Approximate range searching in external memory

Please check the document version of this publication:

Approximate Range Searching
in External Memory

Micha Streppel[1,*] and Ke Yi[2,**]

[1] NCIM-Groep, The Netherlands
m.streppel@ncim.nl
[2] Department of Computer Science and Engineering,
Hong Kong University of Science and Technology,
Clear Water Bay, Kowloon, Hong Kong
yike@cse.ust.hk

Abstract. In this paper, we present two linear-size external memory data structures for approximate range searching. Our first structure, the *BAR-B-tree*, stores a set of N points in $\mathbb{R}^d$ and can report all points inside a query range Q by accessing $O(\log_B N + \epsilon^\gamma + k_\epsilon/B)$ disk blocks, where B is the disk block size, $\gamma = 1 - d$ for convex queries and $\gamma = -d$ otherwise, and k_ϵ is the number of points lying within a distance of $\epsilon \cdot \mathrm{diam}(Q)$ to the query range Q. Our second structure, the *object-BAR-B-tree*, is able to store objects of arbitrary shapes of constant complexity and provides similar query guarantees. In addition, both structures also support other types of range searching queries such as range aggregation and nearest-neighbor. Finally, we present I/O-efficient algorithms to build these structures.

1 Introduction

Range searching is one of the most studied topics in computational geometry. In the basic problem, range reporting, we would like to build a data structure on a set of N points in $\mathbb{R}^d$ such that given a query range Q, all points inside Q can be reported efficiently. As the data sets are often massive in modern applications such as spatial databases, GIS, etc., the resulting data structures often have to be stored on disks. Thus, it is important to design efficient *external memory*, or *I/O-efficient* data structures that optimize disk block transfers instead of CPU time. The design of external memory range searching structures, sometimes called *spatial indexes*, has attracted a lot of interest in both the algorithm and database communities in the past decades.

Ideally, one would like the structure to have a linear size and logarithmic query cost. Unfortunately, this cannot be achieved except for very restricted query ranges. In two dimensions, when the queries are half-spaces, linear space

---

and $O(\log_B N + k/B)$ queries can be simultaneously achieved [1], where B is the disk block size, and k is the output size. However, if the queries are axis-parallel rectangles, in order to achieve a query bound of $O(\log_B N + k/B)$, a super-linear space of $\Omega(\frac{N}{B}\frac{\log(N/B)}{\log\log_B N})$ disk blocks is required, and there is also such a structure matching this bound [7]. If linear space is required, the best obtainable query bound is $O((N/B)^\epsilon + k/B)$ for any constant $\epsilon > 0$ [7]. These bounds become much higher in dimensions greater than 2. We refer the reader to the surveys [14,20,5] for other external memory range searching structures.

Given the theoretical hardness of the problem, practitioners often seek heuristic-based structures. Among them the R-tree and its many variants [16] tend to perform well. In addition, R-trees have several appealing features that make them a popular choice in practice. First, an R-tree usually has a small size, typically not much larger than the raw data size. Second, they support arbitrary query ranges, and can store not only points, but also objects of other shapes. Third, besides the basic range reporting query, they also support other kinds of range searching queries, for example nearest-neighbor, point location, aggregation queries, etc. Fourth, they easily generalize to higher dimensions. As a result, the R-trees have received tremendous research attention with many variants proposed, and are heavily used in practice, in spite of the their lack of good performance guarantees. In fact it was shown [3] that in the worst case, a query has to visit $\Omega((N/B)^{1-1/d} + k/B)$ blocks using *any* variant of R-trees built on N points in $\mathbb{R}^d$. This lower bound is reached by a recently developed R-tree variant [6], but the result holds only if both the queries and objects are axis-parallel hypercubes. If the objects are just points in $\mathbb{R}^d$, other practical structures such as the K-D-B tree [19], the quad tree [20], etc., are also often used.

Approximate range searching and the BAR-tree. Given the fact that exact range searching either uses non-linear storage or incurs super-logarithmic query time, it is natural to seek for approximate solutions. The concept of ϵ-*approximate range searching* was first introduced by Arya and Mount [8]. Here one considers, for a parameter $\epsilon > 0$ and a query range Q of constant complexity, the ϵ-extended query range Q_ϵ, which is the locus of points lying at distance at most $\epsilon \cdot \mathrm{diam}(Q)$ from Q, where $\mathrm{diam}(Q)$ is the diameter of Q. For a point set P of N points in $\mathbb{R}^d$, the following approximate range searching queries can be defined:

(Q1) *Range reporting:* Return a set P^* such that $P \cap Q \subseteq P^* \subseteq P \cap Q_\epsilon$.

(Q2) *Range aggregation:* Supposing each point $p \in P$ is associated with a weight $\omega(p) \in \mathbb{R}$, compute $\bigoplus_{p \in P^*} \omega(p)$ for some P^* such that $P \cap Q \subseteq P^* \subseteq P \cap Q_\epsilon$, where $\oplus$ is an associative and commutative operator. We say that the aggregation is *duplicate-insensitive* if $x \oplus x = x$ for any x. For example MAX is a duplicate-insensitive aggregation while $+$ is not.

(Q3) *Nearest-neighbor:* For a query point q, return a point $p \in P$ such that $d(p,q) \leq (1+\epsilon)d(p^*,q)$, where p^* is the true nearest neighbor of q and $d(p,q)$ is the Euclidean distance between p and q.

