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. These trees are defined with fixed degree and are thus suited for internal memory implementations. Given an efficient wp-tree construction algorithm, we present a general framework for automatically obtaining a new 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 of e.g. kd-trees and BBD-trees.

* 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.

1 Introduction

Both in the database and algorithm communities, much attention has recently been given to the development of I/O-efficient data structures for indexing point data. A large number of data structures (or *indexes*) have been developed, reflecting the many different requirements put on such structures; small (often linear) size, efficient query of update bounds, capabilities of answering a wide variety of queries (mainly range and proximity queries), and simplicity. See recent surveys [19, 3, 43]. The proposed data structures can roughly be divided into two classes, namely practically used (and often heuristics based) structures, for which worst-case query performance guarantees can only be given (if at all) in the *static* case, and theoretically optimal *dynamic* structures, which have yet to be proven practically efficient. The first class of structures are often external versions of well-known simple internal memory structures.

In this paper, we try to combine the advantages of the two classes of structures by developing 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 I/O model and previous results

In this paper we analyze the I/O and space complexity of data structures in the standard two-level I/O model defined by the following parameters [2, 28]: 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 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 [2] developed optimal algorithms for sorting a set of N elements in external memory, in $\Theta(n \log_m n)$ I/Os. Subsequently, I/O-efficient algorithms have been developed for large number of problems. Recently, many efficient (and often optimal) data structures have also been developed. Ideally, an external data structure should use linear space, $O(N/B)$ 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 [9, 15]. For two-dimensional range searching, $O(\sqrt{n} + K/B)$ is the best obtainable query bound with linear space [40]. Structures that use more than linear space are generally infeasible in practical applications. Refer to surveys by Vitter [42] and Arge [3] for references.

One main challenge in the design of external indexing data structures is obtaining good query performance in a dynamic environment. Early structures, such as the grid file [32], the various quad-trees [36, 33], and the kdB-tree [34], were poorly equipped

to handle dynamic updates. Later structures tried to employ various (heuristic) techniques to preserve the query performance and space usage under dynamic updates. They include the LSD-tree [23], the buddy tree [37], the hB-tree [31], and R-tree variants [22, 20, 38, 10, 25]. 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 highly dynamic environment they are all very query suboptimal 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 [11] 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 data structures have been developed. The cross-tree [21] and the O-tree [27] 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 careful theoretical analysis shows that their average query performance is close to the worst-case performance. In contrast, the average case performance of the kd-tree (and the structures based on it) is much better than the worst case performance [39]. Other linear-space and query and update optimal data structures have been designed for special types of range queries, like 2- or 3-sided two-dimensional range queries [7, 5, 26, 40] and halfspace range queries [1]. 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 [4], 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 [35, 24, 17, 30, 41, 13, 16]. Although not optimal, relatively efficient algorithms can often be obtained by constructing an index level-by-level.

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, quad-trees, BBD-trees [8], and BAR trees [18]. 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-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 whereas previously know 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 bulk loading technique, this allows 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 present the class of trees on which our framework can be applied. To simplify the presentation, we develop this framework in $\mathbb{R}^2$. All the results can easily be generalized to any dimension.

Definition 21 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 [11], pseudo-quad-trees [33], BBD-trees [8], and BAR-trees [18] are all wp-trees.

The weight condition insures that wp-trees are balanced. Intuitively, it says that only a constant number of partition steps (κ) is required to obtain regions containing a fraction (δ) of the points each.

Lemma 22 *The height of a wp-tree is at most* $\kappa(\log_{1/\delta} N + 1) - 1$.

We first show how to store a wp-tree on disk using $O(n)$ disk blocks so that a root-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—refer to Figure 1. 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 differentiate them from wp-tree nodes. The leaf block nodes of the blocked wp-tree are potentially underfull (contain less than 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 and since each non-leaf block node contains a subtree of height $O(\log_2 B)$ we obtain the following.

