OPTIMAL EXTERNAL MEMORY INTERVAL MANAGEMENT[*]

LARS ARGE[†] AND JEFFREY SCOTT VITTER[‡]

Abstract. In this paper we present the external interval tree, an optimal external memory data structure for answering stabbing queries on a set of dynamically maintained intervals. The external interval tree can be used in an optimal solution to the dynamic interval management problem, which is a central problem for object-oriented and temporal databases and for constraint logic programming. Part of the structure uses a weight-balancing technique for efficient worst-case manipulation of balanced trees, which is of independent interest. The external interval tree, as well as our new balancing technique, have recently been used to develop several efficient external data structures.

1. Introduction. In recent years external memory data structures have been developed for a wide range of applications, including spatial, temporal, and object-oriented databases and geographic information systems. Often the amount of data manipulated in such applications is too large to fit in main memory, and the data must reside on disk. In such cases the input/output (I/O) communication between main memory and disk can become a bottleneck. In this paper we develop an I/O-optimal and space-optimal external interval tree data structure for answering stabbing queries among a changing set of intervals. The structure is the central part of an optimal solution to the dynamic interval management problem.

1.1. Memory model and previous results. We will be working in the standard model for external memory with one (logical) disk [31, 4]. We assume that each external memory access (called an *I/O operation* or just *I/O*) transmits one page of B elements. We measure the efficiency of an algorithm in terms of the number of I/Os it performs and the number of disk blocks it uses.[1]

The *dynamic interval management problem* is the problem of maintaining a set of intervals such that, given a query interval I_q, all intervals intersecting I_q can be reported efficiently. As discussed in [29, 30, 38], the problem is crucial for indexing constraints in constraint databases and in temporal databases. The key component of dynamic interval management is the ability to answer *stabbing queries* [30]. Given a set of intervals, a stabbing query with a point q asks for all intervals containing q.

By representing an interval $[x, y]$ as the point (x, y) in the plane, a stabbing query reduces to the special case of 2-sided 2-dimensional range searching called a *diagonal corner query*. In the diagonal corner query problem a set of points in the plane above the diagonal line $x = y$ should be stored such that, given a query point (q, q), all points (x, y) with $x \leq q$ and $y \geq q$ can be reported efficiently. The problem of 2-dimensional range searching has been the subject of much research. While B-trees and their variants [12, 21] have been an unqualified success in supporting 1-dimensional external range searching, they are inefficient at handling higher-dimensional problems. In internal memory many worst-case efficient structures have been proposed for 2-dimensional and higher-dimensional range search; see [3] for a survey. Unfortunately, most of these structures are not efficient when mapped to external memory. The practical need for I/O support has also led to the development of a large number of external data structures that do not have good theoretical worst-case update and query I/O bounds but do have good average-case behavior for common problems; see [24, 35] for surveys. The worst-case performance of these data structures is much worse than the optimal bounds achievable for dynamic external 1-dimensional range search using a B-tree.

Prior to the development of the structure presented in this paper, a number of attempts had been made to solve the external stabbing query problem. Kanellakis et al. [30] developed the metablock tree for answering diagonal corner queries in optimal $O(\log_B N + T/B)$ I/Os using optimal $O(N/B)$ blocks of external memory. Here T denotes the number of points reported. The structure supports insertions only in $O(\log_B N + (\log_B^2 N)/B)$ I/Os amortized. A simpler static structure with the same bounds was described by Ramaswamy [37]. In internal memory, the priority search tree of McCreight [32] can be used to answer more general queries than diagonal corner queries, namely 3-sided range queries, and a number of attempts have been made at externalizing this structure [16, 28, 39]. The structure by Icking, Klein, and Ottoman [28] uses optimal space but answers queries in $O(\log_2 N + T/B)$ I/Os. The structure by Blankenagel and Güting [16] also uses optimal space but answers queries in $O(\log_B N + T)$ I/Os (see also [14]). In both papers a number of nonoptimal dynamic versions of the structures are also developed. Ramaswamy and Subramanian [39] developed a technique called *path caching* for transforming an efficient internal memory data structure into an I/O-efficient structure. Using this technique on the priority search tree results in a structure that can be used to answer 2-sided queries, which are more general than diagonal corner queries but less general than 3-sided queries. This structure answers queries in the optimal $O(\log_B N + T/B)$ I/Os and supports updates in amortized $O(\log_B N)$ I/Os but uses nonoptimal $O((N/B) \log_2 \log_2 B)$ space. Various other external data structures for answering 3-sided queries are also developed in [30] and [39]. Subramanian and Ramaswamy also designed the p-range tree for answering 3-sided queries [40]. The structure uses linear space, answers queries in $O(\log_B N + T/B + IL^*(B))$ I/Os, and supports updates in $O(\log_B N + (\log_B^2 N)/B)$ I/Os amortized. ($IL^*(\cdot)$ denotes the iterated log* function, that is, the number of times log* must be applied to get below 2). Finally, following the publication of the extended abstract version of this paper (and based on the results in this paper), Arge, Samoladas, and Vitter [8] developed an optimal external priority search tree, immediately implying an optimal stabbing query structure. Several structures have also been developed for the general 2- and higher-dimensional range searching problem, as well as for several other related problems. See [5, 6, 41] for surveys.

TABLE 1.1
Comparison of our data structure for stabbing queries with other data structures.

	Space (blocks)	Query I/O bound	Update I/O bound
Pri. search tree [28]	$O(\frac{N}{B})$	$O(\log_2 N + T/B)$	
XP-tree [16]	$O(\frac{N}{B})$	$O(\log_B N + T)$	
[37]	$O(\frac{N}{B})$	$O(\log_B N + T/B)$	
Metablock tree [30]	$O(\frac{N}{B})$	$O(\log_B N + T/B)$	$O(\log_B N + (\log_B N)^2/B)$ amortized (inserts only)
P-range tree [40]	$O(\frac{N}{B})$	$O(\log_B N + T/B + IL^*(B))$	$O(\log_B N + (\log_B N)^2/B)$ amortized
Path caching [39]	$O(\frac{N}{B} \log_2 \log_2 B)$	$O(\log_B N + T/B)$	$O(\log_B N)$ amortized
Our result [11] ([8])	$O(\frac{N}{B})$	$O(\log_B N + T/B)$	$O(\log_B N)$

1.2. Overview of our results. The main contribution of this paper is an optimal external memory data structure for the stabbing query problem. As mentioned, our data structure gives an optimal solution to the interval management problem, and thus it settles an open problem highlighted in [30, 39, 40]. The structure uses $O(N/B)$ disk blocks to maintain a set of N intervals such that insertions and deletions can be performed in $O(\log_B N)$ I/Os and such that stabbing queries can be answered in $O(\log_B N + T/B)$ I/Os. In Table 1.1 we compare our result with previous solutions. Unlike previous nonoptimal structures, the update I/O bounds for our data structure are worst-case, and our structure works without assuming that the internal memory is capable of holding $\Omega(B^2)$ elements. Our structure is significantly different from the recently developed external priority search tree [8] and is probably of greater practical interest since it uses relatively fewer random I/Os when answering a query. Most disk systems are optimized for sequential I/O, and, consequently, random I/Os often take a much longer time than sequential I/Os.

Our solution to the stabbing query problem is an external version of the interval tree [22, 23]. In section 2, we present the basic structure, where the endpoints of the intervals stored in the structure belong to a fixed set of N points. In section 3, we then remove this "fixed endpoint-set assumption." In internal memory, the assumption is normally removed using a BB[α]-tree [34] as the base search tree structure [33], and this leads to amortized update bounds. However, as BB[α]-trees are unsuitable for implementation in external memory, we develop a new *weight-balanced B-tree* for use in external memory. This structure resembles the *k-fold tree* of Willard [43]. Like in internal memory, the use of a weight-balanced B-tree as the base tree results in amortized update bounds. In section 4, we then show how to remove the amortization from the structure.