These problems were first considered by Arya and Mount [8], who proposed the BBD-tree. Later, a similar, but simpler structure, called the BAR-tree, was

proposed by Duncan et al. [13]. Our structures will be based on the BAR-tree, which we describe briefly below.

A BAR-tree [13,12] is a binary tree $\mathcal{T}$ that represents a binary space partition (BSP). We briefly describe the two-dimensional version below; generalization to $\mathbb{R}^d$ is similar. Each data point $p \in P$ is stored at a leaf of $\mathcal{T}$. Each node $u \in \mathcal{T}$ is associated with a region R_u that encloses all the points stored below u. The region associated with the root of $\mathcal{T}$ is the entire $\mathbb{R}^2$. For any internal node u, R_u is partitioned into two sub-regions R_v and R_w where v and w are the children of u. A (Q1) range reporting query Q can be answered using such a BSP by visiting all nodes of $\mathcal{T}$ recursively whose regions intersect Q. To support range aggregation queries, we in addition store at u the aggregate of the weights of all points stored below u. For a (Q2) query Q, we start from the root of $\mathcal{T}$ and traverse the tree while keeping a running aggregate. The only difference here is that we skip an entire subtree at some u if R_u is either completely inside Q_ϵ or outside Q. A (Q3) query can be answered by keeping a priority queue storing all the candidate nodes [12].

In a BAR-tree, all the regions R_u are convex, have aspect ratios bounded by some constant, and the boundary of each R_u consists of a constant number of vertical, horizontal, and diagonal line segments. Duncan et al. [13,12] proved that even under these constraints, using at most two splits, any R_u can be partitioned into 2 or 3 cells, such that the number of points in any cell is at most a constant fraction of the number of points in R_u. Thus the height of the tree can be bounded by $O(\log N)$. Note however that some subtrees in a BAR-tree may not be balanced, since sometimes the first split may have to partition the points in R_u into two subsets with drastically different cardinalities.

A BAR-tree obviously uses linear space, and because of the properties of the regions, the number of nodes visited during a query can be effectively bounded using a packing argument [8]. As a result, it is shown [13] that (Q1) takes $O(\log N + \epsilon^\gamma + k_\epsilon)$ time for any query range Q, where $\gamma = 1 - d$ for convex ranges and $\gamma = -d$ otherwise, and k_ϵ is the number of points inside Q_ϵ.[1] (Q2) can be answered in time $O(\log N + \epsilon^\gamma)$ and (Q3) in time $O(\log N + \epsilon^{1-d} \log(1/\epsilon))$.

The I/O-model and previous work. For the analysis of external memory data structures, the standard *I/O model* by Aggarwal and Vitter [4] is often used. In this model, the memory has a limited size M but any computation in memory is free. In one I/O a disk block consisting of B items are read from or written to the external memory. Only the number of I/Os is considered when analyzing the cost of an algorithm. The size of a data structure is measured in the number of disk blocks it occupies. Many fundamental problems have been solved in the I/O model. For example, sorting N elements takes $\text{sort}(N) = \Theta(N/B \log_{M/B}(N/B))$ I/Os. Please refer to [21,5] for comprehensive surveys on I/O-efficient algorithms and data structures.

---

[1] As noted by Haverkort et al. [15], the actual bound of (Q1) is $O(\log N + \min_\epsilon \{\epsilon^\gamma + k_\epsilon\})$ since Q_ϵ is only used in the analysis and not by the query algorithm, which just uses Q to visit $\mathcal{T}$ and always reports the correct answers $P \cap Q$.

Table 1. Summary of our results for the BAR-B-tree on a set of N points and the object-BAR-B-tree on a set of N objects in $\mathbb{R}^d$ for any fixed d. For (Q1) and (Q2), $\gamma = 1 - d$ if the range is convex, and $-d$ otherwise. The update bound is amortized. $^{(*)}$This bound holds only for duplicate-insensitive aggregations.

	BAR-B-tree	object-BAR-B-tree
Size	$O(N/B)$	$O(\lambda N/B)$
Construction	$O(\text{sort}(N))$	$O(\text{sort}(\lambda N))$
(Q1)	$O(\log_B N + \epsilon^\gamma + k_\epsilon/B)$	$O(\log_B N + \lceil \lambda/B \rceil \epsilon^\gamma + \lambda k_\epsilon/B)$
(Q2)	$O(\log_B N + \epsilon^\gamma)$	$O(\log_B N + \lceil \lambda/B \rceil \epsilon^\gamma)^{(*)}$
(Q3)	$O(\log_B N + \epsilon^{1-d}(1 + \frac{1}{B}\log_{M/B}(1/\epsilon)))$	
		$O(\log_B N + \lceil \lambda/B \rceil \epsilon^{1-d}(1 + \frac{1}{B}\log_{M/B}(1/\epsilon)))$
Update	$O(\log_B N + \frac{1}{B}\log_{M/B}(N/B)\log(N/B))$	-

There has been some work on efficient disk layouts of the BAR-tree. By using standard techniques such as a breadth-first blocking scheme and I/O-efficient priority queues, it is pretty straightforward to lay out a BAR-tree on disk such that (Q2) and (Q3) can be answered with $O(\log_B N + \epsilon^\gamma)$ and $O(\log_B N + \epsilon^{1-d}(1 + \frac{1}{B}\log_{M/B}(1/\epsilon)))$ I/Os, respectively. For a range reporting query (Q1) that has a potentially large output, it is crucial to have an output term of $O(k/B)$ rather than $O(k)$. As typical values of B are on the order of hundreds to thousands, the difference between $O(k/B)$ I/Os and $O(k)$ I/Os can be significant.

