External Memory Data Structures
(Invited Paper)

Lars Arge*

Department of Computer Science, Duke University, Durham, NC 27708, USA

Abstract. 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 data structures on external storage devices, and in such cases the Input/Output (or I/O) communication between internal and external memory can become a major performance bottleneck. In this paper we survey recent advances in the development of worst-case I/O-efficient external memory data structures.

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, and 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 which can be interpreted geometrically. Such applications often perform queries which 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 this paper we survey recent advances in the development of worst-case efficient external memory data structures. We will concentrate on data structures for geometric problems—especially the important one- and two-dimensional range searching problems—but mention other structures when appropriate. A more comprehensive discussion can be found in a recent survey by the author [16].

* Supported in part by the National Science Foundation through ESS grant EIA–9870734, RI grant EIA–9972879, and CAREER grant EIA–9984099.

Model of computation. Accurately modeling memory and disk systems is a complex task [131]. 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 [13, 153, 104]:

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

where $B < M < N$. 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 in this model are the number of I/Os used to solve a problem, as well as the amount of space (disk blocks) used and the internal memory computation time.

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. See e.g. the recent survey by Vitter [151]. 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. For brevity we will also ignore internal computation time.

Outline of paper. The rest of this paper is organized as follows. In Section 2 we discuss the B-tree, the most fundamental (one-dimensional) external data structure, as well as recent variants and extensions of the structure, and in Section 3 we discuss the so-called buffer trees. In Section 4 we illustrate some of the important techniques and ideas used in the development of provably I/O-efficient data structures for higher-dimensional problems through a discussion of the external priority search tree for 3-sided planar range searching. We also discuss a general method for obtaining a dynamic data structure from a static

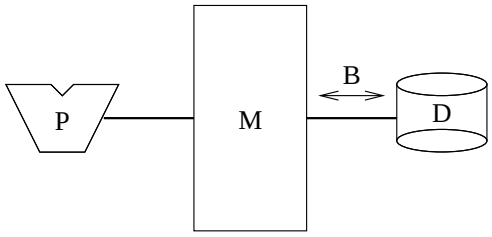

Fig. 1. Disk model; An I/O moves B contiguous elements between disk and main memory (of size M).

one. In Section 5 we discuss data structures for general (4-sided) planar range searching, and in Section 6 we survey results on external data structures for interval management, point location, range counting, higher-dimensional range searching, halfspace range searching, range searching among moving points, and proximity queries. Several of the worst-case efficient structures we consider are simple enough to be of practical interest. Still, there are many good reasons for developing simpler (heuristic) and general purpose structures without worst-case performance guarantees, and a large number of such structures have been developed in the database community. Even though the focus of this paper is on provably worst-case efficient data structures, in Section 7 we give a short survey of some of the major classes of such heuristic-based structures. The reader is referred to recent surveys for a more complete discussion [12, 85, 121]. Finally, in Section 8 we discuss some of the efforts which have been made to implement the developed worst-case efficient structures.

2 B-trees

The B-tree is the most fundamental external memory data structure, corresponding to an internal memory balanced search tree [35, 63, 104, 95]. 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.

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 $O(\log N)$ internal memory search-term, and an $O(T/B)$ reporting-term accounting for the $O(T/B)$ I/Os needed to report T elements. Recently, the above bounds have been obtained for a number of problems (e.g [30, 26, 149, 5, 47, 87]) but higher lower bounds have also been established for some problems [141, 26, 93, 101, 106, 135, 102]. We discuss these results in later sections.

B-trees come in several variants, like B^+ and B^* trees (see e.g. [35, 63, 95, 30, 104, 3] and their references). A basic B-tree is a $\Theta(B)$-ary tree (with the root possibly having smaller degree) built on top of $\Theta(N/B)$ leaves. The degree of internal nodes, as well as the number of elements in a leaf, is typically kept in the range $[B/2 \ldots B]$ such that a node or leaf can be stored in one disk block. All leaves are on the same level and the tree has height $O(\log_B N)$—refer to Figure 2. In the most popular B-tree variants, the N data elements are stored in the leaves (in sorted order) and each internal node holds $\Theta(B)$ "routing" (or "splitting") elements used to guide searches.

To answer a range query $[q_1, q_2]$ on a B-tree we first search down the tree for q_1 and q_2 using $O(\log_B N)$ I/Os, and then we report the elements in the

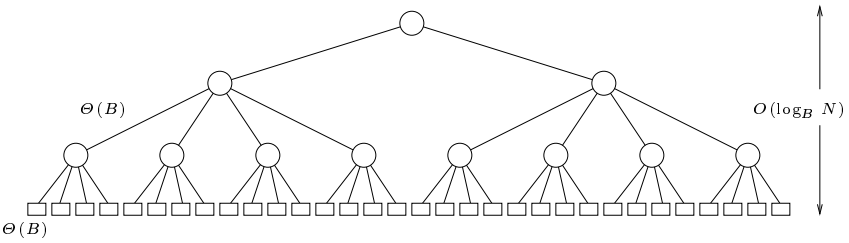

Fig. 2. B-tree; All internal nodes (except possibly the root) have fan-out $\Theta(B)$ and there are $\Theta(N/B)$ leaves. The tree has height $O(\log_B N)$.

$O(T/B)$ leaves between the leaves containing q_1 and q_2. We perform an insertion in $O(\log_B N)$ I/Os by first searching down the tree for the relevant leaf l. If there is room for the new element in l we simply store it there. If not, we *split* l into two leaves l' and l'' of approximately the same size and insert the new element in the relevant leaf. The split of l results in the insertion of a new routing element in the parent of l, and thus the need for a split may propagate up the tree. Propagation of splits can often be avoided by *sharing* some of the (routing) elements of the full node with a non-full sibling. A new (degree 2) root is produced when the root splits and the height of the tree grows by one. Similarly, we can perform a deletion in $O(\log_B N)$ I/Os by first searching for the relevant leaf l and then removing the deleted element. If this results in l containing too few elements we either *fuse* it with one of its siblings (corresponding to deleting l and inserting its elements in the sibling), or we perform a *share* operation by moving elements from a sibling to l. As splits, fuse operations may propagate up the tree and eventually result in the height of the tree decreasing by one.

B-tree variants and extensions. Recently, several important variants and extensions of B-trees have been considered.

Arge and Vitter [30] developed the *weight-balanced B-trees*, which can be viewed as an external version of BB[α] trees [120]. Weight-balanced B-trees are very similar to B-trees but with a weight constraint imposed on each node in the tree instead of a degree constraint. The weight of a node v is defined as the number of elements in the leaves of the subtree rooted in v. Like B-trees, weight-balanced B-trees are rebalanced using split and fuse operations, and a key property of weight-balanced B-trees is that after performing a rebalance operation on a weight $\Theta(w)$ node v, $\Omega(w)$ updates have to be performed below v before another rebalance operation needs to be performed on v. This means that an update can still be performed in $O(\log_B N)$ I/Os (amortized) even if v has a large associated secondary structure that needs to be updated when a rebalance operation is performed on v, provided that the secondary structure can be updated in $O(w)$ I/Os. Weight-balanced B-trees have been used in numerous efficient data structure (see e.g. [30, 26, 89, 90, 38, 3, 28]).

In some applications we need to be able to traverse a path in a B-tree from a leaf to the root. To do so we need a *parent-pointer* from each node to its parent.

Such pointers can easily be maintained efficiently in normal B-trees or weight-balanced B-trees, but cannot be maintained efficiently if we also want to support *divide* and *merge* operations. A divide operation at element x constructs two trees containing all elements less than and greater than x, respectively. A merge operation performs the inverse operation. Without parent pointers, B-trees and weight-balanced B-trees supports the two operations in $O(\log_B N)$ I/Os. Agarwal et al. [3] developed the *level-balance B-trees* in which divide and merge operations can be supported I/O-efficiently while maintaining parent pointers. In level-balanced B-trees a global balance condition is used instead of the local degree or weight conditions used in B-trees or weight-balanced B-trees; a constraint is imposed on the number of nodes on each level of the tree. When the constraint is violated the whole subtree at that level and above is rebuilt. Level-balanced B-trees e.g. have applications in dynamic maintenance of planar st-graphs [3].

Partial persistent B-trees, that is, B-trees where each update results in a new *version* of the structure, and where both the current and older versions can be queried (in the database community often called *multiversion B-trees*), are useful not only in database applications where previous versions of the database needs to be stored and queried, but (as we will discuss in Section 4) also in the solution of many geometric problems. Using standard persistent techniques [137, 70], a persistent B-tree can be designed such that updates can be performed in $O(\log_B N)$ I/Os and such that any version of the tree can be queried in $O(\log_B N + T/B)$ I/Os. Here N is the number of updates and the tree uses $O(N/B)$ disk blocks [36, 147].

In string applications a data element (string of characters) can often be arbitrarily long or different elements can be of different length. Such elements cannot be manipulated efficiently in standard B-trees, which assumes that elements (and thus routing elements) are of unit size. Ferragina and Grossi developed the elegant *string B-tree* where a query string q is routed through a node using a so-called *blind trie* data structure [78]. A blind trie is a variant of the compacted trie [104, 117], which fits in one disk block. In this way a query can be answered in $O(\log_B N + |q|/B)$ I/O. See [64, 80, 77, 20] for other results on string B-trees and external string processing.