Lemma 23 *A blocked wp-tree $\mathcal{T}$ is a multi-way tree of height $O(\log_B N)$. $\mathcal{T}$ can be stored using $O(n)$ blocks.*

2.1 The restricted wp-tree

The definition of a wp-tree emphasizes the structure of the tree more than the geometry of the partitioning. The dynamization methods that will be presented in Section 4 can be applied to any wp-tree. However, without information about the partitioning used, 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 geomtric constraints on the extent of a node and the partitioning method used. The resulting *restricted wp-tree* is general enough to encompass all data structures that interest us, and at the same time is restrictive enough to alow us to prove general update, bulk loading, and query bounds.

Definition 24 A *restricted (β, δ, κ) wp-tree* is a (β, δ, κ) wp-tree in which each node v satisfies the following constraints:

1. The extent of v is the set theoretic difference of two convex polygons, $r_v = b_O \setminus b_I$. The *inner polygon* b_I must be 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 cuts a constant number of times:
   (a) A *geometric cut* l. A geometric cut is a line l, which has to be along a direction $e \in D$ and should not intersect b_I.
   (b) A *rank cut* (e, α). A rank cut is a line l along direction e, where $e \in D$. Let l' be the line along e such that $\alpha w(v)$ of the $w(v)$ points stored in the subtree rooted in v is 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 ratio between the longest and shortest sides is at most 3). Then the 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 [11], 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 into subspaces by means of hyperplanes orthogonal to the coordinate axes. In $\mathbb{R}^2$, the partition is by axes-orthogonal lines—refer to Figure 1. Each partition line divides the point-set into two equal subsets. On even levels of the tree the line is orthogonal to the x-axis, while on odd levels it is orthogonal to

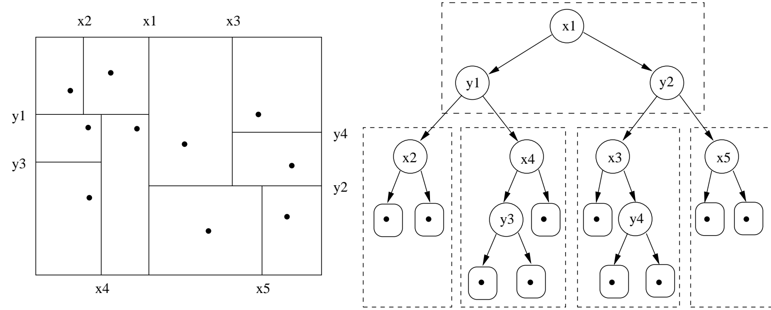

Fig. 1. kd-tree partitioning. The values shown are the coordinates of the dividers. Points are stored in leaves and disk blocks are outlined with dashed lines.

the y-axis. These partitions are rank cuts $(e, 1/2)$, where e is orthogonal to the x- or y-axis. Thus, it is easy to see that 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 [8] for answering approximate nearest neighbor queries. Like the kd-tree, the BBD-tree is a binary tree representing a recursive decomposition of the space into subspaces. The region associated with a BBD-tree node is the set theoretic difference of two rectangles, b_I and b_O (with b_I included in b_O), where the rectangles are fat, meaning that the ratio between the longest and shortest sides is bounded.

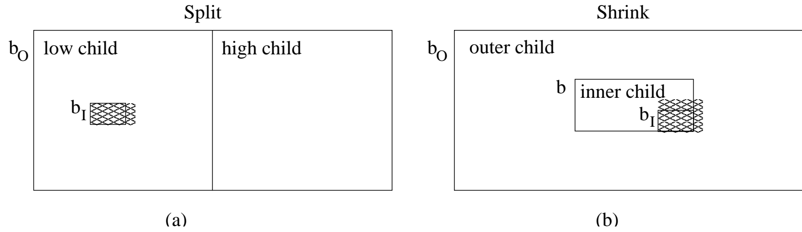

Fig. 2. BBD-tree partitions. (a) Split node. (b) Shrink node.

