A Framework for Index Bulk Loading and Dynamization

Pankaj K. Agarwal[⋆], Lars Arge[⋆⋆],
Octavian Procopiuc[⋆ ⋆ ⋆], and Jeffrey Scott Vitter[†]

Center for Geometric Computing, Dept. of Computer Science,
Duke University, Durham, NC 27708-0129, USA.
{pankaj,large,tavi,jsv}@cs.duke.edu

Abstract. In this paper we investigate automated methods for externalizing internal memory data structures. We consider a class of balanced trees that we call *weight-balanced partitioning trees* (or *wp-trees*) for indexing a set of points in $\mathbb{R}^d$. Well-known examples of wp-trees include kd-trees, BBD-trees, pseudo-quad-trees, and BAR-trees. Given an efficient external wp-tree construction algorithm, we present a general framework for automatically obtaining a dynamic external data structure. Using this framework together with a new general construction (bulk loading) technique of independent interest, we obtain data structures with guaranteed good update performance in terms of I/O transfers. Our approach gives considerably improved construction and update I/O bounds for e.g. external kd-trees and BBD-trees.

1 Introduction

Both in the database and algorithm communities, much attention has recently been given to the development of I/O-efficient external data structures for indexing point data. A large number of external structures have been developed, reflecting the many different requirements put on such structures; small size, efficient query and update bounds, capability of answering a wide range of queries (mainly range and proximity queries), and simplicity. See recent surveys [2, 14, 27]. The proposed data structures can roughly be divided into two classes, namely simple and practical (often heuristics

⋆ Supported by Army Research Office MURI grant DAAH04–96–1–0013, by a Sloan fellowship, by NSF grants ITR–333–1050, EIA–9870724 and CCR–9732787 and by a grant from the U.S.-Israeli Binational Science Foundation.

⋆⋆ Supported in part by the National Science Foundation through ESS grant EIA–9870734, RI grant EIA–9972879 and CAREER grant EIA–9984099. Part of this work was done while visiting BRICS, University of Aarhus, Denmark.

⋆ ⋆ ⋆ Supported by the National Science Foundation through research grant EIA–9870734 and by the Army Research Office through MURI grant DAAH04–96–1–0013. Part of this work was done while visiting BRICS, University of Aarhus, Denmark.

† Supported in part by the National Science Foundation through research grants CCR–9877133 and EIA–9870734 and by the Army Research Office through MURI grant DAAH04–96–1–0013. Part of this work was done while visiting BRICS, University of Aarhus, Denmark.

based) structures, for which worst-case query performance guarantees can only be given in the *static* case (if at all), and theoretically optimal but usually complicated *dynamic* structures. The first class of structures are often external versions of well-known simple internal memory structures.

In this paper, we develop a general mechanism for obtaining efficient external data structures from a general class of simple internal memory structures, such that the external structures are efficient in the *dynamic* case. Part of our result is a new general index construction (bulk loading) technique which is of independent interest.

1.1 Computational Model and Previous Results

In this paper we analyze data structures in the standard two-level external memory model defined by the following parameters [1, 18]: N, the number of input elements, M, the number of elements that fit in main memory, and B, the number of elements that fit in one disk block, where $N \gg M$ and $1 \leq B \leq M/2$. One *I/O operation* (or simply *I/O*) in this model consists of reading one block of contiguous elements from disk into main memory or writing one block from main memory to disk. The measure of performance of an algorithm or data structure is the number of I/O operations it performs and the maximum disk space (blocks) it uses. For notational simplicity, we use $n = N/B$ and $m = M/B$ to denote the input size and memory size in units of data blocks.

Aggarwal and Vitter [1] developed algorithms for sorting a set of N elements in external memory in optimal $\Theta(n \log_m n)$ I/Os. Subsequently, I/O-efficient algorithms have been developed for large number of problems. Recently, many provably efficient (and often optimal) external data structures have also been developed. Ideally, an external data structure should use linear space, $O(n)$ blocks, and answer a query in $O(\log_B N + K/B)$ I/Os, where K is the number of elements reported by the query. These bounds are obtained by the B-tree data structure for one-dimensional range searching [8, 11]. For two-dimensional range searching, $O(\sqrt{n} + K/B)$ is the best obtainable query bound with linear space [26]. Structures that use more than linear space are often infeasible in practical applications. Below we discuss the known external memory data structures most relevant to this paper. See [27, 2] for complete surveys of known results.