Our external interval tree has been used to develop I/O-efficient structures for dynamic point location [1, 9].[2] It has also been used in several visualization applications [18, 19, 20]. Our weight-balanced B-tree has also found several other applications. In internal memory it can, for example, be used to convert amortized bounds to worst-case bounds. (Fixing B to a constant in our result yields an internal-memory interval tree with worst-case update bounds.) It can also be used as a (simpler) alternative to the rather complicated structure developed in [42] in order to add range restriction capabilities to internal-memory dynamic data structures. (It seems possible to

---

[2]Even though the external priority search tree [8] solves a more general problem than the external interval tree, it cannot be used as an alternative to the external interval tree in the point location structures.

use the techniques in [42] to remove the amortization from the update bound of the internal interval tree, but our method is much simpler.) In external memory, it has been used in the recently developed optimal external priority search tree [8], as well as in numerous other structures (e.g., [25, 26, 13, 1, 9]).

Finally, in section 5, we discuss how to use the ideas utilized in our external interval tree to develop an external segment tree using $O((N/B) \log_B N)$ space. This improves upon previously known external segment tree structures, which use $O((N/B) \log_2 N)$ disk blocks [15, 39].

2. External memory interval tree with fixed endpoint set. In this section, we present our external interval tree structure, assuming that the endpoints of the intervals stored in the structure belong to a fixed set E of size N. We also assume that the internal memory is capable of holding $O(B)$ blocks. We remove these assumptions in sections 3 and 4.

2.1. Preliminaries. Our external interval tree makes extensive use of two kinds of auxiliary structures: the B-tree [12, 21] and the "corner structure" [30]. B-trees, or more generally (a, b)-trees [27], are search tree structures suitable for external memory.

LEMMA 2.1. *A set of N elements can be stored in a B-tree structure using $O(N/B)$ disk blocks such that updates and queries can be performed in $O(\log_B N)$ I/Os. The T smallest (largest) elements can be reported in $O(T/B + 1)$ I/Os. Given N sorted elements a B-tree can be built in $O(N/B)$ I/Os.*

A "corner structure" [30] is a data structure that can be used to answer stabbing queries on $O(B^2)$ intervals.

LEMMA 2.2. *(Kanellakis et al. [30]) A set of $K \leq B^2$ intervals can be stored in an external data structure using $O(K/B)$ disk blocks such that a stabbing query can be answered in $O(T/B + 1)$ I/Os, where T is the number of reported intervals.*

As discussed in [30], the corner structure can easily be made dynamic: updates are inserted into an update block, and the structure is rebuilt using $O(B)$ I/Os once B updates have been performed. The rebuilding is performed simply by loading the structure into internal memory, rebuilding it, and writing it back to external memory.

LEMMA 2.3. *Assuming $M \geq B^2$, a set of $K \leq B^2$ intervals can be stored in an external data structure using $O(K/B)$ disk blocks such that a stabbing query can be answered in $O(T/B + 1)$ I/Os and such that an update can be performed in $O(1)$ I/Os amortized. The structure can be constructed in $O(K/B)$ I/Os.*

In section 4.2 (where it will become clearer why the structure is called a "corner structure"), we show how the update bound can be made worst-case. In the process we also remove the assumption on the size of the internal memory.

2.2. The structure. An internal memory interval tree consists of a binary base tree on the sorted set of endpoints E, with the intervals stored in secondary structures associated with internal nodes of the tree [22]. 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 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}). 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 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.

In order to externalize the interval tree structure in an efficient way, we need to increase the fan-out of the base tree to decrease its height to $O(\log_B N)$. This creates several problems. The main idea behind our successful externalization of the structure, as compared with previous attempts [16, 39], is to use a fan-out of $\sqrt{B}$ instead of B (following ideas from [7, 10]).

Structure. The external interval tree on a set of intervals I with endpoints in a fixed set E of size N is defined as follows. (We assume without loss of generality that the endpoints of the intervals in I are distinct.) The *base tree* $\mathcal{T}$ is a perfectly balanced fan-out $\sqrt{B}$ tree over the sorted set of endpoints E. Each leaf represents B consecutive points from E. (If $|E|$ is not $(\sqrt{B})^i B$ for some $i \geq 0$ we adjust the degree of the root of $\mathcal{T}$ to be smaller than $\sqrt{B}$.) The tree has height $O(\log_{\sqrt{B}}(N/B)) + 1 = O(\log_B N)$. 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 $\sqrt{B}$ subintervals by the intervals associated with the children $v_1, v_2, \ldots, v_{\sqrt{B}}$ of v. Refer to Figure 2.1. 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 2.1. In a node v 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 l we store intervals with both endpoints among the endpoints in l. 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 $\sqrt{B} - 1$ slab boundaries b_i, $1 < i \leq \sqrt{B}$, we store the following:
  - 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 and the underflow structure using a corner structure (Lemmas 2.1 and 2.3). In each node v, in $O(1)$ *index blocks*, we also maintain information about the size and place of each
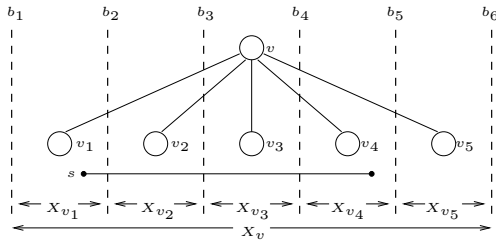
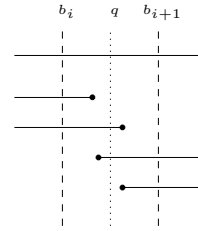FIG. 2.1. *A node in the base tree. Interval s is stored in L_2, R_4, and either $M_{2,4}$ or U.*

FIG. 2.2. *Intervals containing q are stored in R_{b_i}, $L_{b_{i+1}}$, the multislab lists spanning the slab, and U.*

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 2.1 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.

The external interval tree uses linear space: the base tree $\mathcal{T}$ itself uses $O(|E|/B)$ blocks, and each interval is stored in a constant number of linear space secondary structures (Lemmas 2.1 and 2.3). 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 $O(|E|/(B\sqrt{B}))$ internal nodes, the structure uses a total of $O(|E|/B)$ 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 superlinear 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 in I_v as follows:

- We load the $O(1)$ index blocks.
- We 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 $k > i$.
- We perform a stabbing query with q on the underflow structure U and report the result.
- We report intervals in R_i from the largest toward the smallest (according to right endpoint) until we encounter an interval not containing q.
- We report 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 stored either 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 2.2. We correctly report 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}.

The query algorithm uses an optimal $O(\log_B N + T/B)$ I/Os. In $\mathcal{T}$ we visit $O(\log_{\sqrt{B}} N) = O(\log_B N)$ nodes. In each node v we use only $O(1)$ I/Os that are not "paid for" by reportings (blocks read that contain $\Theta(B)$ output intervals): we use $O(1)$ I/Os to load the index blocks, $O(1)$ overhead to query U, and $O(1)$ overhead for R_i and L_{i+1}. Note how U is crucial for obtaining the $O(\log_B N + T/B)$ bound since it guarantees that all visited multislab lists contain $\Theta(B)$ intervals.

LEMMA 2.4. *Assuming $M \geq B^2$, there exists a data structure using $O(N/B)$ disk blocks to store intervals with unique endpoints in a set E of size N such that stabbing queries can be answered in $O(\log_B N + T/B)$ I/Os.*

Updates. We insert a new interval s in the external interval tree as follows: we search down $\mathcal{T}$ to find the first node v where s contains one or more slab boundaries. Then we load the $O(1)$ index blocks of v and 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 $M_{i,j}$. Finally, we update and store the index blocks. Similarly, in order to delete an interval s, we search down $\mathcal{T}$ until we find the node storing s. We then 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 again update and store the index blocks.

To analyze the number of I/Os used to perform an update, first note that 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. The manipulation of the underflow structure U uses $O(1)$ I/Os, except in the cases where $\Theta(B)$ intervals are moved between U and a multislab list $M_{i,j}$. In the latter case we use $O(B)$ I/Os, but then there must have been at least $B/2$ updates involving intervals in $M_{i,j}$ and requiring only $O(1)$ I/Os since the last time an $O(B)$ cost was incurred. Hence the amortized I/O cost is $O(1)$, and we obtain the following.

