Online Data Structures in External Memory

Jeffrey Scott Vitter[1,2]*

[1] Duke University, Center for Geometric Computing,
Department of Computer Science, Durham, NC 27708–0129, USA
http://www.cs.duke.edu/~jsv/
jsv@cs.duke.edu
[2] I.N.R.I.A. Sophia Antipolis, 2004, route des Lucioles, B. P. 93,
06902 Sophia Antipolis Cedex, France

Abstract. The data sets for many of today's computer applications are too large to fit within the computer's internal memory and must instead be stored on external storage devices such as disks. A major performance bottleneck can be the input/output communication (or I/O) between the external and internal memories. In this paper we discuss a variety of online data structures for external memory, some very old and some very new, such as hashing (for dictionaries), B-trees (for dictionaries and 1-D range search), buffer trees (for batched dynamic problems), interval trees with weight-balanced B-trees (for stabbing queries), priority search trees (for 3-sided 2-D range search), and R-trees and other spatial structures. We also discuss several open problems along the way.

1 Introduction

The *Input/Output* communication (or simply *I/O*) between the fast internal memory and the slow external memory (such as disk) can be a bottleneck in applications that process massive amounts of data [33]. One promising approach is to design algorithms and data structures that bypass the virtual memory system and explicitly manage their own I/O. We refer to such algorithms and data structures as *external memory* (or *EM*) *algorithms and data structures*. (The terms *out-of-core algorithms* and *I/O algorithms* are also sometimes used.) We concentrate in this paper on the design and analysis of online EM memory data structures.

The three primary measures of performance of an algorithm or data structure are *the number of I/O operations performed, the amount of disk space used, and the internal (parallel) computation time*. For reasons of brevity we shall focus in this paper on only the first two measures. Most of the algorithms we mention run in optimal CPU time, at least for the single-processor case.

1.1 Disk Model

We can capture the main properties of magnetic disks and multiple disk systems by the commonly-used *parallel disk model* (PDM) introduced by Vitter and Shriver [69]. Data is transferred in large units of *blocks* of size B so as to amortize the latency of moving the read-write head and waiting for the disk to spin into position. Storage systems such

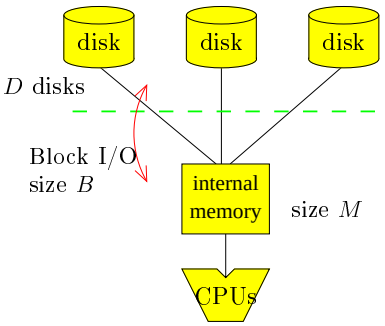
* Supported in part by the Army Research Office through MURI grant DAAH04–96–1–0013 and by the National Science Foundation through research grants CCR–9522047 and EIA–9870734.

as RAID use multiple disks to get more bandwidth [22, 39]. The principal parameters of PDM are the following:

N = problem input data size (items);
Z = problem output data size (items);
M = size of internal memory (items);
B = size of disk block (items);
D = # independent disks,

where $M < N$ and $1 \leq DB \leq M$. The first four parameters are all defined in units of items. For notational convenience, we define the corresponding parameters in units of blocks:

$$n = \frac{N}{B}; \qquad z = \frac{Z}{B}; \qquad m = \frac{M}{B}.$$



For simplicity, we restrict our attention in this paper to the single-disk case $D = 1$, since online data structures that use a single disk can generally be transformed automatically by the technique of disk striping to make optimal use of multiple disks [68].

Programs that perform well in terms of PDM will generally perform well when implemented on real systems [68]. More complex and precise models have been formulated [59, 62, 10]. Hierarchical (multilevel) memory models are discussed in [68] and its references.

1.2 Design Goals for Online Data Structures

Online data structures support the operation of *query* on a collection of data items. The nature of the query depends upon the application at hand. For example, in dictionary data structures, a query consists of finding the item (if any) that has a specified key value. In orthogonal range search, the data items are points in d-dimensional space $\mathbb{R}^d$, for some d, and a query involves finding all the points in a specified query d-dimensional rectangle. Other types of queries include point location, nearest neighbor, finding intersections, etc.

When the data items do not change and the data structure can be preprocessed before any queries are done, the data structure is known as *static*. When the data structure supports insertions and deletions of items, intermixed with the queries, the data structure is called *dynamic*. The primary theoretical challenges in the design and analysis of online EM data structures are three-fold:

1. to answer queries in $O(\log_B N + z)$ I/Os,
2. to use only a linear amount of disk storage space, and
3. to do updates (in the case of dynamic data structures) in $O(\log_B N)$ I/Os.

These criteria correspond to the natural lower bounds for online search in the comparison model. The three criteria are problem-dependent, and for some problems they cannot be met. For dictionary queries, we can do better using hashing, achieving $O(1)$ I/Os per query on the average.

Criterion 1 combines together the I/O cost $O(\log_B N)$ of the search component of queries with the I/O cost $O(\lceil z \rceil)$ for reporting the output, because when one cost is much larger than the other, the query algorithm has the extra freedom to follow a

filtering paradigm [19], in which both the search component and the output reporting are allowed to use the larger number of I/Os. For example, when the output size Z is large, the search component can afford to be somewhat sloppy as long as it doesn't use more than $O(z)$ I/Os; and when Z is small, the Z output items do not have to reside compactly in only $O(\lceil z \rceil)$ blocks. Filtering is an important design paradigm in online EM data structures.

For many of the online problems we consider, there is a data structure (such as binary search trees) for the internal memory version of the problem that can answer queries in $O(\log N + Z)$ CPU time, but if we use the same data structure naively in an external memory setting (using virtual memory to handle page management), a query may require $\Omega(\log N + Z)$ I/Os, which is excessive.[1] The goal is to build locality directly into the data structure and explicitly manage I/O so that the $\log N$ and Z terms in the I/O bounds of the naive approach are replaced by $\log_B N$ and z, respectively. The relative speedup in I/O performance, namely, $(\log N + Z)/(\log_B N + z)$, is at least $(\log N)/\log_B N = \log B$, which is significant in practice, and it can be as much as $Z/z = B$ for large Z.

1.3 Overview of Paper

In Section 2 we discuss EM hashing methods for dictionary applications. The most popular EM data structure is the B-tree structure, which provides excellent performance for dictionary operations and one-dimensional range searching. We give several variants and applications of B-trees in Section 3. We look at several aspects of multi-dimensional range search in Section 4. The contents of this paper are modifications of a broader survey by the author [68] with several additions. The reader is also referred to other surveys of online data structures for external memory [4, 27, 32, 56].