One main challenge in the design of external data structures is obtaining good query performance in a dynamic environment. Early structures, such as the grid file [20], the various quad-trees [21, 23], and the kdB-tree [22], were poorly equipped to handle updates. Later structures tried to employ various (heuristic) techniques to preserve the query performance and space usage under updates. They include the LSD-tree [16], the buddy tree [24], the hB-tree [19], and R-tree variants (see [14] and the references therein). These data structures are often the methods of choice in practical applications because they use linear space and reportedly perform well in practice. However, in a dynamic environment, the query time is high in the worst-case. The hB-tree (or holey brick tree), for example, is based on the statically query-efficient kdB-tree, which combines the spatial query capabilities of the kd-tree [9] with the I/O-efficiency of the B-tree. While nodes in a kdB-tree represent rectangular regions of the space, nodes in an hB-tree represent so-called “holey bricks,” or rectangles from which smaller rectangles have been cut out. This allows for the underlying B-tree to be maintained during updates (insertions). Unfortunately, a similar claim cannot be made about the underlying kd-tree, and thus good query-efficiency cannot be maintained.

Recently, a number of theoretical worst-case efficient dynamic external data structures have been developed. The cross-tree [15] and the O-tree [17], for example, both use linear-space, answer range queries in the optimal number of I/Os, and they can be updated I/O-efficiently. However, their practical efficiency has not been investigated, probably because a theoretical analysis shows that their average query performance is close to the worst-case performance. By contrast, the average-case performance of the kd-tree (and the structures based on it) is much better than the worst-case performance [25]. Other linear-space and query and update optimal external data structures have been designed for special types of range queries, like 2- or 3-sided two-dimensional range queries and halfspace range queries (see e.g. [2, 4]). The practical efficiency of these structures still has to be established.

In the database literature, the term bulk loading is often used to refer to the process of constructing an external data structure. Since bulk loading an index using repeated insertion is often highly non-efficient [3], the development of specialized bulk loading algorithms has received a lot of attention recently. Most work on bulk loading has concentrated on the R-tree (see [3, 12] and the references therein).

1.2 Our Results.

In Section 2 of this paper, we define a class of linear-space trees for indexing a set of points in $\mathbb{R}^d$. These so-called *wp-trees* generalize known internal memory data structures like kd-trees [9], pseudo-quad-trees [21], BBD-trees [7], and BAR trees [13]. We also show how a wp-tree can be efficiently mapped to external memory, that is, how it can be stored in external memory using $O(n)$ blocks such that a root-to-leaf path can be traversed I/O-efficiently. In Section 3, we then design a general technique for bulk loading wp-trees. Using this technique we obtain the first I/O-optimal bulk loading algorithms for kd-trees, pseudo-quad-trees, BBD-trees and BAR-trees. Our algorithms use $O(n \log_m n)$ I/Os while previously known algorithms use at least $\Omega(n \log_2 n)$ I/Os. Finally, in Section 4, we describe several techniques for making a wp-tree dynamic. Our techniques are based on dynamization methods developed for internal memory (*partial rebuilding* and the *logarithmic method*) but adapted for external memory. Together with our external bulk loading technique, they allow us to obtain provably I/O-efficient dynamic versions of structures like the kd-trees, pseudo-quad-trees, BBD-trees, and BAR-trees. Previously, no such structures were known.

2 The wp-tree Framework

In this section, we introduce wp-trees and show how they can mapped to external memory. To simplify the presentation, we discuss our results in $\mathbb{R}^2$. They can all easily be generalized to higher dimensions.

Definition 1 A (β, δ, κ) *weight-balanced partitioning tree* (or *wp-tree*) on a set S of N points in $\mathbb{R}^2$ satisfies the following constraints:

1. Each node v corresponds to a region r_v in $\mathbb{R}^2$, called the extent of v. The extent of the root node is $\mathbb{R}^2$;
2. Each non-leaf node v has $\beta \geq 2$ children corresponding to a partition of r_v into β disjoint regions;
3. Each leaf node v stores exactly one point p from S inside r_v;
4. Let $w(v)$ be the *weight* of node v, defined as the number of data points stored in the subtree rooted at v, and let $v^{(\kappa)}$ be the κ'th ancestor of v. Then $w(v) \leq \delta w(v^{(\kappa)})$ for all nodes v and $v^{(\kappa)}$.

The wp-tree generalizes a number of internal memory data structures used to index point data sets: kd-trees [9], pseudo-quad-trees [21], BBD-trees [7], and BAR-trees [13] are all wp-trees.

Condition 4 (the *weight condition*) insures that wp-trees are balanced; only a constant number of partition steps (κ) is required to obtain regions containing a fraction (δ) of the points.

