DIMACS Series in Discrete Mathematics
and Theoretical Computer Science

External Memory Algorithms and Data Structures

Jeffrey Scott Vitter

Abstract. Data sets in large applications are often too massive to fit completely inside the computer's internal memory. The resulting input/output communication (or I/O) between fast internal memory and slower external memory (such as disks) can be a major performance bottleneck. In this paper, we survey the state of the art in the design and analysis of *external memory algorithms and data structures* (which are sometimes referred to as "EM" or "I/O" or "out-of-core" algorithms and data structures). EM algorithms and data structures are often designed and analyzed using the parallel disk model (PDM). The three machine-independent measures of performance in PDM are the number of I/O operations, the CPU time, and the amount of disk space. PDM allows for multiple disks (or disk arrays) and parallel CPUs, and it can be generalized to handle tertiary storage and hierarchical memory.

We discuss several important paradigms for how to solve batched and online problems efficiently in external memory. Programming tools and environments are available for simplifying the programming task. The TPIE system (Transparent Parallel I/O programming Environment) is both easy to use and efficient in terms of execution speed. We report on some experiments using TPIE in the domain of spatial databases. The newly developed EM algorithms and data structures that incorporate the paradigms we discuss are significantly faster than methods currently used in practice.

1. Introduction

1.1. Background. For reasons of economy, general-purpose computer systems usually contain a hierarchy of memory levels, each level with its own cost and performance characteristics. At the smallest scale, CPU registers and caches are built with the fastest but most expensive memory. For internal main memory, dynamic random access memory (DRAM) is typical. At a larger scale, inexpensive

---

1991 *Mathematics Subject Classification.* 68–02, 68Q10, 68Q20, 68Q25, 65Y20, 65Y25.

Key words and phrases. external memory, secondary storage, disk, block, input/output, I/O, out-of-core, hierarchical memory, multilevel memory, batched, online, external sorting.

Supported in part by Army Research Office MURI grant DAAH04–96–1–0013 and by National Science Foundation research grants CCR–9522047 and EIA–9870734. Part of this work was done at BRICS, University of Aarhus, Århus, Denmark and at INRIA, Sophia Antipolis, France.

Earlier versions of this paper, entitled "External Memory Algorithms", appeared as an invited tutorial in *Proceedings of the 17th Annual ACM Symposium on Principles of Database Systems*, Seattle, WA, June 1998, and as an invited paper in *Proceedings of the 6th Annual European Symposium on Algorithms*, Venice, August 1998. An updated version of this paper is available electronically on the author's web page at http://www.cs.duke.edu/~jsv/.

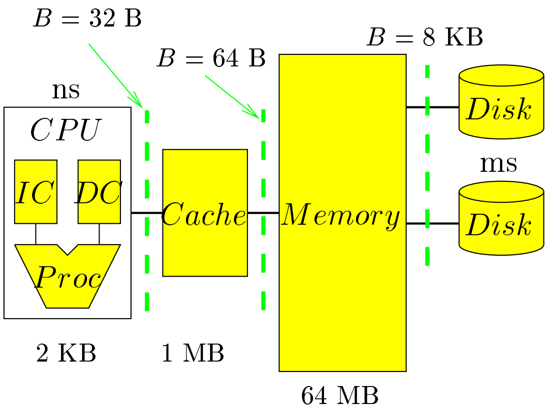


FIGURE 1. The memory hierarchy of a typical uniprocessor, including registers, instruction cache, data cache, level 2 cache, internal memory, and disk. The size of each memory level is indicated below it. The B parameter denotes the block transfer size between two adjacent levels of the hierarchy. All sizes are given in units of bytes (B), kilobytes (KB), or megabytes (MB). In this example, 8 KB is the block transfer size between internal memory and disk. (In the PDM model described in in Section 2, we measure B in units of items rather than in units of bytes. It is often more appropriate in batched applications to use a larger value of B than shown above.)

but slower magnetic disks are used for external mass storage, and even slower but larger-capacity devices such as tapes and optical disks are used for archival storage. Figure 1 depicts a typical memory hierarchy and its characteristics.

Most modern programming languages are based upon a programming model in which memory consists of one uniform address space. The notion of virtual memory allows the address space to be far larger than what can fit in the internal memory of the computer. A natural tendency for programmers is to assume that all memory references require the same access time. In many cases, such an assumption is reasonable (or at least doesn't do any harm), especially when the data sets are not large. The utility and elegance of this programming model are to a large extent why it has flourished, contributing to the productivity of the software industry.

However, not all memory references are created equal. Large address spaces span multiple levels of memory hierarchy, and accessing the data in the lowest levels of memory is orders of magnitude faster than accessing the data at the higher levels. For example, loading a register takes on the order of a nanosecond (10^{-9} seconds), whereas the latency of accessing data from a disk is several milliseconds (10^{-3} seconds). The relative difference in access time is more than one million! The *Input/Output* communication (or simply *I/O*) between levels of memory is often the bottleneck in applications that process massive amounts of data.

Many computer programs exhibit some degree of locality in their pattern of memory references: Certain data are referenced repeatedly for a while, and then the program shifts attention to other sets of data. Modern operating systems can take advantage of such access patterns by tracking the program's so-called "working set"—a vague notion that roughly corresponds to the recently referenced data items [**55**]. If the working set is small, it can be cached in high-speed memory so that access to it is fast. Caching and prefetching heuristics have been developed to reduce the number of occurrences of a "fault", in which the referenced data item is not in cache and must be retrieved by an I/O from a higher level of memory. For example, in a page fault, I/O is needed to retrieve a disk page from disk and bring it into internal memory.

By their nature, caching and prefetching methods are general-purpose, and thus they cannot be expected in all cases to take full advantage of the locality present in a computation. Some computations themselves are inherently non-local, and even with omniscient cache management decisions they are doomed to perform large amounts of I/O and suffer poor performance. Substantial gains in performance may be possible by incorporating locality directly into the algorithm design and by explicit management of the contents of each level of the memory hierarchy, thereby bypassing the virtual memory system.

We refer to algorithms and data structures that explicitly manage data placement and movement as *external memory* (or *EM*) *algorithms and data structures*. Some authors use the terms *I/O algorithms* or *out-of-core algorithms*. We concentrate in this paper on the I/O communication between the random access internal memory and the magnetic disk external memory, where the relative difference in access speeds is most apparent. We therefore use the term I/O to designate the communication between the internal memory and disk.

1.2. Overview of the Paper. In this paper we survey several paradigms for solving problems efficiently in external memory. The problems we consider fall into two general categories:

1. *Batched problems*, in which no preprocessing is done and the entire file of data items must be processed, often in stream mode with one or more passes over the data.
2. *Online problems*, in which computation is done in response to a continuous series of query operations. A common technique for online problems is to organize the data items via a hierarchical index, so that only a very small portion of the data needs to be examined in response to each query. The data being queried can be either *static*, which can be preprocessed for efficient query processing, or *dynamic*, where the queries are intermixed with updates such as insertions and deletions.

We base our approach on the *parallel disk model* (PDM) described in the next section. PDM provides an elegant and reasonably accurate model for analyzing the relative performance of EM algorithms and data structures. The three main performance measures of PDM are number of I/O operations, disk space usage, and CPU time. For reasons of brevity, we focus on the first two measures. Most of the algorithms we discuss are also efficient in terms of CPU time.

In Section 3, we look at the canonical batched EM problem of external sorting and the related problems of permuting and Fast Fourier Transform. The two important paradigms of distribution and merging account for all well-known external sorting algorithms. We provide fundamental lower bounds on the number of I/Os needed to perform sorting and several other batched problems in external memory.

We briefly discuss grid and linear algebra batched computations in Section 4. In Section 5 we mention several effective paradigms for batched EM problems in computational geometry. The paradigms include distribution sweep (for spatial join and finding all nearest neighbors), persistent B-trees (batched point location and graph drawing), batched filtering (for 3-D convex hulls and batched point location), external fractional cascading (for red-blue line segment intersection), online filtering (for cooperative search in fractionally cascaded data structures), external marriage-before-conquest (for output-sensitive convex hulls), and randomized incremental construction with gradations (for line segment intersections and other geometric problems). In Section 6 we look at EM algorithms for combinatorial problems on graphs. In many cases, I/O-efficient algorithms can be obtained by using sorting to simulate some well-known parallel algorithms.

In Sections 7 and 8 we consider spatial data structures in the online setting. Section 7 begins with a discussion of B-trees, the most important dynamic online EM data structure. B-trees are the method of choice for dictionary operations and one-dimensional range queries. Weight-balanced B-trees provide a uniform mechanism for dynamically rebuilding substructures, and level-balanced B-trees permit maintenance of parent pointers. They are useful for building interval trees and doing dynamic point location in external memory. The buffer tree is a so-called "batched dynamic" version of the B-tree for efficient implementation of search trees and priority queues in EM sweep line applications, We also consider multidimensional extensions of the B-tree. R-trees and variants work well in practice for several multidimensional spatial applications such as range searching and spatial joins. In Section 8, we discuss specialized spatial structures for online multidimensional range search, some of which yield optimal bounds for different cases of 2-D range searching. Nonlinear disk space is required in order to achieve optimal query performance for general 2-D range searching. In contrast, R-trees use linear space, but have bad worst-case performance.

Data structures for sorting, searching, and finding matches in strings are the focus of Section 9. In Section 10 we discuss programming environments and tools that facilitate high-level development of efficient EM algorithms. In Section 11, we demonstrate for two problems arising in spatial databases that significant speedups can be obtained in practice by use of efficient EM techniques. We use the TPIE system (Transparent Parallel I/O programming Environment) covered in Section 10 for the implementations. In Section 12 we discuss EM algorithms that adapt optimally to dynamically changing memory allocations. We conclude with some final remarks and observations in Section 13.