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 2(a). In a *shrink node* v, the partition is done using a box rather than a line. This box b lies inside b_O 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 2(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 in the tree, both the geometric size and the number of points associated with each node decrease exponentially as we descend a constant number of levels in the BBD-tree (see [8] 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. 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 bound is a factor of $\log_2 m$ bigger 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—instead of just one—in a linear number of I/Os 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 two-dimensional case and when $\beta = 2$ and D contains only 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 sets S_x and S_y, respectively. Next the recursive procedure **Bulk_load** is called with S_x and S_y as input sets. **Bulk_load** builds a subtree of height $\Theta(\log_2 m)$ in each recursive call, until the input fits in internal memory. The main idea in the algorithm is to impose a grid on the set of input points. The grid is computed so that it can be used as an estimate of the point distribution, allowing partitions to be computed 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. These slabs form a $t \times t$ grid. The number of points in each grid cell is then computed and stored in a matrix A, which is kept in memory. All three types of cuts can now be computed fast using A. A rank cut (e, α) for a node v, for example, is computed by first finding the slab E_k along e that contains the cutting line. This can be done without performing I/Os, by scanning the entries from A in the appropriate order and adding them until the sum exceeds $\alpha w(v)$. The slab where the scanning stopped is E_k. Then, in $O(N/t)$ I/Os, E_k is scanned in order to find the exact cutting line.

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 **Bulk_load** is called recursively on each pair of these sets in order to build the rest of the tree.

procedure Bulk_load(S_x, S_y, v)

> *The sets S_x and S_y contain the same N points, sorted on x and y, respectively. The node v is the root of the tree $\mathcal{T}$ being built.*

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.

   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 ith horizontal slab and the jth vertical slab. The bounding box of the cell is known from the boundary coordinates stored in memory in steps 1 and 2.

3. Create a $t \times t$ matrix A in memory. Scan S_x and compute the grid cell counts: $A_{i,j} = |\mathcal{C}_{i,j}|$, $1 \le i, j \le t$.

   Let u=v.

4. (a) If u is partitioned using a *geometric cut* orthogonal to the x-axis, determine the slab X_k containing the cut line l using the boundary x-coordinates.
   Next scan X_k and, for each cell $\mathcal{C}_{j,k}$, $1 \le j \le t$, compute the counts of "subcells" $\mathcal{C}^{<}_{j,k}$ and $\mathcal{C}^{>}_{j,k}$ obtained by splitting cell $\mathcal{C}_{j,k}$ at l—refer to Figure 3(b). Store these counts in main memory, by splitting the matrix A into two: $A^{<}$ 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 $A^{>}_{j,1} = |\mathcal{C}^{>}_{j,k}|$, $1 \le j \le k$. Go to 4.(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 l using A, then scan X_k to determine the exact position of the cut line. Next split A into $A^{<}$ and $A^{>}$ as above. Go to 4.(d).

   (c) If u is partitioned using a *rectangle cut*, use the following algorithm to determine the sides of b'. Let l 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 it does, repeat the process in that region. Otherwise, the region with the largest number of points (of the two) 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 $A^{<}$, a cell count matrix for $b' \setminus b_I$, and $A^{>}$, a cell count matrix for $b_O \setminus b'$. Go to 4.(d).

   (d) For each of the two regions constructed, create a new wp-tree node. For each of these two nodes, determine its partition by repeating step 4, in which the role of A is played by $A^{<}$ and $A^{>}$, respectively. Stop when reaching level $\log_2 t$.

5. 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 $\mathcal{T}$.

6. For each pair of sets (S_x^i, S_y^i) computed in step 5, either load them in memory and construct the remaining wp-tree nodes, or, if they don't fit in memory, recursively call **Bulk_load** on (S_x^i, S_y^i, v_i).

Theorem 31 *A blocked restricted wp-tree can be bulk loaded in $O(n \log_m n)$ I/Os.*

Proof. The value $t = \Theta(\min\{m, \sqrt{M}\})$ for the number of slabs was chosen so that all necessary items fit in internal memory. Indeed, A consists of t^2 integers, so $t \leq \sqrt{M}$, and the t-wise distribution in Step 5 uses $t+1$ disk blocks, so $t < m$.