2 Hashing for Online Dictionary Search

Dictionary operations consist of insert, delete, and lookup. Given a value x, the lookup operation returns the item(s), if any, in the structure with key value x. The two main types of EM dictionaries are tree-based approaches (which we defer to Section 3) and hashing. The common element of all EM hashing algorithms is a pre-defined hash function $hash : \{\text{all possible keys}\} \rightarrow \{0, 1, 2, \ldots, K-1\}$ that assigns the N items to K address locations in a uniform manner.

The goals in EM hashing are to achieve an average of $O(1)$ I/Os per insert and delete, $O(\lceil z \rceil)$ I/Os per lookup, and linear disk space. Most traditional hashing methods use a statically allocated table and thus can handle only a fixed range of N. The challenge is to develop dynamic EM structures that adapt smoothly to widely varying values of N.

EM hashing methods fall into one of two categories: *directory* methods and *directoryless* methods. Fagin et al. [29] proposed the following directory scheme, called *extendible hashing*: Let us assume that the size K of the range of the hash function $hash$ is sufficiently large. The directory, for $d \geq 0$, consists of a table of 2^d pointers. Each item is assigned to the table location corresponding to the d least significant bits of its hash address. The value of d is set to the smallest value for which each table location has at most B items assigned to it. Each table location contains a pointer to a block where its items are stored. Thus, a lookup takes two I/Os: one to access the

[1] We use the notation $\log N$ to denote the binary (base 2) logarithm $\log_2 N$. For bases other than 2, the base will be specified explicitly, as in the base-B logarithm $\log_B N$.

directory and one to access the block storing the item. If the directory fits in internal memory, only one I/O is needed.

Many table locations may few items assigned to them, and for purposes of minimizing storage utilization, they can share the same disk block for storing their items. A table location shares a disk block with all the locations having the same k least significant bits, where k is chosen to be as small as possible so that the pooled items fit into a single disk block. Different table locations may have different values of k.

When a new item is inserted, and its disk block overflows, the items in the block are redistributed so that the invariants on d and k once again hold. Each time d is incremented by 1, the directory doubles in size, which is how extendible hashing adapts to a growing N. The pointers in the new directory are initialized to point to the appropriate disk blocks. The important point is that the disk blocks themselves do not need to be disturbed during doubling, except for the one block that splits.

Extendible hashing can handle deletions in a symmetric way by merging blocks. The combined size of the blocks being merged must be sufficiently less than B to prevent immediate splitting after a subsequent insertion. The directory shrinks by half (and d is decremented by 1) when all the local depths are less than the current value of d.

The expected number of disk blocks required to store the data items is asymptotically $n/\ln 2 \approx n/0.69$; that is, the blocks tend to be about 69% full [54]. At least $\Omega(n/B)$ blocks are needed to store the directory. Flajolet [30] showed on the average that the directory uses $\Theta(N^{1/B}n/B) = \Theta(N^{1+1/B}/B^2)$ blocks, which can be superlinear in N asymptotically! However, in practice the $N^{1/B}$ term is a small constant, typically less than 2.

A disadvantage of directory schemes is that two I/Os rather than one I/O are required when the directory is stored in external memory. Litwin [50] developed a directoryless method called *linear hashing* that expands the number of data blocks in a controlled regular fashion. In contrast to directory schemes, the blocks in directoryless methods are chosen for splitting in a predefined order. Thus the block that splits is usually not the block that has overflowed, so some of the blocks may require auxiliary overflow lists to store items assigned to them. On the other hand, directoryless methods have the advantage that there is no need for access to a directory structure, and thus searches often require only one I/O. A more detailed survey of methods for dynamic hashing is given in [27].

The above hashing schemes and their many variants work very well for dictionary applications in the average case, but have poor worst-case performance. They also do not support sequential search, such as retrieving all the items with key value in a specified range. Some clever work has been done on order-preserving hash functions, in which items with sequential keys are stored in the same block or in adjacent blocks, but the search performance is less robust and tends to deteriorate because of unwanted collisions. (See [32] for a survey.). A much more popular approach is to use multiway trees, which we explore next.

3 Spatial Data Structures

In this section we consider online EM data structures for storing and querying spatial data. A fundamental database primitive in spatial databases and geographic information systems (GIS) is orthogonal range search, which includes dictionary lookup as a special case. A range query, for a given d-dimensional rectangle, returns all the points in the interior of the rectangle. We use range searching in this section as the canonical

query on spatial data. Other types of spatial queries include point location queries, ray shooting queries, nearest neighbor queries, and intersection queries, but for brevity we restrict our attention primarily to range searching.

Spatial data structures tend to be of two types: space-driven or data-driven. Quad trees and grid files are space-driven since they are based upon a partitioning of the embedding space, somewhat akin to using order-preserving hash functions, whereas methods like R-trees and kd-trees are organized by partitioning the data items themselves. We shall discuss primarily the latter type in this section.

3.1 B-trees and Variants

Tree-based data structures arise naturally in the online setting, in which the data can be updated and queries must be processed immediately. Binary trees have a host of applications in the RAM model. In order to exploit block transfer, trees in external memory use a block for each node, which can store $\Theta(B)$ pointers and data values. The well-known *B-tree* due to Bayer and McCreight [12, 24, 46], which is probably the most widely used EM nontrivial data structure in practice, is a balanced multiway tree with height roughly $\log_B N$ and with node degree $\Theta(B)$. (The root node is allowed to have smaller degree.) B-trees support dynamic dictionary operations and one-dimensional range search optimally in the comparison model, satisfying the three design criteria of Section 1.2. When a node overflows during an insertion, it splits into two half-full nodes, and if the splitting causes the parent node to overflow, the parent node splits, and so on. Splittings can thus propagate up to the root, which is how the tree grows in height.