2. Parallel Disk Model (PDM)

External memory algorithms explicitly control data placement and movement, and thus it is important for algorithm designers to have a simple but reasonably accurate model of the memory system's characteristics. Magnetic disks consist of one or more rotating platters and one read/write head per platter surface. The data are stored in concentric circles on the platters called *tracks*, as shown in Figure 2. To read or write a data item at a certain address on disk, the read/write head must mechanically *seek* to the correct track and then wait for the desired address to pass by. The seek time to move from one random track to another is often on the order of 5–10 milliseconds, and the average rotational latency, which is the time for half a revolution, has the same order of magnitude. In order to amortize this delay, it pays to transfer a large collection of contiguous data items, called a *block*. Similar

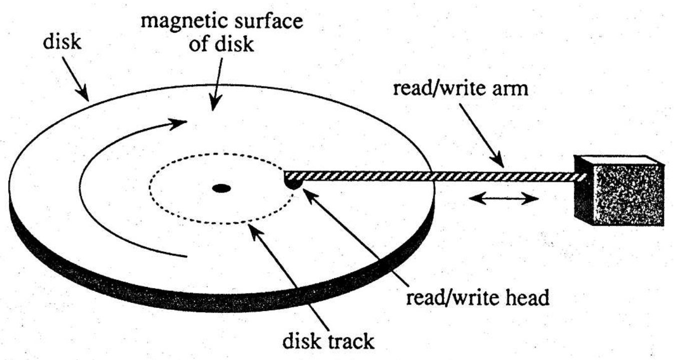


FIGURE 2. Platter of a magnetic disk drive.

considerations apply to all levels of the memory hierarchy. Typical block sizes are shown in Figure 1.

Even if an application can structure its pattern of memory accesses to exploit locality and take full advantage of disk block transfer, there is still a substantial *access gap* between internal memory performance and external memory performance. In fact the access gap is growing, since the speed of memory chips is increasing more quickly than disk bandwidth. Use of parallel processors further widens the gap. Storage systems such as RAID are being developed that deploy multiple disks to get additional bandwidth [**40, 76**].

In the next section, we describe the high-level parallel disk model (PDM), which we use in this paper for the design and analysis of algorithms and data structures. In Section 2.2, we list four fundamental I/O bounds that pertain to most of the problems considered in this paper. Practical considerations of PDM and alternative memory models are discussed in Sections 2.3 and 2.4.

2.1. PDM and Problem Parameters. 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 [**137**]:

$$\begin{aligned} N &= \text{problem size (in units of data items)}; \\ M &= \text{internal memory size (in units of data items)}; \\ B &= \text{block transfer size (in units of data items)}; \\ D &= \#\ \text{independent disk drives}; \\ P &= \#\ \text{CPUs}, \end{aligned}$$

where $M < N$, and $1 \leq DB \leq M/2$. In a single I/O, each of the D disks can simultaneously transfer a block of B contiguous data items. If $P \leq D$, each of the P processors can drive about D/P disks; if $D < P$, each disk is shared by about P/D processors. The internal memory size is M/P per processor, and the P processors are connected by an interconnection network. One desired property for the network is the capability to sort the M data items in the collective main memories of the processors in parallel in optimal $O\big((M/P)\log M\big)$ time.[1] The special cases of PDM for $P = 1$ and $P = D$ are pictured in Figure 3.

[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.

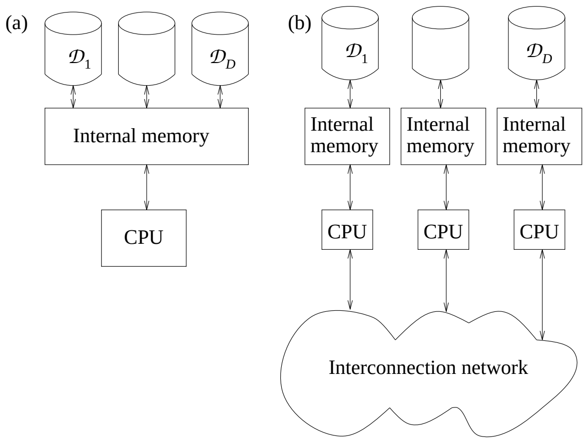


FIGURE 3. Parallel disk model: (a) $P = 1$, (b) $P = D$.

Queries are naturally associated with online computations, but they can also be done in batched mode. For example, in the batched orthogonal 2-D range searching problem discussed in Section 5, we are given a set of N points in the plane and a set of Q queries in the form of rectangles, and the problem is to report the points lying in each of the Q query rectangles. In both the batched and online settings, the number of items reported in response to each query may vary. We thus need to define two more performance parameters:

$$\begin{aligned} Q &= \text{\# queries (for a batched problem)};\\ Z &= \text{query output size (in units of data items)}. \end{aligned}$$

It is convenient to refer to some of the above PDM parameters in units of disk blocks rather than in units of data items. We define the lower-case notation

$$n = \frac{N}{B}, \qquad m = \frac{M}{B}, \qquad q = \frac{Q}{B}, \qquad z = \frac{Z}{B} \tag{2.1}$$

to be the problem input size, internal memory size, query specification size, and query output size, respectively, in units of disk blocks. We assume that the input data are initially "striped" across the D disks, in units of blocks, as illustrated in Figure 4, and we require the output data to be similarly striped. Striped format allows a file of N data items to be read or written in $O(N/DB) = O(n/D)$ I/Os, which is optimal.

	$\mathcal{D}_0$	$\mathcal{D}_1$	$\mathcal{D}_2$	$\mathcal{D}_3$	$\mathcal{D}_4$
stripe 0	0 1	2 3	4 5	6 7	8 9
stripe 1	10 11	12 13	14 15	16 17	18 19
stripe 2	20 21	22 23	24 25	26 27	28 29
stripe 3	30 31	32 33	34 35	36 37	38 39

FIGURE 4. Initial data layout on the disks, for $D = 5$ disks and block size $B = 2$. The input data items are initially striped block-by-block across the disks. For example, data items 16 and 17 are stored in the second block (i.e., in stripe 1) of disk $\mathcal{D}_3$.

2.2. Fundamental Bounds and Objectives. The primary measures of performance in PDM are

1. the number of I/O operations performed,
2. the amount of disk space used, and
3. the internal (parallel) computation time.

For reasons of brevity we will 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. Ideally algorithms and data structures should use linear space, which means $O(N/B) = O(n)$ disk blocks of storage.

The I/O performance of many algorithms and data structures can be expressed in terms of the bounds for the following four fundamental operations:

1. *Scanning* (or *streaming* or *touching*) a file of N data items, which takes $\Theta(N/DB) = \Theta(n/D)$ I/Os.
2. *Sorting* N items, which can be done in $\Theta\big((N/DB)\log_{M/B}(N/B)\big) = \Theta\big((n/D)\log_m n\big)$ I/Os.
3. *Online search* among N items, which takes $\Theta(\log_{DB} N)$ I/Os.
4. *Reporting the answers to a query* in blocked fashion onto external memory, which takes $\Theta\big(\lceil Z/DB\rceil\big) = \Theta\big(\lceil z/D\rceil\big)$ I/Os.

The first two of these I/O bounds—scanning and sorting—apply to batched problems. As mentioned earlier, some batched problems also involve queries, in which case the I/O bound for query reporting is relevant. The last two I/O bounds—online search and query reporting—apply to online problems. We typically assume for online problems that there is only one disk, namely, $D = 1$, and the bounds for online search and query reporting become simply $\Theta(\log_B N)$ and $\Theta(z)$; any discussion of the D-disk bounds is purely for theoretical completeness.[2]

For many of the batched problems we consider, such as sorting, FFTs, triangulation, and computing convex hulls, there are algorithms to solve the corresponding internal memory versions of the problems in $O(N \log N)$ CPU time. But if we deploy such an algorithm naively in an external memory setting (using virtual memory to handle page management), it may require $\Theta(N \log n)$ I/Os, which is excessive. Similarly, in the online setting, many problems can be solved in $O(\log N + Z)$ query time when they fit in internal memory, but the same data structure in an external memory setting may require $\Theta(\log N + Z)$ I/Os per query.

We would like instead to achieve the I/O bounds $O\big((n/D)\log_m n\big)$ in the batched example and $O(\log_{DB} N + z/D)$ for the online case. At the risk of oversimplifying, we can paraphrase the goal of EM algorithm design in the following syntactic way: to derive efficient algorithms so that the N and Z terms in the I/O bounds of the naive algorithms are replaced by n/D and z/D, and so that the base of the logarithm terms is not 2 but instead m (in the case of batched problems) or DB (in the case of online problems). The relative speedup in I/O performance can be very significant, both theoretically and in practice.

[2]The restriction to $D = 1$ for online problems is justified from a theoretical point of view because we can automatically get the optimal bounds for $D > 1$ by means of the disk striping technique applied to the single-disk data structure, as we explain in Section 3. Each conceptual disk block would be enlarged to have size DB, and it would be striped across the D disks, one normal block per disk. From a practical point of view, the improvement in online search time by using multiple disks would be minor, and the searches would require simultaneous access to all the disks, which may cause delays because of access conflicts with other processes.

For example, for batched problems, the I/O performance improvement can be a factor of $(N \log n)/(n/D)\log_m n = DB \log m$, which is very large, even for $D = 1$. For online problems, the performance improvement can be a factor of $(\log N + Z)/(\log_{DB} N + z/D)$, which is at least $(\log N)/\log_{DB} N = \log DB$, which is significant in practice, and it can be as much as $Z/(z/D) = DB$ for large Z.

The I/O bound $\Theta(N/DB) = \Theta(n/D)$ for the trivial batched problem of scanning is considered to be a *linear number of I/Os* in the PDM model. An interesting feature of the PDM model is that almost all nontrivial batched problems require a nonlinear number of I/Os, even those that can be solved easily in linear CPU time in the (internal memory) RAM model. Examples we will discuss later include permuting, transposing a matrix, and several combinatorial graph problems. Sorting is equivalent in I/O complexity to several of these problems.

