External Memory Geometric Data Structures

Lars Arge[*]

Department of Computer Science
University of Aarhus and Duke University
large@daimi.au.dk

1 Introduction

Many modern applications store and process datasets much larger than the main memory of even state-of-the-art high-end machines. Thus massive and dynamically changing datasets often need to be stored in space efficient data structures on external storage devices such as disks. In such cases the Input/Output (or I/O) communication between internal and external memory can become a major performance bottleneck. Many massive dataset applications involve geometric data (for example points, lines, and polygons) or data that can be interpreted geometrically. Such applications often perform queries that correspond to searching in massive multidimensional geometric databases for objects that satisfy certain spatial constraints. Typical queries include reporting the objects intersecting a query region, reporting the objects containing a query point, and reporting objects near a query point.

While development of practically efficient (and ideally also multi-purpose) external memory data structures (or *indexes*) has always been a main concern in the database community, most data structure research in the algorithms community has focused on worst-case efficient internal memory data structures. Recently however, there has been some cross-fertilization between the two areas. In these lecture notes we discuss some of the recent advances in the development of worst-case efficient external memory geometric data structures. We will focus on fundamental dynamic structures for one- and two-dimensional orthogonal range searching, and try to highlight some of the fundamental techniques used to develop such structures.

Accurately modeling memory and disk systems is a complex task. The primary feature of disks we want to model is their extremely long access time relative to that of internal memory. In order to amortize the access time over a large amount of data, typical disks read or write large blocks of contiguous data at once and therefore the standard two-level disk model has the following parameters:

[*] This work was supported in part by the US National Science Foundation through RI grant EIA–9972879, CAREER grant CCR–9984099, ITR grant EIA–0112849, and U.S.-Germany Cooperative Research Program grant INT–0129182, by the US Army Research Office through grant W911NF-04-1-0278, and by an Ole Rømer Scholarship from the Danish National Science Research Council.

$$
\begin{aligned}
N &= \text{number of objects in the problem instance;}\\
T &= \text{number of objects in the problem solution;}\\
M &= \text{number of objects that can fit into internal memory;}\\
B &= \text{number of objects per disk block;}
\end{aligned}
$$

where $B^2 \leq M < N$. An *I/O operation* (or simply *I/O*) is the operation of reading (or writing) a block from (or into) disk. Refer to Figure 1. Computation can only be performed on objects in internal memory. The measures of performance are the number of I/Os used to solve a problem, the amount of space (disk blocks) used, and sometimes the internal memory computation time.

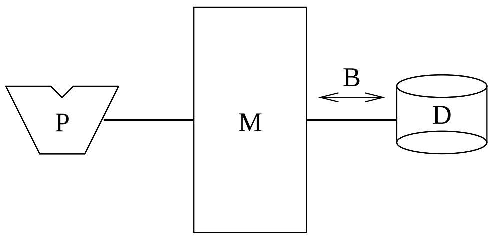

Fig. 1. Disk model. An I/O moves B contiguous elements between disk and main memory (of size M).

Several authors have considered more accurate and complex multi-level memory models than the two-level model. An increasingly popular approach to increase the performance of I/O systems is to use several disks in parallel so work has especially been done in multi disk models. We will concentrate on the two-level one-disk model, since the data structures and data structure design techniques developed in this model often work well in more complex models.

Outline of note. The rest of this note is organized as follows. In Section 2 we discuss the B-tree, the most fundamental (one-dimensional) external data structure. In Sections 3 to 5 we then discuss variants of B-trees, namely weight-balanced B-tress, persistent B-trees, and buffer-tress. In Section 6 we discuss the interval stabbing problem, which illustrates many of the important techniques and ideas used in the development of I/O-efficient data structures for higher-dimensional problems. In Section 7 and Section 8 we discuss data structures for 3-sided and general (4-sided) two-dimensional orthogonal range searching, respectively. Throughout the note we assume that the reader is familiar with basic internal memory data structures and design and analysis methods, such as balanced search trees and amortized analysis

Remarks. Ruemmler and Wilkes [69] discuss modern disk systems and why they are hard to model accurately. The two-level disk model were introduced by

Aggarwal and Vitter [11]; see also e.g. [80, 57]. For convenience we in this note assume that $M \geq B^2$ (such that $M/B \geq B$) instead of the normal assumption that $M \geq 2B$; all the structures we discuss can be modified to work under the weaker assumption. Aggarwal and Vitter also showed that external sorting requires $\Theta(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os. For a discussion of parallel disk result see e.g. the recent survey by Vitter [79]. A somewhat more comprehensive survey of external geometric data structures than the one given in these notes can be found in a recent survey by the author [12]. While the focus of this note and the above surveys are on worst-case efficient structures, there are many good reasons for developing simpler (heuristic) and general purpose structures without worst-case performance guarantees. A large number of such structures have been developed in the database community. See for example the surveys in [10, 46, 64].

2 B-trees

The B-tree is the most fundamental external memory data structure. It corresponds to an internal memory balanced search tree. It uses linear space—$O(N/B)$ disk blocks—and supports insertions and deletions in $O(\log_B N)$ I/Os. One-dimensional range queries, asking for all elements in the tree in a query interval $[q_1, q_2]$, can be answered in $O(\log_B N + T/B)$ I/Os, where T is the number of reported elements. The space, update, and query bounds obtained by the B-tree are the bounds we would like to obtain in general for more complicated problems. The bounds are significantly better than the bounds we would obtain if we just used an internal memory data structure and virtual memory. The $O(N/B)$ space bound is obviously optimal and the $O(\log_B N + T/B)$ query bound is optimal in a comparison model of computation. Note that the query bound consists of an $O(\log_B N)$ search-term corresponding to the familiar $O(\log_2 N)$ internal memory search-term, and an $O(T/B)$ reporting-term accounting for the $O(T/B)$ I/Os needed to report T elements.

B-trees come in several variants, which are all special cases of a more general class of trees called (a, b)-trees:

Definition 1. *A tree $\mathcal{T}$ is an (a, b)-tree ($a \geq 2, b \geq 2a - 1$) if the following conditions hold:*

- *All leaves of $\mathcal{T}$ are on the same level and contain between a and b elements.*
- *Except for the root, all nodes have degree between a and b (contain between $a - 1$ and $b - 1$ elements)*
- *The root has degree between 2 and b (contain between 1 and $b - 1$ elements).*

Normally, the N data elements are stored in the leaves (in sorted order) of an (a, b)-tree $\mathcal{T}$, and elements in the internal nodes are only used to guide searches. This way $\mathcal{T}$ use linear space and has height $O(\log_a N)$. To answer a *range query* $[q_1, q_2]$, we first search down $\mathcal{T}$ for q_1 and q_2 and then we report the elements in the leaves between the leaves containing q_1 and q_2. If we choose $a, b = \Theta(B)$ each node and leaf can be stored in $O(1)$ disk blocks. Thus we obtain a tree of

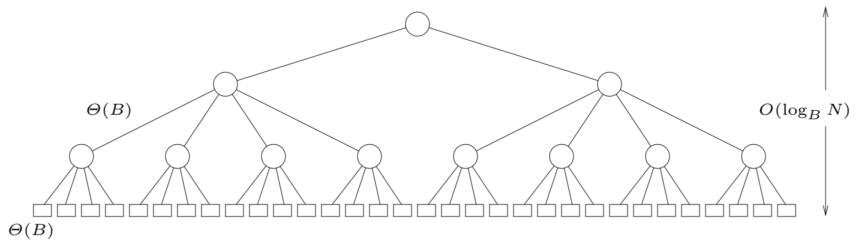

Fig. 2. B-tree. All internal nodes (except possibly the root) have fan-out $\Theta(B)$ and there are $\Theta(N/B)$ leaves. The tree has height $O(\log_B N)$.

height $O(\log_B N)$ using $O(N/B)$ disk blocks, where a query can be perform in $O(\log_B N + T/B)$ I/Os. Refer to Figure 2.

To *insert* an element x in an (a, b)-tree $\mathcal{T}$ we first searching down $\mathcal{T}$ for the relevant leaf u and insert x in u. If u now contains $b + 1$ elements we *split* it into two leaves u' and u'' with $\lceil \frac{b+1}{2} \rceil$ and $\lfloor \frac{b+1}{2} \rfloor$ elements respectively. Both new leaves now have between $\lfloor \frac{b+1}{2} \rfloor \geq a$ (since $b \geq 2a - 1$) and $\lceil \frac{b+1}{2} \rceil \leq b$ (since $b \geq 2a - 1 \geq 3$) elements. Then we remove the reference to u in $parent(u)$ and insert references to u' and u'' instead (that is, we insert a new routing element in $parent(u)$). If $parent(u)$ now has degree $b + 1$ we recursively split it. Refer to Figure 3. This way the need for a split may propagate up through $O(\log_a N)$ nodes of the tree. A new (degree 2) root is produced when the root splits and the height of the tree grows by one.

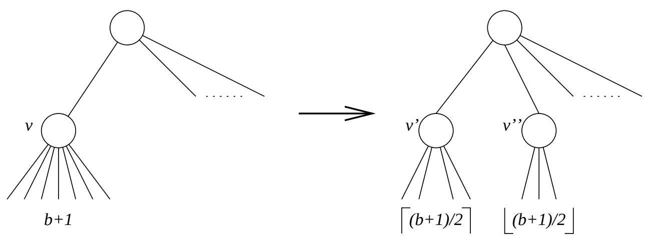

Fig. 3. Splitting degree $b + 1$ node v (or leaf with $b + 1$ elements) into nodes v' and v''.

Similarly, to *delete* an element x we first find and remove x from the relevant leaf u. If u now contains $a - 1$ elements we *fuse* it with one of its siblings u', that is, we delete u' and inserts its elements in u. If this results in u containing more then b (but less than $a - 1 + b < 2b$) elements we split it into two leaves. As before, we also update $parent(u)$ appropriately. If u was split, the routing elements in $parent(u)$ are updated but its degree remains unchanged; thus effectively we have taken an appropriate number of elements from u' and inserted them in u—also called a *share*. Otherwise, the degree decreases by one and we may need to recursively fuse $parent(u)$ with a sibling. Refer to Figure 4. As before, fuse

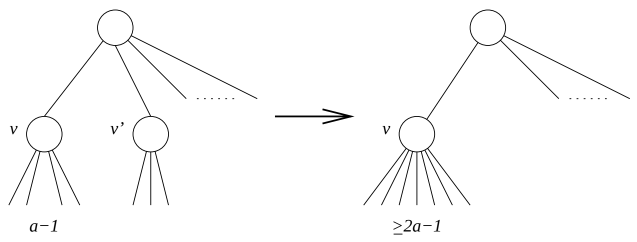

Fig. 4. Fusing degree $a-1$ node v (or leaf with $a-1$ elements) with sibling v'.

operations may propagate up through $O(\log_a N)$ nodes. If this results in the root having degree one, it is removed and its child becomes the new root, that is, the height of the tree decreases by one.

A single update in an (a, b)-tree can at most cause $O(\log_a N)$ rebalancing operations (splits or fuses). In fact, its easy to see that if $b = 2a - 1$ there exists a sequence of updates where each update results in $\Theta(\log_a N)$ rebalancing operations. Refer to Figure 5. However, if $b = 4a$ we only need to modify the delete algorithm slightly to obtain a structure where an update can only cause $O(\frac{1}{a}\log_a N)$ rebalancing operations amortized: A new leaf u constructed when performing a split during insertion rebalancing contains approximately $4a/2 = 2a$ elements. Thus a rebalancing operation will not be needed on u until at least a updates (a deletions or $2a$ insertions) have been performed in it. Just after a fuse of a leaf u during delete rebalancing, u contains between $a - 1 + a = 2a - 1$ and $a - 1 + 4a = 5a - 1$ elements. By splitting u if it contains more than $3a$ elements (performing a share), we can guarantee that leaves involved in rebalancing operations during a delete contain between approximately $\frac{3}{2}a$ and $3a$ elements (since $\frac{3}{2}a < 2a - 1 < \frac{5a-1}{2} < 3a$). Thus a rebalancing operation will not be needed for at least $\frac{1}{2}a$ operations ($\frac{1}{2}a$ deletions or a insertions). Therefore $O(N/a)$ leaf rebalancing operations are needed during N operations. The result

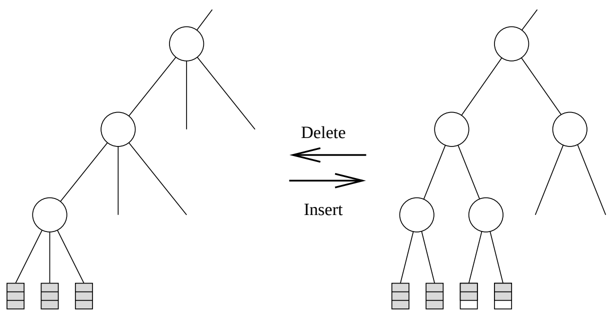

Fig. 5. Alternating insertion and deletion of the same element in an (2,3)-tree can cause $\Theta(\log_a N)$ rebalancing operations each.

then follows, since one such operations trivially can at most lead to $O(\log_a N)$ rebalancing operations on internal nodes.

As mentioned, a B-tree is basically just an (a,b)-tree with $a,b = \Theta(B)$. Sometimes, especially in the context of external data structures, it can be useful to use variants of (a,b)-trees where the constraints on the degree of internal nodes and on the number of elements in the leaves are not the same. For the purpose of these notes we therefore define the following.

Definition 2. *$\mathcal{T}$ is a B-tree with branching parameter b and leaf parameter k ($b,k \geq 8$) if the following conditions hold:*

- *All leaves of $\mathcal{T}$ are on the same level and contain between $\frac{1}{4}k$ and k elements.*
- *Except for the root, all nodes have degree between $\frac{1}{4}b$ and b*
- *The root has degree between 2 and b.*

If we choose $k = \Omega(B)$ we obtain a B-tree that can be stored in a linear number of disk blocks. By modifying the update algorithms as discussed above for the leaves as well as the nodes one level above the leaves, we then obtain the following.

Theorem 1. *An N-element B-tree $\mathcal{T}$ with branching parameter b and leaf parameter $k = \Omega(B)$ uses $O(N/B)$ space, has height $O(\log_b \frac{N}{B})$, and the amortized number of internal node rebalancing operations (split/fuse) needed after an update is $O(\frac{1}{b \cdot k} \log_b \frac{N}{B})$.*

Corollary 1. *A B-tree with branching parameter $\Theta(B^c)$, $0 < c \leq 1$, and leaf parameter $\Theta(B)$ uses $O(N/B)$ space, supports updates in $O(\log_{B^c} N) = O(\log_B N)$ I/Os and queries in $O(\log_B N + T/B)$ I/Os.*

In internal memory, an N element search tree can be built in $O(N \log N)$ time, which is optimal, simply by inserting the elements one by one. In external memory we would use $O(N \log_B N)$ I/Os to construct a B-tree using the same method. Interestingly, this is not optimal since sorting N elements in external memory takes $\Theta(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os. We can construct a B-tree in the same bound by first sorting the elements and then construct the tree level-by-level bottom-up.

Theorem 2. *An N-element B-tree $\mathcal{T}$ with branching parameter b and leaf parameter $k = \Omega(B)$ can be constructed in $\Theta(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os.*

Remarks. B-trees with elements in the leaves, as the ones described here, are normally called B^+-trees in the database literature. Good references on B-tree variants and properties include [24, 36, 57]; (a,b)-tree properties are discussed extensively in [53]. As mentioned, a single update in an (a,b)-tree with $b = 2a-1$ can cause $\Theta(\log_a N)$ rebalancing operations. However, in an (a,b)-tree with $b \geq 2a$ the number of rebalancing operations caused by an update can be reduced to $O(1/a)$ amortized [53]. In this note we only need the weaker $O(\frac{1}{a} \log_a N)$ bound that is easy to shown for $b = 4a$.