In the *B^+-tree* variant, pictured in Figure 1, all the items are stored in the leaves, and the leaves are linked together in symmetric order to facilitate range queries and sequential access. The internal nodes store only key values and pointers and thus can have a higher branching factor. In the most popular variant of B^+-trees, called *B^*-trees*, splitting can usually be postponed when a node overflows, by instead "sharing" the node's data with one of its adjacent siblings. The node needs to be split only if the sibling is also full; when that happens, the node splits into two, and its data and those of its full sibling are evenly redistributed, making each of the three nodes about 2/3 full. This local optimization reduces how often new nodes must be created and thus increases the storage utilization. And since there are fewer nodes in the tree, search I/O costs are lower. When no sharing is done (as in B^+-trees), Yao [71] shows that nodes are roughly $\ln 2 \approx 69\%$ full on the average, assuming random insertions. With sharing (as in B*-trees), the average storage utilization increases to about $2\ln(3/2) \approx 81\%$ [9, 49]. Storage utilization can be increased further by sharing among several siblings, but insertions and deletions get more complicated.

Persistent versions of B-trees have been developed by Becker et al. [13] and Varman and Verma [65]. Lomet and Salzberg [52] explore mechanisms to add concurrency and recovery to B-trees.

Arge and Vitter [8] give a useful variant of B-trees called *weight-balanced B-trees* with the property that the number of data items in any subtree of height h is $\Theta(a^h)$, for some fixed parameter a of order B. (By contrast, the sizes of subtrees at level h in a regular B-tree can differ by a multiplicative factor that is exponential in h.) When a node on level h gets rebalanced, no further rebalancing is needed until its subtree is updated $\Omega(a^h)$ times. This feature can support applications in which the cost to rebalance a node is $O(w)$, allowing the rebalancing to be done in an amortized (and often worst-case) way with $O(1)$ I/Os. Weight-balanced B-trees were originally conceived as part of an optimal dynamic EM interval tree data structure for answering

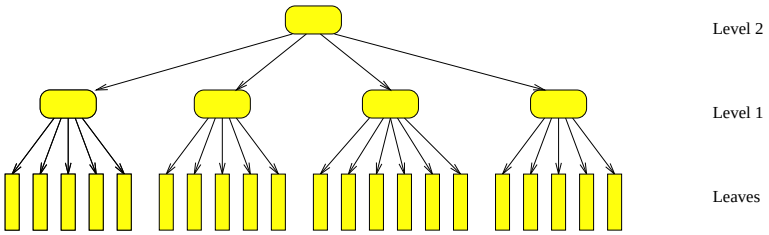


Fig. 1. B^+-tree multiway search tree. Each internal and leaf node corresponds to a disk block. All the items are stored in the leaves. The internal nodes store only key values and pointers, $\Theta(B)$ of them per node. Although not indicated here, the leaf blocks are linked together sequentially.

stabbing queries, which we discuss in Section 4.1, but they also have applications to the internal memory RAM model [8, 36]. For example, by setting a to a constant, we get a simple, worst-case implementation of interval trees in internal memory. They also serve as a simpler and worst-case alternative to the data structure in [70] for augmenting one-dimensional data structures with range restriction capabilities.

Weight-balanced B-trees can also be used to maintain parent pointers efficiently in the worst case: When a node splits during overflow, it costs $\Theta(B)$ I/Os to update parent pointers. We can reduce the cost via amortization arguments and global rebuilding to only $\Theta(\log_B N)$ I/Os, since nodes do not split too often. However, this approach will not work if the B-tree needs to support cut and concatenate operations. Agarwal et al. [1] develop an interesting variant of B-trees with parent pointers, called *level-balanced B-trees*, in which the local balancing condition on the degree of nodes is replaced by a global balancing condition on the number of nodes at each level of the tree. Level-balanced B-trees support search and order operations in $O(\log_B N + z)$ I/Os, and the update operations insert, delete, cut, and concatenate can be done in $O\big((1 + (b/B)(\log_m n)\log_b N\big)$ I/Os amortized, for any $2 \le b \le B/2$, which is bounded by $O\big((\log_B N)^2\big)$. Agarwal et al. [1] use level-balanced B-trees in a data structure for point location in monotone subdivisions, which supports queries and (amortized) updates in $O\big((\log_B N)^2\big)$ I/Os. They also use it to dynamically maintain planar st-graphs using $O\big((1 + (b/B)(\log_m n)\log_b N\big)$ I/Os (amortized) per update, so that reachability queries can be answered in $O(\log_B N)$ I/Os (worst-case). It is open as to whether these results can be improved. One question is how to deal with non-monotone subdivisions. Another question is whether level-balanced B-trees can be implemented in $O(\log_B N)$ I/Os per update, so as to satisfy all three design criteria. Such an improvement would immediately give an optimal dynamic structure for reachability queries in planar st-graphs.

3.2 Buffer Trees

Many batched problems in computational geometry can be solved by plane sweep techniques. For example, to compute orthogonal segment intersections, we can keep maintain the vertical segments hit by a horizontal sweep line moving from top to bottom. If we use a B-tree to store the active vertical segments, each insertion and query will take $\Omega(\log_B N)$ I/Os, resulting in a huge I/O cost of $\Omega(N \log_B N)$, which can be more than B times larger than the desired bound of $O(n \log_m n)$. One solution suggested in [67] is to use a binary tree in which items are pushed lazily down the tree in blocks of B items at a time. The binary nature of the tree results in a data structure

of height $\sim \log n$, yielding a total I/O bound of $O(n \log n)$, which is still nonoptimal by a significant $\log m$ factor.

Arge [5] developed the elegant *buffer tree* data structure to support *batched dynamic* operations such as in the sweep line example, where the queries do not have to be answered right away or in any particular order. The buffer tree is a balanced multiway tree, but with degree $\Theta(m)$, except possibly for the root. Its key distinguishing feature is that each node has a buffer that can store M items (i.e., m blocks of items). Items in a node are not pushed down to the children until the buffer fills. Emptying the buffer requires $O(m)$ I/Os, which amortizes the cost of distributing the M items to the $\Theta(m)$ children. Each item incurs an amortized cost of $O(m/M) = O(1/B)$ I/Os per level. Queries and updates thus take $O\big((1/B)\log_m n\big)$ I/Os amortized. Buffer trees can be used as a subroutine in the standard sweep line algorithm in order to get an optimal EM algorithm for orthogonal segment intersection. Arge showed how to extend buffer trees to implement segment trees [15] in external memory in a batched dynamic setting by reducing the node degrees to $\Theta(\sqrt{m})$ and by introducing *multislabs* in each node, which we explain later in a different context.