Lemma 1. *The height of a wp-tree is at most* $\kappa(\log_{1/\delta} N + 1) - 1$.

We want to store a wp-tree on disk using $O(n)$ disk blocks so that a root-to-leaf path can be traversed I/O-efficiently. Starting with the root v we fill disk blocks with the subtree obtained by performing a breadth-first search traversal from v until we have traversed at most B nodes. We recursively block the tree starting in the leaves of this subtree. The *blocked wp-tree* obtained in this way can be viewed as a fanout $\Theta(B)$ tree with each disk block corresponding to a node. We call these nodes *block nodes* in order to distinguish them from wp-tree nodes. The leaf block nodes of the blocked wp-tree are potentially underfull (contain less than $\Theta(B)$ wp-tree nodes), and thus $O(N)$ blocks are needed to block the tree in the worst case. To alleviate this problem, we let certain block nodes share the same disk block. More precisely, if v is a non-leaf block node, we reorganize all v's children that are leaf block nodes, such that at most one disk block is non-full. This way we only use $O(n)$ disk blocks. Since each non-leaf block node contains a subtree of height $O(\log_2 B)$ we obtain the following.

Lemma 2. *A blocked wp-tree $\mathcal{T}$ is a multi-way tree of height $O(\log_B N)$. It can be stored using $O(n)$ blocks.*

2.1 The Restricted wp-tree

The wp-tree definition emphasizes the structure of the tree more than the geometry of the partitioning. The dynamization methods we will discuss in Section 4 can be applied to any wp-tree. However, without specifying the properties of the partitioning, we cannot quantify the update and query I/O-bounds obtained using these methods. Therefore we now restrict the definition of a wp-tree by adding geometric constraints on the extent of a node and the partitioning methods used. On the one hand the resulting class of *restricted wp-trees* is general enough to encompass many interesting data structures, and on the other hand it is restrictive enough to allow us to prove general bulk loading, update, and query bounds.

Definition 2 A *restricted* (β, δ, κ) *wp-tree* is a (β, δ, κ) wp-tree in which each node v satisfies the following constraints:

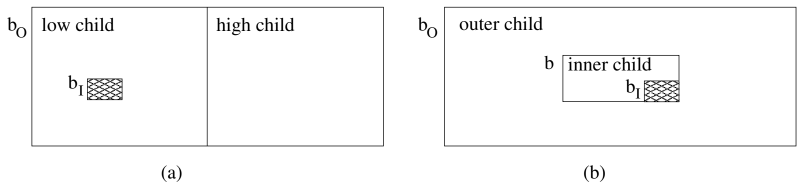

Fig. 1. BBD-tree partitions. (a) Split node. (b) Shrink node.

1. The extent r_v of v is the region lying between two convex polygons; $r_v = b_O \setminus b_I$. The *inner polygon* b_I must be completely inside the *outer polygon* b_O, and the orientations of edges forming b_I and b_O must be taken from a constant set of directions D.
2. The extents of the β children of v are obtained from r_v by applying the following *cut rules* a constant number of times:
   (a) A *geometric cut* (ℓ). A geometric cut is a line ℓ along a direction $e \in D$ not intersecting b_I.
   (b) A *rank cut* (e, α). A rank cut is a line ℓ along a direction $e \in D$. Let ℓ' be the line along e such that $\alpha w(v)$ of the $w(v)$ points corresponding to v are to the left of ℓ'. Then ℓ is the closest line to ℓ' not intersecting the interior of b_I.
   (c) A *rectangle cut*. A rectangle cut can be applied to v only if b_I and b_O are both fat rectangles (i.e., the aspect ratio is at most 3) and $2b_I \subset b_O$. A rectangle cut is a fat rectangle b' such that $b_I \subset b' \subset b_O$ and both $b' \setminus b_I$ and $b_O \setminus b'$ contain at most $2w(v)/3$ points.

2.2 Examples of Restricted wp-trees

Like wp-trees, restricted wp-trees generalize internal memory data structures like kd-trees, BBD-trees, pseudo-quad-trees, and BAR-trees. Below we further discuss kd-trees and BBD-trees. In the full paper we show how pseudo-quad-trees and BAR-trees are also captured by the restricted wp-tree definition.