THEOREM 2.5. *Assuming $M \geq B^2$, there exists a data structure using $O(N/B)$ disk blocks to store intervals with unique endpoints in a set E of size N such that stabbing queries can be answered in $O(\log_B N + T/B)$ I/Os in the worst case and such that updates can be performed in $O(\log_B N)$ I/Os amortized.*

3. General external interval tree. In order to remove the fixed endpoint assumption from our external interval tree, we need to use a dynamic search tree as the base tree. In internal memory a BB[α]-tree [34] is often used as the base tree for structures with secondary structures. In such a tree, a node v with weight w (i.e., with w elements below it) can be involved in a rebalancing operation only once for every $\Omega(w)$ updates that access (i.e., pass through) v [17, 33]. If the necessary reorganization of the secondary structures after a rebalance operation on v can be performed in $O(w)$ time, we then obtain an $O(1)$ amortized bound on performing a rebalancing operation. Unfortunately, a BB[α]-tree is not suitable for implementation in external memory; it is binary, and there seems to be no easy way of grouping nodes together in order to increase the fan-out while at the same time maintaining its other useful properties. On the other hand, a B-tree, which is the natural choice as dynamic base structure, does not have the property that a node v of weight w can be involved in a rebalance operation only for every $\Omega(w)$ updates accessing v.

In section 3.1, we describe a variant of B-trees, called *weight-balanced B-trees*, combining the useful properties of B-trees and BB[α]-trees. They are balanced using normal B-tree operations (split and fusion of nodes) while at the same time having the weight property of a BB[α]-tree. An important feature of a weight-balanced B-tree is that the ratio between the largest and smallest weight subtree rooted in children of a node v is a small constant factor. In a B-tree this ratio can be exponential in the height of the subtrees. In section 3.2, we use the weight-balanced B-tree to remove the fixed endpoint assumption from our external interval tree.

3.1. Weight-balanced B-tree. In a normal B-tree [12, 21] all leaves are on the same level, and each internal node has between a and $2a-1$ children for some constant a. In a weak B-tree, or (a, b)-tree [27], a wider range in the number of children is allowed. We define the weight-balanced B-tree by imposing constraints on the weight of subtrees rather than on the number of children. The other B-tree characteristics remain the same: the leaves are all on the same level (level 0), and rebalancing is performed by splitting and fusing internal nodes.

DEFINITION 3.1. *The weight $w(v_l)$ of a leaf v_l is defined as the number of elements stored in it. The weight of an internal node v is defined as $w(v) = \sum_{v=parent(c)} w(c)$.*

COROLLARY 3.2. *The weight $w(v)$ of an internal node v is equal to the number of elements in leaves below v.*

DEFINITION 3.3. *$\mathcal{T}$ is a weight-balanced B-tree with branching parameter a and leaf parameter k, $a > 4$ and $k > 0$, if the following conditions hold:*
- *All leaves of $\mathcal{T}$ are on the same level and have weight between k and $2k - 1$.*
- *An internal node on level l has weight less than $2a^l k$.*
- *Except for the root, an internal node on level l has weight larger than $\frac{1}{2}a^l k$.*
- *The root has more than one child.*

LEMMA 3.4. *Except for the root, all nodes in a weight-balanced B-tree with parameters a and k have between $a/4$ and $4a$ children. The root has between 2 and $4a$ children.*

Proof. The leaves fulfill the internal node weight constraint, since $k > \frac{1}{2}a^0 k$ and $2k - 1 < 2a^0 k$. Thus the minimal number of children an internal node on level l can have is $\frac{1}{2}a^l k / 2a^{l-1} k = a/4$, and the maximal number of children v can have is $2a^l k / \frac{1}{2}a^{l-1} k = 4a$. The root upper bound follows from the same argument, and the lower bound is by definition. □

COROLLARY 3.5. *The height of an N element weight-balanced B-tree with parameters a and k is $O(\log_a(N/k))$.*

To perform an *update* on a weight-balanced B-tree $\mathcal{T}$, we first search down $\mathcal{T}$ for the relevant leaf. After performing the actual update, we may need to rebalance $\mathcal{T}$ in order to fulfill the constraints in Definition 3.3. For simplicity we consider only insertions in this section. Deletions can easily be handled using global rebuilding [36] (as discussed further in the next section). After inserting an element in leaf u of $\mathcal{T}$, the nodes on the path from u to the root of $\mathcal{T}$ can be out of balance; that is, the node v_l on level l can have weight $2a^l k$. In order to rebalance the tree we split all such nodes starting with u and working towards the root. If u is a leaf containing $2k$ elements we split it into two leaves u and u', each containing k elements, and insert a reference to u' in $parent(u)$. In general, on level l we want to split a node v_l of weight $2a^l k$ into two nodes v_l' and v_l'' of weight $a^l k$ and insert a reference in $parent(v_l)$. (If $parent(v_l)$ does not exist, that is, if we are splitting the root, we create a new root with two children.) However, a perfect split is generally not possible if we want to perform the split so that v_l' gets the first (leftmost) i of v's children and v_l'' gets the rest of the

children. Nonetheless, since nodes on level $l-1$ have weight less than $2a^{l-1}k$, we can always find an i such that if we split at the ith child the weights of both v_l' and v_l'' are between $a^lk - 2a^{l-1}k$ and $a^lk + 2a^{l-1}k$. Since $a > 4$, v_l' and v_l'' fulfill the constraints of Definition 3.3; that is, their weights are strictly between $\frac{1}{2}a^lk$ and $2a^lk$.[3] Note that splitting node v does not change the weight of $parent(v)$. As a result, the structure is relatively simple to implement. In each node we need only to store its level and the weight of each of its children, information we can easily maintain during an update. The previous discussion and Corollary 3.5 combine to prove the following.

LEMMA 3.6. *The number of rebalancing operations (splits) after an insertion in a weight-balanced B-tree $\mathcal{T}$ with parameters a and k is bounded by $O(\log_a(|T|/k))$.*

The following lemma will be crucial in our application.

LEMMA 3.7. *After a split of a node v_l on level l into two nodes v_l' and v_l'', at least $a^lk/2$ inserts have to be performed below v_l' (or v_l'') before it splits again. After a new root r in a tree containing N elements is created, at least $3N$ insertions have to be performed before r splits again.*

Proof. After a split of v_l the weight of each of v_l' and v_l'' is less than $a^lk+2a^{l-1}k < 3/2a^lk$. Each such node will split again when its weight reaches $2a^lk$. It follows that the weight must increase by at least $a^lk/2$. When a root r is created on level l it has weight $2a^{l-1}k = N$. It will not split before it has weight $2a^lk > 2 \cdot 4a^{l-1}k = 4N$. □

One example of how the weight-balanced B-tree can be used as a simpler alternative to existing *internal memory* data structures is in adding range restriction capabilities to dynamic data structures [42]. The general technique for adding range restrictions developed in [42] utilizes a base BB[α]-tree with each interval node v augmented with a dynamic data structure on the set of elements below v. This structure needs to be rebuilt when a rebalancing operation is performed on v, and the use of a BB[α]-tree leads to amortized bounds. In [42] it is shown how worst-case bounds can be obtained by a relatively complicated redefinition of the BB[α]-tree. On the other hand, using our weight-balanced B-tree with branching parameter $a = 5$ and leaf parameter $k = 1$ as base tree we immediately obtain worst-case bounds: the large number of updates between splits of a node immediately implies good amortized bounds, and the bounds can easily be made worst-case by performing the secondary structure rebuilding lazily. The ideas and techniques used in this construction are very similar to the ones presented in the succeeding sections of this paper and are therefore omitted. The use of lazy rebuilding in the BB[α]-tree solution is complicated because rebalancing is performed using rotations, which means that we cannot simply continue to query and update the old secondary structure while lazily building new ones.

As mentioned, the weight-balanced B-tree has been used in the development of numerous efficient internal as well as external data structures (e.g., [26, 13, 8, 25, 9, 2, 41]). In order to obtain an external tree structure suitable for use in our interval tree, we choose $4a = \sqrt{B}$ and $2k = B$ and obtain the following.

THEOREM 3.8. *There exists an N element search tree data structure using $O(N/B)$ disk blocks such that a search or an insertion can be performed in $O(\log_B N)$ I/Os in the worst case.*

Each internal node v at level l in the structure, except for the root, has $\Theta(\sqrt{B})$ children, dividing the $w(v) = \Theta((\sqrt{B})^l B)$ elements below v into $\Theta(\sqrt{B})$ sets. The root

---

[3]If $a > 8$ we can even split the node at child $i-1$ or $i+1$ instead of at child i and still fulfill the constraints. We will use this property in the next section.

has $O(\sqrt{B})$ *children. Rebalancing after an insertion is performed by splitting nodes. When a node v is split, at least $\Theta(w(v))$ elements must have been inserted below v since the last time v was split. In order for a new root to be created, $\Theta(N)$ elements have to be inserted into the data structure.*

Proof. Each internal node can be represented using $O(1)$ blocks, and the space bound follows since each leaf contains $\Theta(B)$ elements. A split at v can be performed in $O(1)$ I/Os: we load the $O(1)$ blocks storing v into internal memory, split v, and write the $O(1)$ blocks defining the two new nodes back to disk. Finally, we update the information in the parent using $O(1)$ I/Os. Thus the insertion and search I/O bounds follow directly from Corollary 3.5 and Lemma 3.6.