Buffer trees have an ever-expanding list of applications. They provide, for example, a natural amortized implementation of priority queues for use in applications like discrete event simulation, sweeping, and list ranking. Brodal and Katajainen [17] provide a worst-case optimal priority queue, in the sense that every sequence of B insert and delete-min operations requires only $O(\log_m n)$ I/Os.

3.3 R-trees and Multidimensional Spatial Structures

The *R-tree* of Guttman [37] and its many variants are an elegant multidimensional generalization of the B-tree for storing a variety of geometric objects, such as points, segments, polygons, and polyhedra, using linear storage space. Internal nodes have degree $\Theta(B)$ (except possibly the root), and leaves store $\Theta(B)$ items. Each node in the tree has associated with it a bounding box (or bounding polygon) of all the elements in its subtree. A big difference between R-trees and B-trees is that in R-trees the bounding boxes of sibling nodes are allowed overlap. If an R-tree is being used for point location, for example, a point may lie within the bounding box of several children of the current node in the search. In that case the search must proceed to all such children.

Several heuristics for where to insert new items into an R-tree and how to rebalance it are surveyed in [4, 32, 34]. The methods perform well in many practical cases, especially in low dimensions, but they have poor worst-case bounds. An interesting open problem is whether nontrivial bounds can be proven for the "typical-case" behavior of R-trees for problems such as range searching and point location. Similar questions apply to the methods discussed in the previous section.

The *R*-tree* variant of Beckmann et al. [14] seems to give best overall query performance. Precomputing an R*-tree by repeated insertions, however, is extremely slow. A faster alternative is to use the Hilbert R-tree of Kamel and Faloutsos [41, 42]. Each item is labeled with the position of its center on the Hilbert space-filling curve, and a B-tree is built in a bottom-up manner on the totally ordered labels. Bulk loading a Hilbert R-tree is therefore easy once the center points are presorted, but the quality of the Hilbert R-tree in terms of query performance is not as good as that of an R*-tree, especially for higher-dimensional data [16, 43].

Arge et al. [6] and van den Bercken et al. [64] have independently devised fast bulk loading methods for R*-trees that are based upon buffer trees. Experiments indicate that the former method is especially efficient and can even support dynamic batched updates and queries.

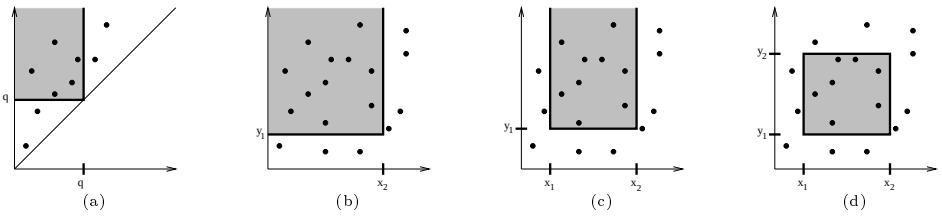


Fig. 2. Different types of 2-D orthogonal range queries: (a) Diagonal corner 2-sided query, (b) 2-sided query, (c) 3-sided query, (d) general 4-sided query.

Related linear-space multidimensional structures, which correspond to multiway versions of well-known internal memory structures like quad trees and kd-trees, include *grid files* [40, 48, 55], *kd-B-trees* [58], *buddy trees* [61], and *hB-trees* [28, 51]. We refer the reader to [4, 32, 56] for a broad survey of these and other interesting methods.

4 Online Multidimensional Range Searching

Multidimensional range search is a fundamental primitive in several online geometric applications, and it provides indexing support for new constraint data models and object-oriented data models. (See [44] for background.) For many types of range searching problems, it is very difficult to develop theoretically optimal algorithms that satisfy the three design criteria of Section 1.2. We have seen some linear-space online data structures in Section 3.3, but their query performance is not optimal. Many open problems remain.

We shall see in Section 4.3 for general 2-D orthogonal queries that it is not possible to satisfy criteria 1 and 2 simultaneously, for a fairly general computational model: At least $\Omega\big(n(\log n)/\log(\log_B N + 1)\big)$ disk blocks of space must be used to achieve a query bound of $O\big((\log_B N)^c + z\big)$ I/Os per query, for any constant c. A natural question is whether criterion 1 can be met if the disk space allowance is increased to $O\big(n(\log n)/\log(\log_B N + 1)\big)$ blocks. And since the lower bound applies only to general rectangular queries, it is natural to ask whether there are data structures that meet criteria 1–3 for interesting special cases of 2-D range searching, such as those pictured in Figure 2. Fortunately, the answers to both questions are "yes!", as we shall explore in the next section.

4.1 Data Structures for 2-D Orthogonal Range Searching

An obvious paradigm for developing an efficient EM data structure is to "externalize" an existing data structure that works well when the problem fits into internal memory. If the internal memory data structure uses a binary tree, then a multiway tree has to be used instead. However, it can be difficult when searching a B-tree to report the outputs in an output-sensitive manner. For example, for certain searching applications, each of the $\Theta(B)$ subtrees of a given node in a B-tree may contribute one item to the query output, which will require each subtree to be explored (costing several I/Os) just to report a single output item. Fortunately, the data structure can sometimes be augmented with a set of filtering substructures, each of which is a data structure for a smaller version of the same problem, in order to achieve output-sensitive reporting. We refer to this approach as the *bootstrapping* paradigm. Each substructure typically needs to store only $O(B^2)$ items and to answer queries in $O(\log_B B^2 + Z'/B) = O(\lceil Z'/B \rceil)$ I/Os, where Z' is the number of items reported. The substructure is allowed to be

static if it can be constructed in $O(B)$ I/Os, since we can keep updates in a separate buffer and do a global rebuilding in $O(B)$ I/Os when there are $\Theta(B)$ updates. Such a rebuilding costs $O(1)$ I/Os per update in the amortized sense, but the amortization for the substructures can often be removed and made worst-case by use of weight-balanced B-trees as the underlying B-tree structure.