The kd-tree. Introduced by Bentley [9], the kd-tree is a classical structure for answering range (or window) queries. It is a binary tree that represents a recursive decomposition of the space by means of hyperplanes orthogonal to the coordinate axes. In $\mathbb{R}^2$ the partition is by axes-orthogonal lines. Each partition line divides the point-set into two equal sized subsets. On even levels of the tree the line is orthogonal to the x-axis, while on odd levels it is orthogonal to the y-axis. These partitions are *rank cuts* $(e, 1/2)$, where e is orthogonal to the x- or y-axis. Thus the kd-tree is a restricted $(2, 1/2, 1)$ wp-tree.

The BBD-tree The *balanced box decomposition tree*, or *BBD-tree*, was introduced by Arya et al [7] for answering approximate nearest-neighbor queries. Like the kd-tree, the BBD-tree is a binary tree representing a recursive decomposition of the space. The region associated with a BBD-tree node is the set theoretic difference of two fat rectangles, b_I and b_O (with b_I included in b_O). More precisely, a BBD-tree consists of two types of nodes: *split nodes* and *shrink nodes*. In a *split node*, the partition is done using an axis-orthogonal line that cuts the longest side of b_O so that the resulting rectangles are fat and b_I lies entirely inside one of them—refer to Figure 1(a). In a *shrink node* v, the partition is done using a box rather than a line. This box b lies inside b_O, contains b_I, and determines the extent of the two children: $b \setminus b_I$ is the extent of the inner child and $b_O \setminus b$ is the extent of the outer child—refer to Figure 1(b). While split nodes reduce the geometric size, the box b used in shrink nodes is chosen so as to reduce the number of points by a factor of 1.5. By alternating split nodes and shrink nodes, both the geometric size and the number of points associated with each node decrease by a constant factor as we descend a constant number of levels in the BBD-tree (see [7] for details). It is easy to see that the split node uses a geometric cut and the shrink node uses a rectangle cut. In the full paper we show that a BBD-tree is a restricted (2,2/3,3) wp-tree.

3 Bulk Loading Restricted wp-trees

In this section we describe an optimal algorithm for bulk loading (constructing) a blocked restricted wp-tree.

It is natural to bulk load a wp-tree using a top-down approach. For example, to construct a kd-tree on N points in $\mathbb{R}^2$ we first find the point with the median x-coordinate in $O(n)$ I/Os [1]. We then distribute the points into two sets based on this point and proceed recursively in each set, alternating between using the median x-coordinate and y-coordinate to define the distribution. This way each level of the wp-tree is constructed in a linear number of I/Os, so in total we use $O(n \log_2 n)$ I/Os to bulk load the tree. This is a factor of $\log_2 m$ larger than the optimal $O(n \log_m n)$ bound (the sorting bound). Intuitively, we need to construct $\Theta(\log_2 m)$ levels of the wp-tree in a linear number of I/Os—instead of just one—in order to obtain this bound. Doing so seems difficult because of the way the points are alternately split by x- and y-coordinates. Nevertheless, below we show how to bulk load a blocked restricted wp-tree, and thus a kd-tree, in $O(n \log_m n)$ I/Os.