Several alternative B-tree balancing schemes, such as level-balance [1] and weigh-balance [23], as well as several B-tree extensions, such as persistent B-trees [39, 25, 77], buffer-trees [13], and string B-trees [44], have been developed. In the following sections we discuss some of these structures (weight-balanced B-trees, persistent B-trees, and buffer-trees).

3 Weight-balanced B-trees

The *weight* $w(v)$ of node v in a search tree $\mathcal{T}$ is defined as the the number of elements in the leaves of the subtree rooted in v. When secondary structures are attached to internal nodes, it is often useful if rebalancing operations are not performed too often on heavy nodes. It is especially useful to use a search tree where a node v of weight $w(v)$ is only involved in a rebalancing operation once for every $\Omega(w(v))$ updates below it. Unfortunately, (a, b)-trees (and consequently B-trees) do not have this property. Instead weight-balanced B-trees, balanced using weight constraints rather than degree constraints, are used. Refer to Figure 6.

Definition 3. *$\mathcal{T}$ is a weight-balanced B-tree with branching parameter b and leaf parameter k ($b, k \geq 8$) if the following conditions hold:*

- *All leaves of $\mathcal{T}$ are on the same level (level 0) and each leaf u has weight $\frac{1}{4}k \leq w(u) \leq k$.*
- *An internal node v on level l has weight $v(w) \leq b^l k$.*
- *Except for the root, an internal node v on level l has weight $v(w) \geq \frac{1}{4}b^l k$.*
- *The root has more than one child.*

The weight constraints on nodes of a weight-balanced B-tree actually implies bounded degree similar to a B-tree; a node minimally has degree $\frac{1}{4}b^l k / b^{l-1}k = \frac{1}{4}b$ and maximally degree $b^l k / \frac{1}{4}b^{l-1}k = 4b$. Thus a weight-balanced B-tree on N elements has height $O(\log_b \frac{N}{k})$. Weight-balanced B-tree are similar to normal B-trees in that all leaves are on the same level and, as we will discuss below, rebalancing can be done by splitting and fusing nodes. However, in some sense weight-balanced B-trees are more balanced than normal B-trees. While the children of a node v on level l of a weight-balanced B-tree are of approximately the

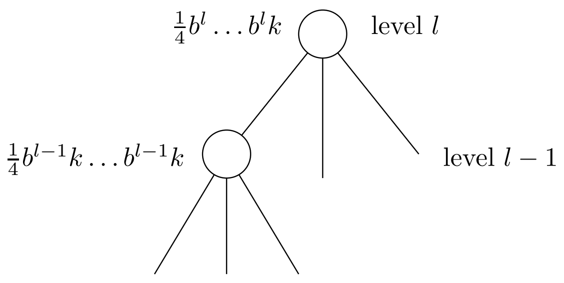
$\frac{1}{4}b^l \ldots b^l k$
level l
$\frac{1}{4}b^{l-1}k \ldots b^{l-1}k$
level $l-1$

Fig. 6. Level l node in weight balanced B-tree has weight between $\frac{1}{4}b^l k$ and $b^l k$.

same weight $\Theta(b^{l-1}k)$, their weight can differ by an exponential factor in l in a B-tree.

To *insert* an element x in a weight-balanced B-tree $\mathcal{T}$ we first search down $\mathcal{T}$ for the relevant leaf u and insert x. After this some of the nodes on the path from u to the root of $\mathcal{T}$ may be out of balance, that is, a node on level l might have weight $b^l k + 1$ (leaf u can have weight $b^0 k + 1$). To rebalance $\mathcal{T}$ we visit these $O(\log_b N)$ nodes starting with u and working towards the root: If a node node v on level l has weight $b^l k + 1$ we would like to split it into two nodes v' and v'' of weight $\lfloor \frac{1}{2}(b^l k + 1) \rfloor$ and $\lceil \frac{1}{2}(b^l k + 1) \rceil$. However except for u, a perfect split is generally not possible since we have to perform the split so that v' gets the, say i, first (leftmost) of v's children and v'' gets the rest of the children. However, since nodes on level $l-1$ have weight at most $b^{l-1}k \leq \frac{1}{8}b^l k$, we can always find an i such that if we split at the i'th child the weight of both v' and v'' is between $\frac{1}{2}b^l k - b^{l-1}k \geq \frac{3}{8}b^l k$ ($> \frac{1}{4}b^l k$) and $\frac{1}{2}b^l k + b^{l-1}k + 1 \leq \frac{5}{8}b^l k + 1$ ($< b^l k$). Since the new nodes v' and v'' are relatively rebalanced, $\Omega(b^l k)$ updates have to be performed below each such node before it needs to be rebalanced again. More precisely, at least $\frac{1}{8}b^l k + 1$ deletions or $\frac{3}{8}b^l k$ insertions have to be performed below v' or v'' before they need to be rebalanced again.

Similarly, to perform a *deletion* we first delete x from the relevant leaf u. Then we rebalance the relevant nodes on the path to the root of $\mathcal{T}$. If a node v on level l has weight $\frac{1}{4}b^l k - 1$ we fuse it with one of its siblings. The resulting node v' has weight at least $\frac{1}{4}b^l k - 1 + \frac{1}{4}b^l k = \frac{1}{2}b^l k - 1$ and at most $\frac{1}{4}b^l k - 1 + b^l k = \frac{5}{4}b^l k - 1$. If v' has weight greater than $\frac{7}{8}b^l k$ we split it into two nodes (and thus effectively perform a share) with weight at least $\frac{7}{16}b^l k - b^{l-1}k - 1 \geq \frac{5}{16}b^l k - 1$ ($> \frac{1}{4}b^l k$) and at most $\frac{5}{8}b^l k + b^{l-1}k \leq \frac{6}{8}b^l k < \frac{7}{8}b^l k$ ($< b^l k$). Again $\Omega(b^l k)$ updates ($\frac{1}{8}b^l k + 1$ insertions or $\frac{1}{16}b^l k$ deletions) need to be performed below a new node before it needs rebalancing.

Theorem 3. *An N-element weight-balanced B-tree with branching parameter b and leaf parameter $k = \Omega(B)$ uses $O(N/B)$ space, has height $O(\log_b \frac{N}{B})$, and the number of rebalancing operations (splits of fuses) needed after an update is bounded by $O(\log_b \frac{N}{B})$. Between two consecutive rebalancing operations on a node v, $\Omega(w(v))$ updates have to be performed in leaves below v.*

As previously, we can construct a weight-balanced B-tree in $\Theta(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os by first sorting the elements and then constructing the tree level-by-level bottom-up.

Corollary 2. *A weight-balanced B-tree with branching parameter $\Theta(B^c)$, $0 < c \leq 1$, and leaf parameter $\Theta(B)$ uses $O(N/B)$ space and can be constructed in $\Theta(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os. It supports updates in $O(\log_B N)$ I/Os and queries in $O(\log_B N + T/B)$ I/Os.*

Remarks. The weight-balance B-tree was introduced by Arge and Vitter [23]. The structure resembles the *k-fold tree* of Willard [81]. It can be viewed as an external version of the BB[α]-trees [63], which are binary and (normally) rebalanced using rotations and therefore not efficient in external memory. Weight-balanced

B-trees combines the useful properties of B-trees and BB[α]-trees. They are defined using weight constraints like BB[α]-trees and therefore they have the useful weight-property (node v of weight $w(v)$ only rebalanced every $\Omega(w(v))$ operations), while at the same time being balanced using normal B-tree operations (splitting and fusion of nodes). Like BB[α]-trees, the weight-property means that the weight-balanced B-tree can also be rebalanced using *partial-rebuilding* (see e.g. [65]): Instead of splitting or fusing nodes on the path to the root after performing an update in a leaf, we can simply rebuild the tree rooted in the highest unbalanced node on this path. Since the (sub-) tree can easily be rebuilt in a linear number of I/Os we obtain an $O(\log_B N)$ amortized update bound.

4 Persistent B-trees

In some database applications one need to be able to update the current database while querying both the current and earlier versions of the database (data structure). One simple but very inefficient way of supporting this functionality is to copy the whole data structure every time an update is performed. Another and much more efficient way is through the (partially) *persistent* technique, also sometimes referred to as the *multiversion* method. Instead of making copies of the structure, the idea in this technique is to maintain one structure at all times but for each element keep track of the time interval (*existence interval*) it is really present in the structure.

Definition 4. *A persistent B-tree $\mathcal{T}$ with parameter $b > 16$ consists of a directed graph $\mathcal{T}_\mathcal{G}$ and a B-tree $\mathcal{T}_\mathcal{B}$ with the following properties:*

- *Each node of $\mathcal{T}_\mathcal{G}$ contains a number of elements each augmented with an* existence interval *defined by an insertion and a deletion* time *(or* version*).*
- *For any time/version t, the nodes of $\mathcal{T}_\mathcal{G}$ with at least one element with existence interval containing t form a B-tree with leaf and branching parameter b.*
- *$\mathcal{T}_\mathcal{B}$ is a B-tree with leaf and branching parameter b on the indegree 0 node (roots) of $\mathcal{T}_\mathcal{G}$ ordered by minimal existence interval beginning time (insert version).*

If we use $b = B$, we can easily *query* any version of a persistent B-tree in $O(\log_B N + T/B)$ I/Os: To perform a query at time t, we simply find the appropriate root node in $\mathcal{T}_\mathcal{G}$ using $\mathcal{T}_\mathcal{B}$ and then we perform the search in $\mathcal{T}_\mathcal{G}$ as in a normal B-tree. Below we discuss how to update the most recent (the current) version of a persistent B-tree efficiently. We will say that an element is *alive* at time t (version t) if t is in its existence interval; otherwise we call it *dead*. In order to insure linear space use, we will maintain the *new-node invariant* that whenever a new node is created it contains between $\frac{3}{8}b$ and $\frac{7}{8}b$ alive elements (and no dead elements); note that this also means that its a valid parameter b B-tree node.

To *insert* a new element x in the current version t of a persistent B-tree we first search for the relevant leaf u (in the B-tree defined by $\mathcal{T}_\mathcal{G}$ at time t) and insert x. If u now contains $b+1$ elements (dead or alive) we perform a *version-split*: We make a copy of the *alive* elements in u and mark all elements in u as deleted at time t (i.e. we delete u from the B-tree "embedded" in $\mathcal{T}$ at the current time). If the number of copied elements is between $\frac{3}{8}b$ and $\frac{7}{8}b$ we then simply create one new leaf node u' with these elements and recursively update $parent(u)$ by persistently deleting the reference to u (as described below) and inserting a reference to u'. If the number of copied elements is greater than $\frac{7}{8}b$ we instead create two new leaves u' and u'' with approximately half of the elements each, and update $parent(u)$ recursively in the appropriate way. The two new leaves u' and u'' both contain at most $\lceil \frac{b+1}{2} \rceil < \frac{7}{8}b$ elements and more than $\lfloor \frac{1}{2}\frac{7}{8}b \rfloor > \frac{3}{8}b$ elements so the new-node invariant is fulfilled. Note now this corresponds to a split in an (a, b)-tree. Finally, if the number of copied elements is smaller than $\frac{3}{8}b$ we perform a version-split on a sibling u to obtain between $\frac{1}{4}b$ and b other alive elements, for a total of between $\frac{1}{4}b + \frac{1}{4}b = \frac{1}{2}b$ and $\frac{3}{8}b + b = \frac{11}{8}b$ elements. If we have less than $\frac{7}{8}a$ elements we simply create a new leaf node u'. Otherwise, we create two new leaves u' and u' containing between $\lfloor \frac{1}{2}\frac{7}{8}b \rfloor > \frac{3}{8}b$ and $\lceil \frac{1}{2}\frac{11}{8}b \rceil < \frac{7}{8}b$ elements (perform a split). This way the new-node invariant is fulfilled. Finally, we recursively update $parent(u)$ appropriately. The first case corresponds to a fuse in the B-tree embedded in $\mathcal{T}$ at time t, while the second corresponds to a share. It is easy to see that nodes of $\mathcal{T}_{\mathcal{G}}$ with live elements at the current time form a B-tree with leaf and branching parameter b, that is, that $\mathcal{T}$ is a correct persistent B-tree.

A *deletion* is handled similarly to an insertion. First we find the relevant element x in a leaf u and mark it as deleted. This may result in u containing $\frac{1}{4}b - 1$ alive elements and therefore we perform what corresponds to a fuse or share as previously: We perform a version-split on u and one of its siblings to obtain a total of between $\frac{1}{4}b + \frac{1}{4}b - 1 = \frac{1}{2}b - 1$ and $b + \frac{1}{4}b - 1 = \frac{5}{4}b - 1$ alive elements. We then either create a new leaf u' with the obtained elements, or split them and create two new leaves u' and u'' precisely as previously, fulfilling the new-node invariant. We also recursively update $parent(u)$ as previously. Again it is easy to see that $\mathcal{T}$ is a correct persistent B-tree after the deletion.

An insertion or deletion after N operations on an initially empty persistent B-tree performs $O(\log_b N)$ rebalancing operations since the rebalancing at most propagates from u to the root of the B-tree corresponding to the current version. To see that the update algorithms (the new-node invariant) ensure linear space use, first note that a rebalance operation on a leaf creates at most two new leaves. Once a new leaf u' is created, at least $\frac{1}{8}b$ updates have to be performed on u' before a rebalance operation is needed on it again. Thus at most $2\frac{N}{b/8}$ leaves are created during N updates. Similarly, we can argue that the number of leaf version-splits during N updates is $2\frac{N}{b/8}$ (two version splits might only create one new leaf). Each time a leaf is created or a leaf version-split performed, a corresponding insertion or deletion is performed recursively one level up the tree. Thus by the same argument the number of nodes created one level up the tree is bounded by $2^2\frac{N}{(\frac{1}{8}b)^2}$. By induction, the number of nodes created l levels up the tree is bounded by $2^{l+1}\frac{N}{(\frac{1}{8}b)^{l+1}}$. The total number of nodes constructed over N updates is therefore bounded by $\frac{2N}{\frac{1}{8}b}\sum_{l=0}^{\log_B N}(\frac{2}{\frac{1}{8}b})^l$, which is $O(\frac{N}{b})$ since $b > 16$.

Theorem 4. *A persistent B-tree with parameter $\Theta(B)$ can be implemented such that after N insertions and deletions in an initially empty structure, it uses $O(N/B)$ space and supports range queries in any version in $O(\log_B N + T/B)$ I/Os. An update can be performed on the newest version in $O(\log_B N)$ I/Os.*

Remarks. General techniques for making data structures persistent were developed by Driscoll et al. [39]; see also [72]. They can be used to develop persistent B-trees. Partially persistent B-trees (and in general partially persistent structures) are sometimes referred to as *multiversion B-trees* (multiversion structures) [25, 77]. Our description of persistent B-trees follow that of Arge et al. [14] and Becker et al. [25].

Several times in later sections we will construct a data structure by performing N insertions and deletions on an initially empty persistent B-tree, and then use the resulting (static) structure to answer queries. Using the update algorithms discussed in this section the construction takes $O(N \log_B N)$ I/Os. Unlike B-trees and weight-balanced B-trees, it seems hard to construct the structure efficiently bottom-up. However, as discussed in the next section, the so-called buffer-tree technique can be used to improve the $O(N \log_B N)$ bound to $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$. Utilizing the *distribution-sweeping* technique, Goodrich et al. [47] showed how to construct a persistent B-tree structure (different from the one described above) in the same bound.

The persistent B-tree as described, as well as the structure by Goodrich et al. [47], requires that every pair of elements in the structure are comparable—even a pair of elements not present in the structure at the same time. This sometimes create problems, e.g. when working with geometric objects (such as line segments). Arge et al. [14] described a modified version of the persistent B-tree that only requires that elements present at the same time are comparable. Unfortunately, this structure cannot be constructed efficiently using the buffer-tree technique.