Arge and Vitter [8] first uncovered the bootstrapping paradigm while designing an optimal dynamic EM data structure for diagonal corner 2-sided 2-D queries (see Figure 2(a)) that meets all three design criteria of Section 1.2. Diagonal corner 2-sided queries are equivalent to stabbing queries: Given a set of one-dimensional intervals, report all the intervals that contain the query value x. (Such intervals are said to be "stabbed" by x.) The global data structure is a multiway version of the well-known interval tree data structure [25, 26], which supports stabbing queries in $O(\log N + Z)$ CPU time and updates in $O(\log N)$ CPU time and uses $O(N)$ space. It is externalized by using a weight-balanced B-tree as the underlying base tree, where the nodes have degree $\Theta(\sqrt{B}\,)$ so that multislabs can be introduced. Each node in the base tree corresponds in a natural way to a one-dimensional range of x-values; its $\Theta(\sqrt{B}\,)$ children correspond to subranges called slabs, and the $\Theta(\sqrt{B}^2) = \Theta(B)$ contiguous sets of slabs are called *multislabs*.

Each inputed interval is stored in the lowest node v in the base tree whose range completely contains the interval. The interval is decomposed by v's slabs into at most three parts: the middle part that completely spans one or more slabs of v, the left end that partially protrudes into a slab w_{left}, and the right end that partially protrudes into a slab w_{right}. The three parts are stored in substructures of v: The middle part is stored in a list associated with the multislab it spans, the left part is stored in a list for w_{left} ordered by left endpoint, and the right part is stored in a list for w_{right} ordered by right endpoint.

Given a query value x, the intervals stabbed by x reside in the substructures of the nodes of the base tree along the search path for x. For each such node v, we consider each of v's multislabs that contains x and report all the intervals in its list. We also walk sequentially through the right-ordered list and left-ordered list for the slab of v that contains x, reporting intervals in an output-sensitive way.

The big problem with this approach is that we have to look at the list for each of v's multislabs that contains x, regardless of how many intervals are in the list. For example, there may be $\Theta(B)$ such multislab lists, but each list may contain only a few stabbed intervals (or worse yet, none at all!). The resulting query performance will be highly nonoptimal. The solution, according to the bootstrapping paradigm, is to use a substructure in each node consisting of an optimal static data structure for a smaller version of the same problem; a good choice is the corner data structure developed by Kanellakis et al. [44]. The corner substructure is used to store all the intervals from the "sparse" multislab lists, namely, those that contain fewer than B intervals, and thus the substructure contains only $O(B^2)$ intervals. When visiting node v, we access only v's non-sparse multislabs lists, each of which contributes $Z' \geq B$ intervals to the output, at an output-sensitive cost of $O(Z'/B)$ I/Os, for some Z'. The remaining Z'' stabbed intervals stored in v can be found by querying v's corner substructure of size $O(B^2)$, at a cost of $O(\lceil Z''/B \rceil)$ I/Os, which is output-sensitive. Since there are $O(\log_B N)$ nodes along the search path, the total collection of Z stabbed intervals are reported in a $O(\log_B N + z)$ I/Os, which is optimal. The use of a weight-balanced B-tree as the underlying base tree permits the rebuilding of the static substructures in worst-case optimal I/O bounds.

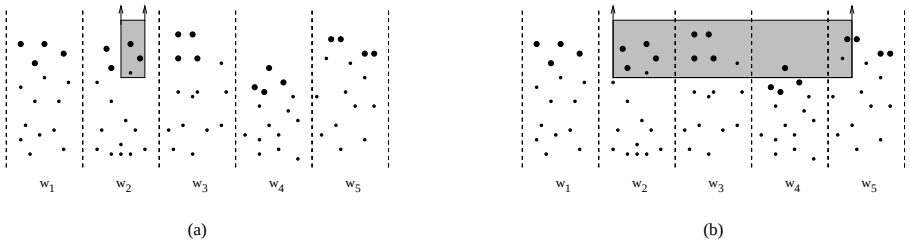


Fig. 3. Internal node v of the EM priority search tree, with slabs (children) w_1, w_2, ..., w_5. The Y-sets of each slab, which are stored collectively in v's substructure, are indicated by the bold points. (a) The 3-sided query is completely contained in the x-range of w_2. The relevant (bold) points are reported from v's substructure, and the query is recursively answered in w_2. (b) The 3-sided query spans several slabs. The relevant (bold) points are reported from v's substructure, and the query is recursively answered in w_2, w_3, and w_5. The query is *not* extended to w_4 in this case because not all of its Y-set $Y(w_4)$ (stored in v's substructure) satisfies the query, and as a result none of the points stored in w_4's subtree can satisfy the query.

Stabbing queries are important because, when combined with one-dimensional range queries, they provide a solution to *dynamic interval management*, in which one-dimensional intervals can be inserted and deleted, and intersection queries can be performed. These operations support indexing of one-dimensional constraints in constraint databases. Other applications of stabbing queries arise in graphics and GIS. For example, Chiang and Silva [23] apply the EM interval tree structure to extract at query time the boundary components of the isosurface (or contour) of a surface. A data structure for a related problem, which in addition has optimal output complexity, appears in [3]. The above bootstrapping approach also yields dynamic EM segment trees with optimal query and update bound and $O(n \log_B N)$-block space usage.

Arge et al. [7] provide another example of the bootstrapping paradigm by developing an optimal dynamic EM data structure for 3-sided 2-D range searching (see Figure 2(c)) that meets all three design criteria. The global structure is an externalization of the optimal structure for internal memory—the priority search tree [53]—using a weight-balanced B-tree as the underlying base tree. Each node in the base tree corresponds to a one-dimensional range of x-values, and its $\Theta(B)$ children correspond to subranges consisting of vertical slabs. Each node v contains a small substructure that supports 3-sided queries. Its substructure stores the "Y-set" $Y(w)$ for each of the $\Theta(B)$ slabs (children) w of v. The Y-set $Y(w)$ consists of the highest $\Theta(B)$ points in w's slab that are not already stored in an ancestor of v. Thus, there are a total of $\Theta(B^2)$ points stored in v's substructure.

A 3-sided query of the form $[x_1, x_2] \times [y_1, \infty)$ is answered by visiting a set of nodes in the base tree, starting with the root, and querying the substructure of each node. The following rule is used to determine which children of a visited node v should be visited: We visit v's child w if either

1. w is along the leftmost search path for x_1 or the rightmost search path for x_2 in the base tree, or
2. the entire Y-set $Y(w)$ is reported when v is visited.

(See Figure 3.) Rule 2 provides an effective filtering mechanism to guarantee output-sensitive reporting when Rule 1 is not satisfied: The I/O cost for initially accessing a

child node w can be charged to the $\Theta(B)$ points in $Y(w)$ reported from v's substructure; conversely, if not all of $Y(w)$ is reported, then the points stored in w's subtree will be too low to satisfy the query, and there is no need to visit w. (See Figure 3(b).)