In his thesis [12] Duncan gave an I/O-efficient variant of the BAR-tree, which uses a breadth-first blocking scheme. The number of blocks visited for answering a (Q1) query is claimed to be $O(\log_B N + \epsilon^\gamma + k_\epsilon/B)$. However this result relies on the incorrect premise that all blocks contain $\Theta(B)$ nodes. Some leaves may contain a small number of points and the query bound is in fact $O(\log_B N + \epsilon^\gamma + k_\epsilon)$ in the worst case. Agarwal et al. [2] gave a general framework for externalizing and dynamizing *weight-balanced partitioning trees* such as the BAR-tree. Like Duncan [12], they use a breadth-first blocking scheme for storing the BAR-tree on disk. To remove the assumption made by Duncan they group blocks together which contain too few nodes. As a result there is at most one block containing too few nodes. This improvement ensures that the resulted layout only uses $O(N/B)$ disk blocks, but (Q1) query cost is still $O(\log_B N + \epsilon^\gamma + k_\epsilon)$, since the k_ϵ points that need to be visited could spread to $\Omega(k_\epsilon)$ blocks.

Our results. We obtain two main results in this paper. We first give a new blocking scheme for the BAR-tree that yields the first disk-based data structure, the *BAR-B-tree*, which answers all of the aforementioned approximate queries efficiently. In particular, the BAR-B-tree answers an approximate range reporting query (Q1) in the desired $O(\log_B N + \epsilon^\gamma + k_\epsilon/B)$ I/Os[2] , i.e., achieving an $O(\log_B N)$ search term and an $O(k_\epsilon/B)$ output term simultaneously. Such terms are optimal when external memory structures are concerned [5]. Unfortunately it seems difficult to reduce the $O(\epsilon^\gamma)$ term. The bounds for other queries and

---

[2] By the same observation of [15], the actual bound is $O(\log_B N + \min_\epsilon\{\epsilon^\gamma + k_\epsilon/B\})$, but we will not write out $\min_\epsilon$ explicitly.

operations match the previous results on externalizing BAR-trees [12,2]. In addition, we can also construct and update the BAR-B-tree efficiently. Please see Table 1 for the detailed results.

Next, we generalize the BAR-B-tree to the *object-BAR-B-tree*, which stores not just points, but arbitrary spatial objects of constant complexity. The approximate range searching queries (Q1), (Q2), and (Q3) are generalized to objects as follows. Let S be a set of objects in $\mathbb{R}^d$ and Q the query range. For (Q1) and (Q2), we return a subset $S^* \subseteq S$ (or the aggregation $\bigoplus_{o \in S^*} \omega(o)$) where S^* includes all objects in S that intersect Q, does not include any object that does not intersect Q_ϵ, and may optionally include some objects that intersect Q_ϵ but not Q. For (Q3), the definition remains the same with the distance definition between an object o and the query point q being $d(o, q) = \min_{p \in o} d(p, q)$. Our idea is based on range searching data structures for *low-density scenes* [11,10]. The *density* of S is the smallest number λ such that the following holds: any ball b is intersected by at most λ objects $o \in S$ with $\rho(o) \geq \rho(b)$ where $\rho(o)$ denotes the radius of the smallest enclosing ball of o [10]. It is believed that for many realistic inputs, λ is small. For example, if all objects of S are disjoint and *fat* (i.e., have bounded aspect ratio), then λ is a constant. The object-BAR-B-tree exhibits the same performance bounds as the BAR-B-tree if λ is a constant, and the costs grow roughly linearly with λ for all operations except for updates. Please refer to Table 1 for the detailed bounds of various operations.

To summarize, with the BAR-B-tree and the object-BAR-B-tree, we present the first external memory data structures that have all the nice features the R-trees have, and in addition provide provable guarantees, albeit in the approximate sense. In addition, these two structures are not difficult to implement, and we expect them to be fairly practical as well. It would be interesting to compare them with R-trees, K-D-B trees, etc., to see how well they behave in practice.

2 The BAR-B-tree

In this section we describe the BAR-B-tree, an efficient layout for the BAR-tree on disk that achieves all the desired bounds listed in Table 1. We introduce our two-stage blocking scheme in Section 2.1, and analyze its query cost when answering a range searching query (Q1) in Section 2.2. The analysis for (Q2) and (Q3) is similar to that in [2,12], and hence is omitted. Finally we briefly talk about construction and updates in Section 2.3. For the remainder of the paper we assume that $\mathcal{T}$ has at least B nodes, otherwise the problem is trivial.

2.1 The Blocking Scheme

For any node $u \in \mathcal{T}$, let $\mathcal{T}_u$ be the subtree rooted at u, and we define $|\mathcal{T}_u|$, the size of $\mathcal{T}_u$, to be the number of nodes in $\mathcal{T}_u$ (including u). Our blocking scheme consists of two stages. In the first stage the tree is blocked such that for any $u \in \mathcal{T}$, $\mathcal{T}_u$ is stored in $O(\lceil |\mathcal{T}_u|/B \rceil)$ blocks. As we will see, this property will guarantee the $O(k_\epsilon/B)$ term in the query bound. However, a root-to-leaf path

in $\mathcal{T}$ may be covered by $\Theta(\log N)$ such blocks. In the second stage we make sure that any root-to-leaf path can be traversed by accessing $O(\log_B N)$ blocks, leading to the desired bound.