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. 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. Step 5 can also be performed in $O(n)$ I/Os since S_x and S_y are distributed into $O(m)$ sets.

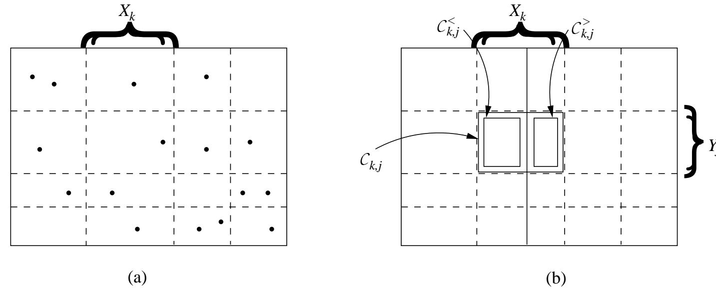

Fig. 3. Finding the median using the grid cells (grid lines are dashed). (a) Slab X_k containing l is computed using A. (b) $A^<$ and $A^>$ are computed by splitting X_k along l.

Step 4 recursively computes a subtree of height $\log_2 t$, using a different algorithm for each of the three patition types. A geometric or rank cut (Step 4.(a) or 4.(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 4.(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 [8]. Summing up over the $2^{\log_2 t} = O(t)$ nodes built, we obtain that Step 4 performs $O(n)$ 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 $O(n \log_m n)$ I/Os.

Corollary 32 *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

the wp-trees and uses partial rebuilding to maintain the tree balanced [7, 33], and the other two methods are based on the so-called logarithmic method [12, 33]. While these 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 method of choice depends on the specific application and on the insertion, deletion, and query bounds. All three methods take advantage of the improved bulk loading bounds obtained in the previous section.

4.1 Partial Rebuilding

In the definition of wp-trees, let δ_0 be the minimum value of δ that satisfies the balancing condition. However, if we relax the balancing condition by choosing a constant $\delta > \delta_0$, we will be able to perform updates with good amortized complexity. A node v of a wp-tree is *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 it has too much weight in one of its descendants. Also, a node v is *perfectly balanced* if any node u such that $u^{(\kappa)} = v$ satisfies $w(u) \leq \delta_0 w(v)$.

In order to allow dynamic updates on a blocked wp-tree, we employ a partial rebuilding technique, used by Overmars [33] for maintaining quad-trees and kd-trees balanced, and first adapted to external memory data structures by Arge and Vitter [7]. When inserting a new point into the data structure, we find its place in the appropriate leaf, and then check for nodes on the path from that leaf to the root that are out of balance. If v is the highest node on this path out of balance, we rebuild the whole subtree rooted at v into a perfectly balanced tree.

Theorem 41 *Let $\mathcal{T}$ be a blocked restricted wp-tree on N points. Then 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.*

Proof. Deletions do not use the partial rebuilding technique. Instead, a global rebuilding technique [33] is used, whereby the whole tree is rebuilt during $\Theta(N)$ updates, to keep it balanced. The deletion cost is obtained by adding the cost of searching for the relevant leaf, $O(\log_B N)$, to the global rebuilding cost charged to this deletion, $O(\frac{1}{B}(\log_m n))$.

To insert a point we first search for the relevant leaf using $O(\log_B n)$ I/Os, and then insert the point into that leaf. Let v be the node that becomes out of balance. Rebuilding the tree rooted at v takes $O\left(\frac{w(v)}{B}(\log_m w(v))\right)$ I/Os. This rebuilding cost is charged to the $\Omega(w(v))$ updates that have been performed on the subtree rooted at v since the last rebuilding, resulting in an $O\left(\frac{1}{B}(\log_m w(v))\right)$ I/O cost per update. Since we are charging this cost to the same update for each node on the path from root to the corresponding leaf, the resulting complexity of an insertion is $O\left(\frac{1}{B}(\log_m n)(\log_2 n) + \log_B n\right)$ I/Os.

As n tends 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.

4.2 Logarithmic methods

An alternative method of obtaining a dynamic blocked wp-tree is by adapting the logarithmic method [12, 33] to the external memory setting.