Arge et al. [7] also provide an elegant and optimal static data structure for 3-sided range search, which can be used in the EM priority search tree described above to implement the substructures containing $O(B^2)$ points. The static structure is a persistent version of a data structure for one-dimensional range search. When used for $O(B^2)$ points, it occupies $O(B)$ blocks, can be built in $O(B)$ I/Os, and supports 3-sided queries in $O(\lceil Z'/B \rceil)$ I/Os per query, where Z' is the number of points reported. The static structure is so simple that it may be useful in practice on its own.

The dynamic data structure for 3-sided range searching can be generalized using the filtering technique of Chazelle [19] to handle general 4-sided queries with optimal query bound $O(\log_B N)$ and optimal disk space usage $O\big(n(\log n)/\log(\log_B N + 1)\big)$ [7]. The update bound becomes $O\big((\log_B N)(\log n)/\log(\log_B N + 1)\big)$. The outer level of the structure is a $(\log_B N + 1)$-way one-dimensional search tree; each 4-sided query is reduced to two 3-sided queries, a stabbing query, and $\log_B N$ list traversals.

Earlier work on 2-sided and 3-sided queries was done by Ramaswamy and Subramanian [57] using the notion of *path caching*; their structure met criterion 1 but had higher storage overheads and amortized and/or nonoptimal update bounds. Subramanian and Ramaswamy [63] subsequently developed the *p-range tree* data structure for 3-sided queries, with optimal linear disk space and nearly optimal query and amortized update bounds. They got a static data structure for 4-sided range search with the same query bound by applying the filtering technique of Chazelle [19]. The structure can be modified to perform updates, by use of a weight-balanced B-tree as the underlying base tree and the dynamization techniques of [7], but the resulting update bound will be amortized and nonoptimal, as a consequence of the use of their 3-sided data structure.

4.2 Other Range Searching Data Structures

For other types of range searching, such as in higher dimensions and for nonorthogonal queries, different filtering techniques are needed. So far, relatively little work has been done, and many open problems remain.

Vengroff and Vitter [66] develop the first theoretically near-optimal EM data structure for static three-dimensional orthogonal range searching. They create a hierarchical partitioning in which all the points that dominate a query point are densely contained in a set of blocks. Compression techniques are needed to minimize disk storage. With some recent modifications by the author, queries can be done in $O(\log_B N + z)$ I/Os, which is optimal, and the space usage is $O\big(n(\log n)^k/(\log(\log_B N + 1))^k\big)$ disk blocks to support $(3 + k)$-sided 3-D range queries, in which k of the dimensions ($0 \le k \le 3$) have finite ranges. The space bounds are optimal for 3-sided 3-D queries (i.e., $k = 0$) and 4-sided 3-D queries (i.e., $k = 1$). The result also provides optimal $O(\log N + Z)$-time query performance in the RAM model using linear space for answering 3-sided 3-D queries, improving upon the result in [21].

Agarwal et al. [2] consider halfspace range searching, in which a query is specified by a hyperplane and a bit indicating one of its two sides, and the output of the query consists of all the points on that side of the hyperplane. They give various data structures for halfspace range searching in two, three, and higher dimensions, including one that works for simplex (polygon) queries in two dimensions, but with a higher query I/O cost. They have subsequently improved the storage bounds to get an optimal static data structure satisfying criteria 1 and 2 for 2-D halfspace range queries.

The number of I/Os needed to build the data structures for 3-D orthogonal range search and halfspace range search is rather large (more than $\Omega(N)$). Still, the structures shed useful light on the complexity of range searching and may open the way to improved solutions. An open problem is to design efficient construction and update algorithms and to improve upon the constant factors.

Callahan et al. [18] develop dynamic EM data structures for several online problems such as finding an approximately nearest neighbor and maintaining the closest pair of vertices. Numerous other data structures have been developed for range queries and related problems on spatial data. We refer to [4, 32, 56] for a broad survey.

4.3 Lower Bounds for Orthogonal Range Searching

As mentioned above, Subramanian and Ramaswamy [63] prove that no EM data structure for 2-D range searching can achieve criterion 1 using less than $O\big(n(\log n)/\log(\log_B N + 1)\big)$ disk blocks, even if we relax 1 to allow $O\big((\log_B N)^c + z\big)$ I/Os per query, for any constant c. The result holds for an EM version of the pointer machine model, based upon the approach of Chazelle [20] for internal memory.

Hellerstein et al. [38] consider a generalization of the layout-based lower bound argument of Kanellakis et al. [44] for studying the tradeoff between disk space usage and query performance. They develop a model for *indexability*, in which an "efficient" data structure is expected to contain the Z output points to a query compactly within $O\big(\lceil Z/B \rceil\big) = O\big(\lceil z \rceil\big)$ blocks. One shortcoming of the model is that it considers only data layout and ignores the search component of queries, and thus it rules out the important filtering paradigm discussed earlier in Section 4. For example, it is reasonable for any query algorithm to perform at least $\log_B N$ I/Os, so if the output size Z is at most B, an algorithm may still be able to satisfy criterion 1 even if the output is contained within $O(\log_B N)$ blocks rather than $O(z) = O(1)$ blocks. Arge et al. [7] modify the model to rederive the same nonlinear space lower bound $O\big(n(\log n)/\log(\log_B N + 1)\big)$ of Subramanian and Ramaswamy [63] for 2-D range searching by considering only output sizes Z larger than $(\log_B N)^c B$, for which the number of blocks allowed to hold the outputs is $Z/B = O\big((\log_B N)^c + z\big)$. This approach ignores the complexity of how to find the relevant blocks, but as mentioned in Section 4.1 the authors separately provide an optimal 2-D range search data structure that uses the same amount of disk space and does queries in the optimal $O(\log_B N + z)$ I/Os. Thus, despite its shortcomings, the indexability model is elegant and can provide much insight into the complexity of blocking data in external memory. Further results in this model appear in [47, 60].

One intuition from the indexability model is that less disk space is needed to efficiently answer 2-D queries when the queries have bounded aspect ratio (i.e., when the ratio of the longest side length to the shortest side length of the query rectangle is bounded). An interesting question is whether R-trees and the linear-space structures of Section 3.3 can be shown to perform provably well for such queries. Another interesting scenario is where the queries correspond to snapshots of the continuous movement of a sliding rectangle.