Often in practice, the nonlinear $\log_m n$ term in the sorting bound and the $\log_{DB} N$ term in the searching bound are small constants. For example, in units of items, we could have $N = 10^{10}$, $M = 10^7$, $B = 10^4$, and $D = 1$, in which case we get $n = 10^6$, $m = 10^3$, and $\log_m n = 2$. If memory is shared with other processes, the $\log_m n$ term will be somewhat larger. In online applications, a smaller B value, such as $B = 10^2$, is more appropriate, as explained in the next section; the corresponding value of $\log_{DB} N$ for the example would be 5.

It still makes sense to identify terms like $\log_m n$ and $\log_{DB} N$ and not hide them within the big-oh factors, since the terms can make a significant difference in practice. (Of course, it is equally important to consider any other constants hidden in big-oh notations!) A nonlinear bound $O\big((n/D)\log_m n\big)$ usually indicates that multiple or extra passes over the data are required. In truly massive problems, the data will reside on tertiary storage. As we mention briefly in Section 2.4, PDM algorithms can often be generalized in a recursive framework to handle multiple levels of memory. A multilevel algorithm developed from a PDM algorithm that does n/D I/Os is likely to run at least an order of magnitude faster in hierarchical memory than a multilevel algorithm generated from a PDM algorithm that does $(n/D)\log_m n$ I/Os [**138**].

2.3. Practical Modeling Considerations. Track size is a parameter of the disk hardware and cannot be altered; for most disks it is in the range 50–100 KB.[3] For batched applications, the block transfer size B in PDM should be chosen to be a significant fraction of the track size or a small multiple of the track size, so as to better amortize seek time.[4] For online applications, a smaller B value is appropriate, such as in Figure 1.

[3]The actual track size for any given disk varies with the radius of the track. Sets of adjacent tracks are usually formatted to have the same track size, so there are typically a small number of different track sizes for a given disk.

[4]The minimum block transfer size imposed by hardware is often 512 bytes, but operating systems generally use a larger block size, such as 8 KB, as in Figure 1. It is possible to use an even larger block size and further reduce the relative significance of seek and rotational latency, but the wall clock time per I/O will increase accordingly. For example, if we set B to be five times larger than the track size, the time per I/O will correspond to five revolutions of the disk plus the relatively insignificant seek time. (If the disk is smart enough, rotational latency can be avoided, since the block spans entire tracks.) Once the block transfer size becomes larger than the track size, the wall clock time per I/O grows linearly with the block size. For best results in batched applications, the block transfer size B in PDM should be considered to be a fixed hardware parameter a little larger than the track size (on the order of 100 KB for most disks), and the time per I/O should be adjusted accordingly. The particular block size that optimizes performance in an actual implementation may vary somewhat from application to application.

PDM is a good generic programming model that facilitates elegant design of I/O-efficient algorithms, especially when used in conjunction with the programming tools discussed in Section 10. More complex and precise disk models have been developed, such as the ones by Ruemmler and Wilkes [**111**], Shriver et al. [**118**], Barve et al. [**27**], and Farach et al. [**60**]. They distinguish between sequential reads and random reads and consider the effects on throughput of features such as disk buffer caches and shared buses, which can reduce the time per I/O by eliminating or hiding the seek time. In practice, the effects of more complex models can be realized or approximated by PDM with an appropriate choice of parameters. The bottom line is that programs that perform well in terms of PDM will generally perform well when implemented on real systems.

2.4. Other Memory Models. The study of problem complexity and algorithm analysis when using external memory devices began more than 40 years ago with Demuth's Ph.D. thesis on sorting [**54, 88**]. In the early 1970s, Knuth [**88**] did an extensive study of sorting using magnetic tapes and (to a lesser extent) magnetic disks. At about the same time, Floyd [**64, 88**] considered a disk model akin to PDM for $D = 1, \quad P = 1, \quad B = M/2 = \Theta(N^c)$, for constant $c > 0$, and developed optimal upper and lower I/O bounds for sorting and matrix transposition. Hong and Kung [**78**] developed a pebbling model of I/O for straightline computations, and Savage and Vitter [**116**] extended the model to deal with block transfer. Aggarwal and Vitter [**10**] generalized Floyd's I/O model to allow simultaneous block transfers, but the model was unrealistic in that the simultaneous transfers were allowed to take place on a single disk. They developed matching upper and lower I/O bounds for all parameter values for a host of problems. Since the PDM model can be thought of as a more restrictive (and more realistic) version of Aggarwal and Vitter's model, their lower bounds apply as well to PDM. Modified versions of PDM that integrate various aspects of parallel computation are developed in [**53, 96, 121**]. Surveys of I/O models and algorithms appear in [**15, 119**].

The same type of bottleneck that occurs between internal memory and external disk storage can also occur at other levels of the memory hierarchy, such as between registers and data cache, between data cache and level 2 cache, between level 2 cache and DRAM, and between disk storage and tertiary devices. The PDM model can be generalized to model the hierarchy of memories ranging from registers at the small end to tertiary storage at the large end. Optimal algorithms for PDM often generalize in a recursive fashion to yield optimal algorithms in the hierarchical memory models. However, the match between theory and practice is harder to establish with hierarchical models; the simpler models are less practical, and the more practical models can be cumbersome to use.

For reasons of brevity and emphasis, we do not consider such hierarchical models in this paper. We refer the reader to the following references: Aggarwal et al. [**7**] define an elegant hierarchical memory model, and Aggarwal et al. [**8**] augment it with block transfer capability. Alpern et al. [**12**] model levels of memory in which the memory size, block size, and bandwidth grow at uniform rates. Vitter and Shriver [**138**] and Vitter and Nodine [**136**] discuss parallel versions and variants of the hierarchical models. The parallel model of Li et al. [**96**] also applies to hierarchical memory. Savage [**115**] gives a hierarchical pebbling version of [**116**]. Carter and Gatlin [**36**] define pebbling models of nonassociative direct-mapped caches.

3. External Sorting and Related Problems

The problem of *external sorting* (or sorting in external memory) is a central problem in the field of EM algorithms, partly because sorting and sorting-like operations account for a significant percentage of computer use [**88**], and also because sorting is an important paradigm in the design of efficient EM algorithms. With some technical qualifications, many problems that can be solved easily in linear time in internal memory, such as permuting, list ranking, expression tree evaluation, and finding connected components in a sparse graph, require the same number of I/Os in PDM as does sorting.

THEOREM 3.1 ([**10, 106**]). *The average-case and worst-case number of I/Os required for sorting N data items using D disks is*

$$\Theta\left(\frac{n}{D}\log_m n\right) = \Theta\left(\frac{n}{D}\frac{\log n}{\log m}\right). \tag{3.1}$$

It is conceptually much simpler to program for the single-disk case ($D = 1$) than for the multiple-disk case. *Disk striping* is a paradigm that can ease the programming task with multiple disks. I/Os are permitted only on entire stripes, one at a time. For example, in the data layout in Figure 4, data items 20–29 can be accessed in a single I/O step because their blocks are grouped into the same stripe. The net effect of striping is that the D disks behave as a single logical disk, but with a larger logical block size DB.

Let us consider what happens if we use the technique of disk striping in conjunction with an optimal sorting algorithm for one disk. The optimal number of I/Os using one disk is

$$\Theta\left(n\frac{\log n}{\log m}\right) = \Theta\left(\frac{N}{B}\frac{\log(N/B)}{\log(M/B)}\right). \tag{3.2}$$

The effect of disk striping with D disks is to replace B by DB in (3.2), which yields the I/O bound

$$\Theta\left(\frac{N}{DB}\frac{\log(N/DB)}{\log(M/DB)}\right) = \Theta\left(\frac{n}{D}\frac{\log(n/D)}{\log(m/D)}\right). \tag{3.3}$$

The striping I/O bound (3.3) is larger than the optimal bound (3.1) by a multiplicative factor of about $(\log m)/\log(m/D)$, which is significant when D is on the order of m, causing the $\log(m/D)$ term in the denominator to be very small. In order to attain the optimal sorting bound (3.1) theoretically, we must be able to control the disks independently, so that each disk can access a different stripe in the same I/O step. Sorting via disk striping is often more efficient in practice than more complicated techniques that utilize independent disks, since the $(\log m)/\log(m/D)$ factor may be dwarfed by the additional overhead of using the disks independently [**133**].

In Sections 3.1 and 3.2, we consider some recently developed external sorting algorithms based upon the *distribution* and *merge* paradigms. The SRM method, which uses a randomized merge technique, outperforms disk striping in practice for reasonable values of D (see Section 3.2). In Sections 3.3 and 3.4, we consider the related problems of permuting and Fast Fourier Transform. All the methods we cover, with the exception of Greed Sort in Section 3.2, access the disks independently during parallel read operations, but parallel writes are done in a striped manner, which facilitates the writing of parity error correction information. (We refer the reader to [**40, 76**] for a discussion of error correction issues.) In Section 3.5, we discuss some fundamental lower bounds on the number of I/Os needed to perform sorting and other batched problems in external memory.

3.1. Sorting by Distribution: Simultaneous Online Load Balancings. Distribution sort is a recursive process in which the data items to be sorted are partitioned by a set of $S-1$ partitioning elements into S buckets. All the items in one bucket precede all the items in the next bucket. The individual buckets are then sorted recursively and concatenated together to form a single totally sorted list.

The $S-1$ partitioning elements should be chosen so that the buckets are of roughly equal size. When that is the case, the bucket sizes decrease by a $\Theta(S)$ factor from one level of recursion to the next, and there are $O(\log_S n)$ levels of recursion. During each level of recursion, the data are streamed through internal memory, and the S buckets are written to the disks in an online manner as the streaming proceeds. A double buffer of size $2B$ is allocated to each of the S buckets. When one half of the double buffer fills, its block is written to disk in the next I/O, and the other half is used to store the incoming items. Therefore, the maximum number of buckets (and partitioning elements) is $S = \Theta(M/B) = \Theta(m)$, and the resulting number of levels of recursion is $\Theta(\log_m n)$.