Algorithm 1. Algorithm to construct tree-blocks

Input: a binary tree $\mathcal{T}$
Output: a set of tree-blocks stored on disk

1 initialize $\mathcal{S} := \{\text{root of } \mathcal{T}\}$, and a block $\mathcal{B} := \emptyset$;
2 **while** $\mathcal{S} \neq \emptyset$ **do**
3   remove any node u from $\mathcal{S}$;
4   initialize a queue $\mathcal{Q} := \{u\}$;
5   **while** $\mathcal{Q} \neq \emptyset$ **do**
6   remove the first node v from $\mathcal{Q}$, let v_1, v_2 be v's children;
7   **if** $|\mathcal{T}_v| \leq B$ **then**
8   put $\mathcal{T}_v$ in a new block $\mathcal{B}'$;
9   write $\mathcal{B}'$ to disk;
10   **else if** $|\mathcal{T}_{v_1}| \geq B/2$ and $|\mathcal{T}_{v_2}| \geq B/2$ **then**
11   add v to $\mathcal{B}$;
12   add v_1, v_2 to $\mathcal{Q}$;
13   **else**
14   suppose $|\mathcal{T}_{v_1}| < B/2$;
15   **if** $|\mathcal{B}| + |\mathcal{T}_{v_1}| + 1 \leq B$ **then**
16   add v and $\mathcal{T}_{v_1}$ to $\mathcal{B}$;
17   add v_2 to $\mathcal{Q}$;
18   **else**
19   add v to $\mathcal{S}$;
20   **if** $|\mathcal{B}| = B$ **then**
21   write $\mathcal{B}$ to disk and reset $\mathcal{B} := \emptyset$;
22   add all nodes of $\mathcal{Q}$ to $\mathcal{S}$;
23   set $\mathcal{Q} := \emptyset$;
24   **if** $|\mathcal{B}| \neq \emptyset$ **then**
25   write $\mathcal{B}$ to disk and reset $\mathcal{B} := \emptyset$;


First stage. In the first stage we block the tree into *tree-blocks* that satisfy the property mentioned above. The blocking procedure is detailed in Algorithm 1. We traverse the tree $\mathcal{T}$ in a top-down fashion, and keep in a set $\mathcal{S}$ all nodes u for which a block will be allocated such that u is the topmost node in the block. Initially $\mathcal{S}$ only contains the root of $\mathcal{T}$. For any node $u \in \mathcal{S}$, we find a connected subtree rooted at u to fit in one block using an adapted breadth-first strategy with a queue $\mathcal{Q}$. Throughout the blocking algorithm we maintain the invariant that $|\mathcal{T}_u| \geq B/2$ for any u that is ever added to $\mathcal{S}$ or $\mathcal{Q}$. The invariant is certainly true when the algorithm initializes (line 1).

For a node $u \in \mathcal{S}$, we fill a block with a top portion of $\mathcal{T}_u$ by an adapted breadth-first search (line 4–23). The BFS starts with $\mathcal{Q} = \{u\}$ (line 4), which is consistent with the invariant since u is a node from $\mathcal{S}$. For each node v encountered in the BFS search, we distinguish among the following three cases. (a) If $|\mathcal{T}_v| \leq B$, then we allocate a new block to store the entire $\mathcal{T}_v$ (line 7–9). Note that this block contains at least $B/2$ nodes by the invariant. (b) Let v_1, v_2 be the two children of v. If both $\mathcal{T}_{v_1}$ and $\mathcal{T}_{v_2}$ have more than $B/2$ nodes, then we add v to the block and continue the BFS process (line 10–12). It is safe to add v_1, v_2 to $\mathcal{Q}$ as we have ensured the invariant. (c) Otherwise, it must be the case that one of the subtrees is smaller than $B/2$ nodes while the other one has more than $B/2$ nodes. Without loss of generality we assume $|\mathcal{T}_{v_1}| < B/2$, and then check if $\mathcal{T}_{v_1}$ plus v itself still fits in the current block. If so we put v and the entire $\mathcal{T}_{v_1}$ in the current block, add v_2 to $\mathcal{Q}$ and continue
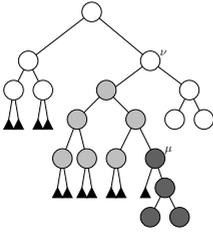
Fig. 1. Three tree-blocks (white, light gray and dark gray) obtained using the blocking scheme for $B = 8$. A black triangles denotes a subtree of size at least $B/2$. The right subtree of ν is placed completely in the white block. The node μ and its right subtree do not fit in the light gray block so a new block must be started at μ.
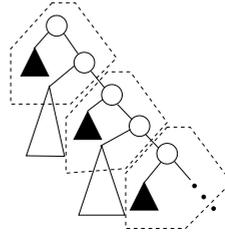
Fig. 2. A bad example for tree-blocks. All black triangles represent subtrees of size $B/2-1$, and all white triangles represent subtrees of sufficiently large size. The rightmost path L will be split into $\Theta(\log N)$ blocks.

the BFS (notice that $|\mathcal{T}_{v_2}| > B/2$); else we put v into $\mathcal{S}$, and will allocate a new block for v (line 14–19). Notice that the second time v is considered by the algorithm, line 19 will never be reached again since with a new empty block, $\mathcal{T}_{v_1}$ and v must be able to fit. Please refer to Figure 1 for an illustration of this blocking algorithm.