When the data structure is restricted to contain only a single copy of each point, Kanth and Singh [45] show for a restricted class of index-based trees that d-dimensional range queries in the worst case require $\Omega(n^{1-1/d} + z)$ I/Os, and they provide a data structure with a matching bound. Another approach to achieve the same bound is the cross tree data structure of Grossi and Italiano [35], which in addition supports the operations of cut and concatenate.

5 Conclusions

In this paper we have surveyed several useful paradigms and techniques for the design and implementation of efficient online data structures for external memory. For lack of space, we didn't cover several interesting geometric search problems, such as point location, ray shooting queries, nearest neighbor queries, where most EM problems remain open, nor the rich areas of string processing and combinatorial graph problems. We refer the reader to [4, 31, 68] and the references therein.

A variety of interesting challenges remain in range searching, such as methods for high dimensions and nonorthogonal searches as well as the analysis of R-trees and linear-space methods for typical-case scenarios. Another problem is to prove lower bounds without the indivisibility assumption. A continuing goal is to translate theoretical gains into observable improvements in practice. For some of the problems that can be solved optimally up to a constant factor, the constant overhead is too large for the algorithm to be of practical use, and simpler approaches are needed.

Online issue also arise in the analysis of batched EM algorithms: In practice, batched algorithms must adapt in a robust and online way when the memory allocation changes, and online techniques can play an important role. Some initial work has been done on memory-adaptive EM algorithms in a competitive framework [11].

Acknowledgements. The author wishes to thank Lars Arge, Ricardo Baeza-Yates, Vasilis Samoladas, and the members of the Center for Geometric Computing at Duke University for helpful comments and suggestions.

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 ACM-SIAM Symposium on Discrete Algorithms*, 11–20, 1999.
2. P. K. Agarwal, L. Arge, J. Erickson, P. G. Franciosa, and J. S. Vitter. Efficient searching with linear constraints. In *Proc. 17th ACM Symposium on Principles of Database Systems*, 169–178, 1998.
3. 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 *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 117–126, 1998.
4. 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 23 of *Contemporary Mathematics*, 1–56. AMS Press, Providence, RI, 1999.
5. L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Proceedings of the Workshop on Algorithms and Data Structures*, volume 955 of *Lecture Notes in Computer Science*, 334–345. Springer-Verlag, 1995. A complete version appears as BRICS technical report RS–96–28, University of Aarhus.
6. L. Arge, K. H. Hinrichs, J. Vahrenhold, and J. S. Vitter. Efficient bulk operations on dynamic R-trees. In *Proceedings of the 1st Workshop on Algorithm Engineering and Experimentation*, Baltimore, January 1999.
7. L. Arge, V. Samoladas, and J. S. Vitter. Two-dimensional indexability and optimal range search indexing. In *Proceedings of the ACM Symposium Principles of Database Systems*, Philadelphia, PA, May–June 1999.
8. L. Arge and J. S. Vitter. Optimal dynamic interval management in external memory. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, 560–569, Burlington, VT, October 1996.
9. R. A. Baeza-Yates. Expected behaviour of B^+-trees under random insertions. *Acta Informatica*, 26(5), 439–472, 1989.
10. R. D. Barve, E. A. M. Shriver, P. B. Gibbons, B. K. Hillyer, Y. Matias, and J. S. Vitter. Modeling and optimizing I/O throughput of multiple disks on a bus: the long version. Technical report, Bell Labs, 1997.
11. R. D. Barve and J. S. Vitter. External memory algorithms with dynamically changing memory allocations: Long version. Technical Report CS–1998–09, Duke University, 1998.
12. R. Bayer and E. McCreight. Organization of large ordered indexes. *Acta Inform.*, 1, 173–189, 1972.

13. B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion B-tree. *The VLDB Journal*, 5(4), 264–275, December 1996.
14. N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the SIGMOD International Conference on Management of Data*, 322–331, 1990.
15. J. L. Bentley. Multidimensional divide and conquer. *Communications of the ACM*, 23(6), 214–229, 1980.
16. S. Berchtold, C. Böhm, and H.-P. Kriegel. Improving the query performance of high-dimensional index structures by bulk load operations. In *Proceedings of the International Conference on Extending Database Technology*, 1998.
17. G. S. Brodal and J. Katajainen. Worst-case efficient external-memory priority queues. In *Proceedings of the Scandinavian Workshop on Algorithms Theory*, volume 1432 of *Lecture Notes in Computer Science*, 107–118, Stockholm, Sweden, July 1998. Springer-Verlag.
18. P. Callahan, M. T. Goodrich, and K. Ramaiyer. Topology B-trees and their applications. In *Proceedings of the Workshop on Algorithms and Data Structures*, volume 955 of *Lecture Notes in Computer Science*, 381–392. Springer-Verlag, 1995.
19. B. Chazelle. Filtering search: a new approach to query-answering. *SIAM Journal on Computing*, 15, 703–724, 1986.
20. B. Chazelle. Lower bounds for orthogonal range searching: I. The reporting case. *Journal of the ACM*, 37(2), 200–212, April 1990.
21. B. Chazelle and H. Edelsbrunner. Linear space data structures for two types of range search. *Discrete & Computational Geometry*, 2, 113–126, 1987.
22. P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2), 145–185, June 1994.
23. 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*, Providence, RI, 1999. AMS Press.
24. D. Comer. The ubiquitous B-tree. *Comput. Surveys*, 11(2), 121–137, 1979.
25. H. Edelsbrunner. A new approach to rectangle intersections, part I. *Int. J. Computer Mathematics*, 13, 209–219, 1983.
26. H. Edelsbrunner. A new approach to rectangle intersections, part II. *Int. J. Computer Mathematics*, 13, 221–229, 1983.
27. R. J. Enbody and H. C. Du. Dynamic hashing schemes. *ACM Computing Surveys*, 20(2), 85–113, June 1988.
28. G. Evangelidis, D. B. Lomet, and B. Salzberg. The hB$^{\Pi}$-tree: A multi-attribute index supporting concurrency, recovery and node consolidation. *VLDB Journal*, 6, 1–25, 1997.
29. R. Fagin, J. Nievergelt, N. Pippinger, and H. R. Strong. Extendible hashing—a fast access method for dynamic files. *ACM Transactions on Database Systems*, 4(3), 315–344, 1979.
30. P. Flajolet. On the performance evaluation of extendible hashing and trie searching. *Acta Informatica*, 20(4), 345–369, 1983.
31. W. Frakes and R. Baeza-Yates, editors. *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, 1992.
32. V. Gaede and O. Günther. Multidimensional access methods. *Computing Surveys*, 30(2), 170–231, June 1998.
33. G. A. Gibson, J. S. Vitter, and J. Wilkes. Report of the working group on storage I/O issues in large-scale computing. *ACM Computing Surveys*, 28(4), 779–793, December 1996.
34. D. Greene. An implementation and performance analysis of spatial data access methods. In *Proceedings of the IEEE International Conference on Data Engineering*, 606–615, 1989.
35. R. Grossi and G. F. Italiano. Efficient cross-trees for external memory. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*. AMS Press, Providence, RI, 1999.
36. R. Grossi and G. F. Italiano. Efficient splitting and merging algorithms for order decomposable problems. *Information and Computation*, in press. An earlier version appears in *Proceedings of the 24th International Colloquium on Automata, Languages and Programming*, volume 1256 of Lecture Notes in Computer Science, Springer Verlag, 605–615, 1997.
37. A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, 47–57, 1985.
38. J. M. Hellerstein, E. Koutsoupias, and C. H. Papadimitriou. On the analysis of indexing schemes. In *Proceedings of the 16th ACM Symposium on Principles of Database Systems*, 249–256, Tucson, AZ, May 1997.
39. L. Hellerstein, G. Gibson, R. M. Karp, R. H. Katz, and D. A. Patterson. Coding techniques for handling failures in large disk arrays. *Algorithmica*, 12(2–3), 182–208, 1994.
40. K. H. Hinrichs. *The grid file system: Implementation and case studies of applications*. PhD thesis, Dept. Information Science, ETH, Zürich, 1985.
41. I. Kamel and C. Faloutsos. On packing R-trees. In *Proceedings of the 2nd International Conference on Information and Knowledge Management*, 490–499, 1993.
42. I. Kamel and C. Faloutsos. Hilbert R-tree: An improved R-tree using fractals. In *Proceedings of the 20th International Conference on Very Large Databases*, 500–509, 1994.