5 Buffer trees

In internal memory we can sort N elements in optimal $O(N \log N)$ time using $\Theta(N)$ operations on a dynamic balanced search tree. Using the same algorithm and a B-tree in external memory results in an algorithm using $O(N \log_B N)$ I/Os. This is a factor of $\frac{B \log_B N}{\log_{M/B}(N/B)}$ from optimal. In order to obtain an optimal sorting algorithm we need a search tree that supports updates in $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ I/Os. In general, if we were able to perform insertions on various structures, for example on a persistent B-tree, in $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ I/Os, we would be able to construct the structures in the optimal $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os simply by performing N insertions. In this section we discuss the *buffer tree* technique that can be used to obtain this bound amortized by introducing "laziness" in the update algorithms and utilizing the large main memory to process (portions of) a large number of updates simultaneously.

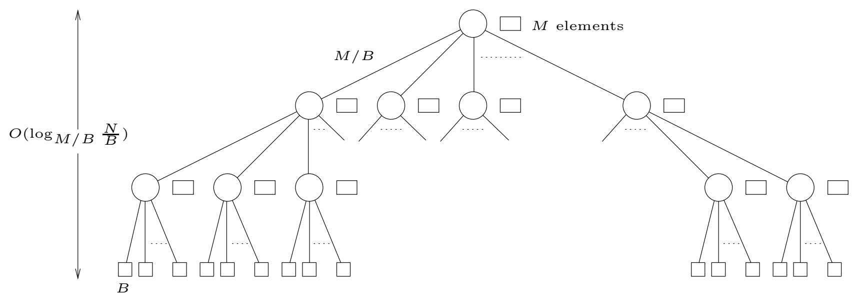

Fig. 7. Buffer tree.

The basic buffer tree is simply a B-tree with branching parameter M/B and leaf parameter B, where each internal node has been augmented with a *buffer* of size M. Refer to Figure 7. The idea is then to perform operations in a "lazy" manner using the buffers. For example, to perform an insertion we do not search all the way down the tree for the relevant leaf. Instead, we simply insert it in the buffer of the root. When a buffer "runs full" the elements in the buffer are then "pushed" one level down to buffers on the next level. We can perform such a *buffer-emptying process* in $O(M/B)$ I/Os since the elements in the buffer fit in main memory and the fan-out of the tree is $O(M/B)$. If the buffer of any of the nodes on the next level becomes full by this process, the buffer-emptying process is applied recursively. Since we push $\Theta(M)$ elements one level down the tree using $O(M/B)$ I/Os (that is, we use $O(1)$ I/Os to push one block one level down), we can argue that every block of elements is touched a constant number of times on each of the $O(\log_{M/B} \frac{N}{B})$ levels of the tree. Thus, disregarding rebalancing, inserting N elements requires $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os in total. Below we discuss the buffer technique in more detail and show how to perform both insertions and deletions. Note that as a result of the laziness, we can have several insertions and deletions of the same element in the tree at the same time and we therefore "time stamp" elements when they are inserted in the root buffer.

Definition 5. *A basic buffer tree $\mathcal{T}$ is*

- *A B-tree with branching parameter $\frac{M}{B}$ and leaf parameter B where;*
- *Each internal node has a buffer of size M.*

We perform an *insertion* or *deletion* on a buffer tree $\mathcal{T}$ as follows: We construct an element consisting of the element in question, a time stamp, and an indication of whether the element corresponds to an insertion or a deletion. When we have collected B such elements in internal memory we insert them in the buffer of the root. If this buffer now contains more than M elements we perform a buffer-emptying process. During such a process, and the resulting recursive buffer-emptying processes, buffers can contain many more than M elements (when many elements from a buffer is distributed to the same child). However, by distributing elements in sorted order, we maintain that a full buffer consists of at most M unsorted elements followed by a list of sorted elements; it can thus be sorted in a linear number of I/Os by first loading and sorting the M unsorted elements and then merging them with the list of sorted elements. We perform buffer-emptying processes differently on *internal nodes* (nodes that do not have leaves as children) and on *leaf nodes*. On internal nodes we basically proceed as described above: We first sort the elements while removing corresponding insert and delete element (with time stamps in the correct order). Then we in a simple scan distribute the remaining elements to buffers one level down. Finally, we apply the buffer-emptying process on children with full buffers, *provided* they are internal nodes. We proceed to empty leaf node buffers only after finishing *all* internal node buffer-emptying processes. The reason is of course that a buffer-emptying process on a leaf node may result in the need for rebalancing. By only emptying leaf nodes after all internal node buffer-emptying processes have been performed we prevent rebalancing and buffer-emptying processes from interfering with each other.

We empty all relevant leaf nodes buffers one-by-one while maintaining the *leaf-emptying invariant* that all buffers of nodes on the path from the root of $\mathcal{T}$ to a leaf node with full buffer are empty. The invariant is obviously true when we start emptying leaf buffers. To empty the buffer of a leaf node v we first sort the buffer while removing matching inserts and deletes as in the internal node case. Next we merge this list with the elements in the, say k, leaves below v, again removing matching elements. Then we replace the elements in the k leaves with the resulting set of sorted elements (and update the "routing elements" in v). If we do not have enough elements to fill the k leaves we instead insert "placeholder-elements". If we have more than enough elements, we insert the remaining elements one-by-one and rebalance the B-tree. We can rebalance as normally using splits (Figure 3), since the leaf-emptying invariant insures that all nodes from u to the root of $\mathcal{T}$ have empty buffers.

After we have emptied all leaf-node buffers we remove the placeholder-elements one-by-one. We do so basically as in a normal B-tree, except that we slightly modify the rebalancing operations. Recall that rebalancing after a delete in a leaf u involves fusing (Figure 4) a number of nodes on the path to the root of $\mathcal{T}$ with one of their siblings (possibly followed by a split). The leaf-emptying invariant ensures that a node v on the path from u to the root has an empty buffer, but its siblings may not have empty buffers. Therefore we perform a buffer-emptying processes on v's (one or two) immediate sibling(s) before performing the actual fuse. The emptying of the buffer of a sibling node v' can result in leaf node buffer's running full; in such cases the leaf-emptying invariant is still fulfilled since all nodes on the path from the parent of v' have empty buffers (since it is also the parent of v). We empty all relevant buffers, excluding leaf-node buffers, before performing the fuse on v. Note that the special way of handling deletes with placeholder-elements ensures that we are only in the process of handling (rebalancing after) one delete operation at any given time (insert rebalancing, splits, after a leaf buffer emptying cannot result in buffer-emptying processes). Note also that we empty the buffers of both immediate siblings of v because insert rebalancing may result in one (but not both) of them not having the same parent as v (if the parent of v splits).

A buffer-emptying process on a node containing X elements, not counting recursive buffer-emptyings or rebalancing, takes $O(X/B + M/B)$ I/Os: Scanning the X elements takes $O(X/B)$ I/Os and distributing them to the $\Theta(M/B)$ buffers one level down (in the internal node case) or scanning the $\Theta(M)$ elements below it (in the leaf node case) takes another $O(X/B + M/B)$ I/O. Thus the cost of emptying a full buffer is $O(X/B + M/B) = O(X/B)$ I/Os, and the argument in the beginning of this chapter can be used to show that the total cost of all full buffer-emptying, not counting rebalancing, is $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os. By Theorem 1 the total number of internal node rebalancing operations performed during N updates is $O(\frac{N}{B\cdot M/B}\log_{M/B}\frac{N}{B})$. Since each such operation takes $O(M/B)$ I/Os (to empty a non-empty buffer), the total cost of the rebalancing is also $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os.

Theorem 5. *The total cost of a sequence of N update operation on an initially empty buffer tree is $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os.*

In order to use the buffer tree in a simple sorting algorithm we need an empty operation that empties all buffers and then reports the elements in the leaves in sorted order. All buffers can be emptied simply by performing a buffer-emptying process on all nodes in the tree in BFS order. As emptying one buffer costs $O(\frac{M}{B})$ I/Os (not counting recursive processes), and as the total number of buffers in the tree is $O(\frac{N}{B}/\frac{M}{B})$, we obtain the following.

Theorem 6. *The total cost of emptying all buffers of a buffer tree after performing N updates on it is $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os.*

Corollary 3. *A set of N elements can be sorted in $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os using a buffer tree.*

Using the buffer tree techniques, a persistent B-tree can be constructed efficiently (while not performing queries), simply by performing the N updates using buffers and then empty all the buffers as above.

Corollary 4. *A sequence of N updates can be performed on an initially empty persistent B-tree (the tree can be constructed) in $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os.*

The buffer tree technique can also be used to develop an efficient priority queue structure. A search tree structure can normally be used to implement a priority queue because the smallest element in a search tree is in the leftmost leaf. The same strategy cannot immediately be used on the buffer tree, since the smallest element is not necessarily stored in the leftmost leaf—smaller elements could reside in buffers of the nodes on the leftmost root-leaf path. However, there is a simple strategy for performing a deletemin operation in the desired amortized I/O bound. We simply perform a buffer-emptying process on all nodes on the path from the root to the leftmost leaf using $O(\frac{M}{B} \cdot \log_{M/B} \frac{M}{B})$ I/Os amortized. Then we delete the $\Theta(\frac{M}{B} \cdot B)$ smallest elements stored in the children (leaves) of the leftmost leaf node and keep them in internal memory. This way we can answer the next $\Theta(M)$ deletemin operations without performing any I/Os. Of course we then also have to update the minimal elements in internal memory as insertions and deletions are performed, but we can do so in a straightforward without performing extra I/Os. Thus, since we use $O(\frac{M}{B} \log_{M/B} \frac{N}{B})$ I/Os to perform $\Theta(M)$ deletemin operations, we obtain the following.

Theorem 7. *Using $O(M)$ internal memory, an arbitrary sequence of N insert, delete and deletemin operations on an initially empty buffer tree can be performed in $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os.*

Remarks. The buffer tree technique was developed by Arge [13] who also showed how the basic buffer tree can support range queries in $O(\frac{1}{B} \log_{M/B} \frac{N}{B} + \frac{T}{B})$ I/Os amortized. The range queries are *batched* in the sense that we do not obtain the result of a query immediately; instead parts of the result will be reported at different times as the query is pushed down the tree. Arge [13] also showed how to implement a buffered segment tree. The buffer tree based priority queue was described by Arge [13]. Note that in this case queries (deletemins) are not batched. By decreasing the fan-out and the size of buffers to $\Theta((M/B)^c)$ for some $0 < c \leq 1$ the buffer tree priority queue can be modified to use only $\Theta((M/B)^c)$ rather than $\Theta(M/B)$ blocks in internal memory. This is useful in applications that utilize more than one priority queue (see, e.g., [22]). Using the buffer technique on a heap, Fadel et al. [43] and Kumar and Schwabe [60] developed alternative external priority queues. Using a partial rebuilding idea, Brodal and Katajainen [29] developed a worst-case efficient external priority queue, that is, a structure where a sequence of B operations requires $O(\log_{M/B} \frac{N}{B})$ I/Os worst-case. Using the buffer tree technique on a tournament tree, Kumar and Schwabe [60] developed a priority queue supporting update operations in $O(\frac{1}{B} \log \frac{N}{B})$ I/Os amortized. Note that if the key of an element to be updated is known, the update can be performed in $O(\frac{1}{B} \log_{M/B} \frac{N}{B})$ I/Os on a buffer tree using a deletion and an insertion. The buffer tree technique has also been used on several other data structures, such as SB-trees [16] and R-trees [17, 75].

6 Dynamic interval stabbing: External interval tree

After considering simple one-dimensional problems, we now turn to higher-dimensional problems. In this section we consider the "1.5-dimensional" interval stabbing problem: We want to maintain a dynamically changing set of (one-dimensional) intervals I such that given a query point q we can report all T intervals containing q efficiently.

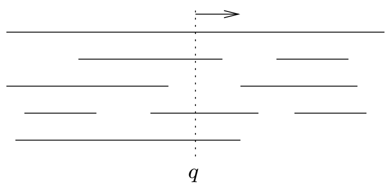

Fig. 8. Static solution to stabbing query problem using persistence.

The static version of the stabbing problem (where the set of intervals is fixed) can easily be solved I/O-efficiently using a sweeping idea and a persistent B-tree. Consider sweeping the N intervals in I along the x-axis starting at $-\infty$, inserting each interval in a B-tree when its left endpoint is reached, and deleting it again when its right endpoint is reached. To answer a stabbing query with q we simply have to report all intervals in the B-tree at "time" q—refer to Figure 8. Thus by Theorem 4 and Corollary 4 we have the following.

Theorem 8. *A static set of N intervals can be stored in a linear space data structures such that a stabbing query can be answered in $O(\log_B N + T/B)$ I/Os. The structure can be constructed in $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os.*

In internal memory, the dynamic interval stabbing problem is solved using an *interval tree.* Such a tree consists of a binary base tree $\mathcal{T}$ on the sorted set of interval endpoints, with the intervals stored in secondary structures associated with internal nodes of the tree. An interval X_v consisting of all endpoints below v is associated with each internal node v in a natural way. The interval X_r of the root r of $\mathcal{T}$ is thus divided in two by the intervals X_{v_l} and X_{v_r} associated with its two children v_l and v_r, and an interval is stored in r if it contains the "boundary" between X_{v_l} and X_{v_r} (if it overlaps both X_{v_l} and X_{v_r}). Refer to Figure 9. Intervals on the left (right) side of the boundary are stored recursively in the subtree rooted in v_l (v_r). Intervals in r are stored in two structures: A search tree sorted according to left endpoints of the intervals and one sorted according to right endpoints. A stabbing query with q is answered by reporting

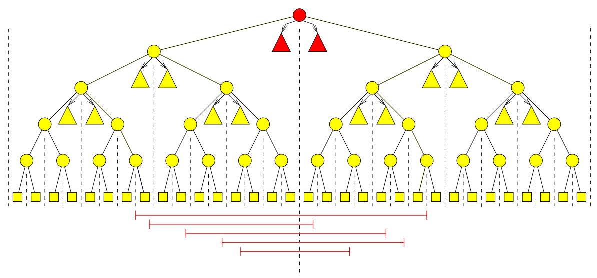

Fig. 9. Internal interval tree and examples of intervals stored in secondary structures of the root.

the intervals in r containing q and recursively reporting the relevant intervals in the subtree containing q. If q is contained in X_{v_l}, the intervals in r containing q are found by traversing the intervals in r sorted according to left endpoints, from the intervals with smallest left endpoints toward the ones with largest left endpoints, until an interval not containing q is encountered. None of the intervals in the sorted order after this interval can contain q. Since $O(T_r)$ time is used to report T_r intervals in r, a query is answered in $O(\log_2 N + T)$ time in total.

A first natural idea to make the interval tree I/O-efficient is to group nodes in $\mathcal{T}$ together into small trees of height $\Theta(\log B)$ ($\Theta(B)$ nodes) and storing them together on disk, effectively obtaining a tree with fanout $\Theta(B)$. This way a root-leaf path can be traversed in $O(\log_B N)$ I/Os. However, to answer a query we may still use $O(\log N)$ I/Os to query the $O(\log N)$ secondary structures on a root-leaf path. Below we show how to modify the structure further in order to overcome this problem, obtaining an *external interval tree.*

Structure. An external interval tree on I consists of a weight-balanced B-tree with branching parameter $\frac{1}{4}\sqrt{B}$ and leaf parameter B (Corollary 2) on the $O(N)$ sorted endpoints of intervals in I. This *base tree* $\mathcal{T}$ tree has height $O(\log_{\sqrt{B}} N) = O(\log_B N)$. As in the internal case, with each internal node v we associate an interval X_v consisting of all endpoints below v. The interval X_v is divided into at most $\sqrt{B}$ subintervals by the intervals associated with the children $v_1, v_2, \ldots$ of v. Refer to Figure 10. For illustrative purposes, we call the subintervals *slabs* and the left (right) endpoint of a slab a *slab boundary.* We define a *multislab* to be a contiguous range of slabs, such as for example $X_{v_2}X_{v_3}X_{v_4}$ in Figure 10. In a node v of $\mathcal{T}$ we store intervals from I that cross one or more of the slab boundaries associated with v, but none of the slab boundaries associated with *parent*(v). In a leaf u we store intervals with both endpoints among the endpoints in u; we assume without loss of generality that the endpoints of all intervals in I are distinct, such that the number of intervals stored in a leaf is less than $B/2$ and can therefore be stored in one block. We store the set of intervals $I_v \subset I$ associated with v in the following $\Theta(B)$ secondary structures associated with v.