3 Buffer trees

In internal memory, an N element search tree can be constructed in optimal $O(N \log N)$ time simply by inserting the elements one by one. This construction algorithm can also be used as on optimal sorting algorithm. In external memory, we would use $O(N \log_B N)$ I/Os to build a B-tree using the same method. Interestingly, this is not optimal since Aggarwal and Vitter showed that sorting N elements in external memory takes $\Theta(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os [13]. We can of course build a B-tree in the same bound by first sorting the elements and then building the tree level-by-level bottom-up.

In order to obtain an optimal sorting algorithm based on a search tree structure, we would need a structure that supports updates in $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ I/Os. The inefficiency of the B-tree sorting algorithm is a consequence of the B-tree being designed to be used in an "on-line" setting where queries should be answered immediately—updates and queries are handled on an individual basis. This way we are not able to take full advantage of the large internal memory. It turns out that in an "off-line" environment where we are only interested in the overall I/O use of a series of operations and where we are willing to relax the demands on the query operations, we can develop data structures on which a series of N operations can be performed in $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os in total. To do so we use the *buffer tree* technique developed by Arge [14].

Basically the buffer tree is just a fan-out $\Theta(M/B)$ B-tree where each internal node has a buffer of size $\Theta(M)$. The tree has height $O(\log_{M/B}\frac{N}{B})$; refer to Figure 3. Operations are performed in a "lazy" manner: In order to perform an insertion we do not (like in a normal B-tree) search all the way down the tree for the relevant leaf. Instead, we wait until we have collected a block of insertions and then we insert this block in the buffer of the root (which is stored on disk). When a buffer "runs full" its elements are "pushed" one level down to buffers on the next level. We can do so 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 levels of the tree. Thus, not counting rebalancing, inserting N elements requires $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os in total, or $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ amortized. Arge showed that rebalancing can be handled in the same bound [14].

The basic buffer tree supporting insertions only can be used to construct a B-trees in $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os (without explicitly sorting). This of course means that buffer trees can be used to design an optimal sorting algorithm. Note that unlike other sorting algorithm, the N elements to be sorted do not all need to be given at the start of this algorithm. Deletions and (one-dimensional) range

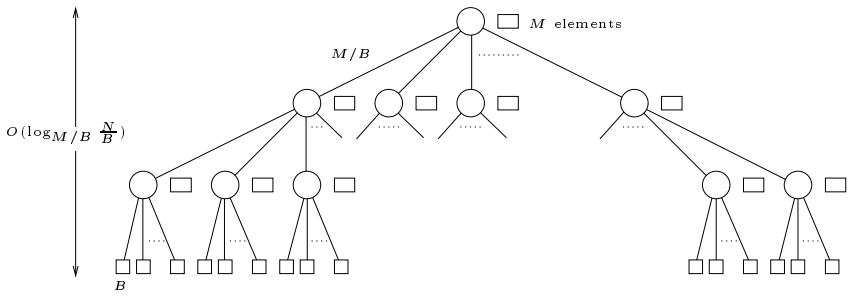

Fig. 3. Buffer tree; Fan-out M/B tree where each node has a buffer of size M. Operations are performed in a lazy way using the buffers.

queries can also be supported I/O-efficiently using buffers [14]. 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. This means that the data structure can only be used in algorithms where future updates and queries do not depend on the result of the queries. Luckily this is the case in many plane-sweep algorithms [73, 14]. In general, problems where the entire sequence of updates and queries is known in advance, and the only requirement on the queries is that they must all eventually be answered, are known as *batched dynamic problems* [73].

As mentioned, persistent B-trees are often used in geometric data structures. Often, a data structure is constructed by performing N insertion and deletions on an initially empty persistent B-tree, and then the resulting (static) structure is used to answer queries. Using the standard update algorithms the construction would take $O(N \log_B N)$ I/Os. A straightforward application of the buffer tree technique improves this to the optimal $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os [145, 21] (another optimal, but not linear space, algorithm can be designed using the *distribution-sweeping* technique [86]). Several other data structures can be constructed efficiently using buffers, and the buffer tree technique has been used to develop several other data structures which in turn have been used to develop algorithms in many different areas [25, 29, 20, 21, 107, 15, 76, 46, 144, 145, 96, 44, 136].

Priority queues. External buffered *priority queues* have been extensively researched because of their applications in graph algorithms. Arge showed how to perform deletemin operations on a basic buffer tree in amortized $O(\frac{1}{B} \log_{M/B} \frac{N}{B})$ I/Os [14]. Note that in this case the deletemin occurs right away, that is, it is not batched. This is accomplished by periodically computing the $O(M)$ smallest elements in the structure and storing them in internal memory. Kumar and Schwabe [107] and Fadel et al. [76] developed similar buffered heaps. Using a partial rebuilding idea, Brodal and Katajainen [45] developed a worst-case efficient external priority queue. Using the buffer tree technique on a tournament tree, Kumar and Schwabe [107] developed a priority queue supporting update operations in $O(\frac{1}{B} \log \frac{N}{B})$ I/Os. They also showed how to use their structure in several efficient external graph algorithms (see e.g [2, 7, 18, 22, 27, 46, 59, 81, 97, 107, 110, 111, 116, 118, 122, 142, 156] for other results on external graph algorithms and data structures). Note that if the priority of an element is known, an update operation can be performed in $O(\frac{1}{B} \log_{M/B} \frac{N}{B})$ I/Os on a buffer tree using a delete and an insert operation.

4 3-sided planar range searching

In internal memory many elegant data structures have been developed for higher-dimensional problems like range searching—see e.g. the recent survey by Agarwal and Erickson [12]. Unfortunately, most of these structures are not efficient when mapped directly to external memory—mainly because they are normally based on binary trees. The main challenge when developing efficient external structures

is to use B-trees as base structures, that is, to use multiway trees instead of binary trees. Recently, some progress has been made in the development of provably I/O-efficient data structures based on multi-way trees. In this section we consider a special case of two-dimensional range searching, namely the *3-sided planar range searching* problem: Given a set of points in the plane, the solution to a 3-sided query (q_1, q_2, q_3) consists of all points (x, y) with $q_1 \leq x \leq q_2$ and $y \geq q_3$. The solution to this problem is not only an important component of the solution to the general planar range searching problem we discuss in Section 5, but it also illustrate many of the techniques and ideas utilized in the development of other external data structures.

The static version of the 3-sided problem where the points are fixed can easily be solved I/O-efficiently using a sweeping idea and a persistent B-tree; consider sweeping the plane with a horizontal line from $y = \infty$ to $y = -\infty$ and inserting the x-coordinate of points in a persistent B-tree as they are met. To answer a query (q_1, q_2, q_3) we simply perform a one-dimensional range query $[q_1, q_2]$ on the version of the persistent B-tree we had when the sweep-line was at $y = q_3$. Following the discussion in Section 2, the structure obtained this way uses linear space and queries 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 using the buffer tree technique.

From the static solution we can obtain a linear space dynamic structure which answers a query in $O(\log_B^2 N + T/B)$ I/Os and can be updated in $O(\log_B^2 N)$ I/Os using an external version of the *logarithmic method* for transforming a static structure into a dynamic structure [39, 125]. This technique was developed by Arge and Vahrenhold as part of the design of a dynamic external planar point location structure (See Section 6). Due to its general interest, we describe the technique in Section 4.1 below. An optimal $O(\log_B N)$ query structure can be obtained in a completely different way, and we discuss this result in Section 4.2.

4.1 The logarithmic method

In internal memory, the main idea in the logarithmic method is to partition the set of N elements into $\log N$ subsets of exponentially increasing size 2^i, $i = 0, 1, 2, \ldots$, and build a static structure $\mathcal{D}_i$ for each of these subsets. Queries are then performed by querying each $\mathcal{D}_i$ and combining the answers, while insertions are performed by finding the first empty $\mathcal{D}_i$, discarding all structures $\mathcal{D}_j$, $j < i$, and building $\mathcal{D}_i$ from the new element and the $\sum_{l=0}^{i-1} 2^l = 2^i - 1$ elements in the discarded structures.

To make the logarithmic method I/O-efficient we need to decrease the number of subsets to $\log_B N$, which in turn means increasing the size of $\mathcal{D}_i$ to B^i. When doing so $\mathcal{D}_j$, $j < i$, does not contain enough objects to build $\mathcal{D}_i$ (since $1 + \sum_{l=0}^{i-1} B^l < B^i$). However, it turns out that if we can build a static structure I/O-efficiently enough, this problem can be resolved and we can make a modified version of the method work in external memory.

Consider a static structure $\mathcal{D}$ that can be constructed in $O(\frac{N}{B} \log_B N)$ I/Os and that answers queries in $O(\log_B N)$ I/Os (note that $O(\frac{N}{B} \log_{M/B} \frac{N}{B}) =$

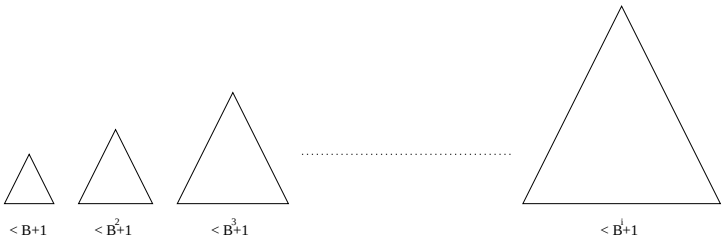

Fig. 4. Logarithmic method; $\log_B N$ structures—$\mathcal{D}_i$ contains less than B^i+1 elements. $\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_j$ do not contain enough elements to build $\mathcal{D}_{j+1}$ of size B^{j+1}.