Lemma 1. *For any $u \in \mathcal{T}$, the nodes in $\mathcal{T}_u$ are stored in $O(\lceil |\mathcal{T}_u|/B \rceil)$ blocks.*

Proof. First consider the case where u is the topmost node in some block, i.e., u has been added to $\mathcal{S}$. Suppose a tree-block $\mathcal{B}$ contains less than $B/2$ nodes of the tree $\mathcal{T}_u$. There is at least one block below $\mathcal{B}$ since by the invariant the subtree of $\mathcal{T}$ rooted at the topmost node in $\mathcal{B}$ has size at least $B/2$. We claim that at least one block $\mathcal{B}'$ directly below $\mathcal{B}$ contains at least $B/2$ nodes. Now suppose for a contradiction that no block directly below $\mathcal{B}$ contains more than $B/2$ nodes. Let $\mathcal{B}'$ be a block below $\mathcal{B}$ containing less than $B/2$ nodes and let v be the highest node in $\mathcal{B}'$. The node v was not placed in $\mathcal{B}$ either because the size of $\mathcal{T}_v$ is between $B/2$ and B (case (a)) or because the size of one of its subtrees, say $\mathcal{T}_{v_1}$, is less than $B/2$ (case(c)). The former case immediately leads to a contradiction. The latter case also leads to a contradiction since $\mathcal{T}_{v_1}$, together with v, fits in $\mathcal{B}$ and would then have been placed in $\mathcal{B}$. So there must be a block $\mathcal{B}'$ below $\mathcal{B}$ whose size is at least $B/2$. We charge $\mathcal{B}$ to $\mathcal{B}'$. Each block containing at least $B/2$ nodes is charged at most once, namely by the block directly above it. The number of tree-blocks is thus $O(\lceil |\mathcal{T}_u|/B \rceil)$.

Next consider the case where $u \in \mathcal{B}$ but u is not the topmost node in $\mathcal{B}$. Let $u_1, \ldots, u_t$ be the t nodes of $\mathcal{T}_u$ stored immediately below $\mathcal{B}$. By the blocking algorithm's invariant, we have $|\mathcal{T}_{u_i}| \geq B/2$, so $|\mathcal{T}_u| > t \cdot B/2$, or $t < 2|\mathcal{T}_u|/B$. Applying the case above, the number of blocks used to store $\mathcal{T}_u$ is thus $1 + O(\sum_{i=1}^{t}(\lceil |\mathcal{T}_{u_i}|/B \rceil) = O(|\mathcal{T}_u|/B + t + 1) = O(\lceil |\mathcal{T}_u|/B \rceil)$.

The blocked BAR-tree resulting after the first stage might have depth as bad as $\Theta(\log N)$, as illustrated by the following example. Consider a root-to-leaf path L in a BAR-tree $\mathcal{T}$ of length $\Theta(\log N)$. In a BAR-tree, at every other node on L the split may be unbalanced. In particular each unbalanced split at a node v might have one subtree, say $\mathcal{T}_{v_1}$, of size $B/2 - 1$ (see Figure 2). Therefore our algorithm will try to store v and $\mathcal{T}_{v_1}$ in the current block. However this attempt will always fail, and we will end up with storing every two nodes on L in a different block, thus L is split over $\Theta(\log N)$ blocks. In the second stage we introduce *path-blocks*, which ensure that $O(\log_B N)$ blocks have to be accessed in order to visit all nodes on any root-to-leaf path.

Second stage. To identify the places where a path-block has to be introduced, we visit $\mathcal{T}$ in a top-down fashion. For a node u, if $|\mathcal{T}_u| \leq B$ we stop. From Lemma 1 we know that $\mathcal{T}_u$ is already covered by $O(1)$ tree-blocks. Otherwise we consider the top subtree of B nodes of $\mathcal{T}_u$ obtained by a BFS starting from u. We denote this subtree by $\widehat{\mathcal{T}}_u$. We check all root-to-leaf paths in $\widehat{\mathcal{T}}_u$. If there is at least one such path that is covered by more than c tree-blocks for some integer constant $c \geq 2$, then we introduce a path-block that stores $\widehat{\mathcal{T}}_u$. We also remove all nodes of $\widehat{\mathcal{T}}_u$ from the tree-blocks where they are stored. Finally we continue this process recursively with each subtree below $\widehat{\mathcal{T}}_u$.

This completes our two-stage blocking scheme. With the introduction of path-blocks, now we have the following.

Lemma 2. *Any root-to-leaf path in $\mathcal{T}$ can be traversed by accessing $O(\log_B N)$ blocks.*

Proof. Note that for any root-to-leaf path L in any $\widehat{\mathcal{T}}_u$, if it does not reach a leaf of $\mathcal{T}$, then L is at least $\log B$ long. Therefore, we can traverse $\log B$ consecutive nodes in any root-to-leaf path of $\mathcal{T}$ by accessing $O(1)$ blocks, hence the proof.

Since any path-block has at least $B/2$ nodes, it is easy to see that Lemma 1 still holds. In particular, we obtain the desired space bound for the BAR-B-tree.