It seems difficult to find $S = \Theta(m)$ partitioning elements using $\Theta(n/D)$ I/Os and guarantee that the bucket sizes are within a constant factor of one another. Efficient deterministic methods exist for choosing $S = \sqrt{m}$ partitioning elements [**105, 137**], which has the effect of doubling the number of levels of recursion. Probabilistic methods based upon random sampling can be found in [**61**].

In order to meet the sorting bound (3.1), the formation of the buckets at each level of recursion must be done in $O(n/D)$ I/Os, which is easy to do for the single-disk case. In the more general multiple-disk case, each read step and each write step during the bucket formation must involve on the average $\Theta(D)$ blocks. The file of items being partitioned was itself one of the buckets formed in the previous level of recursion. In order to read that file efficiently, its blocks must be spread uniformly among the disks, so that no one disk is a bottleneck. The challenge in distribution sort is to write the blocks of the buckets to the disks in an online manner and achieve a global load balance by the end of the partitioning, so that the bucket can be read efficiently during the next level of the recursion.

Partial striping is an effective technique for reducing the amount of information that must be stored in internal memory in order to manage the disks. The disks are grouped into clusters of size C and data are written in "logical blocks" of size CB, one per cluster. Choosing $C = \sqrt{D}$ won't change the optimal sorting time by more than a constant factor, but as pointed out earlier, full striping (in which $C = D$) can be nonoptimal.

Vitter and Shriver [**137**] use two randomized online techniques during the partitioning so that with high probability each bucket is well balanced across the D disks. (Partial striping is used so that the pointers needed to keep track of the layout of the buckets on the disks can fit in internal memory.) The first technique is used when the size N of the file to partition is sufficiently large or when $M/DB = \Omega(\log D)$, so that the number $\Theta(n/S)$ of blocks in each bucket is $\Omega(D \log D)$. Each parallel write operation writes its D blocks in random order to a disk stripe, with all $D!$ orders equally likely. At the end of the partitioning, with high probability each block is evenly distributed among the disks. This situation is analogous to a hashing scenario in which the number of inserted items is larger by at least a logarithmic factor than the number of bins in the hash table, thereby causing items to be spread fairly evenly, so that the expected maximum bin size is within a constant factor of the expected bin size [**135**].

If the number of blocks per bucket is not $\Omega(D \log D)$, however, the technique breaks down and the distribution of each bucket among the disks tends to be uneven. For these smaller values of N, Vitter and Shriver use a different technique: In one pass, the file is read, one memoryload at a time. Each memoryload is randomly permuted and written back to the disks in the new order. In a second pass, the file is accessed one memoryload at a time in a "diagonally striped" manner. They show that with very high probability each individual "diagonal stripe" contributes about the same number of items to each bucket, so the blocks of the buckets in each memoryload can be assigned to the disks in a balanced round robin manner using an optimal number of I/Os.

An even better way to do distribution sort, and deterministically at that, is the BalanceSort method developed by Nodine and Vitter [**105**]. During the partitioning process, the algorithm keeps track of how evenly each bucket has been distributed so far among the disks. For each $1 \leq b \leq S$ and $1 \leq d \leq D$, let num_b be the total number of items in bucket b processed so far during the partitioning and let $num_b(d)$ be the number of those items written to disk d; that is, $num_b = \sum_{1 \leq d \leq D} num_b(d)$. The algorithm is able to write at least half of any given memoryload to the disks and still maintain the invariant for each bucket b that the $\lfloor D/2 \rfloor$ largest values of $num_b(1)$, $num_b(2)$, ..., $num_b(D)$ differ by at most 1, and hence each $num_b(d)$ is at most about twice the ideal value num_b/D.

An alternative sorting technique, with higher overhead, is to use the buffer tree data structure [**13**] described in Section 7.2, which was developed for batched dynamic applications.

DeWitt et al. [**56**] present a randomized distribution sort algorithm in a similar model to handle the case when sorting can be done in two passes. They use a sampling technique to find the partitioning elements and route the items in each bucket to a particular processor. The buckets are sorted individually in the second pass.

Matias et al. [**99**] develop optimal in-place distribution sort algorithms for one disk as a function of the number K of distinct key values. The corresponding I/O bound is $O\bigl(n \log_m \min\{K, n\}\bigr)$. Their technique can be extended within the same I/O bounds to merge sort.

Distribution sort algorithms may have an advantage over the merge approaches presented in the next section in that they typically make better use of lower levels of cache in the memory hierarchy of real systems. Such an intuition comes from analysis of distribution sort and merge sort algorithms on models of hierarchical memory, such as the RUMH model of Vitter and Nodine [**136**].

3.2. Sorting by Merging. The merge paradigm is somewhat orthogonal to the distribution paradigm discussed in Section 3.1. A typical merge sort algorithm works as follows: In the "run formation" phase, the n blocks of data are streamed into memory, one memoryload at a time; each memoryload is sorted into a single "run", which is then output to stripes on disk. At the end of the run formation phase, there are $N/M = n/m$ (sorted) runs, each striped across the disks. (In actual implementations, the "replacement-selection" technique can be used to get runs of $2M$ data items, on the average, when $M \gg B$ [**88**].)

After the initial runs are formed, the merging phase begins. In each pass of the merging phase, groups of R runs are merged together. During each merge, one block from each run resides in internal memory. When the data items of a block expire, the next block for that run is input. Double buffering is used to keep the disks busy. Hence, at most $R = \Theta(m)$ runs can be merged at a time; the resulting number of passes is $O(\log_m n)$.

To achieve the optimal sorting bound (3.1), each merging pass must be done in $O(n/D)$ I/Os, which is easy to do for the single-disk case. In the more general multiple-disk case, each parallel read operation during the merging must on the average bring in the next $\Theta(D)$ blocks needed for the merging. The challenge is to ensure that those blocks reside on different disks so that they can be read in a single I/O (or a small constant number of I/Os). The difficulty lies in the fact that the runs being merged were themselves formed during the previous merge pass. Their blocks were written to the disks in the previous pass without knowledge of how they would interact with other runs in later merges.

A perfect solution, in which the next D blocks needed for the merge are guaranteed to be on distinct disks, can be devised for the binary merging case $R = 2$ based upon the Gilbreath principle [**67, 88**]: The first run is striped in ascending order by disk number, and the other run is striped in descending order. Regardless of how the items in the two runs interleave during the merge, it is always the case that the next D blocks needed for the output can be accessed via a single I/O operation, and thus the amount of internal memory buffer space needed for binary merging can be kept to a minimum. Unfortunately there is no analog to the Gilbreath principle for $R > 2$, and as we have seen above, we need the value of R to be large in order to get an optimal sorting algorithm.

The Greed Sort method of Nodine and Vitter [**106**] was the first optimal deterministic EM algorithm for sorting with multiple disks. It handles the case $R > 2$ by relaxing the condition on the merging process. In each step, the following two blocks from each disk are brought into internal memory: the block b_1 with the smallest data item value and the block b_2 whose largest item value is smallest. If $b_1 = b_2$, only one block is read into memory, and it is added to the next output stripe. Otherwise, the two blocks b_1 and b_2 are merged in memory; the smaller B items are written to the output stripe, and the remaining items are written back to the disk. The resulting run that is produced is only an "approximately" merged run, but its saving grace is that no two inverted items are too far apart. A final application of Columnsort [**94**] in conjunction with partial striping suffices to restore total order.

An optimal deterministic merge sort, with somewhat higher constant factors than those of the distribution sort algorithms, was developed by Aggarwal and Plaxton [**9**], based upon the Sharesort hypercube sorting algorithm [**52**]. To guarantee even distribution during the merging, it employs two high-level merging schemes in which the scheduling is almost oblivious.

The most practical method for sorting is the simple randomized merge sort (SRM) algorithm of Barve et al. [**26**] (referred to as "randomized striping" by Knuth [**88**]). Each run is striped across the disks, but with a random starting point (the only place in the algorithm where randomness is utilized). During the merging process, the next block needed from each disk is read into memory, and if there is not

	$D = 5$	$D = 10$	$D = 50$
$k = 5$	0.56	0.47	0.37
$k = 10$	0.61	0.52	0.40
$k = 50$	0.71	0.63	0.51

TABLE 1. The ratio of the number of I/Os used by simple randomized merge sort (SRM) to the number of I/Os used by merge sort with disk striping, during a merge of kD runs. The figures were obtained by simulation; they back up the (more pessimistic) analytic upper bound in [**26**].

enough room, the least needed blocks are "flushed" (without any I/Os required) to free up space. The expected performance of SRM is not optimal for some parameter values, but it significantly outperforms the use of disk striping for reasonable values of the parameters, as shown in Table 1. Barve et al. [**26**] derive an upper bound on the I/O performance; the precise analysis is an interesting open problem [**88**]. Work is beginning on applying the SRM buffer management techniques to distribution sort. The hope is to get better overall sorting performance by means of improved cache utilization, based upon the intuition mentioned at the end of the previous section.

3.3. Permuting and Transposition. Permuting is the special case of sorting in which the key values of the N data items form a permutation of $\{1, 2, \ldots, N\}$.

THEOREM 3.2 ([**10**]). *The average-case and worst-case number of I/Os required for permuting N data items using D disks is*

$$\Theta\left(\min\left\{\frac{N}{D}, \frac{n}{D}\log_m n\right\}\right). \tag{3.4}$$

The I/O bound (3.4) for permuting can be realized by using one of the sorting algorithms from Section 3 except in the extreme case $B \log m = o(\log n)$, in which case it is faster to move the data items one by one in a non-blocked way. The one-by-one method is trivial if $D = 1$, but with multiple disks there may be bottlenecks on individual disks; one solution for doing the permuting in $O(N/D)$ I/Os is to apply the randomized balancing strategies of [**137**].

Matrix transposition is the special case of permuting in which the permutation can be represented as a transposition of a matrix from row-major order into column-major order.