To simplify the presentation, we present our restricted wp-tree bulk loading algorithm only for the case where $\beta = 2$ and where D contains the two directions orthogonal to the coordinate axes. The details of the general algorithm will be given in the full paper. Let S be a set of N points in $\mathbb{R}^2$. The first step in constructing a blocked wp-tree for S is to sort the N points twice: once according to their x-coordinate, and once according to their y-coordinate. Call the resulting ordered sets S_x and S_y, respectively. Next a recursive procedure **Bulk_load** is called on S_x and S_y. **Bulk_load** builds a subtree of height $\Theta(\log_2 m)$ in each recursive call, The main idea in the algorithm is to impose a grid on the set of input points and count the number of points in each grid cell. The grid counts allow us to compute partitions without reading all the points. More precisely, **Bulk_load** starts by dividing the current region (initially $\mathbb{R}^2$) into $t = \Theta(\min\{m, \sqrt{M}\})$ vertical slabs and t horizontal slabs, each containing N/t points—refer to Figure 2(a). These slabs form a $t \times t$ grid. Note that the grid size t^2 is at most M, and thus the grid can fit into internal memory. The number of points in each grid cell is then computed and stored in a matrix A in main memory. All three types of cuts can now be computed efficiently using A. A rank cut (e, α) for a node v, for example, is computed by first finding the slab E_k along e containing the cutting line. This can be done without performing I/Os. The exact cutting line can then be computed in $O(N/(Bt))$ I/Os by scanning the points in E_k. After a subtree $\mathcal{T}$ of height $\Theta(\log_2 t)$ is built, S_x and S_y are distributed into t sets each, corresponding to the leaves of $\mathcal{T}$ and the rest of the tree is built by calling **Bulk_load** recursively. The detailed **Bulk_load** procedure is given below.

```
procedure Bulk_load($S_x$, $S_y$, $v$)
  1. Divide $S_x$ into $t$ sets, corresponding to $t$ vertical slabs $X_1, \ldots, X_t$, each containing $|S_x|/t$
     points. Store the $t+1$ boundary $x$-coordinates in memory.
  2. Divide $S_y$ into $t$ sets, corresponding to $t$ horizontal slabs $Y_1, \ldots, Y_t$, each containing $|S_y|/t$
     points. Store the $t+1$ boundary $y$-coordinates in memory.
  3. The vertical and horizontal slabs form a grid. Let $C_{i,j}$ be the set of points in the grid cell
     formed at the intersection of the $i$th horizontal slab and the $j$th vertical slab. Create a $t \times t$
     matrix $A$ in memory. Scan $S_x$ and compute the grid cell counts: $A_{i,j} = |\mathcal{C}_{i,j}|$, $1 \leq i, j \leq t$.
  4. Let $u$=$v$.
  5. (a) If $u$ is partitioned using a geometric cut orthogonal to the $x$-axis, determine the slab
         $X_k$ containing the cut line $\ell$ using the boundary $x$-coordinates. Next scan $X_k$ and, for
         each cell $\mathcal{C}_{j,k}$, $1 \leq j \leq t$, compute the counts of "subcells" $\mathcal{C}^{<}_{j,k}$ and $\mathcal{C}^{>}_{j,k}$ obtained
         by splitting cell $\mathcal{C}_{j,k}$ at $\ell$—refer to Figure 2(b). Store these counts in main memory
         by splitting the matrix $A$ into $A^{<}$ and $A^{>}$, containing the first $k$ columns and the last
         $(t-k+1)$ columns of $A$, respectively (column $k$ from matrix $A$ appears in both $A^{<}$
         and $A^{>}$). Then let $A^{<}_{j,k} = |\mathcal{C}^{<}_{j,k}|$ and $A^{>}_{j,1} = |\mathcal{C}^{>}_{j,k}|$, $1 \leq j \leq k$. Go to 5.(d).
     (b) If $u$ is partitioned using a rank cut orthogonal to the $x$-axis, first determine the slab
         $X_k$ containing the cut line $\ell$ using $A$, then scan $X_k$ to determine the exact position of
         the cut line. Next split $A$ into $A^{<}$ and $A^{>}$ as above. A cut orthogonal to the $y$-axis is
         handled similarly. Go to 5.(d).
     (c) If $u$ is partitioned using a rectangle cut, use the following algorithm to determine the
         sides of $b'$. Let $\ell$ be a line orthogonal to the longest side of $b_O$ that cuts $b_O$ into two fat
         rectangles and does not intersect $b_I$. Using only the grid cell counts, decide whether
         any of the two new regions contains more than $2w(u)/3$ points. If this is the case,
         then repeat the process in that region. Otherwise, the region with the largest number of
         points becomes $b'$. Scan the (up to) four slabs that contain the sides of $b'$ and compute
         the counts of the "subcells". These counts will be stored in $A^{<}$, the cell count matrix
         for $b' \setminus b_I$, and $A^{>}$, the cell count matrix for $b_O \setminus b'$. Go to 5.(d).
     (d) Create a new wp-tree node for each of the two regions constructed. For each of these
         two nodes, determine its partition by repeating step 5, in which the role of $A$ is played
         by $A^{<}$ or $A^{>}$. Stop when reaching level $\log_2 t$.
  6. Scan $S_x$ and $S_y$ and distribute the $N$ points into $t$ pairs of sets $(S^i_x, S^i_y)$, corresponding to
     the $t$ leaves $v_i$ of $\mathcal{T}$.
  7. For each pair of sets $(S^i_x, S^i_y)$ do the following. If $(S^i_x, S^i_y)$ fits in memory, then construct
     the remaining wp-tree nodes. Otherwise, recursively call Bulk_load on $(S^i_x, S^i_y, v_i)$.
```

Theorem 1. *A blocked restricted wp-tree can be bulk loaded in $O(n \log_m n)$ I/Os.*

Proof. First note that sorting the points takes $O(n \log_m n)$ I/Os. Once sorted, the points are kept sorted throughout the recursive calls to the **Bulk_load** procedure. Next note that the choice of $t = \Theta(\min\{m, \sqrt{M}\}) = O(\sqrt{m})$ means that the original $t \times t$ count matrix A fits in memory. In fact, since each of the $2^{\log_2 t} = t$ nodes built in one call to **Bulk_load** adds at most t counts, all count matrices produced during one such call fit in memory.