$O(\frac{N}{B}\log_B N)$ if $M > B^2$). We partition the N elements into $\log_B N$ sets such that the ith set has size *less than* $B^i + 1$ and construct an external memory static data structure $\mathcal{D}_i$ for each set—refer to Figure 4. To answer a query, we simply query each $\mathcal{D}_i$ and combine the results using $O(\sum_{j=1}^{\log_B N} \log_B |\mathcal{D}_j|) = O(\log_B^2 N)$ I/Os. We perform an insertion by finding the first structure $\mathcal{D}_i$ such that $\sum_{j=1}^{i} |\mathcal{D}_j| \leq B^i$, discarding all structures $\mathcal{D}_j$, $j \leq i$, and building a new D_i from the elements in these structures using $O((B^i/B)\log_B B^i) = O(B^{i-1}\log_B N)$ I/Os. Now because of the way $\mathcal{D}_i$ was chosen, we know that $\sum_{j=1}^{i-1} |\mathcal{D}_j| > B^{i-1}$. This means that at least B^{i-1} objects are moved from lower indexed structures to $\mathcal{D}_i$. If we divide the $\mathcal{D}_i$ construction cost between these objects, each object is charged $O(\log_B N)$ I/Os. Since an object never moves to a lower indexed structure we can at most charge it $O(\log_B N)$ times during N insertions. Thus the amortized cost of an insertion is $O(\log_B^2 N)$ I/Os. Note that the key to making the method work is that the factor of B we lose when charging the construction of a structure of size B^i to only B^{i-1} objects is offset by the $1/B$ factor in the construction bound. Deletions can also be handled I/O-efficiently using a global rebuilding idea.

4.2 Optimal dynamic structure

Following several earlier attempts [101, 127, 141, 43, 98], Arge et al. [26] developed an optimal dynamic structure for the 3-sided planar range searching problem. The structure is an external version of the internal memory *priority search tree* structure [113]. The external priority search tree consists of a *base B-tree* on the x-coordinates of the N points. A range X_v (containing all points below v) can be associated with each node v in a natural way. This range is subdivided into $\Theta(B)$ subranges associated with the children of v. For illustrative purposes we call the subranges *slabs*. In each node v we store $O(B)$ points for each of v's $\Theta(B)$ children v_i, namely the B points with the highest y-coordinates in the x-range 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 "$O(B^2)$–structure") such that a 3-sided query on them can be answered in $O(\log_B B^2 + T/B) = O(1 + T/B)$

I/Os. Since every point is stored in precisely one $O(B^2)$–structure, the structure uses $O(N/B)$ space in total.

To answer a 3-sided query (q_1, q_2, q_3) we start at the root of the external priority search tree and proceed recursively to the appropriate subtrees; when visiting a node v we query the $O(B^2)$–structure and report the relevant points, and then we advance the search to some of the children of v. The search is advanced to child v_i if v_i is either along the leftmost search path for q_1 or the rightmost search path for q_2, or if the entire set of points corresponding to v_i in the $O(B^2)$–structure were reported—refer to Figure 5. 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 $O(B^2)$–structure corresponding to v_i does not satisfy the query. This in turn means that none of the points in the subtree rooted at v_i can satisfy the query. That we use $O(\log_B N + T/B)$ I/Os to answer a query can be seen as follows. In every internal node v 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 the tree 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 is not on the search path but it is visited because $\Theta(B)$ points corresponding to it were reported when we visited 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.

To insert a point $p = (x, y)$ in the external priority search tree we search down the tree for the leaf containing x, until we reach the node v where p needs to be inserted in the $O(B^2)$–structure. The $O(B^2)$–structure is static but since it has size $O(B^2)$ we can use a global rebuilding idea to make it dynamic [125]; we simply store the update in a special "update block" and once B updates have been collected we rebuild the structure using $O(\frac{B^2}{B} \log_{M/B} \frac{B^2}{B})$ I/Os. Assuming $M > B^2$, that is, that the internal memory is capable of holding B blocks, this is $O(B)$ and we obtain an $O(1)$ amortized update bound. Arge et al. [26] showed how to make this worst-case, even without the assumption on the main

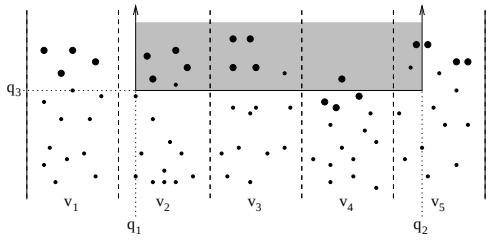

Fig. 5. Internal node v with children $v_1, v_2, \ldots, v_5$. The points in bold are stored in the $O(B^2)$–structure. To answer a 3-sided query we report the relevant of the $O(B^2)$ points and answer the query recursively in v_2, v_3, and v_5. The query is not extended to v_4 because not all of the points from v_4 in the $O(B^2)$–structure satisfy the query.

memory size. Insertion of p in v (may) result in the $O(B^2)$–structure containing one too many points from the slab corresponding to the child v_j containing x. Therefore, apart from inserting p in the $O(B^2)$–structure, we also remove the point p' with the lowest y-coordinate among the points corresponding to v_j. We insert p' recursively in the tree rooted in v_j. Since we use $O(1)$ I/Os in each of the nodes on the search path, the insertion takes $O(\log_B N)$ I/Os. We also need to insert x in the base B-tree. This may result in split and/or share operations and each such operation may require rebuilding an $O(B^2)$–structure (as well as movement of some points between $O(B^2)$–structures). Using weight-balanced B-tress, Arge et al. [26] showed how the rebalancing after an insertion can be performed in $O(\log_B N)$ I/Os worst case. Deletions can be handled in $O(\log_B N)$ I/Os in a similar way [26].

The above solution to the 3-sided planar range searching problem illustrates some of the problems encountered when developing I/O-efficient dynamic data structures, as well as the techniques commonly used to overcome these problems. As already discussed, the main problem is that in order to be efficient, external tree data structures need to have large fan-out. In the above example this resulted in the need for what we called the $O(B^2)$–structure. This structure solved a static version of the problem on $O(B^2)$ points. The structure was necessary since to "pay" for a visit to a child node v_i, we needed to find $\Theta(B)$ points in the slab corresponding to v_i satisfying the query. The idea of charging some of the query cost to the output size is often called *filtering* [51], and the idea of using a static structure on $O(B^2)$ elements in each node has been called the *bootstrapping* paradigm [151, 152]. Finally, the ideas of weight-balancing and global rebuilding were used to obtain worst-case efficient update bounds. All these ideas have been used in the development of other efficient external data structures.

5 General planar range searching

After discussing 3-sided planar range searching we are now ready to consider general planar range searching; given a set of points in the plane we want to be able to find all points contained in a query rectangle. While linear space and $O(\log_B N + T/B)$ query structures exist for special cases of this problem—like the 3-sided problem described in Section 4—Subramanian and Ramaswamy showed that one cannot obtain an $O(\log_B N + T/B)$ query bound using less than $\Theta(\frac{N \log(N/B)}{B \log\log_B N})$ disk blocks [141].[1] This lower bound holds in a natural external memory version of the *pointer machine model* [53]. A similar bound in a slightly different model where the search component of the query is ignored was proved by Arge et al. [26]. This *indexability model* was defined by Hellerstein et al. [93] and considered by several authors [101, 106, 135].

Based on a sub-optimal but linear space structure for answering 3-sided queries, Subramanian and Ramaswamy developed the *P-range tree* that uses optimal $O(\frac{N \log(N/B)}{B \log\log_B N})$ space but uses more than the optimal $O(\log_B N + T/B)$

[1] In fact, this bound even holds for a query bound of $O(\log_B^c N + T/B)$ for any constant c.

I/Os to answer a query [141]. Using their optimal structure for 3-sided queries, Arge et al. obtained an optimal structure [26]. We discuss the structure in Section 5.1 below. In practical applications involving massive datasets it is often crucial that external data structures use linear space. We discuss this further in Section 7. Grossi and Italiano developed the elegant linear space *cross-tree* data structure which answers planar range queries in $O(\sqrt{N/B}+T/B)$ I/Os [89, 90]. This is optimal for linear space data structures—as e.g. proven by Kanth and Singh [102]. The *O-tree* of Kanth and Singh [102] obtains the same bounds using ideas similar to the ones used by van Kreveld and Overmars in *divided k-d trees* [146]. In Section 5.2 below we discuss the cross-tree further.

5.1 Logarithmic query structure

The $O(\log_B N + T/B)$ query data structure is based on ideas from the corresponding internal memory data structure due to Chazelle [51]. The structure consists of a fan-out $\log_B N$ base tree over the x-coordinates of the N points. As previously an x-range is associated with each node v and it is subdivided into $\log_B N$ slabs by v's children $v_1, v_2, \ldots, v_{\log_B N}$. We store *all* the points in the x-range of v in four secondary data structures associated with v. The first structure store the points in a linear list sorted by y-coordinate. The three other structures are external priority search trees. Two of these structures are used for answering 3-sided queries with the opening to the left and to the right, respectively. For the third priority search tree, we consider for each child v_i the points in the x-range of v_i in y-order, and for each pair of consecutive points (x_1, y_1) and (x_2, y_2) we store the point (y_1, y_2) in the tree. With each constructed point we also store pointers to the corresponding two original point in the sorted list of points in a child node. Since we use linear space on each of the $O(\log_{\log_B N}(N/B)) = O(\log(N/B)/\log\log_B N)$ levels of the tree, the structure uses $O(\frac{N\log(N/B)}{B\log\log_B N})$ disk blocks in total.

