma 1, we used the fact that in a cache-oblivious R-tree,  $O(|S_{\nu,i}|/B)$  memory transfers suffice to search any subtree  $\mathcal{T}_{\nu,i}$  of size  $|S_{\nu,i}|$ . In a cache-oblivious augmented R-tree, rectangles may be stored twice, so that traversing the tree may take twice as many memory transfers. The asymptotic bounds remain the same.  $\square$

Now the query analysis again reduces to an analysis of the number of heavy cold nodes visited. We start by analyzing the number of heavy cold nodes visited when querying a separator tree.

LEMMA 6. *Let  $\nu$  be a separator node in a cache-oblivious augmented R-tree. Let  $p$  be the reference point of  $\nu$  and let  $\mathcal{T}_{upper}$  and  $\mathcal{T}_{lower}$  be the upper and lower separator trees of  $\nu$ , respectively. Let  $N$  be the number of rectangles stored under  $\nu$ . A query with a rectangle  $Q$  on the subtree rooted at  $\nu$  visits  $O((N/B)^\epsilon)$  heavy cold nodes.*

**Proof:** We assume that the query rectangle  $Q$  lies at least partially to the right and above the reference point  $p$  of the separator node. Hence, the query will only explore the upper subtree. (The case where  $Q$  lies below the reference point is symmetric — substitute the lower subtree for the upper subtree and bottom edges for top edges — and the case where  $Q$  lies to the left of the reference point is symmetric as well.)

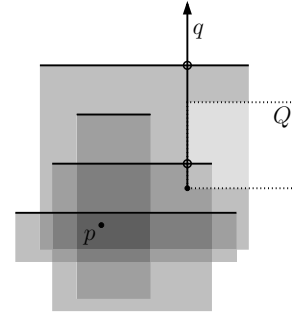


Figure 5: A rectangle intersects  $Q$  iff its top edge intersects  $q$ .

Note that all boxes stored in  $\mathcal{T}_{upper}$  extend below the reference point and therefore cannot lie above  $Q$ . Consequently, the question whether or not a rectangle intersects the query rectangle, reduces to the question whether the top edge of the rectangle lies high enough (that is, above the bottom edge of  $Q$ ) and extends far enough to the right (that is, beyond the left edge of  $Q$ ). In other words, it reduces to the question whether or not the rectangle's top edge intersects the half-line  $q$  that extends upwards from the lower left corner of  $Q$ —see Figure 5. Likewise, the question whether or not a bounding box of a set of rectangles intersects  $Q$ , reduces to the question whether or not the bounding box of the rectangles' top edges intersects  $q$ .

Recall that the rectangles in  $\mathcal{T}_{upper}$  are distributed as if  $\mathcal{T}_{upper}$  were a line-based subtree  $\mathcal{T}'$  on the rectangles' top edges. By the above observations, a bounding box of a node in  $\mathcal{T}_{upper}$  intersects  $Q$  if and only if the bounding box of the corresponding node in  $\mathcal{T}'$  intersects  $q$ . Therefore, the number of heavy cold nodes visited when searching with  $Q$  in  $\mathcal{T}_{upper}$  is exactly the same as when searching with  $q$  in the line-based subtree  $\mathcal{T}'$ . By Lemma 3 the number of heavy cold nodes visited is then  $O((N/B)^\epsilon)$ .  $\square$

We can now analyze the number of heavy cold nodes visited when querying a line-based subtree in an augmented R-tree.

LEMMA 7. *Let  $\mathcal{T}$  be a line-based subtree of an augmented R-tree, and let  $N$  be the number of rectangles stored in  $\mathcal{T}$ . A query with a rectangle  $Q$  on  $\mathcal{T}$  visits  $O((N/B)^\epsilon)$  heavy cold nodes.*

**Proof:** The proof of Lemma 3 already shows that we visit  $O((N/B)^\epsilon)$  heavy cold nodes that are not descendants of separator nodes. Now we have to account for heavy cold nodes in separator trees as well. Again, we distinguish four classes.