Now consider one call to **Bulk_load**. Steps 1, 2 and 3 of **Bulk_load** are linear scans of the input sets S_x and S_y using $O(n)$ I/Os. Step 6 can also be performed in $O(n)$ I/Os since S_x and S_y are distributed into $t = \Theta(\min\{m, \sqrt{M}\}) = O(m)$ sets (which means

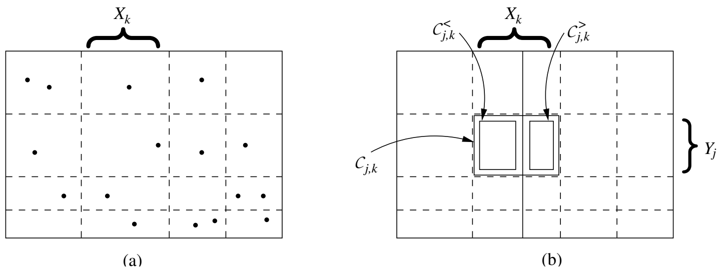

Fig. 2. Finding the median using the grid cells. (a) Slab X_k contains N/t points. (b) $A^<$ and $A^>$ are computed by splitting X_k along ℓ.

that one block for each of the sets can be maintained in memory during the distribution). Step 5 (recursively) computes a subtree of height $\log_2 t$, using a different algorithm for each of the three partition types. A geometric or rank cut (Step 5.(a) or 5.(b)) can be computed in $O(|S_x|/t)$ I/Os since slab X_k is scanned at most three times. Similarly, a rectangle cut (Step 5.(c)) can also be computed in $O(|S_x|/t)$ I/Os. The details of this argument will be given in the full paper. It can also be proven that a rectangle cut always exists [7]. Summing up over the $2^{\log_2 t} = O(t)$ nodes built, we obtain that Step 5 can be performed in $O(n)$ I/Os.

Since a subtree of height $\Theta(\log_2 t)$=$\Theta(\log_2 m)$ can be built in a linear number of I/Os (one call to **Bulk_load**), the cost of building the entire blocked restricted wp-tree is $O(n\frac{\log_2 n}{\log_2 m}) = O(n \log_m n)$ I/Os.

Corollary 1. *A kd-tree, BBD-tree, BAR-tree or pseudo-quad-tree can be bulk loaded in $O(n \log_m n)$ I/Os.*

4 The Dynamization Framework

In this section we present a framework for making wp-trees dynamic. We present three methods: the first one takes advantage of the weight balancing property of wp-trees and uses partial rebuilding to maintain the tree balanced [6, 21], and the other two methods are based on the so-called logarithmic method [10, 21]. All three methods take advantage of the improved bulk loading bounds obtained in the previous section. While the methods are not new, we show how their application to blocked restricted wp-trees produces new dynamic data structures for indexing points in $\mathbb{R}^2$ that are competitive with or better than existing data structures in terms of I/O performance. The choice of method for a given data structure depends on its update and query bounds as well as the application the external structure is to be used in.

4.1 Partial Rebuilding

In the definition of a $(\beta, \delta_0, \kappa)$ wp-tree, the weight condition is satisfied by any $\delta > \delta_0$. This method of relaxing the weight condition allows us to perform updates with good amortized complexity. A node v is said to be *out of balance* if there is another node u such that $u^{(\kappa)} = v$ and $w(u) > \delta w(v)$. In other words, a node is out of balance if one of its descendants is too heavy. A node v is said to be *perfectly balanced* if all nodes u such that $u^{(\kappa)} = v$ satisfy $w(u) \leq \delta_0 w(v)$.

In order to allow dynamic updates on a blocked wp-tree, we employ a partial rebuilding technique, used by Overmars [21] to dynamically maintain quad-trees and kd-trees balanced, and first adapted to external memory by Arge and Vitter [6]. When inserting a new point into the data structure, we first insert it in the appropriate place among the leaves, and then we check for nodes on the path to the root that are out of balance. If v is the highest such node, we rebuild the whole subtree rooted at v into a perfectly balanced tree. In the full paper we prove the following.