Theorem 1. *A BAR-B-tree on N points in $\mathbb{R}^d$ takes $O(N/B)$ disk blocks.*

Since our blocking scheme has no redundancy, i.e., each node of $\mathcal{T}$ is stored in only one block, after the two-stage blocking process we can group blocks together such that all of them are at least half-full. So the space utilization of the BAR-B-tree can be at least 50%.

2.2   Analysis of the Range Reporting Query Time

Since no node is stored in multiple blocks we can use the standard query algorithm for BSPs, that is, we start from the root, and visit all nodes u of $\mathcal{T}$ where the region associated with u intersects with the query range Q. The traversal can be performed in either a BFS or DFS manner, with the use of an I/O-efficient stack or queue such that the extra overhead is $O(1)$ I/Os per B nodes. So we only need to bound the number of blocks that store all the visited nodes.

Theorem 2. *A (Q1) range reporting query Q in a BAR-B-tree can be answered by accessing $O(\log_B N + \epsilon^\gamma + k_\epsilon/B)$ blocks.*

Proof. Fix any ϵ. Note that any visited node must be of one of the following two types: (a) the nodes whose regions intersect Q and also the boundary of Q_ϵ, and (b) the nodes whose regions are completely contained in Q_ϵ. Note that some type-(b) nodes may not be visited by the query algorithm.

We start by proving a bound on the number of blocks containing the type-(a) nodes. From [13,12], we know that the number of such nodes is $O(\log N + \epsilon^\gamma)$. The $O(\log N)$ term comes from a constant number of root-to-leaf paths in $\mathcal{T}$. By Lemma 2 these nodes are covered by $O(\log_B N)$ blocks. So in total we need to access $O(\log_B N + \epsilon^\gamma)$ blocks for nodes of type (a).

Next we give a bound on the number of blocks that cover all the type-(b) nodes. These nodes are organized in t disjoint subtrees $\mathcal{T}_{u_1}, \ldots, \mathcal{T}_{u_t}$, such that $R_{u_i} \subseteq Q_\epsilon$ and $R_{p(u_i)} \not\subseteq Q_\epsilon$, where $p(u_i)$ denotes the parent of u_i. Note that since R_{u_i} is contained in Q_ϵ, $R_{p(u_i)}$ must intersect the boundary of Q_ϵ, i.e., a type-(a) node. Each parent $p(u_i)$ has only one child whose region is inside Q_ϵ, since otherwise $R_{p(u_i)}$ would be completely inside Q_ϵ. From [13,12] we know that there are in total $O(\epsilon^\gamma)$ type-(a) nodes who have a child associated with a region completely inside Q_ϵ, hence $t = O(\epsilon^\gamma)$.

Note that the subtree $\mathcal{T}_{u_i}$ stores at least $|\mathcal{T}_{u_i}|/2$ points of P inside Q_ϵ. By Lemma 1, $\mathcal{T}_{u_i}$ is stored in $O(\lceil|\mathcal{T}_{u_i}|/B\rceil)$ blocks. Thus the total number of blocks covering all the t subtrees is $O\left(\sum_{i=1}^{t}\lceil|\mathcal{T}_{u_i}|/B\rceil\right) = O\left(t + \sum_{i=1}^{t}|\mathcal{T}_{u_i}|/B\right) = O(\epsilon^\gamma + k_\epsilon/B)$.

2.3   Construction and Updates

The construction algorithm of the BAR-B-tree can be based on the "grid" technique [2] and uses $O(\text{sort}(N))$ I/Os. Due to space limit we omit the details from this abstract. We can also use the *partial rebuilding* technique [17,2] to handle insertions and deletions for the BAR-B-tree. To insert a point into the BAR-B-tree, we first follow a root-to-leaf path to find the leaf block where the point should be located. According to Lemma 2 this takes $O(\log_B N)$ I/Os. After inserting the point we check the nodes on this path to see if any of them contains too many points. Among all those nodes, we rebuild the whole subtree rooted at the highest one. Deletions can be handled similarly. Using standard analysis, it can be shown that the amortized cost of an update is $O(\log_B N + \frac{1}{B}\log_{M/B}(N/B)\log(N/B))$ I/Os, and we omit the details.

3   Extension to Objects: the Object-BAR-B-tree

In this section we show how to externalize the *object-BAR-tree* [11], for a set S of objects of constant complexity with density λ. We first briefly review the object-BAR-tree below.

The object-BAR-tree is based on the idea of *guarding sets* [9]. For a subset $X \subseteq S$, a set of points G_X is called a λ-*guarding set* (simply called a *guarding set* in the following) of X if the region associated with any leaf in the BAR-tree constructed on G_X intersects at most $O(\lambda)$ objects of X. To build the object-BAR-tree on S, we first for each object $o \in S$ compute a constant number of points, called the *guards* of o, with the property that the guards of any subset X of S form a guarding set for X. Let G be the set of all guards. We build the object-BAR-tree by first constructing a BAR-tree $\mathcal{T}$ on G, with the adaptation that whenever we are going to build a subtree for a region R with a subset $G' \subset G$, we delete all guards from G' whose objects do not intersect R. Then we store at each leaf of $\mathcal{T}$ all objects of S that intersect the region associated with the leaf. Because G is a guarding set of S, each leaf stores $O(\lambda)$ objects. It was shown [11] that a (Q1) query can be answered in time $O(\log N + \lambda(\epsilon^\gamma + k_\epsilon))$ using the object-BAR-tree.