*Top nodes:* heavy cold nodes that are descendants of separator nodes whose parents' bounding boxes intersect (or contain) the top, but not the bottom edge of  $Q$ ;

*Bottom nodes:* heavy cold nodes that are descendants of separator nodes whose parents' bounding boxes intersect (or contain) the bottom, but not the top edge of  $Q$ ;

*Spanning nodes:* heavy cold nodes that are descendants of separator nodes whose parents' bounding boxes intersect both the top and the bottom edge of  $Q$ ;

*Middle nodes:* heavy cold nodes that are descendants of separator nodes whose parents' bounding boxes lie between the lines containing the horizontal edges of  $Q$ .

The proof of Lemma 3 goes through verbatim for the first three classes, except that in the recurrence for  $V_i(N)$ , we have to add a term  $Q_0(N)$ , which is the maximum number of heavy cold nodes visited in a separator subtree on  $N$  rectangles. From Lemma 6, we know that  $Q_0(N) = O((N/B)^\epsilon)$ , so we can still apply Lemma 2 and the recurrence still solves to  $V_i(N) = O((N/B)^\epsilon)$ .

It remains to bound the number of middle nodes. In the proof of Lemma 3, we showed that if the bounding box of a node  $\nu$  lies between the lines containing the horizontal edges of  $Q$ , all heavy cold descendants (outside separator subtrees) of  $\nu$  are found on a single path of  $1 + \lceil \log_{1/\delta}(|S_\nu|/B) \rceil$  priority nodes below  $\nu$ . Each of the nodes on that path may have a separator subtree that is visited. Because the weights of the nodes on the path are bounded by a geometrically decreasing series and the number of heavy cold nodes visited in a separator subtree is polynomial in its weight, the largest possible separator subtree dominates. Hence, the total number of heavy cold nodes visited in separator subtrees below a node of weight  $N$  is  $V_m^m(N) := O((N/B)^\epsilon)$ . For the total count of middle nodes, we can now derive the same recurrences as in the proof of Lemma 3, leading to the same  $O((N/B)^\epsilon)$  bound.

Adding up the bounds for heavy cold nodes outside separator subtrees and for top, bottom, spanning and middle nodes in separator subtrees yields the claimed bound of  $O((N/B)^\epsilon)$ .  $\square$

Following the same analysis as for Theorem 4, now using Lemma 5 instead of Lemma 1 and Lemma 7 instead of Lemma 3, we get the following.

**THEOREM 8.** *Let  $S$  be a set of  $N$  rectangles in the plane. For any  $\epsilon > 0$ , we can construct a linear-size cache-oblivious augmented R-tree on  $S$  such that the number of memory transfers needed to answer a rectangle query is  $O(\sqrt{N/B} + T/B)$  and the number of memory transfers needed to answer a point query is  $O((N/B)^\epsilon + T/B)$ , where  $T$  is the number of reported answers.*

## 4. CONCLUSIONS AND DISCUSSION

We developed the first cache-oblivious R-tree with non-trivial performance guarantees. It answers rectangle queries using  $O(\sqrt{N/B} + T/B)$  memory transfers and point queries using  $O((N/B)^\epsilon + T/B)$  memory transfers, provided that the input rectangles have a small stabbing number. Our cache-oblivious augmented R-tree, where rectangles can be stored twice, answers queries in the same bounds but without a stabbing-number assumption.

The structures presented in this paper can easily be constructed in  $O((N/B) \log_2 N)$  memory transfers using a cache-oblivious  $O(N/B)$  selection algorithm [19, 26]. This immediately leads to semi-dynamic structures with an amortized

update bound of  $O((\log_2^2 N)/B) = O(\log_B^2 N)$  using the logarithmic method [15, 9]. It is likely that the construction bounds, and consequently the update bounds, can be improved to  $O((N/B) \log_{M/B}(N/B))$  using an I/O-efficient construction technique due to Agarwal et al. [2]. It remains an open problem to also support deletions; the techniques utilized by Agarwal et al. [1] to obtain a dynamic cache-oblivious kd-tree might prove helpful. Of course it also remains an open problem to obtain a true cache-oblivious R-tree with an  $O(\sqrt{N/B} + T/B)$  rectangle query bound (regardless of the stabbing number of the input rectangles).