THEOREM 3.3 ([**10**]). *The number of I/Os required using D disks to transpose a $p \times q$ matrix from row-major order to column-major order is*

$$\Theta\left(\frac{n}{D}\log_m \min\{M, p, q, n\}\right), \tag{3.5}$$

where $N = pq$ and $n = N/B$.

When B is large compared with M, matrix transposition can be as hard as general sorting, but for smaller B, the special structure of the transposition permutation makes transposition easier. In particular, the matrix can be broken up into square submatrices of B^2 elements such that each submatrix contains B blocks of the matrix in row-major order and also B blocks of the matrix in column-major order. Thus, if $B^2 < M$, the transpositions can be done in a simple one-pass operation by transposing the submatrices one-at-a-time in internal memory.

Matrix transposition is a special case of a more general class of permutations called *bit-permute/complement* (BPC) permutations, which in turn is a subset of the class of *bit-matrix-multiply/complement* (BMMC) permutations. BMMC permutations are defined by a $\log N \times \log N$ nonsingular 0-1 matrix A and a $(\log N)$-length 0-1 vector c. An item with binary address x is mapped by the permutation to the binary address given by $Ax \oplus c$. BPC permutations are the special case of BMMC permutations in which A is a permutation matrix, that is, each row and each column of A contain a single 1. BPC permutations include matrix transposition, bit-reversal permutations (which arise in the FFT), vector-reversal permutations, hypercube permutations, and matrix reblocking. Cormen et al. [**49**] characterize the optimal number of I/Os needed to perform any given BMMC permutation solely as a function of the associated matrix A, and they give an optimal algorithm for implementing it.

THEOREM 3.4 ([**49**]). *The number of I/Os required using D disks to perform the BMMC permutation defined by matrix A and vector c is*

$$\Theta\left(\frac{n}{D}\left(1+\frac{\operatorname{rank}(\gamma)}{\log m}\right)\right), \tag{3.6}$$

where γ is the lower-left $\log n \times \log B$ submatrix of A.

An interesting theoretical question is whether there is a simple characterization (as a function of the input) of the I/O cost for a general permutation.

3.4. Fast Fourier Transform. Computing the Fast Fourier Transform (FFT) in external memory consists of a series of I/Os that permit each computation implied by the FFT directed graph (or butterfly) to be done while its arguments are in internal memory. A permutation network computation consists of a fixed pattern of I/Os such that any of the $N!$ possible permutations can be realized; data items can only be reordered when they are in internal memory. A permutation network can be realized by a series of three FFTs [**144**].

THEOREM 3.5. *With D disks, the number of I/Os required for computing the N-input FFT digraph or an N-input permutation network is given by the same bound* (3.1) *as for sorting.*

Cormen and Nicol [**48**] give some practical implementations for one-dimensional FFTs based upon the optimal PDM algorithm of [**137**]. The algorithms for FFT are faster and simpler than for sorting because the computation is nonadaptive in nature, and thus the communication pattern is oblivious.

3.5. Lower Bounds on I/O. In this section we prove the lower bounds from Theorems 3.1–3.5 and mention some related I/O lower bounds for batched problems in computational geometry and graphs.

The most trivial batched problem is that of *scanning* (or *streaming* or *touching*) a file of N data items, which can be done in a linear number $O(N/DB) = O(n/D)$ of I/Os. Permuting is one of several simple problems that can be done in linear CPU time in the (internal memory) RAM model, but require a nonlinear number of I/Os in PDM because of the locality constraints imposed by the block parameter B.

The following proof of the permutation lower bound (3.4) of Theorem 3.2 is due to Aggarwal and Vitter [**10**]. The idea of the proof is to measure, for each $t \geq 0$, the number of distinct orderings that are realizable by at least one sequence of t I/Os. The value of t for which the number of distinct orderings first exceeds $N!/2$ is a lower bound on the average number of I/Os (and hence the worst-case number of I/Os) needed for permuting.

We assume for the moment that there is only one disk, $D = 1$. Let us consider how the number of realizable orderings can change when we read a given disk block into internal memory. There are at most B data items in the block, and they can intersperse among the M items in internal memory in at most $\binom{M}{B}$ ways, so the number of realizable orderings increases by a factor of $\binom{M}{B}$. If the block has never before resided in internal memory, the number of realizable orderings increases by an extra $B!$ factor, since the items in the block can be permuted among themselves. (This extra contribution of $B!$ can only happen once for each of the N/B original blocks.) The effect of writing the disk block is considerably less than that of reading it. There are at most $n + t \leq N \log N$ ways to choose which disk block is involved in the I/O. (We allow the algorithm to use an arbitrary amount of disk space.) Hence, the number of distinct orderings that can be realized by some sequence of t I/Os is at most

$$(B!)^{N/B} \left(N(\log N) \binom{M}{B} \right)^t . \tag{3.7}$$

Setting the expression in (3.7) to be at least $N!/2$, and simplifying by taking the logarithm, we get

$$N \log B + t \left(\log N + B \log \frac{M}{B} \right) = \Omega(N \log N). \tag{3.8}$$

We get the lower bound for the case $D = 1$ by solving for t. The general lower bound (3.4) follows by dividing by D.

Permuting is a special case of sorting, and hence, the permuting lower bound applies also to sorting. In the unlikely case that $B \log m = o(\log n)$, the permuting bound is only $\Omega(N/D)$, and we must resort to the comparison model to get the full lower bound (3.1) of Theorem 3.1 [**10**]. Arge et al. [**18**] show for the comparison model that any problem with an $\Omega(N \log N)$ lower bound in the RAM model requires $\Omega(n \log_m n)$ I/Os in PDM. However, in the typical case in which $B \log m = \Omega(\log n)$, the comparison model is not needed to prove the sorting lower bound; the difficulty of sorting in that case arises not from determining the order of the data but from permuting (or routing) the data.

The proof used above for permuting also works for permutation networks, in which the communication pattern is oblivious. Since the choice of disk block is fixed for each t, there is no $N \log N$ term as there is in (3.7), and correspondingly there is no additive $\log N$ term in the inner expression as there is in (3.8). Hence, when we solve for t, we get the lower bound (3.1) rather than (3.4). The lower bound follows directly from the counting argument; unlike the sorting derivation, it does not require the comparison model for the case $B \log m = o(\log n)$. The lower bound also applies directly to FFTs, since permutation networks can be formed from three FFTs in sequence. The transposition lower bound involves a potential argument based upon a togetherness relation [**10**]. A related argument demonstrates the optimality of the algorithm in [**99**] for sorting N items with K distinct key values.

Chiang et al. [**42**], Arge [**14**], Arge and Miltersen [**19**], and Kameshwar and Ranade [**83**] give models and lower bound reductions for several computational geometry and graph problems. Problems like list ranking and expression tree evaluation have the same nonlinear I/O lower bound as permuting. Other problems like connected components, biconnected components, and minimum spanning trees of sparse graphs with E edges and V vertices require as many I/Os as E/V instances of sorting V items. This situation is in contrast with the RAM model, in which the same problems can all be done in linear CPU time. (The known linear-time RAM algorithm for minimum spanning tree is randomized.) In some cases, there is a gap between the best known upper and lower bounds, which we discuss further in Section 6. The geometry problems discussed in Section 5 are equivalent to sorting in both the internal memory and PDM models.

The lower bounds mentioned above assume that the data items are in some sense "indivisible", in that they are not split up and reassembled in some magic way to get the desired output. It is conjectured that the sorting lower bound (3.1) remains valid even if the indivisibility assumption is lifted. However, for an artificial problem related to transposition, Adler [**2**] showed that removing the indivisibility assumption can lead to faster algorithms. A similar result is shown by Arge and Miltersen [**19**] for the decision problem of determining if N data item values are distinct. Whether or not the conjecture is true is a challenging theoretical problem.

4. Matrix and Grid Computations

Dense matrices are generally represented in memory in row-major or column-major order. Matrix transposition, which is the special case of sorting that involves conversion of a matrix from one representation to the other, was discussed in Section 3.3. For certain operations such as matrix addition, both representations work well. However, for standard matrix multiplication (using only semiring operations) and LU decomposition, a better representation is to block the matrix into square $\sqrt{B} \times \sqrt{B}$ submatrices, which gives the upper bound of the following theorem:

THEOREM 4.1 ([**78, 116, 137, 143**]). *The number of I/Os required for standard matrix multiplication of two $k \times k$ matrices or to compute the LU factorization of a $k \times k$ matrix is* $\Theta\big(k^3 / \min\{k, \sqrt{M}\,\} DB\big)$.

Hong and Kung [**78**] and Nodine et al. [**104**] give optimal EM algorithms for iterative grid computations, and Leiserson et al. [**95**] reduce the number of I/Os of naive multigrid implementations by a $\Theta(M^{1/5})$ factor. Gupta et al. [**73**] show how to derive efficient EM algorithms automatically for computations expressed in tensor form.

If a $k \times k$ matrix A is sparse, that is, if the number N_z of nonzero elements in A is much smaller than k^2, then it may be more efficient to store only the nonzero elements. Each nonzero element $A_{i,j}$ is represented by the triple $(i, j, A_{i,j})$. Unlike the dense case, in which transposition can be easier than sorting (e.g., see Theorem 3.3 when $B^2 \le M$), transposition of sparse matrices is as hard as sorting:

THEOREM 4.2. *For a matrix stored in sparse format and containing $N_z = n_z B$ nonzero elements, the number of I/Os required to convert the matrix from row-major order to column-major order, and vice-versa, is*

$$\Theta\left(\frac{n_z}{D} \log_m n_z\right). \tag{4.1}$$

The lower bound follows by reduction from sorting. If the ith item in the input of the sorting instance has key value $x \neq 0$, there is a nonzero element in matrix position (i, x).

We defer further discussion of numerical EM algorithms and refer the reader to Toledo's survey in this volume [**126**]. Some issues regarding programming environments are discussed in [**47**] and Section 10.