- For each of $O(\sqrt{B})$ slab boundaries b_i we store
  - A right *slab list* R_i containing intervals from I_v with right endpoint between b_i and b_{i+1}. R_i is sorted according to right endpoints.
  - A left *slab list* L_i containing intervals from I_v with left endpoint between b_i and b_{i-1}. L_i is sorted according to left endpoints.
  - $O(\sqrt{B})$ *multislab lists*—one for each boundary to the right of b_i. The list $M_{i,j}$ for boundary b_j ($j > i$) contains intervals from I_v with left endpoint between b_{i-1} and b_i and right endpoint between b_j and b_{j+1}. $M_{i,j}$ is sorted according to right endpoints.
- If the number of intervals stored in a multislab list $M_{i,j}$ is less than $\Theta(B)$, we instead store them in an *underflow structure* U along with intervals associated with all the other multislab lists with fewer than $\Theta(B)$ intervals. More precisely, only if more than B intervals are associated with a multislab do we store the intervals in the multislab list. Similarly, if fewer than

$B/2$ intervals are associated with a multislab, we store the intervals in the underflow structure. If the number of intervals is between $B/2$ and B they can be stored in either the multislab list or in the underflow structure. Since $O((\sqrt{B})^2) = O(B)$ multislabs lists are associated with v, the underflow structure U always contains fewer than B^2 intervals.

We implement all secondary list structures associated with v using B-trees with branching and leaf parameter B (Corollary 1), and the underflow structure using the static interval stabbing structure discussed above (Theorem 8). In each node v, in $O(1)$ *index blocks*, we also maintain information about the size and place of each of the $O(B)$ structures associated with v.

With the definitions above, an interval in I_v is stored in two or three structures: Two slab lists L_i and R_j and possibly in either a multislab list $M_{i,j}$ or in the underflow structure U. For example, we store interval s in Figure 10 in the left slab list L_2 of b_2, in the right slab list R_4 of b_4, and in either the multislab list $M_{2,4}$ corresponding to b_2 and b_4 or the underflow structure U. Note the similarity between the slab lists and the two sorted lists of intervals in the nodes of an internal interval tree. As in the internal case, s is stored in a sorted list for each of its two endpoints. This represents the part of s to the left of the leftmost boundary contained in s, and the part to the right of the rightmost boundary contained in s. Unlike in the internal case, in the external case we also need to represent the part of s between the two extreme boundaries. We do so using one of $O(B)$ multislab lists.

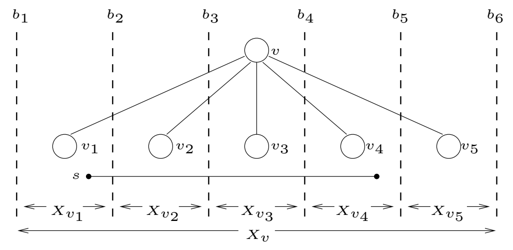

Fig. 10. A node in the base tree. Interval s is stored in L_2, R_4, and either $M_{2,4}$ or U.

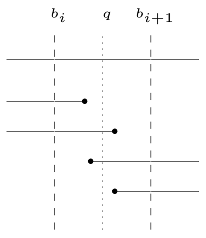

Fig. 11. Intervals containing q are stored in R_{b_i}, $L_{b_{i+1}}$, the multislab lists spanning slab, and U.

The external interval tree uses linear space: The base tree $\mathcal{T}$ itself uses $O(N/B)$ space and each interval is stored in a constant number of linear space secondary structures (Corollary 1 and Theorem 8). The number of other blocks used in a node is $O(\sqrt{B})$: $O(1)$ index blocks and one block for the underflow structure and for each of the $2\sqrt{B}$ slab lists. Since $\mathcal{T}$ has $O(N/(B\sqrt{B}))$ internal nodes, the structure uses a total of $O(N/B)$ blocks. Note that if we did not store the sparse multislab lists in the underflow structure, we could have $\Omega(B)$

sparsely utilized blocks in each node, which would result in a super-linear space bound.

Query. In order to answer a stabbing query q, we search down $\mathcal{T}$ for the leaf containing q, reporting all relevant intervals among the intervals I_v stored in each node v encountered. Assuming q lies between slab boundaries b_i and b_{i+1} in v, we report the relevant intervals by loading the $O(1)$ index blocks and then proceeding as follows: We first report intervals in all multislab lists containing intervals crossing b_i and b_{i+1}, that is, multislab lists $M_{l,k}$ with $l \leq i$ and $k > i$. Then we perform a stabbing query with q on the underflow structure U and report the result. Finally, we report intervals in R_i from the largest toward the smallest (according to right endpoint) until we encounter an interval not containing q, and intervals in L_{i+1} from the smallest toward the largest until we encounter an interval not containing q. It is easy to see that our algorithm reports all intervals in I_v containing q: All relevant intervals are either stored in a multislab list $M_{l,k}$ with $l \leq i < k$, in U, in R_i, or in L_{i+1}. Refer to Figure 11. We correctly reports all intervals in R_i containing q, since if an interval in the right-to-left order of this list does not contain q, then neither does any other interval to the left of it. A similar argument holds for the left-to-right search in L_{i+1}.

That the query algorithm uses $O(\log_B N + T/B)$ I/Os can be seen as follows. We visit $O(\log_B N)$ nodes in $\mathcal{T}$. In each node v we use $O(1)$ I/Os to load the index blocks and $O(1 + T_v/B)$ I/Os to query the two slab list R_i and L_{i+1}, where T_v is the number of intervals reported in v; there is no $O(\log_B N)$-term since we do not search in the lists. Since each visited multislab list contains $\Omega(B)$ intervals, we also use $O(T_v/B)$ I/Os to visit these lists. Note how U is crucial to obtain this bound. Finally, we use $O(\log_B B^2 + T_v/B) = O(1 + T_v/B)$ I/Os to query U. Overall, we use $O(1 + T_v/B)$ I/Os in node v, for a total of $O(\sum_v (1 + T_v/B)) = O(\log_B N + T/B)$ I/Os to answer a query.

Updates. To insert or delete an interval s in the external interval tree, we first update the base tree, that is, we insert or delete the two relevant endpoints. Next we update the secondary structures by first searching down $\mathcal{T}$ to find the first node v where s contains one or more slab boundaries; there we load the $O(1)$ index blocks of v. If performing an insertion, we insert s into the two relevant slab lists L_i and R_j. If the multislab list $M_{i,j}$ exists, we also insert s there. Otherwise, the other intervals (if any) corresponding to $M_{i,j}$ are stored in the underflow structure U and we insert s in this structure. If that brings the number of intervals corresponding to $M_{i,j}$ up to B, we delete them all from U and insert them in $M_{i,j}$. Similarly, if performing a deletion, we delete s from two slab lists L_i and R_j. We also delete s from U or $M_{i,j}$; if s is deleted from $M_{i,j}$ and the list now contains $B/2$ intervals, we delete all intervals in $M_{i,j}$ and insert them into U. Finally, we update and store the index blocks.

Disregarding the update of the base tree $\mathcal{T}$, the number of I/Os needed to perform an update can be analyzed as follows: For both insertions and deletions we use $O(\log_B N)$ I/Os to search down $\mathcal{T}$, and then in *one* node we use $O(\log_B N)$ I/Os to update the secondary list structures. We may also update the underflow structure U, which is static since it it based on persistence. However, since U has size $O(B^2)$ we can use *global rebuilding* to make it dynamic: We simply store updates in a special "update block" and once B updates have been collected we rebuild U using $O(\frac{B^2}{B} \log_{M/B} \frac{B^2}{B}) = O(B)$ I/Os (Theorem 8), or $O(1)$ I/Os amortized. We continue to be able to answer queries on U efficiently, since we only need to use $O(1)$ extra I/Os to check the update block. Thus the manipulation of the underflow structure U uses $O(1)$ I/Os amortized, except in the cases where $\Theta(B)$ intervals are moved between U and a multislab list $M_{i,j}$. In this case we use $O(B)$ I/Os but then there must have been at least $B/2$ $O(1)$ I/O updates involving intervals in $M_{i,j}$ since the last time an $O(B)$ cost was incurred. Hence the amortized I/O cost is $O(1)$ and overall the update is performed in $O(\log_B N)$ I/Os.

Now consider the update of the base tree $\mathcal{T}$, which takes $O(\log_B N)$ I/Os (Corollary 2), except that we have to consider what happens to the secondary structures when we perform a rebalancing operation (split or fuse) on base tree node v. Figure 12 illustrates how the slabs associated with v are affected when v *splits* into nodes v' and v'': All the slabs on one side of a slab boundary b get associated with v', the boundaries on the other side of b get associated with v'', and b becomes a new slab boundary in $\mathit{parent}(v)$. As a result, all intervals in the secondary structures of v that contain b need to be inserted into the secondary structures of $\mathit{parent}(v)$. The rest of the intervals need to be stored in the secondary structures of v' and v''. Furthermore, as a result of the addition of the new boundary b, some of the intervals in $\mathit{parent}(v)$ containing b also need to be moved to new secondary structures. Refer to Figure 13.

First consider the intervals in the secondary structures of v. Since each interval is stored in a left slab list and a right slab list, we can collect all intervals containing b (to be moved to $\mathit{parent}(v)$) by scanning through all of v's slab lists. We first construct a list L_r of the relevant intervals sorted according to right endpoint by scanning through the right slab lists. We scan through each right slab list (stored in the leaves of a B-tree) of v in order, starting with the rightmost slab boundary, adding intervals containing b to L_r. This way L_r will automatically be sorted. We construct a list L_l sorted according to left endpoint by scanning through the left slab lists in a similar way. Since the secondary structures of v contain $O(w(v))$ intervals (they all have an endpoint below v), and since we

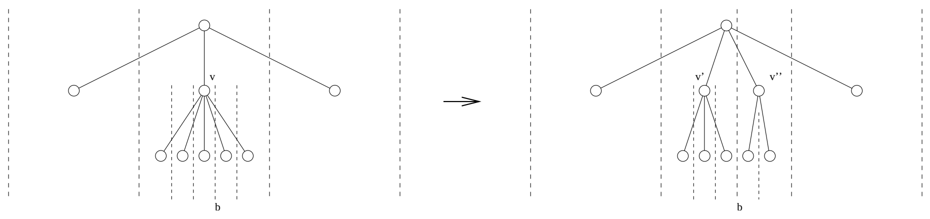

Fig. 12. Splitting a node; v splits along b, which becomes a new boundary in $\mathit{parent}(v)$.

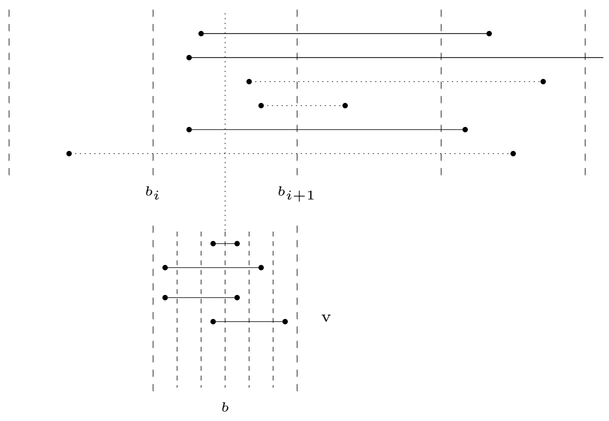

Fig. 13. All solid intervals need to move. Intervals in v containing b move to $parent(v)$ and some intervals move within $parent(v)$.

can scan through each of the $O(\sqrt{B})$ slab lists in a linear number of I/Os, we construct L_r and L_l in $O(\sqrt{B} + w(v)/B) = O(w(v))$ I/Os. Next we construct the slab lists of v' and v'', simply by removing intervals containing b from each slab list of v. We remove the relevant intervals from a given slab list by scanning through the leaves of its B-tree, collecting the intervals for the new list in sorted order, and then constructing a new list (B-tree). This way we construct all the slab lists in $O(w(v))$ I/Os. We construct the multislab lists for v' and v'' simply by removing all multislabs lists containing b. We can do so in $O(w(v))$ I/Os. We construct the underflow structures for v' and v'' by first scanning through the underflow structure for v and collecting the intervals for the two structures, and then constructing them individually using $O(w(v)/B) = O(w(v))$ I/Os (Theorem 8). We complete the construction of v' and v'' in $O(w(v))$ I/Os by scanning though the lists of each of the nodes, collecting the information for the index blocks.

Next consider $parent(v)$. We need to insert the intervals in L_l and L_r into the secondary structures of $parent(v)$ and move some of the intervals already in these structures. The intervals we need to consider all have one of their endpoints in X_v. For simplicity we only consider intervals with *left* endpoint in X_v; intervals with right endpoint in X_v are handled similarly. All intervals with left endpoint in X_v that are stored in $parent(v)$ cross boundary b_{i+1}. Thus we need to consider each of these intervals in one or two of $\sqrt{B} + 1$ lists, namely, in the left slab list L_{i+1} of b_{i+1} and possibly in one of $O(\sqrt{B})$ multislab lists $M_{i+1,j}$. When introducing the new slab boundary b, some of the intervals in L_{i+1} need to be moved to the new left slab list of b. In a scan through L_{i+1} we collect these intervals in sorted order using $O(|X_v|/B) = O(w(v)/B) = O(w(v))$ I/Os. The intervals in L_l also need to be stored in the left slab list of b, so we merge L_l with the collected list of intervals and construct a B-tree on the resulting list. We can easily do so in $O(w(v))$ I/Os and we can update L_{i+1} in the same bound. Similarly, some of the intervals in multislab lists $M_{i+1,j}$ need to be moved to new multislab lists corresponding to multislabs with b as left boundary instead of b_{i+1}. We can easily move the relevant intervals (and thus construct the new multislab lists) in $O(w(v))$ I/Os using a scan through the relevant multislab lists, similarly to the way we moved intervals from the left slab list of b_{i+1} to the left slab list of b. (Note that intervals in the underflow structure do not need to be moved). If any of the new multislab lists contain fewer than $B/2$ intervals, we instead insert the intervals into the underflow structure U. We can easily do so in $O(B) = O(w(v))$ I/Os by rebuilding U. Finally, to complete the split process we update the index blocks of $parent(v)$.

To summarize, we can split a node v in $O(w(v))$ I/Os. By Theorem 3, we know that when performing a split on v (during an insertion) $\Omega(w(v))$ updates must have been performed below v since it was last involved in a rebalance operation. Thus the amortized cost of a split is $O(1)$ I/Os. Since $O(\log_B N)$ nodes split during an insertion (Theorem 3), the update of the base tree $\mathcal{T}$ during an insertion can therefore be performed in $O(\log_B N)$ I/Os amortized. We can analyze the cost of deleting two endpoints in a similar way by considering the cost of fusing two nodes. However, deletes can also be handled in a simpler way using global rebuilding: Instead of deleting the endpoints, we just mark the two endpoints in the leaves of $\mathcal{T}$ as deleted. This does not increase the number of I/Os needed to perform a later update or query operation, but it does not decrease it either. Therefore we periodically rebuild the structure completely. Let N_0 be the number of points in $\mathcal{T}$ just after such a rebuild. As long as $N_0 = \Theta(N)$ the query bound remains $O(\log_B N + T/B)$. After $N_0/2$ deletions have been performed we rebuild the structure in $O(N_0 \log_B N_0) = O(N \log_B N)$I/Os, leading to an $O(\log_B N)$ amortized delete I/O bound: We scan through the leaves of the old base tree and construct a sorted list of the undeleted endpoints. We then use this list to construct the new base tree. All of this can be done in $O(N_0/B)$ I/Os. Finally, we insert the $O(N)$ intervals one-by-one without rebalancing the base tree using $O(N) \cdot O(\log_B N_0)$ I/Os.