The second part of the theorem follows directly from Definition 3.3, Corollary 3.2, and Lemmas 3.4 and 3.7. □

3.2. Using the weight-balanced B-tree to remove the fixed endpoint assumption. We now show how to remove the fixed endpoint assumption from our external interval tree using the weight-balanced B-tree as the base tree $\mathcal{T}$. To insert an interval, we first insert the two new endpoints in the base tree and perform the necessary rebalancing. Then we insert the interval as described in section 2. Since rebalancing is performed by splitting nodes, we need to consider how to split a node v in our interval tree. Figure 3.1 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 *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 *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 *parent*(v) containing b also need to be moved to new secondary structures. Refer to Figure 3.2.
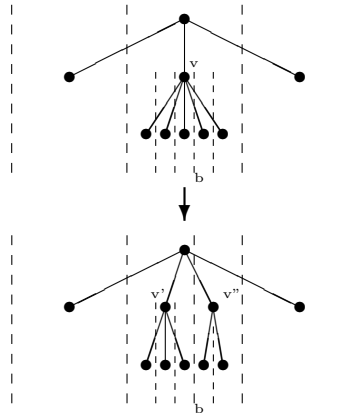

FIG. 3.1. *Splitting a node; v splits along b, which becomes a new boundary in parent(v).*

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 *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 every right slab list (stored in the leaves of a B-tree) of v in order, starting with the rightmost slab boundary,
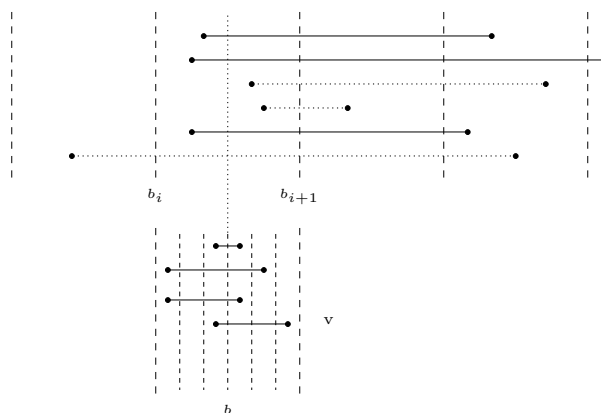
FIG. 3.2. *All solid intervals need to move. Intervals in v containing b move to parent(v), and some intervals move within parent(v).*

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 can scan through each of the $O(\sqrt{B})$ slab lists in a linear number of I/Os (Lemma 2.1), we construct L_r and L_l in $O(\sqrt{B} + w(v)/B) = O(w(v)/B)$ 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 $O(w(v)/B)$ I/Os. We construct the multislab lists for v' and v'' simply by removing all multislabs lists containing b. Since each removed list contains $\Omega(B)$ intervals, we can do so in $O(w(v)/B)$ 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 $O(w(v)/B)$ I/Os (Lemma 2.3). We complete the construction of v' and v'' in $O(w(v)/B)$ 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 consider only 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 in $O(|X_v|/B) = O(w(v)/B)$ 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 $O(w(v)/B)$ I/Os (Lemma 2.1), 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 $O(w(v)/B)$ 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 $O(B) = O(w(v)/\sqrt{B})$ 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 $O(w(v)/\sqrt{B})$ I/Os, and since $O(w(v))$ endpoints must have been inserted below v since it was constructed (Theorem 3.8), the amortized cost of a split is $O(1/\sqrt{B})$ I/Os. Since $O(\log_B N)$ nodes split during an insertion, we obtain the following.

LEMMA 3.9. *Assuming $M \geq B^2$, there exists a data structure using $O(N/B)$ disk blocks to store N intervals such that stabbing queries can be answered in $O(\log_B N + T/B)$ I/Os in the worst case and such that an interval can be inserted in $O(\log_B N)$ I/Os amortized.*

As mentioned in section 3.1, deletions can be handled using global rebuilding [36]. To delete an interval s we first delete it from the secondary structures as described in section 2 *without* deleting the endpoints of s from the base tree $\mathcal{T}$. Instead we just mark the two endpoints in the leaves of the base tree 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. After $N/2$ deletions have been performed we rebuild the structure in $O(N \log_B N)$ I/Os, leading to an $O(\log_B N)$ amortized delete I/O bound: first we scan through the leaves of the old base tree and construct a sorted list of the undeleted endpoints. This list is then used to construct the new base tree. All of this can be done in $O(N/B)$ I/Os. Finally, we insert the $O(N)$ intervals one by one without rebalancing the base tree, using $O(N) \cdot O(\log_B N)$ I/Os.

THEOREM 3.10. *Assuming $M \geq B^2$, there exists an external interval tree using $O(N/B)$ disk blocks to store N intervals such that stabbing queries can be answered in $O(\log_B N + T/B)$ I/Os in the worst case and such that updates can be performed in $O(\log_B N)$ I/Os amortized.*

4. Removing amortization. In this section, we discuss how to make the update bound of Theorem 3.10 worst-case. Amortized bounds appeared in several places in our structure; in the fixed endpoint version of our structure, amortization was introduced as a result of the amortized $O(1)$ update bound of the underflow structure (Lemma 2.3), as well as when moving intervals between the underflow and multislab lists. In the dynamic base tree version, amortization was introduced in the amortized node split bound (insertions) as well as in the use of global rebuilding (deletions).

In sections 4.1 and 4.2, we show how to remove amortization from the node split and underflow (corner structure) update bounds, respectively. In section 4.2, we also show how to remove the $M \geq B^2$ assumption from the corner structure and thus from our external interval tree. The remaining amortization can be removed using standard lazy global rebuilding techniques [36]. We make the global rebuilding (deletion) bound worst-case as follows: instead of using $O(N \log_B N)$ I/Os to rebuild the entire structure when the number of endpoints fall below $N/2$, we distribute the rebuilding over the next $1/3 \cdot N/2$ updates using $O(\log_B N)$ I/Os on rebuilding at each update. We use and update the original structure while constructing the new structure. When the new structure is completed the $1/3 \cdot N/2$ updates that occurred after the rebuilding started still need to be performed in the new structure. We

perform these updates during the next $1/3 \cdot (1/3 \cdot N/2)$ operations. This process continues until both structures store the same set of intervals (with at least $(1 - (1/3 + 1/9 + \cdots))N/2 \geq 1/2 \cdot N/2$ endpoints). Then we dismiss the old structure and use the new one instead. Since the rebuilding is finished before the structure contains only $N/4$ endpoints, we are in the process of constructing at most one new structure at any given time. The amortization introduced when moving intervals between the underflow structure and multislab lists can be removed in a similar way: recall that we moved B intervals using $O(B)$ I/Os when the underflow structure contained B intervals belonging to the same multislab list $M_{i,j}$ and when the number of intervals in a multislab list fell below $B/2$. We remove this amortization by moving the intervals over $B/4$ updates. If the size of a multislab list $M_{i,j}$ falls to $B/2$, we move two intervals from $M_{i,j}$ to the underflow structure over each of the next $B/4$ insertions or deletions involving $M_{i,j}$. Insertions themselves are also performed on the underflow structure, and deletions are performed on the underflow structure or $M_{i,j}$. When all intervals are moved there are between $\frac{1}{4}B$ and $\frac{3}{4}B$ intervals belonging to $M_{i,j}$ stored in U. Similarly, if the number of intervals in the underflow structure belonging to $M_{i,j}$ reaches B, we move the B intervals during the next $B/4$ updates involving $M_{i,j}$. Even though this way $M_{i,j}$ can contain $o(B)$ intervals, the optimal space and query bounds are maintained since $M_{i,j}$ and U together contain $\Theta(B)$ intervals during the process. This proves our main result.