Theorem 2. *Let $\mathcal{T}$ be a blocked restricted wp-tree on N points. We can insert points into $\mathcal{T}$ in $O\left(\frac{1}{B}(\log_m n)(\log_2 n) + \log_B n\right)$ I/Os, amortized, and delete points from $\mathcal{T}$ in $O(\log_B n)$ I/Os, worst case. Point queries take $O(\log_B n)$ I/Os, worst case.*

As n goes to infinity, the first additive term dominates the insertion bound. In practice, however, we expect the behavior to be consistent with the second term, $O(\log_B n)$, because the value of B is in the thousands, thus cancelling the effect of the $\log_2 n$ factor in the first term for all practical values of n.

4.2 Logarithmic Methods

The main idea in the logarithmic method [10, 21] is to partition the set of input objects into $\log_2 N$ subsets of increasing size 2^i, and build a perfectly balanced data structure $\mathcal{T}_i$ for each of these subsets. Queries are performed by querying each of the $\log_2 N$ structures and combining the answers. Insertion is performed by finding the first empty structure $\mathcal{T}_i$, discarding all structures $\mathcal{T}_j$, $0 \leq j < i$, and building $\mathcal{T}_i$ from the new object and all the objects previously stored in $\mathcal{T}_j$, $0 \leq j < i$. One can adapt the method to external memory by letting the ith subset contain either 2^i *blocks* of points or B^i *points*. We call the two resulting methods *the logarithmic method in base 2* and *the logarithmic method in base B*, respectively.

Logarithmic method in base 2. As mentioned, the ith subset contains 2^i blocks, or $B \cdot 2^i$ points, $0 \leq i \leq \log_2 n$. Queries are performed by combining the answers from the $\log_2 n$ structures. Insertions are performed as in the internal memory case, but we need to maintain a block in internal memory. All insertions go into this block until the block is full, at which time the rebuilding is performed using all points in the block. In the full paper we prove the following.

Theorem 3. *A forest of perfectly balanced blocked restricted wp-trees for indexing N points can be maintained such that insertions take $O\left(\frac{1}{B}(\log_m n)(\log_2 n)\right)$ I/Os, amortized, deletions take $O((\log_m n)(\log_2 n))$ I/Os, worst case, and point queries take $O((\log_B n)(\log_2 n))$ I/Os, worst case.*

Note that, for realistic values of n, m and B, we need less than one I/O, amortized, to insert a point This should be compared to the (at least) $O(\log_B n)$ used in the partial rebuilding method. However, the deletion and point query bounds of this method are worse than the bounds obtained using partial rebuilding.

Logarithmic method in base B. Arge and Vahrenhold used the logarithmic method in base B to obtain an I/O-efficient solution to the dynamic point location problem [5]. Following closely the ideas of Arge and Vahrenhold, we obtain the following.

Theorem 4. *A forest of perfectly balanced blocked restricted wp-trees for indexing N points can be maintained such that insertions take $O\left((\log_m n)(\log_B n)\right)$ I/Os, amortized, deletions take $O(\log_B n)$ I/Os, amortized, and point queries take $O(\log_B^2 n)$ I/Os, worst case.*

The insertion bound of the base B method is a factor of $\frac{B}{\log_2 B}$ worse than the bound obtained using the base 2 method. The deletion bound, however, is improved by a factor of $\log_2 n$.

4.3 Applications

We now briefly state the results we obtain when using the three dynamization methods on our two running examples, kd-trees and BBD-trees.

The kd-tree. In the full paper we show how we can exploit a property of the kd-tree partitioning method to obtain worst-case bounds on the number of I/Os needed to perform a range query. We obtain the following.

Theorem 5. *Using partial rebuilding, a dynamic external kd-tree can be designed, which answers range queries in $O(\sqrt{N}^{\log_{1/\delta} 2}/\sqrt{B} + K/B)$ I/Os in the worst case, where K is the number of points reported. Each insertion takes $O\left(\frac{1}{B}(\log_m n)(\log_2 n) + \log_B n\right)$ I/Os, amortized, and each deletion takes $O(\log_B n)$ I/Os, worst case. Using the logarithmic method in base 2 (or in base B), a structure with an $O(\sqrt{n} + K/B)$ worst-case range query bound can be designed. In this case insertions take $O\left(\frac{1}{B}(\log_m n)(\log_2 n)\right)$ I/Os, amortized (or $O((\log_m n)(\log_B n))$ I/Os, amortized), and deletions take $O\left((\log_m n)(\log_2 n)\right)$ I/Os, worst case (or $O(\log_B n)$ I/Os, amortized).*

Using the logarithmic methods, the query bound of the dynamic structure is the same as the bound for the static structure, although a logarithmic number of trees are queried in the worst case. This is true in general for a structure with polynomial query bound, because the cost to search each successive structure is geometrically decreasing. If the query bound on the static structure is polylogarithmic (as in our next example), the bound on the dynamic structure increases.