Theorem 9. *An external interval tree on a set of N intervals uses $O(N/B)$ space and answers stabbing queries in $O(\log_B N + T/B)$ I/Os. Updates can be performed in $O(\log_B N)$ I/Os amortized.*

Remarks. The internal memory interval tree was developed by Edelsbrunner [40, 41]. The external interval tree as described here was developed by Arge and Vitter [23] following earlier attempts by several authors [55, 73, 67, 28, 54]. The global rebuilding ideas used in the structure is due to Overmars [65]. The amortized update bounds can be made worst-case using standard lazy-rebuilding techniques also due to Overmars [65]. Variants of the external interval tree structure—as well as experimental results on applications of it in isosurface extraction—have been considered by Chiang and Silva [33, 35, 34]. The structure also forms the basis for several external planar point location structures [1, 21].

The external interval tree illustrates some of the problems encountered when trying to map multilevel internal memory structures to external memory, mainly the problems encountered when needing to use multi-way trees as base trees, as well as the techniques commonly used to overcome these problems: The multi-level base tree resulted in the need for multislabs. To handle multislabs efficiently we used the notion of underflow structure, as well as the fact that we could decrease the fan-out of $\mathcal{T}$ to $\Theta(\sqrt{B})$ while maintaining the $O(\log_B N)$ tree height. The underflow structure—implemented using sweeping and a persistent B-tree—solved a static version of the problem on $O(B^2)$ interval in $O(1 + T_v/B)$ I/Os. The structure was necessary since if we had just stored the intervals in multislab lists we might have ended up spending $\Theta(B)$ I/Os to visit the $\Theta(B)$ multislab lists of a node without reporting more than $O(B)$ intervals in total. This would have resulted in an $\Omega(B \log_B N + T/B)$ query bound. We did not store intervals in multislab lists containing $\Omega(B)$ intervals in the underflow structure, since the I/Os spent on visiting such lists during a query can always be charged to the $O(T_v/B)$-term in the query bound. The idea of charging some of the query cost to the output size is often called *filtering* [31], and the idea of using a static structure on $O(B^2)$ elements in each node has been called the *bootstrapping* paradigm [79]. Finally, we used weight-balancing and global rebuilding to obtain efficient update bounds.

7 Three-sided planar range searching: External priority search tree

We now move on and consider the special case of planar orthogonal range searching called *3-sided planar range searching*: Maintain a set S of point in the plane such that given a 3-sided query $q = (q_1, q_2, q_3)$ we can report all points $(x, y) \in S$ with $q_1 \leq x \leq q_2$ and $y \geq q_3$. Refer to Figure 14. This problem is actually a generalization of the interval stabbing problem, since interval stabbing is equivalent to performing *diagonal corner queries* on a set of points in the plane: Consider mapping an interval $[x, y]$ to the point (x, y) in the plane. Finding all intervals containing a query point q then corresponds to finding all points (x, y) such that $x \leq q$ and $y \geq q$. Refer to Figure 15.

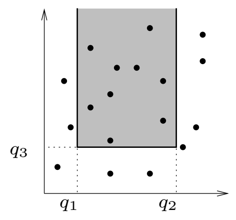

Fig. 14. 3-sided query.

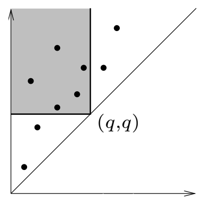

Fig. 15. Diagonal corner query.

Like in the interval stabbing case, the static version of the 3-sided range searching problem can easily be solved using a persistent B-tree. This time we imagine sweeping the plane with a horizontal line from $y = \infty$ to $y = -\infty$ and inserting the x-coordinate of points from S in a persistent B-tree as they are met. To answer a query $q = (q_1, q_2, q_3)$ we perform a one-dimensional range query $[q_1, q_2]$ on the B-tree at "time" q_3. This way we obtain the following (Theorem 4 and Corollary 4).

Theorem 10. *A static set of N points in the plane can be stored in a linear space data structures such that a 3-sided range query can be answered in $O(\log_B N + T/B)$ I/Os. The structure can be constructed in $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os.*

A dynamic solution to the 3-sided range query problem can be obtained using an external *priority search tree.*

Structure. An external priority search tree consists of a weight-balanced base B-tree $\mathcal{T}$ with branching parameter $\frac{1}{4}B$ and leaf parameter B on the x-coordinates of the points in S. As in the interval tree case, each internal node v corresponds naturally to an x-range X_v, which is divided into $\Theta(B)$ slabs by the x-ranges of its children. In each node v we store $O(B)$ points from S for each of its $\Theta(B)$ children v_i, namely the B points with the highest y-coordinates in the x-range X_{v_i} of v_i (if existing) that have not been stored in ancestors of v. We store the $O(B^2)$ points in the linear space static structure discussed above—the "B^2–structure"—such that a 3-sided query on the points can be answered in $O(\log_B B^2 + T_v/B) = O(1 + T_v/B)$ I/Os (Theorem 10). Note that as in the interval tree case, we can update the B^2–structure in $O(1)$ I/Os using an "update block" and global rebuilding. In a leaf u of $\mathcal{T}$ we store the points with x-coordinates among the x-coordinates in u that are not stored further up the tree; assuming without loss of generality that all x-coordinate are distinct, we can store these points in a single block. Overall the external priority search tree uses $O(N/B)$ space, since $\mathcal{T}$ uses linear space and since every point is stored in precisely one B^2–structure.

Query. To answer a 3-sided query $q = (q_1, q_2, q_3)$ we start at the root of $\mathcal{T}$ and proceed recursively to the appropriate subtrees: When visiting a node v we first query the B^2–structure and report the relevant points. Then we advance the search to some of the children of v. The search is advanced to a child v_i if it is either along the search path for q_1 or the search path for q_2, or if the entire set of points corresponding to v_i in the B^2–structure were reported—refer to Figure 16(a). The query procedure reports all points in the query range since if we do not visit child v_i corresponding to a slab completely spanned by the interval $[q_1, q_2]$, it means that at least one of the points in the B^2–structure corresponding to v_i is not in q. This in turn means that none of the points in the subtree rooted at v_i can be in q.

That we use $O(\log_B N + T/B)$ I/Os to answer a query can be seen as follows. In each internal node v of $\mathcal{T}$ visited by the query procedure we spend $O(1+T_v/B)$ I/Os, where T_v is the number of points reported. There are $O(\log_B N)$ nodes visited on the search paths in $\mathcal{T}$ to the leaf containing q_1 and the leaf containing q_2, and thus the number of I/Os used in these nodes adds up to $O(\log_B N + T/B)$. Each remaining visited internal node v in $\mathcal{T}$ is not on the search paths but it is visited because $\Theta(B)$ points corresponding to v were reported in its parent. Thus the cost of visiting these nodes adds up to $O(T/B)$, even if we spend a constant number of I/Os in some nodes without finding $\Theta(B)$ points to report.

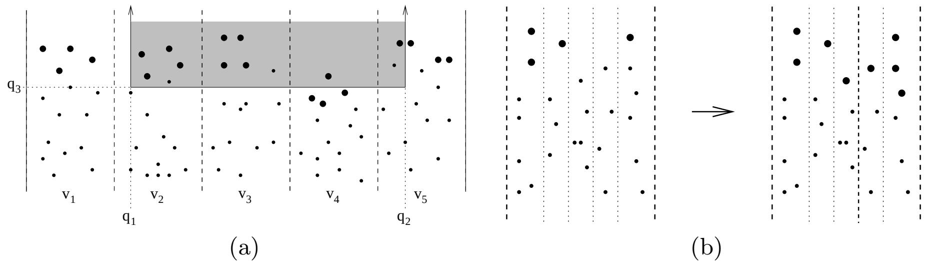

Fig. 16. a) Internal node v with children $v_1, v_2, \ldots, v_5$. The points in bold are stored in the B^2–structure. To answer a 3-sided query $q = (q_1, q_2, q_3, q_4)$ we report the relevant of the $O(B^2)$ points and answer the query recursively in v_2, v_3, and v_5. The query is not extended to v_4 because not all of the points from v_4 in the B^2–structure is in q. (b) The $\Theta(B^2)$ (small) points stored in a node v and $\Theta(B)$ (big) points stored in $parent(v)$ corresponding to v. $O(B)$ new points is needed in $parent(v)$ when v splits an a new slab boundary is inserted in $parent(v)$.

Updates. To insert or delete a point $p = (x, y)$ in the external priority search tree, we first insert or delete x from the base tree $\mathcal{T}$. If we are performing an *insertion* we then update the secondary structures using the following recursive *bubble-down* procedure starting at the root v: We first find the (at most) B points in the B^2–structure of v corresponding to the child v_i whose x-range X_{v_i} contains x; we find these points simply by performing the (degenerate) 3-sided query defined by X_{v_i} and $y = -\infty$ on the B^2–structure. If p is below these points (and there are B of them), we recursively insert p in v_i. Otherwise, we insert p in the B^2–structure. If this means that the B^2–structure contains more than B points corresponding to v_i, we delete the lowest of these points (which can again be identified by a simple query) and insert it recursively in v_i. If v is a leaf, we simply store the point in the associated block. If we are performing a *deletion* we first identify the node v containing p by searching down $\mathcal{T}$ for x while querying the B^2–structure for the (at most) B points corresponding to the relevant child v_i. Then we delete p from the B^2–structure. Since this decreases the number of points from X_{v_i} stored in v, we promote a point from v_i using a recursive *bubble-up* procedure: We first find the topmost point p' (if existing) stored in v_i using a (degenerate) query on its B^2–structure. Then we delete p' from the B^2–structure of v_i and insert it in the B^2–structure of v. Finally, we recursively promote a point from the child of v_i corresponding to the slab containing p'. Thus we may end up promoting points along a path to a leaf; at a leaf we simply load the single block containing points in order to identify, delete, and promote the relevant point.

Disregarding the update of the base tree $\mathcal{T}$, an update is performed in $O(\log_B N)$ I/Os amortized: We search down one path of $\mathcal{T}$ of length $O(\log_B N)$ and in each node we perform a query and a constant number of updates on a B^2–structure. Since we only perform queries that return at most B points, each of them takes $O(\log_B B^2 + B/B) = O(1)$ I/Os (Theorem 4). Each update also takes $O(1)$ I/Os amortized.

The update of the base tree $\mathcal{T}$ also takes $O(\log_B N)$ I/Os (Theorem 3), except that we have to consider what happens to the secondary B^2–structures when we perform a rebalancing operation (split or fuse) on a base tree node v. As discussed in Section 6, when v is *split* into v' and v'' along a boundary b, all slabs on one side of b gets associated with v' and all slabs on the other side with v'', and b becomes a new slab boundary in $\mathit{parent}(v)$. Refer to Figure 12. The B^2–structures of v' and v'' then simply contains the relevant of the points that were stored in the B^2–structure of v. However, since the insertion of b in $\mathit{parent}(v)$ splits a slab into two, the B^2–structure of $\mathit{parent}(v)$ now contains B too few points. Refer to Figure 16(b). Thus we need to promote (a most) B points from v' and v'' to $\mathit{parent}(v)$. We can do so simply by performing $O(B)$ bubble-up operations. Since we can construct the B^2–structures of v' and v'' in $O(B)$ I/Os, and perform the $O(B)$ bubble-up operations in $O(B \log_B w(v))$ (because the height of the tree rooted in v is $O(\log_B w(v))$), we can in total perform a split in $O(B \log_B w(v)) = O(w(v))$ I/Os. By Theorem 3, we know that when performing a split on v (during an insertion) $\Omega(w(v))$ updates must have been performed below v since it was last involved in a rebalance operation. Thus the amortized cost of a split is $O(1)$ I/Os and thus an insertion is performed in $O(\log_B N)$ I/Os amortized. As previously, we can handle deletes on $\mathcal{T}$ in a simple way using global rebuilding rather than fusion of nodes: To delete an x-coordinate we simply mark it as deleted in the relevant leaf of $\mathcal{T}$ and periodically rebuild the structure. Let N_0 be the number of points in the structure just after a rebuild. After $N_0/2$ deletions we rebuild the structure in $O(N_0 \log_B N_0) = O(N \log_B N)$ I/Os. This way the query bound is maintained and we obtain an $O(\log_B N)$ amortized delete I/O bound.

Theorem 11. *An external priority search tree on a set of N points in the plane uses $O(N/B)$ space and answers 3-sided range queries in $O(\log_B N + T/B)$ I/Os. Updates can be performed in $O(\log_B N)$ I/Os amortized.*

Remarks In internal memory the priority search tree was developed by McCreight [62]. The external priority search tree was described by Arge et al. [19], following earlier attempt by several authors [67, 73, 28, 54]. They also showed several ways of removing the amortization from the update bounds. Note how the structure uses many of the ideas already utilized in the development of the external interval tree structure in Section 6: *Bootstrapping* using a static structure, *filtering*, and a *weight-balanced* B-tree.

8 General planar orthogonal range searching

After discussing 2- and 3-sided planar range searching, we are now ready to consider general (4-sided) orthogonal range searching. Given a set of points S in the plane we want to be able to find all points contained in an axis-aligned query rectangle. While linear space and $O(\log_B N + T/B)$ query structures exist for 2- and 3-sided queries, it turns out that in order to obtain an $O(\log_B N + T/B)$ query bound for the general case we have to use $\Omega(\frac{N}{B}\frac{\log_B N}{\log_B \log_B N})$ space. We describe the *external range tree* obtaining these bounds in Section 8.1. In practical applications involving massive datasets it is often crucial that external data structures use linear space. In that case we need $\Omega(\sqrt{N/B} + T/B)$ I/Os to answer a query. We describe the *O-tree* obtaining these bounds in Section 8.2.

8.1 External range tree

Structure. An external range tree on a set of points S in the plane consists of a base weight-balanced B-tree $\mathcal{T}$ with branching parameter $\frac{1}{4}\log_B N$ and leaf parameter B on the x-coordinates of the points. As previously, an x-range X_v is associated with each node v and it is subdivided into $\Theta(\log_B N)$ slabs by v's children. We store *all* the points from S in X_v in four secondary data structures associated with v. The first two structures are priority search trees for answering 3-sided queries—one for answering queries with opening to the left and one for queries with opening to the right. The third structure is a B-tree on the points sorted by y-coordinate. For the fourth structure, we imagine for each child v_i linking together the points in X_{v_i} in y-order, producing a polygonal line monotone with respect to the y-axis. We project all the segments produced in this way onto the y-axis and store them in an external interval tree. Refer to Figure 17(a). With each segment endpoint in X_{v_i} we also store a pointer to the same point in the B-tree of the child node v_i.

Since the tree has $O(\log_{\log_B N}(N/B)) = O(\log_B N/\log_B \log_B N)$ levels and we use linear space on each level, the structure uses $O(\frac{N}{B}\frac{\log_B N}{\log_B \log_B N})$ space in total.