The main idea in the logarithmic method 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 perfectly balanced 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 stored in $\mathcal{T}_j$, $0 \leq j < i$. To adapt this method to work I/O-efficiently, two approaches can be taken: let the ith subset contain 2^i *blocks*, or let it contain 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 also similar to the internal memory case, but we need to maintain a "buffer block" for each tree: all insertions go into this block until the block is full, at which time the rebuilding is performed using all points in this block.

The following theorem summarizes the bounds that we obtain using the logarithmic method.

Theorem 42 *We can maintain a forest of perfectly balanced blocked restricted wp-trees for indexing N points, so 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.*

The proof follows the lines of Overmars [33] and is omitted here, for brevity. Note that, for realistic values of n, m and B, we need less than one I/O to insert a point, amortized over $\Theta(N)$ insertions. Compared with partial rebuilding, this method improves a lot the insertion bound, but has worse deletion and point query performance.

Logarithmic method in base B. Arge and Vahrenhold [6] used the logarithmic method to give an I/O-efficient solution to the point location problem. In contrast to the version explained above, in their method each set contains B^i points, rather than 2^i blocks. Following closely the ideas of Arge and Vahrenhold, we obtain another dynamization technique for the blocked wp-tree.

Theorem 43 *We can maintain a forest of perfectly balanced blocked restricted wp-trees for indexing N points, so that insertions take $O((\log_m n)(\log_B n))$ 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.*

Compared with the previous method, the insertion bound here is a factor of $\frac{B}{log_2 B}$ bigger, while the deletion bound is a factor of $\log_2 n$ smaller.

We now focus on the two running examples, the kd-tree and the BBD-tree, and give query and update bounds for them using the three dynamization methods.

4.3 Applications

The kd-tree. We can exploit a property of the kd-tree partitioning method to derive worst-case bounds on the range query performance. This will effectively give us a new linear-space data structure for indexing large sets of points in $\mathbb{R}^d$, on which updates and range queries can be performed I/O-efficiently.

Theorem 44 *1. Using partial rebuilding as the dynamization method, the blocked kd-tree on N points can be used to answer 4-sided range queries in*

$$O\left(\frac{1}{\sqrt{B}}(\sqrt{N}^{\log_{1/\delta} 2}) + K/B\right)$$

I/Os in the worst case, where K is the number of points reported. 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.

2. Using the logarithmic method in base 2 (or in base B), the range query bound is $O(\sqrt{n} + K/B)$ in the worst 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 $O((\log_m n)(\log_2 n))$ I/Os, worst case (or $O(\log_B n)$ I/Os, amortized).

Proof. Let $\mathcal{T}$ be a blocked kd-tree using the *partial rebuilding* dynamization method. The maximum height of the underlying kd-tree is $\log_{1/\delta} N$, as shown in Lemma 22. To simplify the computation, consider the blocked kd-tree $\mathcal{T}'$, which is a *perfectly balanced* blocked kd-tree whose underlying kd-tree has height $\log_{1/\delta} N$, and $\mathcal{T}$ is a subtree of $\mathcal{T}'$. The range query cost on $\mathcal{T}'$ is an upper bound for the cost of the same query on $\mathcal{T}$.

The number of binary nodes visited during a range query on $\mathcal{T}'$ is $O(\sqrt{N}^{\log_{1/\delta} 2} + K)$ [29]. This is obtained by counting the nodes cut by the four edges of the query rectangle—$O(\sqrt{N}^{\log_{1/\delta} 2}$, and adding the "interior nodes", whose corresponding regions are not cut by the query rectangle—$O(K)$. To obtain the number of block nodes cut by each edge we need only observe that $\Omega(\sqrt{B})$ kd-tree nodes from each block node are cut by an axis-orthogonal line.

The proof of the second part of the theorem is also a consequence of the above considerations, since each kd-tree in the logarithmic method is a perfectly balanced tree of height $O(\log_2 N)$.