## Acknowledgment

We thank Jeff Erickson for inspiration during one of the early discussions that led to the results in this paper.

## 5. REFERENCES

- [1] P. K. Agarwal, L. Arge, A. Danner, and B. Holland-Minkley. Cache-oblivious data structures for orthogonal range searching. In *Proc. ACM Symposium on Computational Geometry*, pages 237–245, 2003.
- [2] P. K. Agarwal, L. Arge, O. Procopiuc, and J. S. Vitter. A framework for index bulk loading and dynamization. In *Proc. International Colloquium on Automata, Languages, and Programming*, pages 115–127, 2001.
- [3] P. K. Agarwal, M. de Berg, J. Gudmundsson, M. Hammar, and H. J. Haverkort. Box-trees and R-trees with near-optimal query time. *Discr. Comput. Geom.* 28: 291–312 (2002).
- [4] 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.
- [5] A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.
- [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 Symposium on Theory of Computation*, pages 268–276, 2002.
- [8] L. Arge, M. de Berg, H. J. Haverkort, and K. Yi. The Priority R-Tree: A Practically Efficient and Worst-Case-Optimal R-Tree. *Symp. of the ACM Special Interest Group on Management of Data (SIGMOD)*, Paris, 2004, pages 347–358.
- [9] L. Arge and J. Vahrenhold. I/O-efficient dynamic planar point location. *Computational Geometry: Theory and Applications*, 29(2):147–162, 2004.
- [10] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.
- [11] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R\*-tree: An efficient and robust access

- method for points and rectangles. In *Proc. SIGMOD International Conference on Management of Data*, pages 322–331, 1990.
- [12] M. A. Bender, R. Cole, and R. Raman. Exponential structures for cache-oblivious algorithms. In *Proc. International Colloquium on Automata, Languages, and Programming*, pages 195–207, 2002.
- [13] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *Proc. IEEE Symposium on Foundations of Computer Science*, pages 339–409, 2000.
- [14] M. A. Bender, Z. Duan, J. Iacono, and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, pages 29–38, 2002.
- [15] J. L. Bentley. Decomposable searching problems. *Information Processing Letters*, 8(5):244–251, 1979.
- [16] G. S. Brodal, R. Fagerberg, and R. Jacob. Cache oblivious search trees via binary trees of small height. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, pages 39–48, 2002.
- [17] G. S. Brodal and R. Fagerberg. Funnel heap - a cache oblivious priority queue. In *Proc. International Symposium on Algorithms and Computation, LNCS 2518*, pages 219–228, 2002.
- [18] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.
- [19] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. IEEE Symposium on Foundations of Computer Science*, pages 285–298, 1999.
- [20] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.
- [21] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. SIGMOD International Conference on Management of Data*, pages 47–57, 1984.
- [22] H. J. Haverkort. Results on Geometric Networks and Data Structures. *PhD thesis*, Utrecht University, 2004.
- [23] I. Kamel and C. Faloutsos. Hilbert R-tree: An improved R-tree using fractals. In *Proc. International Conference on Very Large Databases*, pages 500–509, 1994.
- [24] K. V. R. Kanth and A. K. Singh. Optimal dynamic range searching in non-replicating index structures. In *Proc. International Conference on Database Theory, LNCS 1540*, pages 257–276, 1999.
- [25] Y. Manolopoulos, A. Nanopoulos, A. N. Papadopoulos, and Y. Theodoridis. R-trees have grown everywhere. *Submitted to ACM Computing Surveys*, 2003.
- [26] H. Prokop. Cache-oblivious algorithms. Master’s thesis, Massachusetts Institute of Technology, Cambridge, MA, June 1999.
- [27] 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.
- [28] T. Sellis, N. Roussopoulos, and C. Faloutsos. The  $R^+$ -tree: A dynamic index for multi-dimensional objects. In *Proc. International Conference on Very Large Databases*, pages 507–518, 1987.
- [29] J. S. Vitter. External memory algorithms and data structures: Dealing with MASSIVE data. *ACM Computing Surveys*, 33(2):209–271, 2001.