Query. To answer a 4-sided query $q = (q_1, q_2, q_3, q_4)$ we first find the topmost node v in the base tree $\mathcal{T}$ where the x-range $[q_1, q_2]$ of the query contains a slab boundary. Consider the case where q_1 lies in the x-range X_{v_i} of v_i and q_2 lies in the x-range X_{v_j} of v_j. The query q is naturally decomposed into three parts, consisting of a part in X_{v_i}, a part in X_{v_j}, and a part completely spanning slabs X_{v_k}, for $i < k < j$; refer to Figure 17(b). We find the points contained in the first two parts in $O(\log_B N + T/B)$ I/Os using the right opening 3-sided structure associated with v_i and the left opening 3-sided structure associated with v_j. To find the points in the third part we query the interval tree associated with v with the y-value q_3. This way we obtain the $O(\log_B N)$ segments in the structure containing q_3, and thus (a pointer to) the bottommost point contained in the query for each of the nodes $v_{i+1}, v_{i+2}, \ldots, v_{j-1}$. We then traverse the relevant leaves of the $j - i - 1 = O(\log_B N)$ relevant B-tree and output the remaining

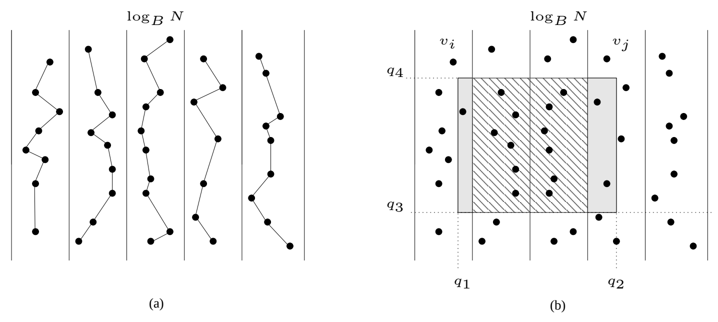

Fig. 17. a) The slabs corresponding to a node v in the base tree and the segments projected onto the y-axis and stored in an interval tree. (b) A query $q = (q_1, q_2, q_3, q_4)$ is naturally decomposed into a part in X_{v_i}, a part in X_{v_j}, and a part that spans all slabs X_{v_k} for $i < k < j$.

points using $O(\log_B N + T/B)$ I/Os. Thus overall the query q is answered in $O(\log_B N + T/B)$ I/Os.

Updates. To insert a point in or delete a point from the external range tree we first update the base tree $\mathcal{T}$; below we discuss how to do so. Then we perform $O(1)$ updates using $O(\log_B N)$ I/Os each in the secondary structures of the $O(\log_B N/\log_B \log_B N)$ nodes on the path from the root of $\mathcal{T}$ to the leaf containing the point, for a total of $O(\log^2_B N/\log_B \log_B N)$ I/Os.

As previously, we update the base tree $\mathcal{T}$ during a deletion using global rebuilding: When deleting a point (x-coordinate) we simply mark it as deleted in a leaf. If N_0 was the number of points in the structure when it was last rebuilt, we then rebuild it again after $N_0/2$ deletes. Since a secondary structure (B-tree, interval tree, or priority search tree) on the $w(v)$ points in the x-range X_v of a node v can easily be constructed in $O(w(v)\log_B N_0)$ I/Os using repeated insertion, the structure is constructed in $O(N_0 \log_B N_0 \cdot \log_B N_0/\log_B \log_B N_0)$ I/Os or $O(\log^2_B N/\log_B \log_B N)$ I/Os amortized.

Insertions in the base tree $\mathcal{T}$ are also relatively easy to handle, since each node stores all points in its x-range (that is, each point is stored on each level of $\mathcal{T}$). When rebalancing a node v during an insertion, that is, splitting it into v' and v'', we simply split the $w(v)$ points in v into two sets and construct the secondary structures for the two new nodes using $O(w(v)\log_B N)$ I/Os. A discussed in Section 6 and 7, the split introduces a new boundary in *parent*(v). This effects the interval tree of *parent*(v), since the $O(w(v))$ intervals generated from the points in X_v change. Refer to Figure 18. The other structures remain unchanged. We can easily update the interval tree in *parent*(v) in $O(w(v)\log_B N)$ I/Os, simply by deleting the $O(w(v))$ superfluous intervals and inserting the $O(w(v))$ new ones generated as a result of the split. This takes $O(w(v)\log_B N)$ I/Os in total. Since $\Omega(w(v))$ updates must have been performed below v since it was last involved in a rebalance operation (Theorem 3), the amortized cost of a split is

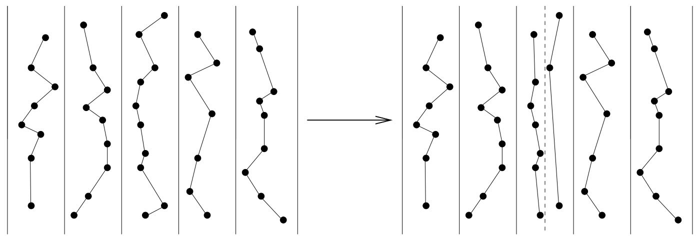

Fig. 18. Change of intervals when new boundary is inserted in $parent(v)$ when v splits.

$O(\log_B N)$ I/O. Thus an insertion is also performed in $O(\log_B^2 N/\log_B \log_B N)$ I/Os amortized.

Theorem 12. *An external range tree on a set of N points in the plane uses $O(\frac{N}{B}\frac{\log_B N}{\log_B \log_B N})$ space and answers orthogonal range queries in $O(\log_B N+T/B)$ I/Os. Updates can be performed in $O(\log_B^2 N/\log_B \log_B N)$ I/Os amortized.*

Remarks. The external range tree is adopted from the internal range tree due to Chazelle [31]. Subramanian and Ramaswamy [73] were the first to attempt to externalize this structure. Based on a sub-optimal linear space structure for answering 3-sided queries they developed the *P-range tree* that uses $O(\frac{N}{B}\frac{\log_B N}{\log_B \log_B N})$ space but slightly more than $O(\log_B N + T/B)$ I/Os to answer a query. Using their optimal structure for 3-sided queries, Arge et al. [19] obtained the structure described above. Subramanian and Ramaswamy [73] proved that one cannot obtain an $O(\log_B N + T/B)$ query bound using less than $\Theta(\frac{N}{B}\frac{\log_B N}{\log_B \log_B N})$ disk blocks, that is, that the external range tree is optimal. In fact, this lower bound even holds for a query bound of $O(\log_B^c N+T/B)$ for any constant $c \geq 1$. It holds in a natural external memory version of the *pointer machine model* [32]. A similar bound in a slightly different (stronger) model where the search component of the query is ignored was proved by Arge et al. [19]. This *indexability model* was defined by Hellerstein et al. [52] and considered by several authors [55, 59, 71, 20].

8.2 O-tree

In this section we describe the linear space O-tree that supports range queries in $O(\sqrt{N/B}+T/B)$ I/Os. We start by describing the external kd-tree, which is an externalization of the internal memory kd-tree. This structure actually supports queries in $O(\sqrt{N/B} + T/B)$ I/Os but updates require $O(\log_B^2 N)$ I/Os. Next we describe the actual O-tree, which improves the update bound to $O(\log_B N)$ utilizing the external kd-tree.

External kd-tree

Structure. An internal memory kd-tree $\mathcal{T}$ on a set S of N points in the plane is a binary tree of height $O(\log_2 N)$ with the points stored in the leaves of the tree. The internal nodes represent a recursive decomposition of the plane by means of axis-orthogonal lines that partition the set of, say, N' points below a node into two subsets of approximately equal size $\lfloor N'/2 \rfloor$ and $\lceil N'/2 \rceil$. On even levels of $\mathcal{T}$ we use horizontal dividing lines and on odd levels vertical dividing lines. In this way a rectangular region R_v is naturally associated with each node v and the nodes on any particular level of $\mathcal{T}$ partition the plane into disjoint regions. In particular, the regions associated with the leaves represent a partition of the plane into rectangular regions containing one point each. Refer to Figure 19.

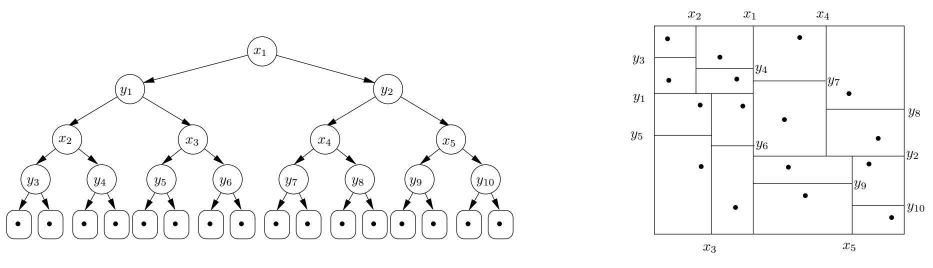

Fig. 19. A kd-tree and the corresponding partitioning.

A (static) external memory kd-tree is simply a kd-tree $\mathcal{T}$ where we stop the recursive partition when the number of point in a region falls below B. This way the structure has $O(N/B)$ leaves containing between $\lfloor B/2 \rfloor$ and B points each. We store the points in each leaf together in a disk block. Since the number of internal nodes is $O(N/B)$, the structures uses $O(N/B)$ blocks regardless of how we store these nodes on disk. However, in order to be able to follow a root-leaf path in $O(\log_B N)$ I/Os, we *block* the internal nodes of $\mathcal{T}$ in a way similar to a B-tree: Starting at the root we visit nodes in breadth-first order until we have visited a subtree consisting of $\log B - 1$ whole levels of internal nodes. This subtree contains less than $2 \cdot 2^{\log B - 1} = B$ nodes and we can therefore store it in one disk block. Then we recursively visit and block the $O(B)$ tress rooted below the subtree; when a subtree has less than $\log B - 1$ levels (contains less than B internal nodes) we simply store it in a disk block. This way a root leaf path in $\mathcal{T}$ of length $O(\log \frac{N}{B})$ is stored in $O(\log \frac{N}{B})/(\log B - 1) + 1 = O(\log_B N)$ blocks, such that it can be traversed in $O(\log_B N)$ I/Os.

We can easily construct an external static kd-tree $\mathcal{T}$ on a set S of N points in $O(\frac{N}{B} \log \frac{N}{B})$ I/Os: We first creating two lists of the points in S sorted by x- and y-coordinates, respectively. Then we create the root of $\mathcal{T}$ using $O(N/B)$ I/Os, simply by scanning the list sorted by y-coordinates and computing the horizontal median split. After this we partition the two sorted lists according to this median, and recursively construct the rest of the tree. After having constructed the tree we can block the $O(N/B)$ internal nodes in $O(N/B)$ I/Os using a simple recursive depth-first traversal of the tree.

Query. A point query on S, that is, a search for a given point, can obviously be answered in $O(\log_B N)$ I/Os by following one root-leaf path in $\mathcal{T}$. A range query $q = (q_1, q_2, q_3, q_4)$ can be answered with a simple recursive procedure starting at the root: At a node v we advance the query to a child w if q intersects the region R_w associated with w. At a leaf u we return the points in u contained in q.

To bound the number of node in $\mathcal{T}$ visited when answering a range query q, or equivalently, the number of nodes v where R_v intersects q, we first bound the number of nodes v where R_v intersects a vertical line l. The region R_r associated with the root r is obviously intersected by l, but as the regions associated with its two children represent a subdivision of R_r with a vertical line, only the region R_w associated with one of these children w is intersected. Because the region R_w is subdivided by a horizontal line, the regions associated with both children of w are intersected. Let $L = O(N/B)$ be the number of leaves in the kd-tree. As the children of w are roots in kd-trees with $L/4$ leaves, the recurrence for the number of regions intersected by l is $Q(L) \leq 2 + 2Q(L/4) = O(\sqrt{L}) = O(\sqrt{N/B})$. Similarly, we can show that the number of regions intersected by a horizontal line is $O(\sqrt{N/B})$. This means that the number of nodes v with regions R_v intersected by the boundary of q is $O(\sqrt{N/B})$. All the additional nodes visited when answering q correspond to regions completely inside q. Since each leaf contains $\Theta(B)$ points there are $O(T/B)$ leaves with regions completely inside q. Since the region R_v corresponding to an internal node v is only completely contained in q if the regions corresponding to the leaves below v are contained in q (and since the kd-tree is binary), the total number of regions completely inside q is also $O(T/B)$. Thus in total $O(\sqrt{N/B} + T/B)$ nodes are visited and therefore a query is answered in $O(\sqrt{N/B} + T/B)$ I/Os.

Theorem 13. *A static external kd-tree for storing a set of N points in the plane uses linear space and can be constructed in $O(\frac{N}{B} \log \frac{N}{B})$ I/Os. It supports point queries in $O(\log_B N)$ I/Os and orthogonal range query in $O(\sqrt{N/B} + T/B)$ I/Os.*

Updates. We first consider the case where we only want to support *insertions.* We do so using the so-called *logarithmic method*: We maintain a set of $O(\log_2 N)$ static kd-trees $\mathcal{T}_0, \mathcal{T}_1, \ldots$, such that $\mathcal{T}_i$ is either empty or has size 2^i. We perform an insertion by finding the first empty structure $\mathcal{T}_i$, discarding all structures $\mathcal{T}_j, j < i$, and building $\mathcal{T}_i$ from the new point and the $\sum_{l=0}^{i-1} 2^l = 2^i - 1$ points in the discarded structures using $O(\frac{2^i}{B} \log \frac{2^i}{B})$ I/Os (Theorem 13). If we divide this cost between the 2^i points, each of them is charged $O(\frac{1}{B} \log \frac{N}{B})$ I/Os. Because points never move from higher to lower indexed structures, we charge each point $O(\log N)$ times. Thus the amortized cost of an insertion is $O(\frac{1}{B} \log \frac{N}{B} \log N) = O(\log_B^2 N)$ I/Os.

To answer a query we simply query each of the $O(\log N)$ structures. In general, querying $\mathcal{T}_i$ takes $O(1 + \sqrt{2^i/B} + T_i/B)$ I/Os, where T_i is the number of reported points. However, if we keep the first $\log B$ structures in memory, using $O(1)$ blocks of main memory, we can query these structures without performing any I/Os. Thus in total we use $\sum_{i=\log B}^{\log N} O(\sqrt{2^i/B} + T_i/B) = O(\sqrt{N/B} + T/B)$ I/Os to answer a query.

Next we also consider *deletions*. We first describe how we can modify the static external kd-tree (Theorem 13) to support deletions efficiently: To delete a point p, we first perform a point query on $\mathcal{T}$ to find the leaf u containing p. Next we simply remove p from u. If the number of points in u falls to $B/4$ we then rebuild the kd-tree rooted in the parent v of u (what we call a *local rebuilding*); note that after a number of rebuildings, v can be root in a very large kd-tree. During the rebuild we store the internal nodes in the rebuilt structure in the same blocks as were previously used for the subtree rooted in v. In particular, we may store v (and a cirtain number of levels of nodes below v) in a block also containing some of v ancestors. Finally, we also periodically rebuild the entire structure (*global rebuilding*): If N_0 is the number of points in the structure just after the last complete rebuild, we rebuild the structure again after $N_0/2$ deletions.

The delete algorithm ensures that the linear space bound and the $O(\log_B N)$ point query and $O(\sqrt{N/B} + T/B)$ range query performance is maintained: The global rebuilding ensures that the space bound remains $O(N_0/B) = O(N/B)$ and that the recurrence for the number of nodes corresponding to regions intersected by a vertical line remains $Q(L) \leq 2 + 2Q(L/4) = O(\sqrt{L}) = O(\sqrt{N_0/B}) = O(\sqrt{N/B})$; the local rebuilding ensures that each leaf always contain $\Theta(B)$ points (between $B/4$ and B) and thus that the number of nodes corresponding to regions completely inside a query q remains $O(T/B)$. This way the range query bound is maintained. The use of the original blocking of internal nodes of the tree during local rebuildings ensure that the point query bound is maintained.