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 where the x-range $[q_1, q_2]$ of the query contains a boundary between two slabs. Consider the case where q_1 lies in the x-range of v_i and q_2 lies in the x-range of v_j—refer to Figure 6. The query q is naturally decomposed into three parts, consisting of a part in v_i, a part in v_j, and a part completely spanning nodes v_k, for $i < k < j$. The points contained in the first two parts can be found in $O(\log_B N+T/B)$ I/Os using the right opening priority search tree in v_i and the left opening priority search tree in v_j. To find the points in the third part we query the third priority search tree associated with v with $(-\infty, q_2, q_2)$, that is, we find all points (y_1, y_2) in the structure with $y_1 \leq q_2$ and $y_2 \geq q_2$. Since a point (y_1, y_2) corresponds to a consecutive pair (x_1, y_1) and (x_2, y_2) of the original points in a slab, we in this way obtain the $O(\log_B N)$ bottommost point contained in the query for each of the nodes v_{i+1}, $v_{i+2}, \ldots, v_{j-1}$. Using the pointers to the same points in these children nodes, we then traverse the $j - i - 1 = O(\log_B N)$ relevant sorted lists and output the remaining points using $O(\log_B N + T/B)$ I/Os.

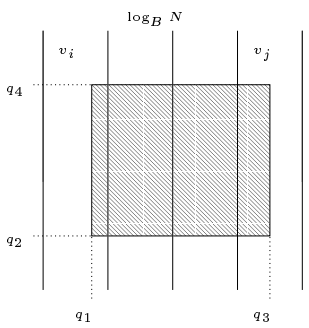

Fig. 6. The slabs corresponding to a node v in the base tree. To answer a query (q_1, q_2, q_3, q_4) we need to answer 3-sided queries on the points in slab v_i and slab v_j, and a range query on the points in the $O(\log_B N)$ slabs between v_i and v_j.

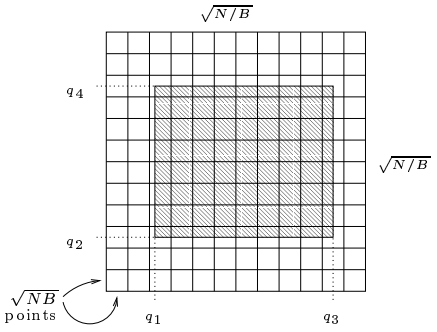

Fig. 7. Basic squares. To answer a query (q_1, q_2, q_3, q_4) we check points in two vertical and two horizontal slabs, and report points in basic squares completely covered by the query.

To insert or delete a point, we need to perform $O(1)$ updates on each of the $O(\log(N/B)/\log\log_B N)$ levels of the base tree. Each of these updates takes $O(\log_B N)$ I/Os. We also need to update the base tree. Using a weight-balanced B-tree, Arge et al. showed how this can be done in $O((\log_B N)(\log \frac{N}{B})/\log\log_B N)$ I/Os [26].

5.2 Linear space structure

The linear space cross-tree structure of Grossi and Italiano consists of two levels [89, 90]. The lower level partitions the plane into $\Theta(\sqrt{N/B})$ vertical slabs and $\Theta(\sqrt{N/B})$ horizontal slabs containing $\Theta(\sqrt{NB})$ points each, forming an irregular grid of $\Theta(N/B)$ *basic squares*—refer to Figure 7. Each basic square can contain between 0 and $\sqrt{N/B}$ points. The points are grouped and stored according to the vertical slabs—points in vertically adjacent basic squares containing less than B points are grouped together to form groups of $\Theta(B)$ points and stored in blocks together. The points in a basic square containing more than B points are stored in a B-tree. Thus the lower level uses $O(N/B)$ space. The upper level consists of a linear space search structure which can be used to determine the basic square containing a given point—for now we can think of the structure as consisting of a fan-out $\sqrt{B}$ B-tree $\mathcal{T}_V$ on the $\sqrt{N/B}$ vertical slabs and a separate fan-out $\sqrt{B}$ B-tree $\mathcal{T}_H$ on the $\sqrt{N/B}$ horizontal slabs.

In order to answer a query (q_1, q_2, q_3, q_4) we use the upper level search tree to find the vertical slabs containing q_1 and q_3 and the horizontal slabs containing q_2 and q_4 using $O(\log_{\sqrt{B}} N) = O(\log_B N)$ I/Os. We then explicitly check all points in these slabs and report all the relevant points. In doing so we use $O(\sqrt{NB}/B) = O(\sqrt{N/B})$ I/Os to traverse the vertical slabs and $O(\sqrt{NB}/B +$

$\sqrt{N/B}) = O(\sqrt{N/B})$ I/Os to traverse the horizontal slabs (the $\sqrt{N/B}$-term in the latter bound is a result of the slabs being blocked vertically—a horizontal slab contains $\sqrt{N/B}$ basic squares). Finally, we report all points corresponding to basic squares fully covered by the query. To do so we use $O(\sqrt{N/B}+T/B)$ I/Os since the slabs are blocked vertically. In total we answer a query in $O(\sqrt{N/B}+T/B)$ I/Os.

In order to perform an update we need to find and update the relevant basic square. We may also need to split slabs (insertion) or merge slabs with neighbor slabs (deletions). In order to do so efficiently while still being able to answer a range query I/O-efficiently, the upper level is actually implemented using a *cross-tree* $\mathcal{T}_{HV}$. $\mathcal{T}_{HV}$ can be viewed as a cross product of $\mathcal{T}_V$ and $\mathcal{T}_H$: For each pair of nodes $u \in \mathcal{T}_H$ and $v \in \mathcal{T}_V$ on the same level we have a node (u, v) in $\mathcal{T}_{HV}$, and for each pair of edges $(u, u') \in \mathcal{T}_H$ and $(v, v') \in \mathcal{T}_V$ we have an edge $((u, v), (u', v'))$ in $\mathcal{T}_{HV}$. Thus the tree has fan-out $O(B)$ and uses $O((\sqrt{N/B})^2) = O(N/B)$ space. Grossi and Italiano showed how we can use the cross-tree to search for a basic square in $O(\log_B N)$ I/Os and how the full structure can be used to answer a range query in $O(\sqrt{N/B}+T/B)$ I/Os [89, 90]. They also showed that if $\mathcal{T}_H$ and $\mathcal{T}_V$ are implemented using weight-balanced B-trees, the structure can be maintained in $O(\log_B N)$ I/Os during an update.

6 Survey of other external data structures

After having discussed planar range searching is some detail, in this section we survey other results on worst-case efficient external data structures.

Interval management. The interval management (or stabbing query) problem is the problem of maintaining a dynamically changing set of (one-dimensional) intervals such that given a query point q all intervals containing q can be reported efficiently. By mapping each interval $[x, y]$ to the point (x, y) in the plane, a query corresponds to finding all points such that $x \leq q$ and $y \geq q$, which means that the external priority search tree describe in Section 4.2 can be used to solve this problem optimally. However, the problem was first solved optimally by Arge and Vitter [30], who developed an external version of the *interval tree* of Edelsbrunner [71, 72].

The external interval tree was the first to use several of the ideas also utilized in the external priority search tree; filtering, bootstrapping, and weight-balanced B-trees. The structure also utilized the notion of *multislabs*, which is useful when storing objects (like intervals) with a spatial extent. Recall that a slab is a subranges of the range associated with a node v of a base tree defined by the range of one of v's children. A multislabs is simply a contiguous sets of slabs. A key idea in the external interval tree is to decrease the fanout of the base tree to $\sqrt{B}$, maintaining the $O(\log_B N)$ tree height, such that each node has $O(\sqrt{(B)^2}) = O(B)$ associated multislabs. This way a constant amount of information can be stored about each multislab in $O(1)$ blocks. Similar ideas have been utilized in several other external data structures [14, 25, 3, 28]. Variants

of the external interval tree structure, as well as applications of it in isosurface extraction, have been considered by Chiang and Silva [29, 60, 62, 61] (see also [7]).

Planar point location. The planar point location problem is defined as follows: Given a planar subdivision with N vertices (i.e., a decomposition of the plane into polygonal regions induced by a straight-line planar graph), construct a data structure so that the face containing a query point p can be reported efficiently. In internal memory, a lot of work has been done on this problem—see e.g. the survey by Snoeyink [140]. Goodrich et al. [86] described the first query optimal $O(\log_B N)$ I/O static solution to the problem, and several structures which can answer a batch of queries I/O-efficiently have also been developed [86, 29, 25, 65, 143].

Recently, progress has been made in the development of I/O-efficient dynamic point location structures. In the dynamic version of the problem one can change the subdivision dynamically (insert and delete edges and vertices). Based on the external interval tree structure and ideas also utilized in several internal memory structures [56, 34], Agarwal et al. [3] developed a dynamic structure for *monotone* subdivisions. Utilizing the logarithmic method and a technique similar to dynamic fractional cascading [54, 114], Arge and Vahrenhold improved the structure to work for general subdivisions. Their structure uses linear space and supports updates and queries in $O(\log_B^2 N)$ I/Os.