5. Batched Problems in Computational Geometry[5]

Problems involving massive amounts of geometric data are ubiquitous in spatial databases [**93, 112, 113**], geographic information systems (GIS) [**93, 112, 129**], constraint logic programming [**84, 85**], object-oriented databases [**146**], statistics, virtual reality systems, and computer graphics [**65**]. NASA's Earth Observing System project, the core part of the Earth Science Enterprise (formerly Mission to Planet Earth), produces petabytes (10^{15} bytes) of raster data per year [**58**]! Microsoft's TerraServer online database of satellite images is over one terabyte in size [**124**]. A major challenge is to develop mechanisms for processing the data, or else much of it will be useless.

For systems of this size to be efficient, we need fast EM algorithms and data structures for basic problems in computational geometry. Luckily, many problems on geometric objects can be reduced to a small core of problems, such as computing intersections, convex hulls, or nearest neighbors. Useful paradigms have been developed for solving these problems in external memory.

THEOREM 5.1. *The following batched problems and several related problems involving N input items, Q queries, and Z output items can be solved using*

$$O\big((n+q)\log_m n + z\big) \tag{5.1}$$

I/Os (where Q and Z are set to 0 if they are not relevant for the particular problem):

1. *Computing the pairwise intersections of N orthogonal segments in the plane,*
2. *Answering Q orthogonal 2-D range queries on N points in the plane (i.e., finding all the points within the Q query rectangles),*
3. *Computing the pairwise intersections of N segments in the plane,*
4. *Finding all intersections between N nonintersecting red line segments and N nonintersecting blue line segments in the plane.*
5. *Constructing the 2-D and 3-D convex hull of N points,*
6. *Voronoi diagram and Triangulation of N points in the plane,*
7. *Performing Q point location queries in a planar subdivision of size N,*
8. *Finding all nearest neighbors for a set of N points in the plane,*
9. *Finding the pairwise intersections of N orthogonal rectangles in the plane,*
10. *Computing the measure of the union of N orthogonal rectangles in the plane,*
11. *Computing the visibility of N segments in the plane from a point,*
12. *Performing Q ray-shooting queries in 2-D Constructive Solid Geometry (CSG) models of size N,*

[5] For brevity, in the remainder of this paper we deal only with the single-disk case $D = 1$. The single-disk I/O bounds for the batched problems can often be cut by a factor of $\Theta(D)$ for the case $D > 1$ by using the load balancing techniques of Section 3. In practice, disk striping may be sufficient. For online problems, disk striping will convert optimal bounds for the case $D = 1$ into optimal bounds for $D > 1$.

Goodrich et al. [**69**], Zhu [**148**], Arge et al. [**23**], Arge et al. [**21**], and Crauser et al. [**50, 51**] develop EM algorithms for those problems using the following EM paradigms for batched problems:

Distribution sweeping: a generalization of the distribution paradigm of Section 3 for externalizing plane sweep algorithms;

Persistent B-trees: an offline method for constructing an optimal-space persistent version of the B-tree data structure (see Section 7.1), yielding a factor of B improvement over the generic persistence techniques of Driscoll et al. [**57**].

Batched filtering: a general method for performing simultaneous external memory searches in data structures that can be modeled as planar layered directed acyclic graphs and in external fractionally cascaded data structures; it is useful for 3-D convex hulls and batched point location.

External fractional cascading: an EM analog to fractional cascading on a segment tree.

Online filtering: a technique based upon the work of Tamassia and Vitter [**123**] for online queries in data structures with fractional cascading.

External marriage-before-conquest: an EM analog to the well-known technique of Kirkpatrick and Seidel [**87**] for performing output-sensitive convex hull constructions.

Randomized incremental construction with gradations: a localized version of the incremental construction paradigm of Clarkson and Shor [**45**].

The distribution sweep paradigm is fundamental to sweep line processes. For example, we can compute the pairwise intersections of N orthogonal segments in the plane by the following recursive distribution sweep: At each level of recursion, the plane is partitioned into $\Theta(m)$ vertical strips, each containing $\Theta(N/m)$ of the segments' endpoints. We sweep a horizontal line from top to bottom to process the N segments. When a vertical segment is encountered by the sweep line, the segment is inserted into the appropriate strip. When a horizontal segment h is encountered by the sweep line, we report h's intersections with all the "active" vertical segments in the strips that are spanned *completely* by h. (A vertical segment is "active" if it is intersected by the current sweep line; vertical segments that are found to be no longer active are deleted from the strips.) The remaining end portions of h (which partially span a strip) are passed recursively to the next level, along with the vertical segments. After the initial sorting preprocessing, each of the $O(\log_m n)$ levels of recursion requires $O(n)$ I/Os, yielding the desired bound (5.1). Arge et al. [**21**] develop a unified approach to distribution sweep in higher dimensions.

A central operation in spatial databases is spatial join. A common preprocessing step is to find the pairwise intersections of the bounding boxes of the objects involved in the spatial join. The problem of intersecting orthogonal rectangles can be solved by combining the previous algorithm for orthogonal segments with one for range searching. A unified approach, extendible to higher dimensions, is taken by Arge et al. [**21**] using distribution sweep. The objects that are stored in the data structure in this case are rectangles, not vertical segments. The branching factor is chosen to be $\Theta(\sqrt{m}\,)$ rather than $\Theta(m)$. Each rectangle is associated with the largest contiguous range of vertical strips that it spans. Each of the possible $\Theta\bigl(\binom{\sqrt{m}}{2}\bigr) = \Theta(m)$ contiguous ranges is called a *multislab*. (The branching factor was chosen to be $\Theta(\sqrt{m}\,)$ rather than $\Theta(m)$ so as to accommodate a buffer in internal memory for each multislab; the height of the tree remains $O(\log_m n)$.) The resulting algorithm outperforms other techniques; empirical timings are given in Section 11.

Arge et al. [**21**] give an algorithm for finding all intersections among N line segments, but the output component of the I/O bound is slightly nonoptimal: $z \log_m n$ rather than z. Crauser et al. [**50, 51**] use an incremental randomized construction to attain the optimal I/O bound (5.1) for line segment intersection and other problems. They also show how to compute the trapezoidal decomposition for intersecting segments.

6. Batched Problems on Graphs

The first work on EM graph algorithms was by Ullman and Yannakakis [**127**] for the problem of transitive closure. Chiang et al. [**42**] consider a variety of graph problems, several of which have upper and lower I/O bounds related to permuting. One key idea Chiang et al. exploit is that efficient EM algorithms can often be developed by a sequential simulation of a parallel algorithm for the same problem. Sorting is done periodically to reblock the data. In list ranking, which is used as a subroutine in the solution of several other graph problems, the number of working processors in the parallel algorithm decreases geometrically with time, so the number of I/Os for the entire simulation is proportional to the number of I/Os used in the first phase, which is given by the sorting bound $\Theta\big((n/D)\log_m n\big)$. Dehne et al. [**53**] and Sibeyn and Kaufmann [**121**] show how to get efficient I/O bounds by exploiting coarse-grained parallel algorithms, under certain assumptions on the parameters of the PDM model (such as assuming that $\log_m n \leq 2$ and that the total disk space usage is $O(n)$) so that the periodic sortings can be done in a linear number of I/Os.

For list ranking, the optimality of the EM algorithm in [**42**] assumes that $\sqrt{m}\log m = \Omega(\log n)$, which is usually true. That assumption can be removed by use of the buffer tree data structure [**13**] (see Section 7.2). A practical, randomized implementation of list ranking appears in [**120**]. Recent work on other EM graph algorithms appears in [**1, 14, 71, 83, 91**]. The problem of how to store graphs on disks for efficient traversal is discussed in [**5, 103**]. EM problems that arise in data mining and On-Line Analytical Processing include constructing classification trees [**141**] and computing wavelet decompositions and histograms [**139, 140**].

The I/O complexity of several of the basic graph problems considered in [**42, 83, 127**] remain open, including connected components, topological sorting, shortest paths, breadth-first search, and depth-first search. For example, for a graph with $V = vB$ vertices and $E = eB$ edges, the best-known EM algorithms for breadth-first search, depth-first search, and transitive closure require $\Theta(e\log_m v + V)$, $\Theta(ve/m + V)$, and $\Theta(Vv\sqrt{e/m}\,)$ I/Os, respectively. Connected components can be determined in $O\big(e(\log_m v)\log\max\{2, \log(vB/e)\}\big)$ I/Os deterministically and in only $O(e\log_m v)$ I/Os probabilistically.

In order for the parallel simulation technique to yield an efficient EM algorithm, the parallel algorithm must not use too many processors, preferably at most N. Unfortunately, the polylog-time algorithms for problems like depth-first search and shortest paths use a polynomial number of processors. The interesting connection between the parallel domain and the EM domain suggests that there may be relationships between computational complexity classes related to parallel computing (such as P-complete problems) and those related to I/O efficiency.

7. Spatial Data Structures

We now turn our attention to some online spatial data structures for massive data applications. For purposes of exposition, we consider dictionary lookup and orthogonal range search as the canonical query operations. That is, we want data structures that can support insert, delete, lookup, and orthogonal range query. Given a value x, the lookup operation returns the item(s), if any, in the structure with key value x. A range query, for a given d-dimensional rectangle, returns all the points in the interior of the rectangle.

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, whereas methods like R-trees and kd-trees are organized by partitioning the data items themselves. We discuss primarily the latter type.

7.1. B-trees and Variants. Tree-based data structures arise naturally in the dynamic 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 generally use a block for each node, which can store $\Theta(B)$ pointers and data values. A tree of degree B^c with n leaf nodes has $\lceil \frac{1}{c} \log_B N \rceil$ levels. The well-known *B-tree* due to Bayer and McCreight [**29, 46, 88**] 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 linear space, $O(\log_B N + z)$ I/Os per query, and $O(\log_B N)$ I/Os per insert or delete. 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. Deletions are handled in a symmetric way by merging nodes.