The BBD-tree. The BBD-tree can be used to answer $(1+\epsilon)$-approximate nearest neighbor queries [7]. Using our dynamization methods we obtain the following.

Theorem 6. *Using partial rebuilding a dynamic external BBD-tree can be designed, which answers a $(1+\epsilon)$-approximate nearest neighbor query in $Q_{BBD}(N) = O(c(\epsilon)(\log_m n)(\log_2 n)/B + \log_B n)$ I/Os, where $c(\epsilon) = 2\left\lceil 1 + \frac{12}{\epsilon}\right\rceil^2$ [7]. Insertions take $O\left(\frac{1}{B}(\log_m n)(\log_2 n) + \log_B n\right)$ I/Os, amortized, and deletions take $O(\log_B n)$ I/Os, worst case. Using the logarithmic method in base 2 (or in base B), the query bound increases to $Q_{BBD}(N)\log_2 n$ (or $Q_{BBD}(N)\log_B n$). Insertions take $O\left(\frac{1}{B}(\log_m n)(\log_2 n)\right)$ I/Os, (or $O((\log_m n)(\log_B n))$ I/Os), amortized, and deletions take $O\left((\log_m n)(\log_2 n)\right)$ I/Os, worst case (or $O(\log_B n)$ I/Os, amortized).*

References

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

2. 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.
3. L. Arge, K. H. Hinrichs, J. Vahrenhold, and J. S. Vitter. Efficient bulk operations on dynamic R-trees. *Algorithmica*, 33(1):104–128, 2002.
4. L. Arge, V. Samoladas, and J. S. Vitter. On two-dimensional indexability and optimal range search indexing. In *Proc. ACM Symp. Principles of Database Systems*, pages 346–357, 1999.
5. L. Arge and J. Vahrenhold. I/O-efficient dynamic planar point location. In *Proc. ACM Symp. on Computational Geometry*, pages 191–200, 2000.
6. L. Arge and J. S. Vitter. Optimal dynamic interval management in external memory. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, pages 560–569, 1996.
7. S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of the ACM*, 45(6):891–923, Nov. 1998.
8. R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.
9. J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, Sept. 1975.
10. J. L. Bentley. Decomposable searching problems. *Inform. Process. Lett.*, 8:244–251, 1979.
11. D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.
12. M. de Berg, J. Gudmundsson, M. Hammar, and M. Overmars. On R-trees with low stabbing number. In *Proc. Annual European Symposium on Algorithms*, pages 167–178, 2000.
13. C. A. Duncan, M. T. Goodrich, and S. Kobourov. Balanced aspect ratio trees: Combining the advantages of k-d trees and octrees. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 300–309, N.Y., Jan. 17–19 1999. ACM-SIAM.
14. V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.
15. R. Grossi and G. F. Italiano. Efficient cross-trees for external memory. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society Press, 1999.
16. A. Henrich, H.-W. Six, and P. Widmayer. Paging binary trees with external balancing. In *Proc. Graph-Theoretic Concepts in Computer Science, LNCS 411*, pages 260–276, 1989.
17. 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.
18. D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading MA, second edition, 1998.
19. D. Lomet and B. Salzberg. The hB-tree: A multiattribute indexing method with good guaranteed performance. *ACM Transactions on Database Systems*, 15(4):625–658, 1990.
20. J. Nievergelt, H. Hinterberger, and K. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1):38–71, 1984.
21. M. H. Overmars. *The Design of Dynamic Data Structures*, volume 156 of *Lecture Notes Comput. Sci.* Springer-Verlag, Heidelberg, West Germany, 1983.
22. J. Robinson. The K-D-B tree: A search structure for large multidimensional dynamic indexes. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 10–18, 1981.
23. H. Samet. *The Design and Analyses of Spatial Data Structures*. Addison Wesley, MA, 1990.
24. B. Seeger and H.-P. Kriegel. The buddy-tree: An efficient and robust access method for spatial data base systems. In *Proc. International Conf. on Very Large Databases*, pages 590–601, 1990.

25. Y. V. Silva Filho. Average case analysis of region search in balanced k-d trees. *Inform. Process. Lett.*, 8:219–223, 1979.
26. S. Subramanian and S. Ramaswamy. The P-range tree: A new data structure for range searching in secondary memory. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 378–387, 1995.
27. J. S. Vitter. External memory algorithms and data structures. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*, pages 1–38. American Mathematical Society, DIMACS series in Discrete Mathematics and Theoretical Computer Science, 1999.