THEOREM 4.1. *There exists an external interval tree using $O(N/B)$ disk blocks to store N intervals such that stabbing queries can be answered in $O(\log_B N + T/B)$ I/Os in the worst case and such that updates can be performed in $O(\log_B N)$ I/Os in the worst case.*

4.1. Splitting nodes lazily. Recall that when a node v splits along a boundary b the intervals in v containing b need to be moved to $parent(v)$, and some of the intervals in $parent(v)$ containing b need to move internally in $parent(v)$ (Figure 3.2). In section 3, we showed how to move the intervals in $O(w(v)/\sqrt{B})$ I/Os, and since $O(w(v))$ updates have to be performed below v between splits (Theorem 3.8) we obtained an $O(1/\sqrt{B})$ amortized split bound. When performing an update in a leaf l it affects the weight of $O(\log_B N)$ nodes on the path from the root to l. These nodes are all *accessed* in the search for l performed before the actual insertion. In this section we show how to split a node v (move the relevant intervals) lazily using $O(1)$ I/Os during the next $O(w(v))$ updates accessing v while still being able to query the secondary structures of v efficiently. This way we are done splitting v before a new split is needed, and we obtain an $O(1)$ worst-case split bound.

Our lazy node splitting algorithm works as follows. When v needs to be split along a boundary b, in $O(1)$ I/Os we first insert b as a *partial* slab boundary into $parent(v)$. The boundary remains partial until we have finished the split. In order to keep different split processes from interfering with each other, we want to avoid splitting nodes along partial boundaries. Since v, or rather the nodes it splits into, cannot split again as long as b is partial, at most every second boundary in $parent(v)$ can be partial. As discussed in section 3.1 (footnote 3), this means that we can always split a node along a nonpartial boundary without violating the constraints on the base weight-balanced B-tree $\mathcal{T}$. Next we move the relevant intervals in two phases: in an *up phase*, we first construct the new secondary structures for v' and v'' as before, by removing the intervals in secondary structures of v containing b and collecting them in two sorted lists L_l and L_r. Below we show how to do so lazily using $O(1)$ I/Os over $O(w(v)/\sqrt{B})$ updates accessing v so that we can still

query and update the "old" secondary structures of v. During this phase we can therefore perform queries as before, simply by ignoring the partial slab boundary b in $parent(v)$. Next, in the *rearrange phase*, in $O(1)$ I/Os, we split v into v' and v'' by switching to the new structures and splitting the index blocks. We also associate the lists L_l and L_r with the partial boundary b in $parent(v)$. Then we move the relevant intervals in $parent(v)$ containing b by constructing new updated versions of all secondary structures containing intervals with endpoints in X_v. Below we discuss how to do so lazily over $O(w(v)/\sqrt{B})$ updates accessing v (v' and v'') so that the "old" structures can still be queried and updated. In order to answer queries between b_i and b_{i+1} in $parent(v)$ correctly during the rearrange phase, we first perform a query as before while ignoring the partial slab boundary b. To report the relevant intervals among the intervals we have removed from v and inserted into L_l and L_r, we then query the relevant one of the two slab lists. Since this adds only $O(1)$ I/Os to the query procedure in $parent(v)$, the optimal query bound is maintained. At the end of the rearrange phase, we finish the split of v in $O(1)$ I/Os by switching to the new structures and marking b as a normal slab boundary.

All that remains is to describe how to perform the up and rearrange phases. Each of these phases can be performed lazily using $O(1)$ I/Os during each of $O(w(v)/\sqrt{B})$ updates accessing v. However, our algorithms will assume that only one up or rearrange phase is in progress on a node v at any given time, which means that when we want to perform a phase on v we might need to wait until we have finished another phase. In fact, other phases may also be waiting, and we may need to wait until v has been accessed enough times for us to finish all of them. Luckily, since an up and rearrange phase requires only $O(w(v)/\sqrt{B})$ accesses to finish, and since we need only to finish our phase in $O(w(v))$ accesses, we can afford to wait for quite a while. Before an up phase can be performed on v we at most have to finish $O(\sqrt{B})$ rearrange phases (one for each partial slab boundary), each requiring $O((w(v)/\sqrt{B})/\sqrt{B}) = O(w(v)/B)$ accesses, for a total of $O(w(v)/\sqrt{B})$ accesses. After finishing the up phase on v we might need to finish $O(\sqrt{B})$ other rearrange phases and one up phase on $parent(v)$ before performing the rearrange phase. These phases require at most $O(\sqrt{B}) \cdot O(w(v)/\sqrt{B}) + O((w(v) \cdot \sqrt{B})/\sqrt{B}) = O(w(v))$ accesses. Thus in total we finish the split of v in the allowed $O(w(v))$ accesses. Note that the waiting can be implemented simply by associating a single block with v storing a queue with information about the $O(\sqrt{B})$ phases waiting at v. When we want to perform a phase on v we simply insert it in the queue, and each time v is accessed $O(1)$ I/Os are performed on the phase in the front of the queue. When v has been accessed enough times to finish one phase we start the next phase in the queue. Note also that while we are waiting to perform a phase on a node v, or even while we are performing the phase, new partial slab boundaries may be inserted in v, and new slab lists may be inserted for such boundaries (due to splits of children of v). However, this does not interfere with the up or rearrange phase, since we do not split along partial boundaries and since the intervals in the two new slab lists for a partial boundary contain only the partial boundary.

After having described how to guarantee that we are working on only one up or rearrange phase in a node v at any given time, we can now describe how to perform such a phase lazily over $O(w(v)/\sqrt{B})$ accesses. We do so basically using lazy global rebuilding: during normal updates we maintain a copy of each secondary structure—called a *shadow* structure. Maintaining such shadow structures along with the original structures does not change the asymptotic space, update, or query bounds of the

interval tree. When we start an up or rearrange phase on v, we first "freeze" the shadow structures of v; that is, instead of performing updates on them we just store updates as they arrive. Then we perform the necessary movement of intervals on the *shadow* structures as described in section 3. It is easy to realize how we can perform these movements over the next $O(w(v)/\sqrt{B})$ updates accessing v such that $O(1)$ I/Os are used at each access. The only slight complication is that we need to make sure that updates performed in v during the process (that is, updates that were stored at v and still need to be performed on the shadow structures) are performed after we have moved the relevant intervals. To do so within the $O(w(v)/\sqrt{B})$ bound, we actually perform $\Theta(\log_B N)$ instead of only $O(1)$ I/Os on the shadow structure movement process every time an update is performed in v. This does not change the overall $O(\log_B N)$ update I/O bound, since an update (insertion or deletion of an interval) takes place only in *one* node of $\mathcal{T}$ (or, equivalently, since we are already using $O(\log_B N)$ I/Os to perform the update on the original secondary structures of v). After finishing the interval movement, we then perform the stored updates lazily using $O(1)$ I/Os each time v is accessed. Since we performed $\Theta(\log_B N)$ I/Os on the shadow structures each time an update was stored at v, we are guaranteed that we will finish performing the updates within $O(w(v)/\sqrt{B})$ accesses to v. Finally, after moving intervals and performing updates, we (lazily) make a copy (shadow) of the new shadow structures in the same I/O bound. We handle updates performed during this copying in the same way we handled updates during global rebuilding of the external interval tree (to remove delete amortization). We finish the phase by discarding the old secondary structures and instead use the updated shadow structures (along with their shadow) as the secondary structures.