Range counting. Given a set of N points in the plane, a range counting query asks for the number of points within a query rectangle. Based on ideas utilized in an internal memory counting structure due to Chazelle [52], Agarwal et al. [6] designed an external data structure for the range counting problem. Their structure use linear space and answers a query in $O(\log_B N)$ I/Os. Based on a reduction due to Edelsbrunner and Overmars [74], they also designed a linear space and $O(\log_B N)$ query structure for the rectangle counting problem. In this problem, given a set of N rectangles in the plane, a query asks for the number of rectangles intersecting a query rectangle. Finally, they extended their structures to the d-dimensional versions of the two problems. See also [157] and references therein.

Higher-dimensional range searching. Vengroff and Vitter [149] presented a data structure for 3-dimensional range searching with a logarithmic query bound. With recent modifications their structure answers queries in $O(\log_B N + T/B)$ I/Os and uses $O(\frac{N}{B} \log^3 \frac{N}{B} / \log\log_B^3 N)$ space [151]. More generally, they presented structures for answering $(3+k)$-sided queries (k of the dimensions, $0 \leq k \leq 3$, have finite ranges) in $O(\log_B N + T/B)$ I/Os using $O(\frac{N}{B} \log^k \frac{N}{B} / \log\log_B^k N)$ space.

As mentioned, space use is often as crucial as query time when manipulating massive datasets. The linear space cross-tree of Grossi and Italiano [89, 90], as well as the O-tree of Kanth and Singh [102], can be extended to support d-dimensional range queries in $O((N/B)^{1-1/d} + T/B)$ I/Os. Updates can be

performed in $O(\log_B N)$ I/Os. The cross-tree can also be used in the design of dynamic data structures for several other problems [89, 90].

Halfspace range searching. Given a set of points in d-dimensional space, a halfspace range query asks for all points on one side of a query hyperplane. Halfspace range searching is the simplest form of non-isothetic (non-orthogonal) range searching. The problem was first considered in external memory by Franciosa and Talamo [83, 82]. Based on an internal memory structure due to Chazelle et al. [55], Agarwal et al. [5] described an optimal $O(\log_B N+T/B)$ query and linear space structure for the 2-dimensional case. Using ideas from an internal memory result of Chan [50], they described a structure for the 3-dimensional case, answering queries in $O(\log_B N+T/B)$ expected I/Os but requiring $O((N/B)\log(N/B))$ space. Based on the internal memory *partition trees* of Matoušek [112], they also gave a linear space data structure for answering d-dimensional halfspace range queries in $O((N/B)^{1-1/d+\epsilon}+T/B)$ I/Os for any constant $\epsilon > 0$. The structure supports updates in $O((\log(N/B))\log_B N)$ expected I/Os amortized. Using an improved construction algorithm, Agarwal et al. [4] obtained an $O(\log_B^2 N)$ amortized and expected update I/O-bound for the planar case. Agarwal et al. [5] also showed how the query bound of the structure can be improved at the expense of extra space. Finally, their linear space structure can also be used to answer very general queries—more precisely, all points within a query polyhedron with m faces can be found in $O(m(N/B)^{1-1/d+\epsilon}+T/B)$ I/Os.

Range searching on moving points. Recently there has been an increasing interest in external memory data structures for storing continuously moving objects. A key goal is to develop structures that only need to be changed when the velocity or direction of an object changes (as opposed to continuously).

Kollios at al. [105] presented initial work on storing moving points in the plane such that all points inside a query range at query time t can be reported in a provably efficient number of I/Os. Their results were improved and extended by Agarwal et al. [4] who developed a linear space structure that answers a query in $O((N/B)^{1/2+\epsilon}+T/B)$ I/Os for any constant $\epsilon > 0$. A point can be updated using $O(\log_B^2 N)$ I/Os. The structure is based on partition trees and can also be used to answer queries where two time values t_1 and t_2 are given and we want to find all points that lie in the query range at any time between t_1 and t_2. Using the notion of *kinetic data structures* introduced by Basch et al. [33], as well as a persistent version of the planar range searching structure [26] discussed in Section 5, Agarwal et al. [4] also developed a number of other structures with improved query performance. One of these structures has the property that queries in the near future are answered faster than queries further away in time. Further structures with this property were developed by Agarwal et al. [10].

Proximity queries. Proximity queries such as nearest neighbor and closest pair queries have become increasingly important in recent years, for example because of their applications in similarity search and data mining. Callahan et al. [47] developed the first worst-case efficient external proximity query data

structures. Their structures are based on an external version of the *topology trees* of Frederickson [84] called *topology B-trees*, which can be used to dynamically maintain arbitrary binary trees I/O-efficiently.

Using topology B-trees and ideas from an internal structure of Bespamyatnikh [42], Callahan et al. [47] designed a linear space data structure for dynamically maintaining the closest pair of a set of points in d-dimensional space. The structure supports updates in $O(\log_B N)$ I/Os. The same result was obtained by Govindarajan et al. [87] using the *well-separated pair decomposition* of Callahan and Kosaraju [48, 49]. Govindarajan et al. [87] also showed how to dynamically maintain a well-separated pair decomposition of a set of d-dimensional points using $O(\log_B N)$ I/Os per update.

Using topology B-trees and ideas from an internal structure due to Arya et al. [31], Callahan et al. [47] developed a linear space data structure for the dynamic approximate nearest neighbor problem. Given a set of points in d-dimensional space, a query point p, and a parameter ϵ, the approximate nearest neighbor problem consists of finding a point q with distance at most $(1+\epsilon)$ times the distance of the actual nearest neighbor of p. The structure answers queries and supports updates in $O(\log_B N)$ I/Os. Agarwal et al. [4] designed I/O-efficient data structures for answering approximate nearest neighbor queries on a set of moving points.

In some applications we are interested in finding not only the nearest but all the k nearest neighbors of a query point. Based on their 3-dimensional halfspace range searching structure, Agarwal et al [5] described a structure that uses $O((N/B)\log(N/B))$ space to store N points in the plane such that a k nearest neighbors query can be answered in $(\log_B N + k/B)$ I/Os.

7 Practical general-purpose structures

Although several of the worst-case efficient (and often optimal) data structures discussed in the previous sections are simple enough to be of practical interest, they are often not the obvious choices when deciding which data structures to use in a real-world application. There are several reasons for this, one of the most important being that in real applications involving massive datasets it is practically feasible to use data structures of size cN/B only for a very small constant c. Since fundamental lower bounds often prevent logarithmic worst-case search cost for even relatively simple problems when restricting the space use to linear, we need to develop heuristic structures which perform well in most practical cases. Space restrictions also motivate us not to use structures for single specialized queries but instead design general structures that can be used to answer several different types of queries. Finally, implementation considerations often motivate us to sacrifice worst-case efficiency for simplicity. All of these considerations have led to the development of a large number of general-purpose data structures that often work well in practice, but which do not come with worst-case performance guarantees. Below we quickly survey the major classes

of such structures. The reader is referred to more complete surveys for details [12, 85, 121, 88, 124, 134].

Range searching in d-dimensions is the most extensively researched problem. A large number of structures have been developed for this problem, including space filling curves (see e.g. [123, 1, 32]), grid-files [119, 94], various quad-trees [133, 134], kd-B tress [128]—and variants like Buddy-trees [138], hB-trees [109, 75] and cell-trees [91]—and various R-trees [92, 88, 139, 37, 100]. Often these structures are broadly classified into two types, namely *space driven* structures (like quad-trees and grid-files), which partition the embedded space containing the data points and *data driven* structures (like kd-B trees and R-trees), which partition the data points themselves. Agarwal et al. [9] describe a general framework for efficient construction and updating of many of the above structures.

As mentioned above, we often want to be able to answer a very diverse set of queries, like halfspace range queries, general polygon range queries, and point location queries, on a single data structure. Many of the above data structures can easily be used to answer many such different queries and that is one main reason for their practical success. Recently, there has also been a lot of work on extensions—or even new structures—which also support e.g. moving objects (see e.g [155, 132, 154, 126] and references therein) or proximity queries (see e.g. [41, 129, 103, 85, 12, 121] and references therein). However, as discussed, most often no guarantee on the worst-case query performance is provided for these structures.

So far we have mostly discussed point data structures. In general, we are interested in storing objects such as lines and polyhedra with a spatial extent. Like in the point case, a large number of heuristic structures, many of which are variations of the ones mentioned above, have been proposed for such objects. However, almost no worst-case efficient structures are known. In practice a *filtering/refinement* method is often used when managing objects with spatial extent. Instead of directly storing the objects in the data structure we store the *minimal bounding (axis-parallel) rectangle* containing each object together with a pointer to the object itself. When answering a query we first find all the minimal bounding rectangles fulfilling the query (the *filtering* step) and then we retrieve the objects corresponding to these rectangles and check each of them to see if they fulfill the query (the *refinement* step). One way of designing data structures for rectangles (or even more general objects) is to transform them into points in higher-dimensional space and store these points in one of the point data structures discussed above (see e.g. [85, 121] for a survey). However, a structure based on another idea has emerged as especially efficient for storing and querying minimal bounding rectangles. Below we further discuss this so-called R-tree and its many variants.