The amortized number of I/Os needed to perform a deletion on an extenal kd-tree $\mathcal{T}$ is $O(\log_B^2 N)$: The search for p requires $O(\log_B N)$ I/Os. The global rebuilding every $N_0/2$ deletions is performed in $O(\frac{N_0}{B} \log \frac{N_0}{B})$ I/Os, or $O(\frac{1}{B} \log \frac{N_0}{B}) = O(\log_B N)$ I/Os amortized. When local rebuilding is performed on a node v, one of v children w is a leaf containing $B/4$ points. This means that all but $B/4$ of the original points stored below w—and thus all but $B/4$ of half of the original points below v—have been deleted since v was constructed (during a local or global rebuild). Since the number of points below v was at least $2\frac{B}{2} = B$ at that time, the number of deleted points is proportional to the number of original points below v. Amortized each deleted point therefore contribute $O(\frac{1}{B} \log \frac{N_0}{B})$ I/Os to the local rebuilding. Since each points contributes on each of the $O(\log \frac{N_0}{B})$ levels of the tree, the total amortized cost is $O(\frac{1}{B} \log^2 \frac{N_0}{B}) = O(\log_B^2 N)$ I/Os.

Finally, we describe how to support both insertions and deletions. To delete a point p from the external kd-tree supporting insertions using the logarithmic method, we simply delete p from the relevant $\mathcal{T}_i$ using the algorithm described above. We globally rebuild the entire structure after every $N_0/2$ deletes, such that the number of structures remains $O(\log N)$. This ensures that a range query can still be answered in $O(\sqrt{N/B} + T/B)$ I/Os. We ignore deletions in terms of the logarithmic method, that is, we destroy and reconstruct structures $\mathcal{T}_i$ as if no deletions were taking place. This way, points still only move from lower to higher index structures, which ensures that the amortized insertion cost remains $O(\log_B^2 N)$. Regarding deleted points as still being present in terms of the logarithmic method also lets us efficiently find the structure $\mathcal{T}_i$ containing a point p to be deleted. We simply maintain a separate B-tree $\mathcal{T}_d$ on the points in the structure. For point p, $\mathcal{T}_d$ stores how many points were inserted since the last global rebuild when p was inserted. Maintenance of $\mathcal{T}_d$ adds $O(\log_B N)$ I/Os to the insertion bound. To find the structure $\mathcal{T}_i$ containing a given point p, we query $\mathcal{T}_d$ with p using $O(\log_B N)$ I/Os. A simple calculation, based on the obtained information and the current number of elements inserted since the last global rebuilding, then determines i. After that p can be deleted in $O(\log_B^2 N)$ I/Os.

Theorem 14. *An external kd-tree for storing a set of N points in the plane uses linear space and can be constructed in $O(\frac{N}{B}\log\frac{N}{B})$ I/Os. It supports point queries in $O(\log_B N)$ I/Os, orthogonal range query in $O(\sqrt{N/B} + T/B)$ I/Os, and updates in $O(\log_B^2 N)$ I/Os amortized.*

O-tree structure

After describing the external kd-tree we are now ready to describe the O-tree with an improved $O(\log_B N)$ update bound.

Structure. Consider dividing the plane into *slabs* using $\Theta(\sqrt{N/B}/\log_B N)$ vertical lines such that the number of points from S in each slab is between $\frac{1}{2}\sqrt{NB}\log_B N$ and $\sqrt{NB}\log_B N$. Each of these slabs are further divided into *cells* using $\Theta(\sqrt{N/B}/\log_B N)$ horizontal lines such that each cell contains between $\frac{1}{2}B\log_B^2 N$ and $B\log_B^2 N$ points. Refer to Figure 20(a). The O-tree consist of a B-tree $\mathcal{T}_v$ (with leaf and branching parameter B) on the vertical lines, a B-tree $\mathcal{T}_i$ (with leaf and branching parameter B) in each slab s_i on the horizontal lines in s_i, and an external kd-tree on the points in each cell; the kd-tree in the j'th cell in slabs s_i is called $\mathcal{T}_{ij}$. Refer to Figure 20(b).

The O-tree uses linear space: The B-trees use $O((\sqrt{N/B}/\log_B N)^2/B) = O(N/(B\log_B N)^2)$ space in total (Theorem 1) and each point is store in exactly one linear space kd-tree (Theorem 14). We can easily construct the structure in $O(\frac{N}{B}\log\frac{N}{B})$ I/Os: We first compute the vertical lines bacically by sorting the points in S by x-coordinates using $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os. Then we construct $\mathcal{T}_v$ in the same bound (Theorem 2). We compute the horizontal lines and construct $\mathcal{T}_i$ in each slab s_i in $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os in total in a similar way. Finally we construct the kd-tree in all cells in $O(\frac{N}{B}\log\frac{N}{B})$ I/Os in total (Theorem 14).

Query. A point query on S, that is, a search for a given point p, can easily be answered in $O(\log_B N)$ I/Os by performing a search in $\mathcal{T}_v$, followed by a search in one $\mathcal{T}_i$, and finally a search in one kd-tree $\mathcal{T}_{ij}$ (Corollary 1 and Theorem 14). To answer a range query $q = (q_1, q_2, q_3, q_4)$, we first perform a query on $\mathcal{T}_v$ to

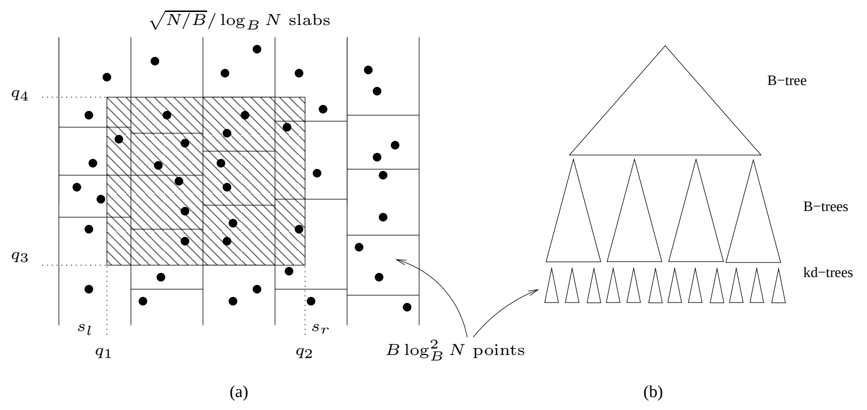

Fig. 20. a) Division of plane into cells containing $\Theta(B\log_B^2 N)$ points using $\Theta(\sqrt{N/B}\log_B N)$ vertical lines and $\Theta(\sqrt{N/B}\log_B N)$ horizontal lines in each slab. (b) O-tree: B-tree on vertical lines, B-tree on horizontal lines in each slab, and kd-tree on points in each cell. A query $q = (q_1, q_2, q_3, q_4)$ is answered by querying all kd-trees in cells in slabs s_l and s_r containing q_1 and q_2, as well as all kd-trees in cells intersected by $[q_3, q_4]$ in slabs spanned by $[q_1, q_2]$.

find the $O(\sqrt{N/B}/\log_B N)$ slabs intersected by the x-range $[q_1, q_2]$ of q. If q_1 is in slab s_l and q_2 in slab s_r we then query *all* kd-trees $\mathcal{T}_{lj}$ and $\mathcal{T}_{rj}$ in slabs s_l and s_r to find all points in these slabs in q. The remaining points in q are all in one of the slabs s_i, $l < i < r$, completely spanned by $[q_1, q_2]$. In each such slab s_i we query $\mathcal{T}_i$ to find the $O(\sqrt{N/B}/\log_B N)$ cells intersected by the y-range $[q_3, q_4]$ of q. Then we query the kd-tree $\mathcal{T}_{ij}$ for each of these cells to find all points in s_i in q. Refer to Figure 20.

That we answer a range query in $O(\sqrt{N/B} + T/B)$ as on a external kd-tree can be seen as follows (using Corollary 1 and Theorem 14): The query on B-tree $\mathcal{T}_v$ is performed in $O(\log_B(\sqrt{N/B}/\log_B N) + \sqrt{N/B}/(B\log_B N)) = O(\sqrt{N/B})$, since we know the output from the query is of size $O(\sqrt{N/B}/\log_B N)$. Querying the $2 \cdot O((\sqrt{N/B}/\log_B N)$ kd-trees in slab s_l and s_r takes $O((\sqrt{N/B}/\log_B N) \cdot O(\sqrt{B\log_B^2 N)/B}) + O(T/B) = O(\sqrt{N/B} + T/B)$ I/Os. The query on B-trees $\mathcal{T}_i$ in the $O(\sqrt{N/B}/\log_B N)$ slabs s_i completely spanned by the x-range $[q_1, q_2]$ takes $O(\sqrt{N/B}/\log_B N) \cdot O(\log_B(\sqrt{N/B}/\log_B N) + O(T/B) = O(\sqrt{N/B} + T/B)$, since we know that the combined output size for all the queries is $O(T + \sqrt{N/B}/\log_B N)$ (because all but 2 reported cells in each slab s_i is complety contained in q). Finally, the total number of I/Os required to query the kd-trees $\mathcal{T}_{ij}$ in cells intersected by the y-range $[q_3, q_4]$ in all slabs s_i completely spanned by the x-range $[q_1, q_2]$ is $2 \cdot O(\sqrt{N/B}/\log_B N) \cdot O(\sqrt{B\log_B^2 N)/B}) + O(T/B) = O(\sqrt{N/B} + T/B)$, since we know that all points are reported in all but two cells in each of the $O(\sqrt{N/B}/\log_B N)$ slabs (i.e. the query-term in the query bound is dominated by the output term in all but $2 \cdot O(\sqrt{N/B}/\log_B N)$ kd-trees).

Updates. We utilize a global rebuilding strategy to update the O-tree efficiently: If N_0 is the number of points in the structure just after rebuilding it, we rebuild it again after $N_0/2$ updates using $O(\frac{N_0}{B}\log\frac{N_0}{B})$ I/Os or $O(\frac{1}{B}\log\frac{N_0}{B}) = O(\log_B N)$ I/Os amortized. By allowing the number of points in each slab to vary between $\frac{1}{4}\sqrt{N_0B}\log_B N_0$ and $\frac{5}{4}\sqrt{N_0B}\log_B N_0$ and in each cell between $\frac{1}{4}B\log_B^2 N_0$ and $\frac{5}{4}B\log_B^2 N_0$, we can then update the O-tree in $O(\log_B N)$ I/Os amortized as described below.

To *insert* a point p we first perform a point query to find the cell (kd-tree $\mathcal{T}_{ij}$) containing p. Then we insert p in $\mathcal{T}_{ij}$. If the cell containing p now contains more than $\frac{5}{4}B\log_B^2 N_0$ points we simply split it into two cells containing approximately $\frac{5}{8}B\log_B^2 N_0$ points each using a horizontal line, remove the kd-tree for the old cell, and construct two new kd-trees for the two new cells. We also insert the new horizontal line in the B-tree $\mathcal{T}_i$ for the slab s_i containing p. Similarly, if slab s_i now contains more than $\frac{5}{4}\sqrt{N_0B}\log_B N_0$ points we split it into two slabs containing approximately $\frac{5}{8}\sqrt{N_0B}\log_B N_0$ points each using a vertical line, insert the line in $\mathcal{T}_v$, and use $\Theta(\sqrt{N_0/B}/\log_B N_0)$ horizontal lines in each new slab to construct new cells containing between $\frac{1}{2}B\log_B^2 N_0$ and $B\log_B^2 N_0$ points each; we discard the B-trees $\mathcal{T}_i$ on the horizontal lines in the old slab and create two new B-trees for the two new slabs, and finally we construct a kd-tree on the points in each of the new cells.

To *delete* a point p we also first perform a point query to find the relevant kd-tree $\mathcal{T}_{ij}$ (cell containing p). Then we delete p from $\mathcal{T}_{ij}$. If the cell now contains less than $\frac{1}{4}B\log_B^2 N_0$ points we merge it with one of its neighbors: We remove the kd-trees for the two cells and collect the between $\frac{1}{2}B\log_B^2 N_0$ and $\frac{6}{4}B\log_B^2 N_0$ points in the cells; we also delete the horizontal splitting line between the cells from the B-tree $\mathcal{T}_i$ of the slab s_i containing p. If the number of collected points is between $\frac{1}{2}B\log_B^2 N_0$ and $B\log_B^2 N_0$ we then simply construct one new kd-tree for the new (merged) cell. Otherwise we split the set of points in two with a horizontal line, that is, we split the new cell in two cells containing between $\frac{1}{2}B\log_B^2 N_0$ and $\frac{3}{4}B\log_B^2 N_0$ points, insert the horizontal line in $\mathcal{T}_i$, and construct two new kd-trees. Similarly, if the slab s_i containing p now contains less than $\frac{1}{4}\sqrt{N_0B}\log_B N_0$ points we merge it with one of its neighbors: We delete the vertical splitting line between the two slabs from $\mathcal{T}_v$, delete the B-trees containing the horizontal lines from the two slabs, and remove all the kd-trees for the two slabs while collecting the between $\frac{1}{2}\sqrt{N_0B}\log_B N_0$ and $\frac{6}{4}\sqrt{N_0B}\log_B N_0$ points. If the number of collected points is between $\frac{1}{2}\sqrt{N_0B}\log_B N_0$ and $\sqrt{N_0B}\log_B N_0$ we simply use $\Theta(\sqrt{N_0/B}/\log_B N_0)$ horizontal lines in the (merged) slab to construct new cells containing between $\frac{1}{2}B\log_B^2 N_0$ and $B\log_B^2 N_0$ points each; we create a new B-tree on the lines for the slab, and construct a kd-tree for each of the new cells. Otherwise we first use a vertical split line to construct two new slabs containing between $\frac{1}{2}\sqrt{N_0B}\log_B N_0$ and $\frac{3}{4}\sqrt{N_0B}\log_B N_0$ points each, insert the line in $\mathcal{T}_v$, and in each slab construct $\Theta(\sqrt{N_0/B}/\log_B N_0)$ cells (B-tree and kd-trees) containing between $\frac{1}{2}B\log^2_B N_0$ and $B\log^2_B N_0$ points as above.

That the update algorithms use $O(\log_B N)$ I/Os amortized can be seen as follows (using Corollary 1 as well as Theorems 1 and 14): The initial point query takes $O(\log_B N_0) = O(\log_B N)$ I/Os. In the case where no cell or slab becomes too large or small the update is then finished with an update on an external kd-tree using $O(\log^2_B(B\log^2_B N_0) = O(\log_B N)$ I/Os. If a cell becomes too large or small (contains less than $\frac{1}{4}B\log^2_B N_0$ or more than $\frac{5}{4}B\log^2_B N_0$ points) and we perform $O(1)$ splits or merges, we in total perform $O(1)$ of updates on a B-trees $\mathcal{T}_i$ in slabs s_i using $O(\log_B(\sqrt{N_0/B}/\log_B N_0)) = O(\log_B N_0)$ I/Os, and remove and construct of $O(1)$ kd-trees of size $O(B\log^2_B N_0)$ using $O(\frac{B\log^2_B N_0}{B}\log\frac{B\log^2_B N_0}{B}) = O(\log^2_B N_0 \cdot \log\log^2_B N_0)$ I/Os. Since a newly created cell always contains between $\frac{1}{2}B\log^2_B N_0$ and $B\log^2_B N_0$ points, the amortized update bound is therefore $O(\log^2_B N_0 \cdot \log\log^2_B N_0)/(\frac{1}{4}B\log^2_B N_0) = O(\frac{\log\log^2_B N_0}{B}) = O(\log_B N)$. If a slab becomes too large or small (contains less than $\frac{1}{4}\sqrt{N_0 B}\log_B N_0$ or more than $\frac{5}{4}\sqrt{N_0 B}\log_B N_0$ points) and we perform $O(1)$ splits or merges of slabs, we in total perform $O(1)$ updates on B-tree $\mathcal{T}_v$ using $O(\log_B(\sqrt{N_0/B}/\log_B N_0) = O(\log_B N)$ I/Os. We also remove and construct $O(1)$ B-trees $\mathcal{T}_i$ in slabs S_i using $O(\frac{\sqrt{N_0/B}/\log_B N_0}{B}\log_{M/B}\frac{\sqrt{N_0/B}/\log_B N_0}{B}) = O(\frac{\sqrt{N_0/B}/\log_B N_0}{B}\log_B N_0)$ I/Os, and remove and construct $O(\sqrt{N_0/B}/\log_B N_0)$ kd-trees of size $O(B\log^2_B N_0)$ each using $O(\sqrt{N_0/B}/\log_B N_0)\cdot O(\log^2_B N_0 \cdot \log\log^2_B N_0)$ I/Os; the last bound dominates the first. Since the number of points in a newly created slab is always between $\frac{1}{2}\sqrt{N_0 B}\log_B N_0$ and $\sqrt{N_0 B}\log_B N_0$, the amortized update bound is therefore $O(\sqrt{N_0/B}/\log_B N_0)\cdot(\log^2_B N_0 \cdot \log\log^2_B N_0)/(\frac{1}{4}\sqrt{N_0 B}\log_B N_0) = O(\frac{\log\log^2_B N_0}{B}) = O(\log_B N)$. The $O(\log_B N)$ bound follows since an update is charged at most twice in the amortization argument (in a slab and a cell).