4.2. Removing amortization from the corner structure. In this section we sketch the "corner structure" of Kannelakis et al. [30] (Lemma 2.3) and discuss how its $O(1)$ amortized update bound can be made worst-case. At the same time we remove the assumption that $M \geq B^2$.
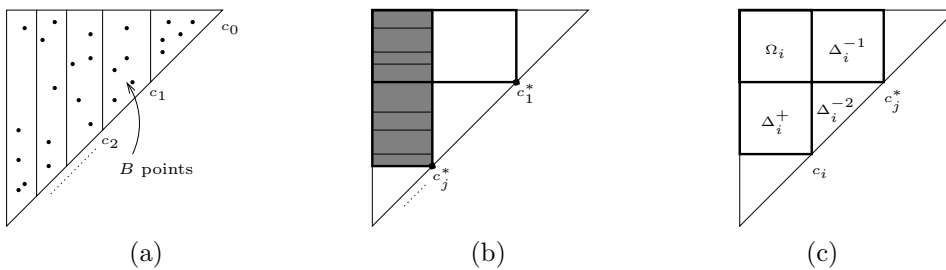

FIG. 4.1. *(a) Vertical regions. (b) The set C^* (the marked points). The dark lines represent the boundaries of queries whose corners are points in C^*. One (horizontally blocked) query is shaded. (c) The sets Ω_i, Δ_i^{-1}, Δ_i^{-2}, and Δ_i^+. c_j^* is the last point that was added to C^*, and c_i is being considered for inclusion in C^*.*

The corner structure is designed to store a set S of $K \leq B^2$ points in the plane above the $x = y$ line such that diagonal corner queries can be answered in $O(1+T/B)$ I/Os [30]. As discussed in the introduction, this problem is equivalent to the stabbing query problem. The structure is defined as follows: First S is divided into $\lceil K/B \rceil$ *vertical* regions containing B points each. Refer to Figure 4.1(a). The points in these regions are stored in $\lceil K/B \rceil$ blocks. Let C be the set of points at which the right boundaries of the $\lceil K/B \rceil$ regions intersect the $y = x$ line. An iterative procedure is

used to choose a subset C^* of these points, and one or more *horizontally* oriented blocks are used to explicitly store the answer to each query with a corner $c^* \in C^*$. Refer to Figure 4.1(b). First the highest point $c_1^* \in C$ is included in C^*. Then each point in C is considered in turn along the $x = y$ line. Let c_j^* be the point of C most recently added to C^*; initially, this is c_1^*. When considering $c_i \in C$, the sets $\Omega_i \subset S$, $\Delta_i^{-1} \subset S$, $\Delta_i^{-2} \subset S$, and $\Delta_i^+ \subset S$ are defined as shown in Figure 4.1(c). The set $S_j^* = \Omega_i \cup \Delta_i^{-1}$ is the answer to a query whose corner is c_j^*, and $S_i = \Omega_i \cup \Delta_i^+$ is the answer to a query whose corner is c_i. Let $\Delta_i^- = \Delta_i^{-1} \cup \Delta_i^{-2}$. The point c_i is then added to C^* if and only if

$$|\Delta_i^-| + |\Delta_i^+| > |S_i|.$$

In [30] it is shown that the total number of blocks used to store the sets S_i^* is $O(K/B)$. The corner structure consists of these blocks, as well as $O(1)$ blocks storing the sets C and C^*. In [30] it is also shown how a diagonal corner query can be answered in $O(1 + T/B)$ I/Os using this representation.

The corner structure can easily be constructed in $O(K/B)$ I/Os when $M \geq B^2$ (since in this case it fits in main memory). As discussed, the structure can therefore easily be made dynamic with an $O(1)$ amortized update bound using an update block and global rebuilding [30]. In the following we show how to construct the structure in $O(K/B)$ I/Os incrementally over $O(K/B)$ updates such that no more than $O(1)$ blocks are loaded into main memory at any time. This immediately removes the $M \geq B^2$ assumption. Using this result and lazy global rebuilding, the $O(1)$ amortized update bound can then be made worst-case: once $B/2$ updates have been collected in the update block, the structure is rebuilt during the next $B/2$ updates using $O(1)$ I/Os at each update.

We first discuss how to construct a corner structure on K points in $O(K/B)$ I/Os using $O(1)$ blocks of main memory. After that we discuss how the construction can be performed lazily. We assume that the K points are given in two lists sorted according to x- and y-coordinates, respectively. We can easily store two such lists along with the corner structure itself using $O(K/B)$ space. At the start of a rebuilding process, we first merge the $O(B)$ points in the update block into these two lists in $O(K/B)$ I/Os using $O(1)$ blocks of main memory. Then we construct the corner structure in three steps: first we compute the vertical blocking, that is, the set C. Then we compute C^*. Finally, we construct the horizontal blocking corresponding to each of the points in C^*.
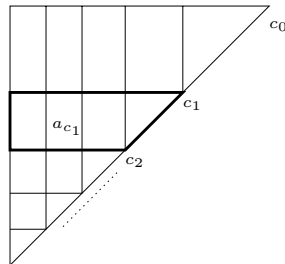

FIG. 4.2. *Definition of a_{c_i}.*

The vertical blocking is simply the list sorted according to x-coordinates. We can easily make a copy of this list and thus compute the set C in $O(K/B)$ I/Os
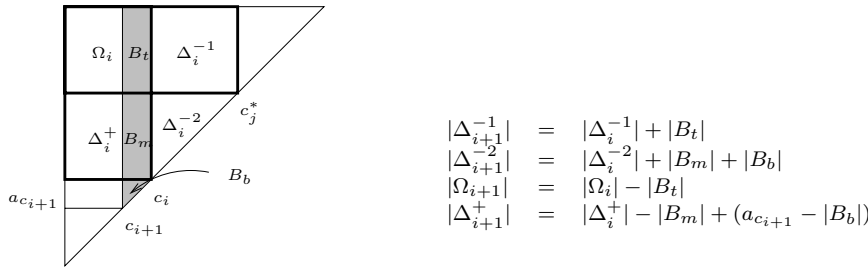
$$
\begin{array}{rcl}
|\Delta_{i+1}^{-1}| & = & |\Delta_i^{-1}| + |B_t| \\
|\Delta_{i+1}^{-2}| & = & |\Delta_i^{-2}| + |B_m| + |B_b| \\
|\Omega_{i+1}| & = & |\Omega_i| - |B_t| \\
|\Delta_{i+1}^{+}| & = & |\Delta_i^{+}| - |B_m| + (a_{c_{i+1}} - |B_b|)
\end{array}
$$

FIG. 4.3. *Definition of B_t, B_m, and b_b and computation of C^*.*

using $O(1)$ blocks of memory. Let $c_i \in C$ be the ith point in the sorted sequence of points on the $x = y$ line. To aid the computation of C^* we first compute for each $c_i \in C$ a number a_{c_i}. We define a_{c_i} to be the number of points with y-coordinates between the y-coordinates of c_i and c_{i+1}. Refer to Figure 4.2. We compute all a_{c_i}s in $O(K/B)$ I/Os using $O(1)$ blocks of main memory in a single scan of the list of points sorted by y-coordinates. We then compute C^* as previously by proceeding along the $x = y$ line and including $c_i \in C$ in C^* if $|\Delta_i^-| + |\Delta_i^+| > |S_i|$. Since the number of points in C is $O(K/B)$, our goal is to decide if c_i should be included in $O(1)$ I/Os. We can do so if we can compute $|\Omega_{i+1}|$, $|\Delta_{i+1}^+|$, $|\Delta_{i+1}^{-1}|$, and $|\Delta_{i+1}^{-2}|$ in $O(1)$ I/Os, given $|\Omega_i|$, $|\Delta_i^+|$, $|\Delta_i^{-1}|$, $|\Delta_i^{-2}|$, c_j^*, c_i, c_{i+1}, and a_{i+1}. (These values/points can all be loaded into main memory in $O(1)$ I/Os.) Figure 4.3 illustrates how we can compute $|\Omega_{i+1}|$, $|\Delta_{i+1}^+|$, $|\Delta_{i+1}^{-1}|$, and $|\Delta_{i+1}^{-2}|$ once B_t, B_m, and B_b have been computed. We compute these three sets, containing a total of B points, in $O(1)$ I/Os, simply by loading the relevant vertical block. Thus overall we can compute C^* in $O(K/B)$ I/Os.

To compute and horizontally block the points in the answer S_j^* to a query at each of the points $c_j^* \in S^*$, we again proceed along the $x = y$ line considering each point $c_i^* \in C^*$ in turn. Assume we have already blocked S_j^* and that we know the position p of the last (lowest y-coordinate) point in S_j^* in the list of points sorted by y-coordinates. Initially, S_j^* is empty, and p is the first point in the list of points sorted by y-coordinates. To block S_{j+1}^*, we scan through the horizontal blocking of S_j^*, collecting the points with x-coordinate smaller than the x-coordinate of c_{j+1}^*. This way we obtain the points in S_{j+1}^* above the y-coordinate of s_j^* horizontally blocked (sorted by y-coordinates). Then we collect the remaining points in S_{j+1}^* horizontally blocked by scanning through the list of points sorted by y-coordinates, starting at p. Altogether we use $O(|S_j^*|/B + |S_{j+1}^*|/B)$ I/Os to compute the blocking of S_{j+1}^*. Thus we use $O(2 \sum_{j \in 1..|S^*|} |S_j^*|/B)$ I/Os to compute the blocking corresponding to all the points in C^*. This is $O(K/B)$ since the structure uses linear space.