R-trees. The R-tree, originally proposed by Guttman [92], is a multiway tree very similar to a B-tree; all leaf nodes are on the same level of the tree and a leaf contains $\Theta(B)$ data rectangles. Each internal node v (except maybe for the root) has $\Theta(B)$ children. For each of its children v_i, v contains the minimal bounding rectangle of all the rectangles in the tree rooted in v_i. An R-tree has height $O(\log_B N)$ and uses $O(N/B)$ space. An example of an R-tree is shown in

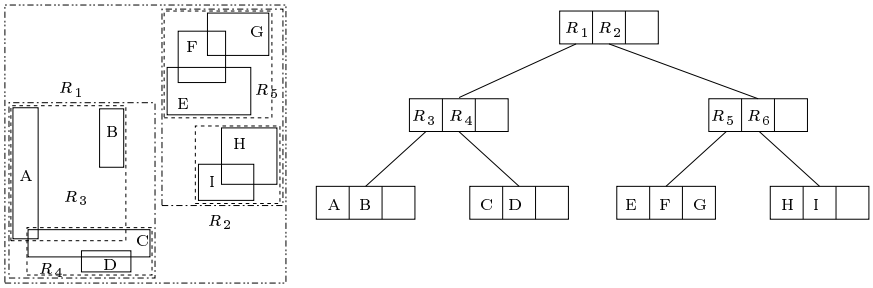

Fig. 8. R-tree constructed on rectangles A, B, C, ... , I ($B = 3$).

Figure 8. Note that there is no unique R-tree for a given set of data rectangles and that minimal bounding rectangles stored within an R-tree node can overlap.

In order to query an R-tree to find, say, all rectangles containing a query point p, we start at the root and recursively visit all children whose minimal bounding rectangle contains p. This way we visit all internal nodes whose minimal bounding rectangle contains p. There can be many more such nodes than actual data rectangles containing p and intuitively we want the minimal bounding rectangles stored in an internal node to overlap as little as possible in order to obtain a query efficient structure.

An insertion can be performed in $O(\log_B N)$ I/Os like in a B-tree. We first traverse the path from the root to the leaf we choose to insert the new rectangle into. The insertion might result in the need for node splittings on the same root-leaf path. As insertion of a new rectangle can increase the overlap in a node, several heuristics for choosing which leaf to insert a new rectangle into, as well as for splitting nodes during rebalancing, have been proposed [88, 139, 37, 100]. The R*-tree variant of Beckmann et al. [37] seems to result in the best performance in many cases. Deletions are also performed similarly to deletions in a B-tree but we cannot guarantee an $O(\log_B N)$ bound since finding the data rectangle to delete may require many more I/Os. Rebalancing after a deletion can be performed by fusing nodes like in a B-tree but some R-tree variants instead delete a node when it underflows and reinsert its children into the tree (often referred to as "forced reinsertion"). The idea is to try to obtain a better structure by forcing a global reorganization of the structure instead of the local reorganization a node fuse constitutes.

Constructing an R-tree using repeated insertion takes $O(N \log_B N)$ I/Os and does not necessarily result in a good tree in terms of query performance. Therefore several sorting based $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/O construction algorithms have been proposed [130, 99, 69, 108, 40]. Several of these algorithms produce an R-tree with practically better query performance than an R-tree built by repeated insertion. Still, no better than a linear worst-case query I/O-bound has been proven for any of them. Very recently, however, de Berg et al. [68] and Agarwal et al. [11] presented R-tree construction algorithms resulting in R-trees with provably efficient worst-case query performance measured in terms

of certain parameters describing the input data. They also discussed how these structures can be efficiently maintained dynamically.

8 Implementation of I/O-efficient data structures

Two ongoing projects aim at developing software packages that facilitates implementation of I/O-efficient algorithms and data structures in a high-level, portable and efficient way. These projects are the LEDA-SM project at MPI in Germany [66, 67] and the TPIE (Transparent Parallel I/O Programming Environment) project at Duke [17, 148]. We briefly discuss these projects and the experiments performed within them below. Outside these projects, a few other authors have reported on stand-alone implementations of geometric algorithms [57, 58], external interval trees [60–62], buffer trees [96], and string B-trees [79].

LEDA-SM. LEDA-SM is an extension of the LEDA library [115] of efficient algorithms and data structures. It consists of a kernel that gives an abstract view of external memory as a collection of disks, each consisting of a collection of blocks. The kernel provides a number of primitives for manipulating blocks, which facilitate efficient implementation of external memory algorithms and data structures. The LEDA-SM distribution also contains a collection of fundamental data structures, such as stacks, queues, heaps, B-trees and buffer-trees, as well as a few fundamental algorithms such as external sorting and matrix operations. It also contains algorithms and data structures for manipulating strings and massive graphs. Results on the practical performance of LEDA-SM implementations of external priority queues and I/O-efficient construction of suffix arrays can be found in [44] and [64], respectively.

TPIE. The first part of the TPIE project took a stream-based approach to computation [148, 17], where the kernel feeds a continuous stream of elements to the user programs in an I/O-efficient manner. This approach is justified by theoretical research on I/O-efficient algorithms, which show that a large number of problems can be solved using a small number of streaming paradigms, all implemented in TPIE. This part of TPIE also contains fundamental data structures such as queues and stacks, algorithms for sorting and matrix operations, as well as a few more specialized geometric algorithms. It has been used in I/O-efficient implementations of several scientific computation [150], spatial join [24, 25, 23], and terrain flow [27, 19] algorithms.

Since most external data structures cannot efficiently be implemented in a stream-based framework, the second part of the TPIE project adds kernel support for a block oriented programming style. Like in LEDA-SM, the external memory is viewed as a collection of blocks and primitives for manipulating such blocks are provided. Fundamental data structure such as B-trees and R-trees are also provided with this part of TPIE. The block oriented part of TPIE is integration with the stream oriented part, and together the two parts have been used to implement I/O-efficient algorithms for R-trees construction [21] based on

the buffer technique, to implement an I/O-efficient algorithms for constriction and updating of kd-tree [8, 9], as well as to implement the recently developed structures for range counting [6]. Other external data structures currently being implemented includes persistent B-trees, structures for planar point location, and the external priority search tree.

9 Conclusions

In this paper we have discussed recent advances in the development of provably efficient external memory dynamic data structures, mainly for geometric objects and especially for the one- and two-dimensional range searching problems. A more detailed survey by the author can be found in [16]. 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.

Acknowledgments

The author thanks Tammy Bailey, Tavi Procopiuc, and Jan Vahrenhold for comments on earlier drafts of this paper.

References

1. D. J. Abel and D. M. Mark. A comparative analysis of some two-dimensional orderings. *Intl. J. Geographic Informations Systems*, 4(1):21–31, 1990.
2. J. Abello, A. L. Buchsbaum, and J. R. Westbrook. A functional approach to external graph algorithms. In *Proc. Annual European Symposium on Algorithms, LNCS 1461*, pages 332–343, 1998.
3. 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 Symp. on Discrete Algorithms*, pages 1116–1127, 1999.
4. P. K. Agarwal, L. Arge, and J. Erickson. Indexing moving points. In *Proc. ACM Symp. Principles of Database Systems*, pages 175–186, 2000.
5. 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.
6. P. K. Agarwal, L. Arge, and S. Govindarajan. CRB-tree: An optimal indexing scheme for 2d aggregate queries. Manuscript, 2001.
7. P. K. Agarwal, L. Arge, T. M. Murali, K. Varadarajan, and J. S. Vitter. I/O-efficient algorithms for contour line extraction and planar graph blocking. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 117–126, 1998.
8. P. K. Agarwal, L. Arge, O. Procopiuc, and J. S. Vitter. Dynamic kd-trees on large data sets. Manuscript, 2001.
9. P. K. Agarwal, L. Arge, O. Procopiuc, and J. S. Vitter. A framework for index bulk loading and dynamization. In *Proc. Annual International Colloquium on Automata, Languages, and Programming*, 2001.

10. P. K. Agarwal, L. Arge, and J. Vahrenhold. A time responsive indexing scheme for moving points. In *Proc. Workshop on Algorithms and Data Structures*, 2001.
11. 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 Symp. on Computational Geometry*, pages 124–133, 2001.
12. 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, Providence, RI, 1999.
13. A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.
14. L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Proc. Workshop on Algorithms and Data Structures, LNCS 955*, pages 334–345, 1995. A complete version appears as BRICS technical report RS-96-28, University of Aarhus.
15. L. Arge. The I/O-complexity of ordered binary-decision diagram manipulation. In *Proc. Int. Symp. on Algorithms and Computation, LNCS 1004*, pages 82–91, 1995. A complete version appears as BRICS technical report RS-96-29, University of Aarhus.
16. L. Arge. External memory data structures. In J. Abello, P. M. Pardalos, and M. G. C. Resende, editors, *Handbook of Massive Data Sets*. Kluwer Academic Publishers, 2001. (To appear).
17. L. Arge, R. Barve, O. Procopiuc, L. Toma, D. E. Vengroff, and R. Wickremesinghe. *TPIE User Manual and Reference (edition 0.9.01a)*. Duke University, 1999. The manual and software distribution are available on the web at `http://www.cs.duke.edu/TPIE/`.
18. L. Arge, G. S. Brodal, and L. Toma. On external memory MST, SSSP and multi-way planar graph separation. In *Proc. Scandinavian Workshop on Algorithms Theory, LNCS 1851*, pages 433–447, 2000.
19. L. Arge, J. Chase, P. Halpin, L. Toma, D. Urban, J. Vitter, and R. Wickremesinghe. Flow computation on massive grids. In *16'th Annual Symposium of the International Association of Landscape Ecology (US-IALE 2001)*, 2001.
20. L. Arge, P. Ferragina, R. Grossi, and J. Vitter. On sorting strings in external memory. In *Proc. ACM Symp. on Theory of Computation*, pages 540–548, 1997.
21. L. Arge, K. H. Hinrichs, J. Vahrenhold, and J. S. Vitter. Efficient bulk operations on dynamic R-trees. In *Proc. Workshop on Algorithm Engineering, LNCS 1619*, pages 328–347, 1999.
22. L. Arge, U. Meyer, L. Toma, and N. Zeh. On external-memory planar depth first search. In *Proc. Workshop on Algorithms and Data Structures*, 2001.
23. L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, J. Vahrenhold, and J. S. Vitter. A unified approach for indexed and non-indexed spatial joins. In *Proc. Conference on Extending Database Technology*, 1999.
24. L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Scalable sweeping-based spatial join. In *Proc. International Conf. on Very Large Databases*, 1998.
25. 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 Symp. on Discrete Algorithms*, pages 685–694, 1998.
26. L. Arge, V. Samoladas, and J. S. Vitter. On two-dimensional indexability and optimal range search indexing. In *Proc. ACM Symp. Principles of Database Systems*, pages 346–357, 1999.