In the *B^+-tree* variant, pictured in Figure 5, 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 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 it increases the relative storage utilization. When no sharing is done (as in B^+-trees), Yao [**145**] 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 $2 \ln(3/2) \approx 81\%$ [**25, 92**]. 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. [**30**] and Varman and Verma [**130**]. Lomet and Salzberg [**98**] explore mechanisms to add concurrency and recovery to B-trees.

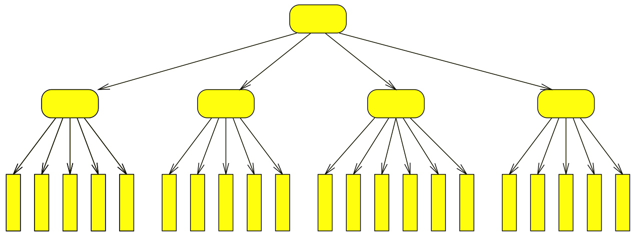


FIGURE 5. B$^+$-tree multiway search tree. Each internal and leaf node corresponds to a disk block. All the items are stored in the leaves. The internal nodes store only key values and pointers, $\Theta(B)$ of them per node. Although not indicated here, the leaf blocks are linked together sequentially.

Arge and Vitter [**24**] 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 optimal dynamic data structures for stabbing queries and segment trees in external memory, which we discuss in Section 8, but they also have applications to the internal memory RAM model [**24, 71**]. 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 [**142**] for augmenting one-dimensional data structures with range restriction capabilities.

Agarwal et al. [**3**] develop an interesting variant of B-trees, called *level-balanced B-trees*, that maintain parent pointers. A straightforward modification of conventional B-trees would require $\Theta(B \log_B N)$ I/Os per split to maintain parent parents. Instead, level-balanced B-trees support insert, delete, merge, and split operations in $O\bigl((1 + (b/B)(\log_m n) \log_b N\bigr)$ I/Os amortized, for any $2 \leq b \leq B/2$, which is bounded by $O\bigl((\log_B N)^2\bigr)$. Agarwal et al. [**3**] use level-balanced B-trees in a data structure for point location in monotone subdivisions, which supports queries and (amortized) updates in $O\bigl((\log_B N)^2\bigr)$ I/Os. They also use it to dynamically maintain planar *st*-graphs using $O\bigl((1 + (b/B)(\log_m n) \log_b N\bigr)$ 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. Such an improvement would immediately give an optimal dynamic structure for reachability queries in planar *st*-graphs.

7.2. Buffer Trees. Many batched problems in computational geometry can be solved by plane sweep techniques. For example, in Section 5 we showed how to compute orthogonal segment intersections by keeping track of the vertical segments hit by a horizontal sweep line. If we use a B-tree to store the hit vertical segments, each insertion and query uses $O(\log_B N)$ I/Os, resulting in a huge I/O bound of $O(N \log_B N)$, which can be more than B times larger than the desired bound of $O(n \log_m n)$. One solution suggested in [**134**] 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 [**13**] 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 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 [**32**] showed how to extend buffer trees to implement segment trees in external memory in a batched dynamic setting by reducing the node degrees to $\Theta(\sqrt{m}\,)$ and by introducing *multislabs* in each node.

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 [**34**] 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.

7.3. Multidimensional Spatial Structures. Grossi and Italiano [**72**] construct a multidimensional version of B-trees, called *cross trees*, that combine the data-driven partitioning of weight-balanced B-trees at the upper levels of the tree with the space-driven partitioning of methods like quad trees at the lower levels of the tree. For $d > 1$, d-dimensional orthogonal range queries can be done in $O(n^{1-1/d} + z)$ I/Os, and inserts and deletes take $O(\log_B N)$ I/Os. The data structure uses linear space and also supports the dynamic operations of split and concatenate in $O(n^{1-1/d})$ I/Os.

One way to get multidimensional EM data structures is to augment known internal memory structures, such as quad trees and kd-trees, with block access capabilities. Examples include *grid files* [**77, 90, 101**], *kd-B-trees* [**110**], *buddy trees* [**117**], and *hB-trees* [**59, 97**]. Another technique is to "linearize" the multidimensional space by imposing a total ordering on it (a so-called space-filling curve), and then the total order is used to organize the points into a B-tree. All the methods described in this paragraph use linear space, and they work well in certain situations; however, their worst-case range query performance is no better than that of cross trees, and for some methods, like grid files, queries can require $\Theta(n)$ I/Os, even if there are no points satisfying the query. We refer the reader to [**6, 66, 102**] for a broad survey of these methods. Space-filling curves arise again in connection with R-trees, which we describe in the next section.

7.4. R-trees. The *R-tree* of Guttman [**74**] 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. 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 [**6, 66, 70**]. 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. [**31**] 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 [**80, 81**]. 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 [**33, 82**].

Arge et al. [**17**] and van den Bercken et al. [**128**] have independently devised fast bulk loading methods for R*-trees that are based upon buffer trees. The former method is especially efficient and can even support dynamic batched updates and queries. Experiments with this technique are discussed in Section 11.

8. 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 [**85**] for background.) We have already discussed multidimensional range searching in a batched setting in Section 5. In this section we concentrate on the important online case.

For many types of range searching problems, it is very difficult to develop theoretically optimal algorithms. We have seen some linear-space online data structures in Sections 7.3 and 7.4, but their query performance is not optimal. Many open problems remain. The primary theoretical challenges are three-fold:

1. to get a combined search and output cost for queries of $O(\log_B N + z)$ I/Os,
2. to use only a linear amount of disk storage space, and
3. to support dynamic updates in $O(\log_B N)$ I/Os.

To develop optimal data structures for queries, it is helpful to combine together the I/O cost $O(\log_B N)$ of the search component with the I/O cost $O(z)$ for reporting the output, as in criterion 1, rather than to consider the search cost separately from the output cost, because when one cost is much larger than the other, the query algorithm has the extra freedom to follow a *filtering* paradigm [**37**], in which both the search component and the output reporting are allowed to use the larger number of I/Os. Subramanian and Ramaswamy [**122**] prove the lower bound 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

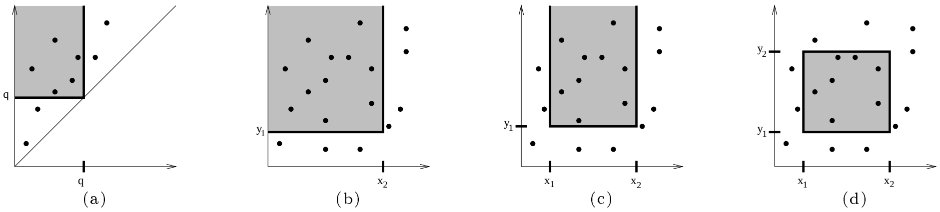


FIGURE 6. Different types of 2-D range queries: (a) Diagonal corner $(1,1)$-sided query, (b) $(1,1)$-sided query, (c) $(2,1)$-sided query, (d) general $(2,2)$-sided query.

$O\big((\log_B N)^c + z\big)$ 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 [**38**] for the internal memory model.

Hellerstein et al. [**75**] consider a generalization of the layout-based lower bound argument of Kanellakis et al. [**85**] for studying the tradeoff between disk space usage and query performance. 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 a filtering approach. 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. One fix is to consider only output sizes Z larger than $(\log_B N)B$, but then the problem of how to find the relevant blocks is ignored. Despite this shortcoming, the model is elegant and provides insight into the complexity of blocking data in external memory. Further results in this model appear in [**89, 114**].

When the data structure is restricted to contain only a single copy of each item, Kanth and Singh [**86**] 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 [**72**] mentioned in Section 7.3, which in addition supports the operations of split and concatenate.

The lower bounds mentioned above for 2-D range search apply to general rectangular queries. A natural question to ask is whether there are data structures that can meet criteria 1–3 for interesting special cases of 2-D orthogonal range searching. Fortunately, the answer is yes.

To be precise, we define a $(s_1, s_2, \ldots, s_d)$-*sided range query* in d-dimensional space, where each $s_i \in \{1,2\}$, to be an orthogonal range query with s_i finite limits in the x_i dimension. For example, the 2-D range query $[3,5] \times [4,\infty)$ is a $(2,1)$-sided range query, since there are two finite limits in the x_1 dimension (namely, $3 \leq x_1$ and $x_1 \leq 5$) but only one finite limit in the x_2 dimension (namely, $x_2 \geq 4$). A general 2-D range query is a $(2,2)$-sided query. (See Figure 6.) In the two-dimensional cases studied in [**22, 85, 109, 122**], the authors use the terms "two-sided", "three-sided", and "four-sided" range query to mean what we call $(1,1)$-sided, $(2,1)$-sided, and $(2,2)$-sided queries, respectively.

Arge and Vitter [**24**] design an EM interval tree data structure based upon the weight-balanced B-tree that meets all three criteria. It uses linear disk space and does queries in $O(\log_B N + z)$ I/Os and updates in $O(\log_B N)$ I/Os. It solves the problems of stabbing queries and dynamic interval management, following up on earlier work by Kanellakis et al. [**85**]. Stabbing queries are equivalent to $(1,1)$-sided range queries where the corner point is on the diagonal. Other applications arise in graphics and GIS. For example, Chiang and Silva [**43**] 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 [**5**]. The interval tree approach also yields dynamic EM segment trees with optimal query and update bound and $O(n \log_B N)$-block space usage.

For nonrestricted $(1,1)$-sided and $(2,1)$-sided 2-D range queries, Ramaswamy and Subramanian [**109**] introduce the notion of *path caching* to develop EM data structures that meet criterion 1 but have higher storage overheads and amortized and/or nonoptimal update bounds. Subramanian and Ramaswamy [**122**] present the *P-range tree* data structure for $(2,1)$-sided queries, which uses optimal linear disk space and has nearly optimal query and amortized update bounds. They get a static data structure for general $(2,2)$-sided 2-D range searching with the same query bound by applying a filtering technique of Chazelle [**37**]: The outer level of the structure is a $(\log_B N + 1)$-way one-dimensional search tree; each $(2,2)$-sided query is reduced to two $(2,1)$-sided queries, a stabbing query, and $\log_B N$ list traversals. The disk space usage is $O\bigl(n(\log n)/\log(\log_B N + 1)\bigr)$, as required by the pointer machine lower bound. The structure could be modified to perform updates, by application of a weight-balanced B-tree and the dynamization techniques of [**22**], but the resulting update time would be amortized and nonoptimal, as a consequence of the use of the $(2,1)$-sided data structure.