Remark. Note that, in 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, if the query bound on the static structure is polynomial. If the query bound on the static structure is polylogarithmic, as in our next example, the bound on the dynamic structure increases asymptotically.

The BBD-tree. The BBD-tree is used in [8] to answer $(1+\epsilon)$-approximate nearest neighbor queries. We can answer this type of queries using the blocked BBD-tree I/O-efficiently.

Theorem 45 *1. Using partial rebuilding as the dynamization method, the blocked BBD-tree can be used to answer a $(1+\epsilon)$-approximate nearest neighbor query in*

$$Q_{BBD}(N) = O\left(\frac{c(\epsilon)}{B}(\log_m n)(\log_2 n) + \log_B n\right)$$

I/Os. Insertions take $O\left(\frac{1}{B}(\log_m n)(\log_2 n) + \log_B n\right)$ I/Os, and deletions take $O(\log_B n)$ I/Os.

2. Using the logarithmic method with base 2 (or the log. method with base B), the query 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, amortized (or $O((\log_m n)(\log_B n))$ I/Os amortized), and deletions take $O((\log_m n)(\log_2 n))$ I/Os, worst case (or $O(\log_B n)$ I/Os, amortized).

Proof. Let $\mathcal{T}$ be a blocked BBD-tree using the partial rebuilding method. The algorithm for finding a $(1+\epsilon)$-approximate nearest neighbor is an external memory version of the algorithm proposed by Arya et al [8]. We first find the leaf containing the query point q. Next, in order to enumerate leaves based on their distance[1] from q, we use an external-memory priority queue [14] of nodes, where the priority of a node v is the distance between q and v. The root of the BBD-tree is initially inserted in the priority queue. Then we extract the node v with the lowest priority from the queue and, starting from v, we descend the BBD-tree to the leaf node closest to the query point q. As we descend, we insert the sibling of each visited node into the priority queue. When we reach the leaf node, we compute the distance from the point found inside that leaf node to the query point q. We maintain the closest point p. The search terminates when the distance between the current node and q is bigger than $d(p,q)/(1+\epsilon)$. Arya et al [8] prove that the number of cells visited during this algorithm is at most $c(\epsilon) = \lceil 1 + 12/\epsilon \rceil^2$.

The query performance is obtained by adding the cost of finding the leaf node containing q to the cost of the actual nearest neighbor finding algorithm. The first is $O(\log_B n)$. The second is $c(\epsilon)$ times the cost of determining the nearest cell. Inserting an item and extracting the lowest priority item from the queue takes $O(\frac{1}{B}(\log_m n))$ I/Os, using the external priority queue by Brodal and Katajainen [14]. So the total cost of determining the nearest cell is $O\left(\frac{1}{B}(\log_m n)(\log_2 n) + \log_B n\right)$.

When dynamization is done using the two logarithmic methods, each tree of the forest has to be queried, hence the added factor in the query bound.

References

1. P. K. Agarwal, L. Arge, J. Erickson, P. Franciosa, and J. Vitter. Efficient searching with linear constraints. In *Proc. ACM Symp. Principles of Database Systems*, pages

[1] The distance between a point p and a node is defined as the closest distance between p and any part of the region associated with the node.