We have shown how to construct the corner structure in $O(K/B)$ I/Os using $O(1)$ blocks of main memory. We can easily modify the algorithm to work in an incremental way, that is, to run in $O(K/B)$ steps of $O(1)$ I/Os without using any main memory between two steps: throughout the algorithm we can represent the current state of the algorithm by a constant number of pointers and values. We can therefore perform one step by loading the current state into main memory using $O(1)$ I/Os, performing the step using $O(1)$ I/Os, and finally using $O(1)$ I/Os to write the new state back to disk. In total we have proved the following.

LEMMA 4.2. *A set of $K \leq B^2$ intervals can be stored in an external data structure using $O(K/B)$ disk blocks such that a stabbing query can be answered in $O(T/B+1)$ I/Os and such that updates can be performed in $O(1)$ I/Os. The structure can be constructed in $O(K/B)$ I/Os.*

5. External segment tree. In this section we sketch how the ideas used in the external interval tree can also be used to develop an external segment tree-like structure with a better space bound than previously known for such structures [15, 39]. Like an interval tree, a segment tree solves the stabbing query problem. Unlike the interval tree, however, it uses superlinear space. It is often used as base tree structure in multidimensional structures (refer, e.g., to [10, 3]).

Structure. In internal memory, as in the case of the interval tree, a segment tree consists of a binary base tree with intervals stored in secondary structures of internal nodes. Unlike for the interval tree, an interval can be stored in the secondary structures of up to two nodes on each level of the base tree. More precisely, an interval s is stored in all nodes v such that s contains the interval associated with at least one of v's children but not the interval X_v associated with v. As in the interval tree case, we externalize the structure by using a weight-balanced B-tree (Theorem 3.8) as the base tree. As previously, an internal node v defines $\Theta(\sqrt{B})$ slabs and $\Theta(B)$ multislabs, and $\Theta(B)$ secondary structures are associated with v. An interval s is stored only in the secondary structures of v if it spans one of v's *slabs* but not the whole interval X_v. Thus, unlike in the external interval tree, where s is stored only in the highest node for which it contains a slab *boundary*, s can be stored in $O(\log_B N)$ nodes—refer to Figure 5.1(a). As in the interval tree, s is stored in a multislab list corresponding to the largest multislab it spans, and as before intervals from multislab lists containing $o(B)$ intervals are stored in an underflow structure. Note how an external segment tree corresponds to an external interval tree, where parts of an interval not completely spanning a slab in a node v are stored recursively instead of in a slab list. Multislab lists are implemented as simple (unordered) lists, and with each interval s we store pointers to the copies of s in the nearest ancestor and descendent of v storing copies of s. These pointers are not directly maintained for intervals in underflow structures. Instead we keep a separate lists of intervals in the underflow structure of each node v, and pointers are maintained for these intervals. This allows us to rebuild an underflow structure containing K intervals in $O(K/B)$ I/Os and thus maintain the $O(1)$ update bound of the underflow structure (Lemma 4.2); maintaining pointers would have required $O(K)$ I/Os. Finally, we maintain an auxiliary B-tree containing all intervals in the structure, ordered according to right endpoint, such that given an interval s we can obtain a pointer to the copy of s stored in the topmost node storing s. This B-tree uses linear space, and since the secondary structures of a node also use linear space the external segment tree uses $O((N/B)\log_B N)$ disk blocks to store N intervals.

Query. We answer a stabbing *query q* on an external segment tree in $O(\log_B N + T/B)$ I/Os, simply by searching down the base tree for q and in each node querying the underflow structure and reporting all intervals in lists corresponding to multislabs containing q.

Updates. To *insert* an interval s into an external segment tree we first insert the endpoints of s in the base tree and rebalance the tree by splitting nodes. Below we show how a node v can be split in $O(w(v))$ I/O such that an endpoint is inserted in $O(\log_B N)$ I/Os amortized. Then we perform the actual insertion of s by traversing two paths in the base tree, inserting s in the relevant multislab lists. As the multislab lists are unsorted, we can insert s in a list in $O(1)$ I/O while at the same time
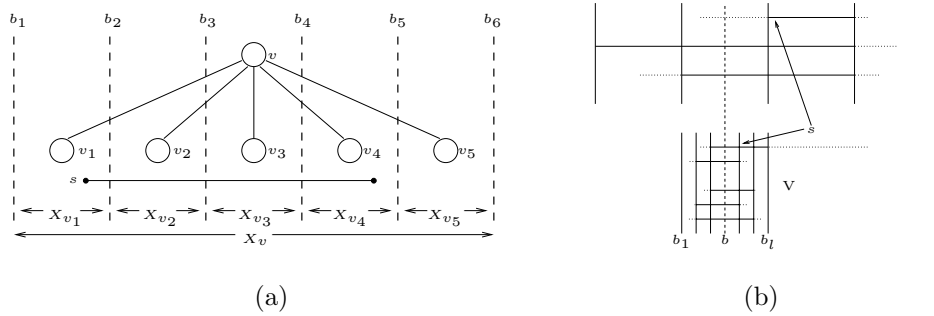
FIG. 5.1. (a) *Node v in the base tree. The interval s is stored in v as well as (recursively) in v_1 and v_4.* (b) *Splitting a segment tree node.*

storing the relevant pointers to other copies of s. We handle the cases involving the underflow structure in $O(1)$ I/Os precisely as in the external interval tree case. In total we can perform an insertion in $O(\log_B N)$ I/Os. We *delete* an interval s from an external segment tree in $O(\log_B N)$ I/Os simply by locating s in the auxiliary B-tree—obtaining a pointer to the topmost of the $O(\log_B N)$ occurrences of s—deleting s from the B-tree, and then using the pointers between copies of s to find and remove all occurrences of s in $O(\log_B N)$ I/Os; note that even though in a node where s is stored in the underflow structure we obtain a pointer to the separate list of intervals, we can still delete s from the underflow structure in $O(1)$ I/Os. As in the interval tree case, we remove the endpoints of s from the base tree in $O(\log_B N)$ I/Os using global rebuilding.

All that remains is to describe how to efficiently split a node v along a slab boundary b. When v splits into v' and v'', all intervals in multislabs containing b need to be moved. These intervals fall into two categories: intervals that contain the leftmost or rightmost slab boundaries b_1 and b_l of v and those that do not. Refer to Figure 5.1(b). Intervals not containing b_1 or b_l (but containing b) need to be stored in multislab lists of both v' and v''. Thus to move these intervals we simply need to make a copy of the relevant lists for both v' and v''. We also need to update the relevant pointers between intervals. We can easily do so in $O(w(v))$ I/Os. Intervals that contain b_1 or b_l (and b) need to be inserted in v' or v'', as well as moved within *parent*(v). Consider, for example, intervals containing b and b_l (e.g., interval s in Figure 5.1(b)). Such intervals need to be inserted in multislab lists for v', as well as either inserted into *parent*(v) or moved within *parent*(v): if one of these intervals s is already stored in *parent*(v) we use the pointer stored with s in v to locate and delete s in *parent*(v), and then we insert s in the relevant new multislab list. Since each interval can be moved in $O(1)$ I/Os, and since all moved intervals have an endpoint in X_v, we use $O(w(v))$ I/Os in total to split a node v. Finally, as in the interval tree case, we can perform the split over $O(w(v))$ updates accessing v and thus obtain worst-case bounds.

THEOREM 5.1. *There exists an external segment tree using $O((N/B)\log_B N)$ disk blocks to store N intervals such that stabbing queries can be answered in $O(\log_B N + T/B)$ I/Os and such that updates can be performed in $O(\log_B N)$ I/Os.*

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 Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms, Baltimore, MD, 1999, pp. 11–20.

[2] P. K. AGARWAL, L. ARGE, AND J. ERICKSON, *Indexing moving points*, J. of Comput. System Sci., 66 (2003), pp. 207–243.

[3] P. K. AGARWAL AND J. ERICKSON, *Geometric range searching and its relatives*, in Advances in Discrete and Computational Geometry, B. Chazelle, J. E. Goodman, and R. Pollack, eds., Contemp. Math. 223, AMS, Providence, RI, 1999, pp. 1–56.