Theorem 15. *An O-tree for storing a set of N points in the plane uses linear space and supports point queries in $O(\log_B N)$ I/Os, orthogonal range query in $O(\sqrt{N/B} + T/B)$ I/Os, and updates in $O(\log_B N)$ I/Os amortized.*

Remarks. The internal memory kd-tree was developed by Bentley [26]. The static external version presented here is similar to the static version of the kdB-tree of Robinson [68], and the dynamic version is similar to the Bkd-tree of Agarwal et al. [66]. Several other dynamic external versions of the kd-tree (without worst-case performance guarantees) have been proposed (e.g. [68, 61, 42]; see also [70]). The logarithmic method was introduced by Bentley [27] (see also [65]). The external kd-tree update bounds can be improved slightly using an improved $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ construction algorithm due to Agarwal et al. [66, 5] and an external version of the logarithmic method (where $O(\log_B N)$ rather than $O(\log N)$ structures are maintained) due to Arge and Vahrenhold [21]. The amortized update bounds can also be made worst-case using standard lazy-rebuilding techniques [65].

The O-tree is due to Kanth and Singh [56]. The ideas used in the structure is similar to the ones utilized by van Kreveld and Overmars [76] in *divided k-d trees*. As for the external kd-tree, the O-tree construction bound (and thus exact update bounds) can be improve slightly and the amortized bounds can be made worst-case. The O-tree described here is slightly different than the structure of Kanth and Singh [56]. Grossi and Italiano [50, 51] developed a structure called a *cross-tree* obtaining the same bounds as the O-tree. The external kd-tree, the O-tree and the cross-tree can all be extended to d-dimensions in a straightforward way obtaining a $O((N/B)^{1-1/d} + T/B)$ query bound. These bounds are optimal for data structures that only store one copy of each data point [56, 9].

9 Conclusions

In this note we have discussed some of the recent advances in the development of provably efficient dynamic external memory data structures for one- and two-dimensional orthogonal range searching. We have discussed some of the most important techniques utilized to obtain efficient structures.

Even though a lot of progress has been made, many problems still remain open. For example, $O(\log_B N)$-query and space efficient structures still need to be found for many higher-dimensional problems. The practical performance of many of the worst-case efficient structures also needs to be researched.

Remarks. While this note only covers a few structures for one- and two-dimensional orthogonal range searching, a large number of worst-case efficient data structures for other (and often more complicated) problems have also be developed in recent years. These include structures for three- and higher-dimensional orthogonal range searching [78, 79, 50, 51, 56], variants such as rang counting, max, and stabbing queries [48, 82, 7, 20, 8], for halfspace range searching [45, 3, 2], for queries on moving objects [58, 2, 6, 4], for closest pair and nearest neighbor queries [30, 49, 2, 3], point location queries [47, 22, 18, 37, 74, 21, 1, 14], and for rectangle range searching [38, 9, 15]. This list is not meant to be exhaustive.

References

1. P. K. Agarwal, L. Arge, G. S. Brodal, and J. S. Vitter. I/O-efficient dynamic point location in monotone planar subdivisions. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, pages 1116–1127, 1999.
2. P. K. Agarwal, L. Arge, and J. Erickson. Indexing moving points. *Journal of Computer and System Sciences*, 66(1):207–243, 2003.
3. P. K. Agarwal, L. Arge, J. Erickson, P. Franciosa, and J. Vitter. Efficient searching with linear constraints. *Journal of Computer and System Sciences*, 61(2):194–216, 2000.
4. P. K. Agarwal, L. Arge, J. Erickson, and H. Yu. Efficient tradeoff schemes in data structures for querying moving objects. In *Proc. European Symposium on Algorithms, LNCS 3221*, pages 4–15, 2004.

5. P. K. Agarwal, L. Arge, O. Procopiuc, and J. S. Vitter. A framework for index bulk loading and dynamization. In *Proc. International Colloquium on Automata, Languages, and Programming, LNCS 2076*, pages 115–127, 2001.
6. P. K. Agarwal, L. Arge, and J. Vahrenhold. A time responsive indexing scheme for moving points. In *Proc. Workshop on Algorithms and Data Structures, LNCS 2076*, pages 50–61, 2001.
7. P. K. Agarwal, L. Arge, J. Yang, and K. Yi. I/O-efficient structures for orthogonal range max and stabbing max queries. In *Proc. European Symposium on Algorithms, LNCS 2832*, pages 7–18, 2003.
8. P. K. Agarwal, L. Arge, and K. Yi. An optimal dynamic interval stabbing-max data structure? In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, pages 803–812, 2005.
9. P. K. Agarwal, M. de Berg, J. Gudmundsson, M. Hammer, and H. J. Haverkort. Box-trees and R-trees with near-optimal query time. In *Proc. ACM Symposium on Computational Geometry*, pages 124–133, 2001.
10. P. K. Agarwal and J. Erickson. Geometric range searching and its relatives. In B. Chazelle, J. E. Goodman, and R. Pollack, editors, *Advances in Discrete and Computational Geometry*, volume 223 of *Contemporary Mathematics*, pages 1–56. American Mathematical Society, 1999.
11. A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.
12. 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.
13. L. Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.
14. L. Arge, A. Danner, and S.-H. Teh. I/O-efficient point location using persistent B-trees. In *Proc. Workshop on Algorithm Engineering and Experimentation*, 2003.
15. L. Arge, M. de Berg, H. J. Haverkort, and K. Yi. The priority R-tree: A practically efficient and worst-case optimal R-tree. In *Proc. SIGMOD International Conference on Management of Data*, pages 347–358, 2004.
16. L. Arge, P. Ferragina, R. Grossi, and J. Vitter. On sorting strings in external memory. In *Proc. ACM Symposium on Theory of Computation*, pages 540–548, 1997.
17. L. Arge, K. H. Hinrichs, J. Vahrenhold, and J. S. Vitter. Efficient bulk operations on dynamic R-trees. *Algorithmica*, 33(1):104–128, 2002.
18. L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Theory and practice of I/O-efficient algorithms for multidimensional batched searching problems. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, pages 685–694, 1998.
19. L. Arge, V. Samoladas, and J. S. Vitter. On two-dimensional indexability and optimal range search indexing. In *Proc. ACM Symposium on Principles of Database Systems*, pages 346–357, 1999.
20. L. Arge, V. Samoladas, and K. Yi. Optimal external memory planar point enclosure. In *Proc. European Symposium on Algorithms, LNCS 3221*, pages 40–52, 2004.
21. L. Arge and J. Vahrenhold. I/O-efficient dynamic planar point location. *Computational Geometry: Theory and Applications*, 29(2):147–162, 2004.
22. L. Arge, D. E. Vengroff, and J. S. Vitter. External-memory algorithms for processing line segments in geographic information systems. *Algorithmica*, 1998.
23. L. Arge and J. S. Vitter. Optimal external memory interval management. *SIAM Journal on Computing*, 32(6):1488–1508, 2003.

24. R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.
25. B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion B-tree. *VLDB Journal*, 5(4):264–275, 1996.
26. J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18:509–517, 1975.
27. J. L. Bentley. Decomposable searching problems. *Information Processing Letters*, 8(5):244–251, 1979.
28. G. Blankenagel and R. H. Güting. XP-trees—External priority search trees. Technical report, FernUniversität Hagen, Informatik-Bericht Nr. 92, 1990.
29. 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.
30. P. Callahan, M. T. Goodrich, and K. Ramaiyer. Topology B-trees and their applications. In *Proc. Workshop on Algorithms and Data Structures, LNCS 955*, pages 381–392, 1995.
31. B. Chazelle. Filtering search: a new approach to query-answering. *SIAM J. Comput.*, 15(3):703–724, 1986.
32. B. Chazelle. Lower bounds for orthogonal range searching: I. the reporting case. *Journal of the ACM*, 37(2):200–212, Apr. 1990.
33. Y.-J. Chiang and C. T. Silva. I/O optimal isosurface extraction. In *Proc. IEEE Visualization*, pages 293–300, 1997.
34. Y.-J. Chiang and C. T. Silva. External memory techniques for isosurface extraction in scientific visualization. In J. Abello and J. S. Vitter, editors, *External memory algorithms and visualization*, pages 247–277. American Mathematical Society, DIMACS series in Discrete Mathematics and Theoretical Computer Science, 1999.
35. Y.-J. Chiang, C. T. Silva, and W. J. Schroeder. Interactive out-of-core isosurface extraction. In *Proc. IEEE Visualization*, pages 167–174, 1998.
36. D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.
37. A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, and E. Ramos. Randomized external-memory algorithms for some geometric problems. *International Journal of Computational Geometry & Applications*, 11(3):305–337, 2001.
38. M. de Berg, J. Gudmundsson, M. Hammar, and M. Overmars. On R-trees with low stabbing number. In *Proc. European Symposium on Algorithms*, pages 167–178, 2000.
39. J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38:86–124, 1989.
40. H. Edelsbrunner. A new approach to rectangle intersections, part I. *Int. J. Computer Mathematics*, 13:209–219, 1983.
41. H. Edelsbrunner. A new approach to rectangle intersections, part II. *Int. J. Computer Mathematics*, 13:221–229, 1983.
42. G. Evangelidis, D. Lomet, and B. Salzberg. The hb^{π}-tree: A multi-attribute index supporting concurrency, recovery and node consolidation. *The VLDB Journal*, 6(1):1–25, 1997.
43. R. Fadel, K. V. Jakobsen, J. Katajainen, and J. Teuhola. Heaps and heapsort on secondary storage. *Theoretical Computer Science*, 220(2):345–362, 1999.
44. P. Ferragina and R. Grossi. A fully-dynamic data structure for external substring search. In *Proc. ACM Symposium on Theory of Computation*, pages 693–702, 1995.

45. P. G. Franciosa and M. Talamo. Orders, k-sets and fast halfplane search on paged memory. In *Proc. Workshop on Orders, Algorithms and Applications , LNCS 831*, pages 117–127, 1994.
46. V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.
47. M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *Proc. IEEE Symposium on Foundations of Computer Science*, pages 714–723, 1993.
48. S. Govindarajan, P. K. Agarwal, and L. Arge. CRB-tree: An efficient indexing scheme for range-aggregate queries. In *Proc. International Conference on Database Theory*, pages 143–157, 2003.
49. S. Govindarajan, T. Lukovszki, A. Maheshwari, and N. Zeh. I/O-efficient well-separated pair decomposition and its applications. In *Proc. European Symposium on Algorithms, LNCS 1879*, pages 220–231, 2000.
50. R. Grossi and G. F. Italiano. Efficient cross-tree for external memory. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*, pages 87–106. American Mathematical Society, 1999.
51. R. Grossi and G. F. Italiano. Efficient splitting and merging algorithms for order decomposable problems. *Information and Computation*, 154(1):1–33, 1999.
52. J. M. Hellerstein, E. Koutsoupias, and C. H. Papadimitriou. On the analysis of indexing schemes. In *Proc. ACM Symposium on Principles of Database Systems*, pages 249–256, 1997.
53. S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982.
54. C. Icking, R. Klein, and T. Ottmann. Priority search trees in secondary memory. In *Proc. Graph-Theoretic Concepts in Computer Science, LNCS 314*, pages 84–93, 1987.
55. 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.
56. 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.
57. D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading MA, second edition, 1998.
58. G. Kollios, D. Gunopulos, and V. J. Tsotras. On indexing mobile objects. In *Proc. ACM Symposium on Principles of Database Systems*, pages 261–272, 1999.
59. E. Koutsoupias and D. S. Taylor. Tight bounds for 2-dimensional indexing schemes. In *Proc. ACM Symposium on Principles of Database Systems*, pages 52–58, 1998.
60. V. Kumar and E. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proc. IEEE Symp. on Parallel and Distributed Processing*, pages 169–177, 1996.
61. 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.
62. E. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, 1985.
63. J. Nievergelt and E. M. Reingold. Binary search tree of bounded balance. *SIAM Journal on Computing*, 2(1):33–43, 1973.

64. J. Nievergelt and P. Widmayer. Spatial data structures: Concepts and design choices. In M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer, editors, *Algorithmic Foundations of GIS*, pages 153–197. Springer-Verlag, LNCS 1340, 1997.
65. M. H. Overmars. *The Design of Dynamic Data Structures.* Springer-Verlag, LNCS 156, 1983.
66. O. Procopiuc, P. K. Agarwal, L. Arge, and J. S. Vitter. Bkd-tree: A dynamic scalable kd-tree. In *Proc. International Symposium on Spatial and Temporal Databases, LNCS 2750*, 2003.
67. S. Ramaswamy and S. Subramanian. Path caching: A technique for optimal external searching. In *Proc. ACM Symposium on Principles of Database Systems*, pages 25–35, 1994.
68. J. Robinson. The K-D-B tree: A search structure for large multidimensional dynamic indexes. In *Proc. SIGMOD International Conference on Management of Data*, pages 10–18, 1981.
69. C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, 1994.
70. H. Samet. *The Design and Analyses of Spatial Data Structures.* Addison Wesley, MA, 1990.
71. V. Samoladas and D. Miranker. A lower bound theorem for indexing schemes and its application to multidimensional range queries. In *Proc. ACM Symposium on Principles of Database Systems*, pages 44–51, 1998.
72. N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29:669–679, 1986.
73. S. Subramanian and S. Ramaswamy. The P-range tree: A new data structure for range searching in secondary memory. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, pages 378–387, 1995.
74. J. Vahrenhold and K. H. Hinrichs. Planar point location for large data sets: To seek or not to seek. In *Proc. Workshop on Algorithm Engineering, LNCS 1982*, pages 184–194, 2001.
75. J. van den Bercken, B. Seeger, and P. Widmayer. A generic approach to bulk loading multidimensional index structures. In *Proc. International Conference on Very Large Databases*, pages 406–415, 1997.
76. M. J. van Kreveld and M. H. Overmars. Divided k-d trees. *Algorithmica*, 6:840–858, 1991.
77. P. J. Varman and R. M. Verma. An efficient multiversion access structure. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):391–409, 1997.
78. D. E. Vengroff and J. S. Vitter. Efficient 3-D range searching in external memory. In *Proc. ACM Symposium on Theory of Computation*, pages 192–201, 1996.
79. J. S. Vitter. External memory algorithms and data structures: Dealing with MASSIVE data. *ACM Computing Surveys*, 33(2):209–271, 2001.
80. J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory, I: Two-level memories. *Algorithmica*, 12(2–3):110–147, 1994.
81. D. E. Willard. Reduced memory space for multi-dimensional search trees. In *Symposium on Theoretical Aspects of Computer Science, LNCS 182*, pages 363–374, 1985.
82. D. Zhang, A. Markowetz, V. Tsotras, D. Gunopulos, and B. Seeger. Efficient computation of temporal aggregates with range predicates. In *Proc. ACM Symposium on Principles of Database Systems*, pages 237–245, 2001.