169–178, 1998. (Full version with improved results—to appear in *Journal of Computer and System Sciences*—can be obtained from the authors www-pages).
2. A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31:1116–1127, 1988.
3. L. Arge. External memory data structures. In J. Abello, P. M. Pardalos, and M. G. C. Resende, editors, *Handbook of Massive Data Sets*. Kluwer Academic Publishers, 2000. (To appear).
4. L. Arge, K. H. Hinrichs, J. Vahrenhold, and J. S. Vitter. Efficient bulk operations on dynamic R-trees. In *Proc. Workshop on Algorithm Engineering*, 1999.
5. 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.
6. L. Arge and J. Vahrenhold. I/O-efficient dynamic planar point location. In *Proc. ACM Symp. on Computational Geometry*, pages 191–200, 2000.
7. 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.
8. 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.
9. R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.
10. 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 Intl. Conf. on Management of Data*, pages 322–331, 1990.
11. J. L. Bentley. Multidimensional binary search trees used for associative searc hing. *Commun. ACM*, 18(9):509–517, 1975.
12. J. L. Bentley. Decomposable searching problems. *Inform. Process. Lett.*, 8:244–251, 1979.
13. S. Berchtold, C. Böhm, and H.-P. Kriegel. Improving the query performance of high-dimensional index structures by bulk load operations. In *Proc. Conference on Extending Database Technology*, pages 216–230, 1998.
14. G. S. Brodal and J. Katajainen. Worst-case efficient external-memory priority queues. In *Proc. Scandinavian Workshop on Algorithms Theory, LNCS 1432*, pages 107–118, 1998.
15. D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.
16. 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.
17. D. J. DeWitt, N. Kabra, J. M. Patel, and J.-B. Yu. Client-server Paradise. In *Proc. 19th Intl. Conf. on Very Large Databases*, pages 558–569, 1994.
18. 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.
19. V. Gaede and O. Günther. Multidimensional access methods. *Computing Surveys*, 30(2):170–231, 1998.
20. D. Greene. An implementation and performance analysis of spatial data access methods. In *Proc. IEEE International Conference on Data Engineering*, pages 606–615, 1989.
21. 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.
22. A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 47–57, 1985.
23. 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.
24. I. Kamel and C. Faloutsos. On packing R-trees. In *Proc. Intl. Conf. on Information and Knowledge Management*, pages 47–57, 1993.
25. I. Kamel and C. Faloutsos. Hilbert R-tree: An improved R-tree using fractals. In *Proc. IEEE International Conf. on Very Large Databases*, pages 500–509, 1994.
26. P. C. Kanellakis, S. Ramaswamy, D. E. Vengroff, and J. S. Vitter. Indexing for data models with constraints and classes. *Journal of Computer and System Sciences*, 52(3):589–612, 1996.
27. K. V. R. Kanth and A. K. Singh. Optimal dynamic range searching in non-replicating index structures. In *Proc. International Conference on Database Theory*, pages 257–276, 1999.
28. D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming.* Addison-Wesley, Reading MA, second edition, 1998.
29. D. T. Lee and C. K. Wong. Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees. *Acta Inform.*, 9:23–29, 1977.
30. S. T. Leutenegger, M. A. Lopez, and J. Edgington. STR: A simple and efficient algorithm for R-tree packing. In *Proc. Annual IEEE Conference on Data Engineering*, pages 497–506, 1996.
31. 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.
32. J. Nievergelt, H. Hinterberger, and K. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1):257–276, 1984.
33. M. H. Overmars. *The Design of Dynamic Data Structures*, volume 156 of *Lecture Notes Comput. Sci.* Springer-Verlag, Heidelberg, West Germany, 1983.
34. 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, 1984.
35. N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed R-trees. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 17–31, 1985.
36. H. Samet. *The Design and Analyses of Spatial Data Structures.* Addison Wesley, MA, 1989.
37. B. Seeger and H.-P. Kriegel. The buddy-tree: An efficient and robust access method for spatial data base systems. In *Proc. IEEE International Conf. on Very Large Databases*, pages 590–601, 1990.
38. T. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+-tree: A dynamic index for multi-dimensional objects. In *Proc. IEEE International Conf. on Very Large Databases*, pages 507–518, 1987.
39. Y. V. Silva Filho. Average case analysis of region search in balanced k-d trees. *Inform. Process. Lett.*, 8:219–223, 1979.
40. 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.

41. J. van den Bercken, B. Seeger, and P. Widmayer. A generic approach to bulk loading multidimensional index structures. In *Proc. IEEE International Conf. on Very Large Databases*, pages 406–415, 1997.
42. J. S. Vitter. External memory algorithms and data structures. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*. American Mathematical Society Press, 1999.
43. J. S. Vitter. Online data structures in external memory. In *Proc. Annual International Colloquium on Automata, Languages, and Programming, LNCS 1644*, pages 119–133, 1999.