[4] A. AGGARWAL AND J. S. VITTER, *The input/output complexity of sorting and related problems*, Comm. ACM, 31 (1988), pp. 1116–1127.

[5] L. ARGE, *External memory data structures (invited paper)*, in Proceedings of the 9th Annual European Symposium on Algorithms, Aarhus, Denmark, 2001, Lecture Notes in Comput. Sci. 2161, Springer-Verlag, Berlin. 2001, pp. 1–29.

[6] L. ARGE, *External memory data structures*, in Handbook of Massive Data Sets, J. Abello, P. M. Pardalos, and M. G. C. Resende, eds., Kluwer Academic Publishers, Dordrecht, The Netherlands, 2002, pp. 313–358.

[7] L. ARGE, *The buffer tree: A technique for designing batched external data structures*, Algorithmica, 37 (2003), pp. 1–24.

[8] L. ARGE, V. SAMOLADAS, AND J. S. VITTER, *On two-dimensional indexability and optimal range search indexing*, in Proceedings of the 18th ACM Symposium on Principles of Database Systems, Philadelphia, PA, 1999, pp. 346–357.

[9] L. ARGE AND J. VAHRENHOLD, *I/O-efficient dynamic planar point location*, Internat. J. Comput. Geom. Appl., to appear.

[10] L. ARGE, D. E. VENGROFF, AND J. S. VITTER, *External-memory algorithms for processing line segments in geographic information systems*, Algorithmica, to appear.

[11] L. ARGE AND J. S. VITTER, *Optimal dynamic interval management in external memory*, in Proceedings of the 37th Annual IEEE Symposium on Foundations of Computer Science, Burlington, VT, 1996, pp. 560–569.

[12] R. BAYER AND E. MCCREIGHT, *Organization and maintenance of large ordered indexes*, Acta Informat., 1 (1972), pp. 173–189.

[13] M. A. BENDER, E. D. DEMAINE, AND M. FARACH-COLTON, *Cache-oblivious B-trees*, in Proceedings of the 41st Annual IEEE Symposium on Foundations of Computer Science, Redondo Beach, CA, 2000, pp. 339–409.

[14] G. BLANKENAGEL, *Intervall-Indexstructuren in Datenbanksystemen*, Informatik-Farberichte 312, Springer-Verlag, Berlin, 1992.

[15] G. BLANKENAGEL AND R. GÜTING, *External segment trees*, Algorithmica, 12 (1994), pp. 498–532.

[16] G. BLANKENAGEL AND R. H. GÜTING, *XP-Trees—External Priority Search Trees*, Technical report, FernUniversität Hagen, Informatik-Bericht 92, Hagen, Germany, 1990.

[17] N. BLUM AND K. MEHLHORN, *On the average number of rebalancing operations in weight-balanced trees*, Theoret. Comput. Sci., 11 (1980), pp. 303–320.

[18] Y.-J. CHIANG AND C. T. SILVA, *I/O optimal isosurface extraction*, in Proceedings of IEEE Visualization, Phoenix, AZ, 1997, pp. 293–300.

[19] Y.-J. CHIANG AND C. T. SILVA, *External memory techniques for isosurface extraction in scientific visualization*, in External Memory Algorithms and Visualization, DIMACS Ser. Discrete Math. Theoret. Comput. Sci. 50, J. Abello and J. S. Vitter, eds., AMS, Providence, RI, 1999, pp. 247–277.

[20] Y.-J. CHIANG, C. T. SILVA, AND W. J. SCHROEDER, *Interactive out-of-core isosurface extraction*, in Proceedings of IEEE Visualization, Research Triangle Park, NC, 1998, pp. 167–174.

[21] D. COMER, *The ubiquitous B-tree*, ACM Computing Surveys, 11 (1979), pp. 121–137.

[22] H. EDELSBRUNNER, *A new approach to rectangle intersections, part* I, Internat. J. Comput. Math., 13 (1983), pp. 209–219.

[23] H. EDELSBRUNNER, *A new approach to rectangle intersections, part* II, Internat. J. Comput. Math., 13 (1983), pp. 221–229.

[24] V. GAEDE AND O. GÜNTHER, *Multidimensional access methods*, ACM Computing Surveys, 30 (1998), pp. 170–231.

[25] R. GROSSI AND G. F. ITALIANO, *Efficient cross-tree for external memory*, in External Memory Algorithms and Visualization, DIMACS Ser. Discrete Math. Theoret. Comput. Sci. 50, J. Abello and J. S. Vitter, eds., AMS, Providence, RI, 1999, pp. 87–106. Revised version available at ftp://ftp.di.unipi.it/pub/techreports/TR-00-16.ps.Z.

[26] R. GROSSI AND G. F. ITALIANO, *Efficient splitting and merging algorithms for order decomposable problems*, Inform. and Comput., 154 (1999), pp. 1–33.

[27] S. HUDDLESTON AND K. MEHLHORN, *A new data structure for representing sorted lists*, Acta Inform., 17 (1982), pp. 157–184.

[28] C. ICKING, R. KLEIN, AND T. OTTMANN, *Priority search trees in secondary memory*, in Proceedings of Graph-Theoretic Concepts in Computer Science, Staffelstein, Germany, 1987, Lecture Notes in Comput. Sci. 314, Springer-Verlag, Berlin, 1988, pp. 84–93.

[29] P. C. KANELLAKIS, G. KUPER, AND P. REVESZ, *Constraint query languages*, in Proceedings of the 9th ACM Symposium on Principles of Database Systems, Nashville, TN, 1990, pp. 299–313.

[30] P. C. KANELLAKIS, S. RAMASWAMY, D. E. VENGROFF, AND J. S. VITTER, *Indexing for data models with constraints and classes*, J. Comput. System Sci., 52 (1996), pp. 589–612.

[31] D. E. KNUTH, *Sorting and Searching, Volume* 3 *of The Art of Computer Programming*, 2nd ed., Addison-Wesley, Reading, MA, 1998.

[32] E. M. MCCREIGHT, *Priority search trees*, SIAM J. Comput., 14 (1985), pp. 257–276.

[33] K. MEHLHORN, *Data Structures and Algorithms* 3*: Multi-dimensional Searching and Computational Geometry*, EATCS Monographs on Theoretical Computer Science, Springer-Verlag, Berlin, 1984.

[34] J. NIEVERGELT AND E. M. REINGOLD, *Binary search trees of bounded balance*, SIAM J. Comput., 2 (1973), pp. 33–43.

[35] J. NIEVERGELT AND P. WIDMAYER, *Spatial data structures: Concepts and design choices*, in Algorithmic Foundations of GIS, Lecture Notes in Comput. Sci. 1340, M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer, eds., Springer-Verlag, Berlin, 1997, pp. 153–197.

[36] M. H. OVERMARS, *The Design of Dynamic Data Structures*, Lecture Notes in Comput. Sci. 156, Springer-Verlag, Berlin, 1983.

[37] S. RAMASWAMY, *Efficient indexing for constraint and temporal databases*, in Proceedings of the 6th International Conference on Database Theory, Delphi, Greece, 1997, Lecture Notes in Comput. Sci. 1186, Springer-Verlag, Berlin, 1997, pp. 419–431.

[38] S. RAMASWAMY AND P. KANELLAKIS, *OODB Indexing by Class Division*, A.P.I.C. Series, Academic Press, New York, 1995.

[39] S. RAMASWAMY AND S. SUBRAMANIAN, *Path caching: A technique for optimal external searching*, in Proceedings of the 13th ACM Symposium on Principles of Database Systems, Minneapolis, MN, 1994, pp. 25–35.

[40] S. SUBRAMANIAN AND S. RAMASWAMY, *The P-range tree: A new data structure for range searching in secondary memory*, in Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms, San Francisco, CA, 1995, pp. 378–387.

[41] J. S. VITTER, *External memory algorithms and data structures: Dealing with MASSIVE data*, ACM Computing Surveys, 33 (2001), pp. 209–271.

[42] D. WILLARD AND G. LUEKER, *Adding range restriction capability to dynamic data structures*, J. Assoc. Comput. Mach., 32 (1985), pp. 597–617.

[43] D. E. WILLARD, *Reduced memory space for multi-dimensional search trees*, in Proceedings of the 2nd Symposium on Theoretical Aspects of Computer Science, Saarbrüken, Germany, Lecture Notes in Comput. Sci. 182, Springer-Verlag, Berlin, 1985, pp. 363–374.