3.1   Building the Object-BAR-B-tree

We first build all the guards with a scan over S. For $\mathbb{R}^2$ we can use the simple construction (Figure 3) of De Berg et al. [11]. For $\mathbb{R}^d$, $d \geq 3$ the construction is more involved and the details can be found in [18].
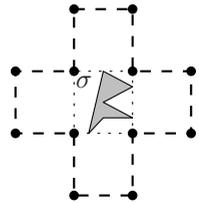
Fig. 3. The guards for an object in $\mathbb{R}^2$

Next we build the BAR-B-tree on the set of all guards G. The adaptation of removing guards during the construction as described above can easily be accommodated in the algorithm, and we can build and lay out the tree $\mathcal{T}$ on disk in $O(\text{sort}(\lambda N))$ I/Os. During the process we can also compute for each leaf v of $\mathcal{T}$, the set of at most $O(\lambda)$ objects that intersect the region R_v. We omit the technical details.

Finally, for each leaf block $\mathcal{B}$ of $\mathcal{T}$, we store all the intersecting objects consecutively on disk. More precisely, consider a block $\mathcal{B}$ and let L be the set of leaves stored in $\mathcal{B}$. The objects intersecting the regions of the nodes in L are stored together in one list as follows. Let $v_1, \cdots, v_{|L|}$ be the leaves in L ordered according to an in-order traversal of $\mathcal{T}$. We first store the objects intersecting R_{v_1}, then the objects intersecting R_{v_2}, etc. Note that an object might be stored more than once in the list. At every leaf v_i we store a pointer to the first and last object in the list intersecting R_{v_i}. Since each leaf has $O(\lambda)$ intersecting objects, each such list occupies $O(\lambda B/B) = O(\lambda)$ disk blocks, so the overall space usage of these lists is $O(\lambda N/B)$ blocks. This completes the description of the object-BAR-B-tree. Note that the object-BAR-B-tree automatically reduces to the BAR-B-tree when all the objects are points.

Theorem 3. *Let S be a set of N objects in $\mathbb{R}^d$ with density λ. An object-BAR-B-tree on S takes $O(\lambda N/B)$ blocks and can be constructed in $O(\text{sort}(\lambda N))$ I/Os.*

3.2   Analysis of the Query Cost

In this section we prove the bounds stated in Table 1 for the object-BAR-B-tree. The query bounds for (Q2) and (Q3) follow from the bounds on (Q2) and (Q3)

for the BAR-B-tree and the fact that for every visited leaf v, we now have to check the $\lceil \lambda/B \rceil$ blocks containing the objects intersecting R_v. However, note that since in an object-BAR-B-tree an object might be stored at several leaves, we can only handle duplicate-insensitive aggregations for (Q2). We are left with proving the query bound on (Q1).

Theorem 4. *Let S be a set of N objects of constant complexity in $\mathbb{R}^d$ with density λ. An object-BAR-B-tree for S answers a (Q1) query Q using $O(\log_B N + \lceil \lambda/B \rceil \epsilon^\gamma + \lambda k_\epsilon/B)$ I/Os, where k_ϵ is the number of objects intersecting Q_ϵ.*

Proof. The query cost of answering a range searching query Q consists of two parts: the cost to visit the nodes of $\mathcal{T}$ and the cost to read the object lists. Since $\mathcal{T}$ is a BAR-B-tree with possible removal of guards during construction, which only reduces the number of nodes, the cost of visiting $\mathcal{T}$ can still be bounded by Theorem 2. So we only concentrate on the cost of reading the object lists of the visited leaves.

Fix any ϵ. Any visited leaf must fall into one of the following two categories: either its region intersects Q and also the boundary of Q_ϵ, or its region is completely contained in Q_ϵ. There are at most ϵ^γ leaves of the former type [12]. For these leaves we can check all objects intersecting their regions using $O(\lceil \lambda/B \rceil)$ I/Os each.

The latter type of leaves can be covered in t disjoint subtrees $\mathcal{T}_{u_1}, \ldots, \mathcal{T}_{u_t}$, such that $R_{u_i} \subseteq Q_\epsilon$ and $R_{p(u_i)} \not\subseteq Q_\epsilon$, where $p(u_i)$ denotes the parent of u_i. Note that there are $O(\epsilon^\gamma)$ such subtrees [12,13]. For any u_i, let $k(u_i)$ denote the number of objects that intersect R_{u_i} and have at least one guard in R_{u_i}. Since $\mathcal{T}_{u_i}$ is a BAR-tree built on the $O(k(u_i))$ guards of these objects (with pruning), and each object has a guard in at least one of the leaves of $\mathcal{T}_{u_i}$, we have $|\mathcal{T}_{u_i}| = O(k(u_i))$. Furthermore, since each object intersecting Q_ϵ has guards in at most a constant number of these subtrees, we have $\sum_{i=1}^t k(u_i) = O(k_\epsilon)$.