43. I. Kamel, M. Khalil, and V. Kouramajian. Bulk insertion in dynamic R-trees. In *Proceedings of the 4th International Symposium on Spatial Data Handling*, 3B, 31–42, 1996.
44. 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 Science*, 52(3), 589–612, 1996.
45. K. V. R. Kanth and A. K. Singh. Optimal dynamic range searching in non-replicating index structures. In *Proceedings of the 7th International Conference on Database Theory*, Jerusalem, January 1999.
46. D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading MA, second edition, 1998.
47. E. Koutsoupias and D. S. Taylor. Tight bounds for 2-dimensional indexing schemes. In *Proceedings of the 17th ACM Symposium on Principles of Database Systems*, Seattle, WA, June 1998.
48. R. Krishnamurthy and K.-Y. Wang. Multilevel grid files. Tech. report, IBM T. J. Watson Center, Yorktown Heights, NY, November 1985.
49. K. Küspert. Storage utilization in B*-trees with a generalized overflow technique. *Acta Informatica*, 19, 35–55, 1983.
50. W. Litwin. Linear hashing: A new tool for files and tables addressing. In *International Conference On Very Large Data Bases*, 212–223, Montreal, Quebec, Canada, October 1980.
51. D. B. 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.
52. D. B. Lomet and B. Salzberg. Concurrency and recovery for index trees. *The VLDB Journal*, 6(3), 224–240, 1997.
53. E. M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2), 257–276, May 1985.
54. H. Mendelson. Analysis of extendible hashing. *IEEE Transactions on Software Engineering*, SE–8, 611–619, November 1982.
55. J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Trans. Database Syst.*, 9, 38–71, 1984.
56. 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*, volume 1340 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.
57. S. Ramaswamy and S. Subramanian. Path caching: a technique for optimal external searching. *Proceedings of the 13th ACM Conference on Principles of Database Systems*, 1994.
58. J. T. Robinson. The k-d-b-tree: a search structure for large multidimensional dynamic indexes. In *Proc. ACM Conference Principles Database Systems*, 10–18, 1981.
59. C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 17–28, March 1994.
60. V. Samoladas and D. Miranker. A lower bound theorem for indexing schemes and its application to multidimensional range queries. In *Proc. 17th ACM Conf. on Princ. of Database Systems*, Seattle, WA, June 1998.
61. B. Seeger and H.-P. Kriegel. The buddy-tree: An efficient and robust access method for spatial data base systems. In *Proc. 16th VLDB Conference*, 590–601, 1990.
62. E. Shriver, A. Merchant, and J. Wilkes. An analytic behavior model for disk drives with readahead caches and request reordering. In *Joint International Conference on Measurement and Modeling of Computer Systems*, June 1998.
63. S. Subramanian and S. Ramaswamy. The P-range tree: a new data structure for range searching in secondary memory. *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 1995.
64. J. van den Bercken, B. Seeger, and P. Widmayer. A generic approach to bulk loading multidimensional index structures. In *Proceedings 23rd VLDB Conference*, 406–415, 1997.
65. P. J. Varman and R. M. Verma. An efficient multiversion access structure. *IEEE Transactions on Knowledge and Data Engineering*, 9(3), 391–409, May/June 1997.
66. D. E. Vengroff and J. S. Vitter. Efficient 3-d range searching in external memory. In *Proceedings of the ACM Symposium on Theory of Computation*, 192–201, Philadelphia, PA, May 1996.
67. J. S. Vitter. Efficient memory access in large-scale computation. In *Proceedings of the 1991 Symposium on Theoretical Aspects of Computer Science*, Lecture Notes in Computer Science. Springer-Verlag, 1991. Invited paper.
68. J. S. Vitter. External memory algorithms and data structures. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*. AMS Press, Providence, RI, 1999. An updated version is available via the author's web page `http://www.cs.duke.edu/~jsv/`.
69. J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2–3), 110–147, 1994.
70. D. Willard and G. Lueker. Adding range restriction capability to dynamic data structures. *Journal of the ACM*, 32(3), 597–617, 1985.
71. A. C. Yao. On random 2-3 trees. *Acta Informatica*, 9, 159–170, 1978.