27. L. Arge, L. Toma, and J. S. Vitter. I/O-efficient algorithms for problems on grid-based terrains. In *Proc. Workshop on Algorithm Engineering and Experimentation*, 2000.
28. L. Arge and J. Vahrenhold. I/O-efficient dynamic planar point location. In *Proc. ACM Symp. on Computational Geometry*, pages 191–200, 2000.
29. L. Arge, D. E. Vengroff, and J. S. Vitter. External-memory algorithms for processing line segments in geographic information systems. In *Proc. Annual European Symposium on Algorithms, LNCS 979*, pages 295–310, 1995. To appear in special issues of *Algorithmica* on Geographical Information Systems.
30. L. Arge and J. S. Vitter. Optimal dynamic interval management in external memory. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, pages 560–569, 1996.
31. S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching. In *Proc. 5th ACM-SIAM Sympos. Discrete Algorithms*, pages 573–582, 1994.
32. T. Asano, D. Ranjan, T. Roos, E. Welzl, and P. Widmayer. Space-filling curves and their use in the design of geometric data structures. *Theoret. Comput. Sci.*, 181(1):3–15, July 1997.
33. J. Basch, L. J. Guibas, and J. Hershberger. Data structures for mobile data. *Journal of Algorithms*, 31(1):1–28, 1999.
34. H. Baumgarten, H. Jung, and K. Mehlhorn. Dynamic point location in general subdivisions. *Journal of Algorithms*, 17:342–380, 1994.
35. R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.
36. B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion B-tree. *VLDB Journal*, 5(4):264–275, 1996.
37. N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 322–331, 1990.
38. M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, pages 339–409, 2000.
39. J. L. Bentley. Decomposable searching problems. *Information Processing Letters*, 8(5):244–251, 1979.
40. S. Berchtold, C. Böhm, and H.-P. Kriegel. Improving the query performance of high-dimensional index structures by bulk load operations. In *Proc. Conference on Extending Database Technology, LNCS 1377*, pages 216–230, 1998.
41. S. Berchtold, B. Ertl, D. A. Keim, H.-P. Kriegel, and T. Seidl. Fast nearest neighbor search in high-dimensional spaces. In *Proc. IEEE International Conference on Data Engineering*, pages 209–218, 1998.
42. S. N. Bespamyatnikh. An optimal algorithm for closets pair maintenance. *Discrete and Computational Geometry*, 19:175–195, 1998.
43. G. Blankenagel and R. H. Güting. XP-trees—External priority search trees. Technical report, FernUniversität Hagen, Informatik-Bericht Nr. 92, 1990.
44. K. Brengel, A. Crauser, P. Ferragina, and U. Meyer. An experimental study of priority queues in external memory. In *Proc. Workshop on Algorithm Engineering, LNCS 1668*, pages 345–358, 1999.
45. 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.

46. A. L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. R. Westbrook. On external memory graph traversal. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 859–860, 2000.
47. 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.
48. P. B. Callahan and S. R. Kosaraju. Algorithms for dynamic closest-pair and n-body potential fields. In *Proc. 6th ACM-SIAM Sympos. Discrete Algorithms*, pages 263–272, 1995.
49. P. B. Callahan and S. R. Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *Journal of the ACM*, 42(1):67–90, 1995.
50. T. M. Chan. Random sampling, halfspace range reporting, and construction of ($\leq k$)-levels in three dimensions. *SIAM Journal of Computing*, 30(2):561–575, 2000.
51. B. Chazelle. Filtering search: a new approach to query-answering. *SIAM J. Comput.*, 15(3):703–724, 1986.
52. B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM J. Comput.*, 17(3):427–462, June 1988.
53. B. Chazelle. Lower bounds for orthogonal range searching: I. the reporting case. *Journal of the ACM*, 37(2):200–212, Apr. 1990.
54. B. Chazelle and L. J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1:133–162, 1986.
55. B. Chazelle, L. J. Guibas, and D. T. Lee. The power of geometric duality. *BIT*, 25(1):76–90, 1985.
56. S. W. Cheng and R. Janardan. New results on dynamic planar point location. *SIAM J. Comput.*, 21(5):972–999, 1992.
57. Y.-J. Chiang. *Dynamic and I/O-Efficient Algorithms for Computational Geometry and Graph Problems: Theoretical and Experimental Results.* PhD thesis, Brown University, August 1995.
58. Y.-J. Chiang. Experiments on the practical I/O efficiency of geometric algorithms: Distribution sweep vs. plane sweep. In *Proc. Workshop on Algorithms and Data Structures, LNCS 955*, pages 346–357, 1995.
59. Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 139–149, 1995.
60. Y.-J. Chiang and C. T. Silva. I/O optimal isosurface extraction. In *Proc. IEEE Visualization*, pages 293–300, 1997.
61. 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.
62. Y.-J. Chiang, C. T. Silva, and W. J. Schroeder. Interactive out-of-core isosurface extraction. In *Proc. IEEE Visualization*, pages 167–174, 1998.
63. D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.
64. A. Crauser and P. Ferragina. On constructing suffix arrays in external memory. In *Proc. Annual European Symposium on Algorithms, LNCS, 1643*, pages 224–235, 1999.

65. A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, and E. Ramos. Randomized external-memory algorithms for some geometric problems. In *Proc. ACM Symp. on Computational Geometry*, pages 259–268, 1998.
66. A. Crauser and K. Mehlhorn. *LEDA-SM: A Platform for Secondary Memory Computation.* Max-Planck-Institut für Informatik, 1999. The manual and software distribution are available on the web at `http://www.mpi-sb.mpg.de/~crauser/leda-sm.html`.
67. A. Crauser and K. Mehlhorn. LEDA-SM: Extending LEDA to secondary memory. In *Proc. Workshop on Algorithm Engineering*, 1999.
68. M. de Berg, J. Gudmundsson, M. Hammar, and M. Overmars. On R-trees with low stabbing number. In *Proc. Annual European Symposium on Algorithms*, pages 167–178, 2000.
69. D. J. DeWitt, N. Kabra, J. Luo, J. M. Patel, and J.-B. Yu. Client-server paradise. In *Proc. International Conf. on Very Large Databases*, pages 558–569, 1994.
70. 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.
71. H. Edelsbrunner. A new approach to rectangle intersections, part I. *Int. J. Computer Mathematics*, 13:209–219, 1983.
72. H. Edelsbrunner. A new approach to rectangle intersections, part II. *Int. J. Computer Mathematics*, 13:221–229, 1983.
73. H. Edelsbrunner and M. Overmars. Batched dynamic solutions to decomposable searching problems. *Journal of Algorithms*, 6:515–542, 1985.
74. H. Edelsbrunner and M. H. Overmars. On the equivalence of some rectangle problems. *Inform. Process. Lett.*, 14:124–127, 1982.
75. 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.
76. R. Fadel, K. V. Jakobsen, J. Katajainen, and J. Teuhola. Heaps and heapsort on secondary storage. *Theoretical Computer Science*, 220(2):345–362, 1999.
77. M. Farach, P. Ferragina, and S. Muthukrishnan. Overcoming the memory bottleneck in suffix tree construction. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, pages 174–183, 1998.
78. P. Ferragina and R. Grossi. A fully-dynamic data structure for external substring search. In *Proc. ACM Symp. on Theory of Computation*, pages 693–702, 1995.
79. P. Ferragina and R. Grossi. Fast string searching in secondary storage: Theoretical developments and experimental results. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 373–382, 1996.
80. P. Ferragina and F. Luccio. Dynamic dictionary matching in external memory. *Information and Computation*, 146(2):85–99, 1998.
81. E. Feuerstein and A. Marchetti-Spaccamela. Memory paging for connectivity and path problems in graphs. In *Proc. Int. Symp. on Algorithms and Computation, LNCS 762*, pages 416–425, 1993.
82. P. Franciosa and M. Talamo. Time optimal halfplane search on external memory. Unpublished manuscript, 1997.
83. P. G. Franciosa and M. Talamo. Orders, k-sets and fast halfplane search on paged memory. In *Proc. Workshop on Orders, Algorithms and Applications (ORDAL'94), LNCS 831*, pages 117–127, 1994.
84. G. N. Frederickson. A structure for dynamically maintaining rooted trees. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 175–184, 1993.
85. V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