Consider some $\mathcal{T}_{u_i}$, and let $v_1, v_2, \ldots$ be the leaves of $\mathcal{T}_{u_i}$ ordered according to an in-order traversal of $\mathcal{T}$. From our blocking algorithm for the BAR-B-tree, we know that these leaves are partitioned into $O(\lceil |\mathcal{T}_{u_i}|/B \rceil)$ pieces, each stored in a block. Since in a block, the objects intersecting consecutive leaves are also stored consecutively in the object list, the total number of I/Os to read these objects is $O(\lceil |\mathcal{T}_{u_i}|/B \rceil + \lambda |\mathcal{T}_{u_i}|/B) = O(\lceil \lambda |\mathcal{T}_{u_i}|/B \rceil)$. Thus, the total number of I/Os for reading the object lists for all the leaves whose regions are completely inside Q_ϵ is

$$O\left(\sum_{i=1}^t \left\lceil \frac{\lambda |\mathcal{T}_{u_i}|}{B} \right\rceil \right) = O\left(t + \sum_{i=1}^t \frac{\lambda |\mathcal{T}_{u_i}|}{B} \right) = O\left(t + \sum_{i=1}^t \frac{\lambda k(u_i)}{B} \right) = O(\epsilon^\gamma + \lambda k_\epsilon/B).$$

Updating the object-BAR-B-tree. The object-BAR-B-tree can be updated by first updating the BAR-B-tree $\mathcal{T}$, followed by updating the object lists. Since each object only has a constant number of guards, the cost of the former is the same as the update cost of the BAR-B-tree asymptotically. However, we do not have a

worst-case or amortized bound for the latter, as one object may intersect many regions of the leaves of $\mathcal{T}$. Nevertheless, since an object only intersects $O(\lambda)$ such regions on average over all stored objects, we expect the actual update cost to be small in practice.

References

1. Agarwal, P.K., Arge, L., Erickson, J., Franciosa, P., Vitter, J.: Efficient searching with linear constraints. Journal of Computer and System Sciences 61(2), 194–216 (2000)
2. Agarwal, P.K., Arge, L., Procopiuc, O., Vitter, J.S.: A framework for index bulk loading and dynamization. In: Proc. International Colloquium on Automata, Languages, and Programming, pp. 115–127 (2001)
3. Agarwal, P.K., de Berg, M., Gudmundsson, J., Hammar, M., Haverkort, H.J.: Box-trees and R-trees with near-optimal query time. Discrete and Computational Geometry 28(3), 291–312 (2002)
4. Aggarwal, A., Vitter, J.S.: The input/output complexity of sorting and related problems. Communications of the ACM 31(9), 1116–1127 (1988)
5. Arge, L.: External memory data structures. In: Abello, J., Pardalos, P.M., Resende, M.G.C. (eds.) Handbook of Massive Data Sets, pp. 313–358. Kluwer Academic Publishers, Dordrecht (2002)
6. Arge, L., de Berg, M., Haverkort, H.J., Yi, K.: The priority R-tree: A practically efficient and worst-case optimal R-tree. In: Proc. SIGMOD International Conference on Management of Data, pp. 347–358 (2004)
7. Arge, L., Samoladas, V., Vitter, J.S.: On two-dimensional indexability and optimal range search indexing. In: Proc. ACM Symposium on Principles of Database Systems, pp. 346–357 (1999)
8. Arya, S., Mount, D.M.: Approximate range searching. In: Proc. 11th Annual ACM Symposium on Computational Geometry, pp. 172–181 (1995)
9. de Berg, M., David, H., Katz, M.J., Overmars, M., van der Stappen, A.F., Vleugels, J.: Guarding scenes againt invasive hypercubes. Computational Geometry: Theory and Applications 26, 99–117 (2003)
10. de Berg, M., Katz, M.J., van der Stappen, A.F., Vleugels, J.: Realistic input models for geometric algorithms. In: Proc. 13th Annual ACM Symposium Computational Geometry, pp. 294–303 (1997)
11. de Berg, M., Streppel, M.: Approximate range searching using binary space partitions. Computational Geometry: Theory and Applications 33(3), 139–151 (2006)
12. Duncan, C.: Balanced Aspect Ratio Trees. PhD thesis, John Hopkins University (1999)
13. Duncan, C., Goodrich, M., Kobourov, S.: Balanced aspect ratio trees: Combining the advantages of k-d trees and octrees. Journal of Algorithms 38, 303–333 (2001)
14. Gaede, V., Günther, O.: Multidimensional access methods. ACM Computing Surveys 30(2), 170–231 (1998)
15. Haverkort, H.J., de Berg, M., Gudmundsson, J.: Box-trees for collision checking in industrial installations. In: Proc. ACM Symposium on Computational Geometry, pp. 53–62 (2002)
16. Manolopoulos, Y., Nanopoulos, A., Papadopoulos, A.N., Theodoridis, Y.: R-trees: Theory and Applications. Springer, Heidelberg (2005)

17. Overmars, M.H.: The Design of Dynamic Data Structures. LNCS, vol. 156. Springer, Heidelberg (1983)
18. Streppel, M.: Multifunctional Geometric Data Structures. PhD thesis, Faculty of Mathematics and Computer Science, TU Eindhoven (2007)
19. Robinson, J.: The K-D-B tree: A search structure for large multidimensional dynamic indexes. In: Proc. SIGMOD International Conference on Management of Data, pp. 10–18 (1981)
20. Samet, H.: Spatial data structures. In: Kim, W. (ed.) Modern Database Systems, The Object Model, Interoperability and Beyond, pp. 361–385. ACM Press and Addison-Wesley (1995)
21. Vitter, J.S.: External memory algorithms and data structures: Dealing with MASSIVE data. ACM Computing Surveys 33(2), 209–271 (2001)