86. M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, pages 714–723, 1993.
87. S. Govindarajan, T. Lukovszki, A. Maheshwari, and N. Zeh. I/O-efficient well-separated pair decomposition and its applications. In *Proc. Annual European Symposium on Algorithms*, pages 220–231, 2000.
88. D. Greene. An implementation and performance analysis of spatial data access methods. In *Proc. IEEE International Conference on Data Engineering*, pages 606–615, 1989.
89. 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, DIMACS series in Discrete Mathematics and Theoretical Computer Science, 1999. Revised version available at ftp://ftp.di.unipi.it/pub/techreports/TR-00-16.ps.Z.
90. R. Grossi and G. F. Italiano. Efficient splitting and merging algorithms for order decomposable problems. *Information and Computation*, 154(1):1–33, 1999.
91. O. Günther. The design of the cell tree: An object-oriented index structure for geometric databases. In *Proc. IEEE International Conference on Data Engineering*, pages 598–605, 1989.
92. A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 47–57, 1984.
93. J. M. Hellerstein, E. Koutsoupias, and C. H. Papadimitriou. On the analysis of indexing schemes. In *Proc. ACM Symp. Principles of Database Systems*, pages 249–256, 1997.
94. K. H. Hinrichs. *The grid file system: Implementation and case studies of applications.* PhD thesis, Dept. Information Science, ETH, Zürich, 1985.
95. S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982.
96. D. Hutchinson, A. Maheshwari, J.-R. Sack, and R. Velicescu. Early experiences in implementing the buffer tree. In *Proc. Workshop on Algorithm Engineering*, pages 92–103, 1997.
97. D. Hutchinson, A. Maheshwari, and N. Zeh. An external-memory data structure for shortest path queries. In *Proc. Annual Combinatorics and Computing Conference, LNCS 1627*, pages 51–60, 1999.
98. 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.
99. I. Kamel and C. Faloutsos. On packing R-trees. In *Proc. International Conference on Information and Knowledge Management*, pages 490–499, 1993.
100. I. Kamel and C. Faloutsos. Hilbert R-tree: An improved R-tree using fractals. In *Proc. International Conf. on Very Large Databases*, pages 500–509, 1994.
101. 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.
102. 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.
103. N. Katayama and S. Satoh. The SR-tree: An index structure for high-dimensional nearest-neighbor queries. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 369–380, 1997.

104. D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading MA, second edition, 1998.
105. G. Kollios, D. Gunopulos, and V. J. Tsotras. On indexing mobile objects. In *Proc. ACM Symp. Principles of Database Systems*, pages 261–272, 1999.
106. E. Koutsoupias and D. S. Taylor. Tight bounds for 2-dimensional indexing schemes. In *Proc. ACM Symp. Principles of Database Systems*, pages 52–58, 1998.
107. 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.
108. S. T. Leutenegger, M. A. López, and J. Edgington. STR: A simple and efficient algorithm for R-tree packing. In *Proc. IEEE International Conference on Data Engineering*, pages 497–506, 1996.
109. 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.
110. A. Maheshwari and N. Zeh. External memory algorithms for outerplanar graphs. In *Proc. Int. Symp. on Algorithms and Computation, LNCS 1741*, pages 307–316, 1999.
111. A. Maheshwari and N. Zeh. I/O-efficient algorithms for graphs of bounded treewidth. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 89–90, 2001.
112. J. Matoušek. Efficient partition trees. *Discrete Comput. Geom.*, 8:315–334, 1992.
113. E. McCreight. Priority search trees. *SIAM Journal of Computing*, 14(2):257–276, 1985.
114. K. Mehlhorn and S. Näher. Dynamic fractional cascading. *Algorithmica*, 5:215–241, 1990.
115. K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, UK, 2000.
116. U. Meyer. External memory bfs on undirected graphs with bounded degree. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 87–88, 2001.
117. D. R. Morrison. PATRICIA: Practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15:514–534, 1968.
118. K. Munagala and A. Ranade. I/O-complexity of graph algorithm. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 687–694, 1999.
119. J. Nievergelt, H. Hinterberger, and K. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1):38–71, 1984.
120. J. Nievergelt and E. M. Reingold. Binary search tree of bounded balance. *SIAM Journal of Computing*, 2(1):33–43, 1973.
121. 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.
122. M. H. Nodine, M. T. Goodrich, and J. S. Vitter. Blocking for external graph searching. *Algorithmica*, 16(2):181–214, 1996.
123. J. Orenstein. Spatial query processing in an object-oriented database system. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 326–336, 1986.
124. J. Orenstein. A comparison of spatial query processing techniques for native and parameter spaces. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 343–352, 1990.

125. M. H. Overmars. *The Design of Dynamic Data Structures.* Springer-Verlag, LNCS 156, 1983.
126. D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel approaches to the indexing of moving objects trajectories. In *Proc. International Conf. on Very Large Databases*, pages 395–406, 2000.
127. S. Ramaswamy and S. Subramanian. Path caching: A technique for optimal external searching. In *Proc. ACM Symp. Principles of Database Systems*, pages 25–35, 1994.
128. J. Robinson. The K-D-B tree: A search structure for large multidimensional dynamic indexes. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 10–18, 1981.
129. N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 71–79, 1995.
130. N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed R-trees. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 17–31, 1985.
131. C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, 1994.
132. B. Salzberg and V. J. Tsotras. A comparison of access methods for time evolving data. *ACM Computing Surveys*, 31(2):158–221, 1999.
133. H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS.* Addison Wesley, MA, 1990.
134. H. Samet. *The Design and Analyses of Spatial Data Structures.* Addison Wesley, MA, 1990.
135. V. Samoladas and D. Miranker. A lower bound theorem for indexing schemes and its application to multidimensional range queries. In *Proc. ACM Symp. Principles of Database Systems*, pages 44–51, 1998.
136. P. Sanders. Fast priority queues for cached memory. In *Proc. Workshop on Algorithm Engineering and Experimentation, LNCS 1619*, pages 312–327, 1999.
137. N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29:669–679, 1986.
138. B. Seeger and H.-P. Kriegel. The buddy-tree: An efficient and robust access method for spatial data base systems. In *Proc. International Conf. on Very Large Databases*, pages 590–601, 1990.
139. T. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+-tree: A dynamic index for multi-dimensional objects. In *Proc. International Conf. on Very Large Databases*, pages 507–518, 1987.
140. J. Snoeyink. Point location. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 30, pages 559–574. CRC Press LLC, Boca Raton, FL, 1997.
141. S. Subramanian and S. Ramaswamy. The P-range tree: A new data structure for range searching in secondary memory. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 378–387, 1995.
142. J. D. Ullman and M. Yannakakis. The input/output complexity of transitive closure. *Annals of Mathematics and Artificial Intellegence*, 3:331–360, 1991.
143. 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*, 2000.
144. J. van den Bercken, B. Seeger, and P. Widmayer. A generic approach to bulk loading multidimensional index structures. In *Proc. International Conf. on Very Large Databases*, pages 406–415, 1997.

145. J. van den Bercken, B. Seeger, and P. Widmayer. A generic approach to processing non-equijoins. Technical Report 14, Philipps-Universität Marburg, Fachbereich Matematik und Informatik, 1998.
146. M. J. van Kreveld and M. H. Overmars. Divided k-d trees. *Algorithmica*, 6:840–858, 1991.
147. P. J. Varman and R. M. Verma. An efficient multiversion access structure. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):391–409, 1997.
148. D. E. Vengroff. A transparent parallel I/O environment. In *Proc. DAGS Symposium on Parallel Computation*, 1994.
149. D. E. Vengroff and J. S. Vitter. Efficient 3-D range searching in external memory. In *Proc. ACM Symp. on Theory of Computation*, pages 192–201, 1996.
150. D. E. Vengroff and J. S. Vitter. I/O-efficient scientific computation using TPIE. In *Proceedings of the Goddard Conference on Mass Storage Systems and Technologies*, NASA Conference Publication 3340, Volume II, pages 553–570, 1996.
151. J. S. Vitter. External memory algorithms and data structures. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*, pages 1–38. American Mathematical Society, DIMACS series in Discrete Mathematics and Theoretical Computer Science, 1999.
152. J. S. Vitter. Online data structures in external memory. In *Proc. Annual International Colloquium on Automata, Languages, and Programming, LNCS 1644*, pages 119–133, 1999.
153. J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory, I: Two-level memories. *Algorithmica*, 12(2–3):110–147, 1994.
154. S. Šaltenis, C. S. Jensen, S. T. Leutenegger, and M. A. López. Indexing the positions of continuously moving objects. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 331–342, 2000.
155. O. Wolfson, A. P. Sistla, S. Chamberlain, and Y. Yesha. Updating and querying databases that track mobile units. *Distributed and Parallel Databases*, 7(3):257–287, 1999.
156. N. Zeh. I/O-efficient planar separators and applications. Manuscript, 2001.
157. D. Zhang, A. Markowetz, V. Tsotras, D. Gunopulos, and B. Seeger. Efficient computation of temporal aggregates with range predicates. In *Proc. ACM Symp. Principles of Database Systems*, pages 237–245, 2001.