Arge et al [**22**] apply notions of persistence to get a simple and optimal static data structure for $(2,1)$-sided range queries; it supports queries in $O(\log_B N + z)$ I/Os and uses linear disk space. They get a fully dynamic data structure for $(2,1)$-sided queries with the same optimal query and space bounds and with optimal update bound $O(\log_B N)$, by combining the static structure with an external priority search tree based upon weight-balanced B-trees. The structure can be generalized using the technique of [**37**] to handle $(2,2)$-sided queries with optimal query bound $O(\log_B N)$, optimal disk space usage $O\bigl(n(\log n)/\log(\log_B N + 1)\bigr)$, and amortized update bound $O\bigl((\log_B N)(\log n)/\log(\log_B N + 1)\bigr)$.

One intuition from [**75**] 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 Sections 7.3 and 7.4 can be shown to perform provably well for such queries.

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. Vengroff and Vitter [**132**] 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 items that dominate a query point are densely contained in a set of blocks. 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 $O\bigl(n(\log n)^k/(\log(\log_B N + 1))^k\bigr)$ disk blocks to support 3-D range queries in which k of the dimensions $(0 \le k \le 3)$ have finite ranges.

The space bounds are optimal for $(1, 1, 1)$-sided queries (i.e., $k = 0$) and $(2, 1, 1)$-sided 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 $(1, 1, 1)$-sided queries, improving upon the result in [**39**]. Agarwal et al. [**4**] give optimal bounds for static halfspace range searching in two dimensions and some variants in higher dimensions. The number of I/Os needed to build the 3-D and halfspace data structures is rather large (more than order N). Still, the structures shed useful light on the complexity of range searching. An open problem is to design efficient construction and update algorithms and to improve upon the constant factors. Some other types of range searching, such as simplex range searching, have not yet been investigated in the external memory setting.

Callahan et al. [**35**] 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 [**6, 66, 102**] for a broad survey.

9. String Processing

Digital trie-based structures, in which branching decisions at each node are made based upon the values of particular bits in strings, are effective for string processing in internal memory. In EM applications, what is needed is a multiway digital structure. Unfortunately, if the strings are long, there is no space to store them completely in each node, and if pointers to strings are stored in each node, the number of I/Os per node access will be large.

Ferragina and Grossi [**63, 62**] develop an elegant generalization of the B-tree for storing strings, called the *String B-tree* or simply *SB-tree*. An SB-tree differs from a conventional B-tree in the way that each $\Theta(B)$-way branching node is represented. In a conventional B-tree, $\Theta(B)$ unit-sized keys are stored in each internal node to guide the searching, and thus the entire node fits in one or two blocks. However, strings can be arbitrarily long, so there may not be enough space to store $\Theta(B)$ strings per node. Pointers to $\Theta(B)$ strings could be stored instead in each node, but access to the strings during search would require more than a constant number of I/Os per node.

Ferragina and Grossi's solution for how to represent each node of the SB-tree is based upon a variant of the *Patricia trie* character-based data structure [**100, 88**] along the lines of Ajtai et al. [**11**]. The Patricia trie achieves B-way branching with a total storage of $O(B)$ characters. Each of its internal nodes stores an index (a number from 0 to N) and a one-character label for each of its outgoing edges. For example, in the example in Figure 7, the right child of the root has index 4 and characters "`a`" and "`b`", which means that the node's left subtrie consists of strings whose fifth character (character 4) is "`a`", and its right subtrie consists of strings whose fifth character is "`b`". The preceding (four) characters in all the strings in the node's subtrie are identically "`bcbc`". To find which of the B branches to take for a search string, a binary search is done in the Patricia trie; each binary branching decision is based upon the character indexed at that node. For search string "`bcbabcba`", a binary search on the trie in Figure 7 traverses the far-right path of the Patricia trie, examining character positions 0, 4, and 6.

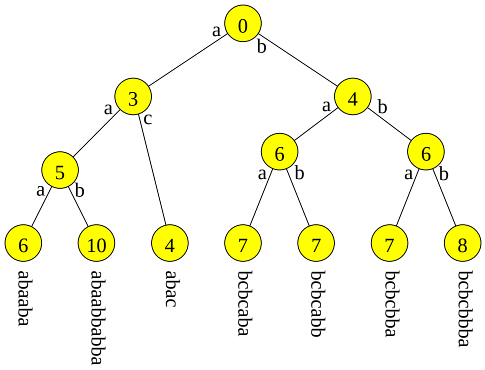


Figure 7. Patricia trie representation of a single node of an SB-tree, with branching factor $B = 8$. The seven strings used for partitioning are pictured at the leaves; in the actual data structure, pointers to the strings, not the strings themselves, are stored at the leaves. The pointers to the B children of the SB-tree node are also stored at the leaves.

Unfortunately, the leaf node that is eventually reached (in our example, the leaf at the far right, corresponding to "`bcbcbbba`") is not in general at the correct branching point, since only certain character !la positions in the string were examined during the search. The key idea to fix this situation is to sequentially compare the search string with the string associated with the leaf, and if they differ, the index where they differ can be found. In the example, the search string "`bcbabcba`" differs from "`bcbcbbba`" in the fourth character (character 3), and the search string therefore is lexicographically smaller than the entire right subtrie of the root. It fits in between the leaves "`abac`" and "`bcbcaba`".

Searching the Patricia trie requires one I/O to load it into memory, plus additional I/Os to do the sequential scan of the string after the leaf of the Patricia trie is reached. Each block of the search string that is examined during a sequential scan does not have to be read in again for lower levels of the SB-tree, so the I/Os for the sequential scan can be charged to the blocks of the search string. The resulting query time to search in an SB-tree for a string of ℓ characters is therefore $O(\log_B N + \ell/B)$, which is optimal.

Ferragina and Grossi apply SB-trees to string matching, prefix search, and substring search. Farach et al. [**60**] show how to construct SB-trees, suffix trees, and suffix arrays on strings of length N using $O(n \log_m n)$ I/Os, which is optimal. Clark and Munro [**44**] give an alternate approach to suffix trees.

Arge et al. [**16**] consider several models for the problem of sorting K strings of total length N in external memory. They develop efficient sorting algorithms in these models, making use of the SB-tree, buffer tree techniques, and a simplified version of the SB-tree for merging called the *lazy trie*. In the RAM model, the problem can be solved in $O(K \log K + N)$ time. By analogy to the problem of sorting integers, it would be natural to expect that the I/O complexity would be $O(k \log_m k + n)$ time, where $k = \lceil K/B \rceil$. Arge et al. show somewhat counterintuitively that for sorting short strings (i.e., strings of length at most B) the I/O complexity depends upon the total *number of characters*, whereas for long strings the complexity depends upon the total *number of strings*.

THEOREM 9.1 ([**16**]). *We can sort K strings of total length N, where N_1 is the total length of the short strings and K_2 is the number of long strings, using the following number of I/Os:*

$$O\left(\frac{N_1}{B}\log_m\left(\frac{N_1}{B}+1\right)+K_2\log_M(K_2+1)+\frac{N}{B}\right), \tag{9.1}$$

Lower bounds for various models of how strings can be manipulated are given in [**16**]. There are gaps in some cases between the upper and lower bounds.

10. The TPIE External Memory Programming Environment

There are three basic approaches to supporting development of I/O-efficient code, which we call array-oriented systems (such as PASSION and ViC*), access-oriented systems (such as the UNIX file system, Panda, and MPI-IO), and framework-oriented systems (such as TPIE). We refer the reader to [**68**] and its references for background.

In this section we describe TPIE (Transparent Parallel I/O programming Environment)[6] [**17, 131, 133**], which is used as the implementation platform for the experiments in the next section. TPIE is a comprehensive software package that helps programmers to develop high-level, portable, and efficient implementations of EM algorithms.

TPIE takes a somewhat non-traditional approach to batched computation: Instead of viewing it as an enterprise in which code reads data, operates on it, and writes results, TPIE views computation as a continuous process during which a program is fed streams of data from an outside source and leaves trails of results behind. Programmers do not need to worry about making explicit calls to I/O routines; instead, they merely specify the functional details of the desired computation, and TPIE automatically choreographs a sequence of data movements to keep the computation fed.

TPIE is written in C++ as a set of templated classes and functions. It consists of three main components: a block transfer engine (BTE), a memory manager (MM), and an access method interface (AMI). The BTE is responsible for moving blocks of data to and from the disk. It is also responsible for scheduling asynchronous read-ahead and write-behind when necessary to allow computation and I/O to overlap. The MM is responsible for managing main memory in coordination with the AMI and BTE. The AMI provides the high-level uniform interface for application programs. The AMI is the only component that programmers normally need to interact with directly. Applications that use the AMI are portable across hardware platforms, since they do not have to deal with the underlying details of how I/O is performed on a particular machine. We have seen in the previous sections that many batched problems in spatial databases, GIS, scientific computing, graphs, and string processing can be solved optimally using a relatively small number of basic paradigms like scanning, multiway distribution, and merging, which TPIE supports as access mechanisms. TPIE also supports block-oriented operations on trees for online problems.

[6]The TPIE software distribution is available at no charge on the World Wide Web at http://www.cs.duke.edu/TPIE/.

11. Empirical Comparisons

In this section we examine the empirical performance of algorithms for two problems that arise in spatial databases. The TPIE system described in the previous section is used as the common implementation platform. Other recent experiments involving the paradigms discussed in this paper appear in [**41, 79**].

11.1. Rectangle Intersection and Spatial Join. In the first experiment, three algorithms are implemented in TPIE for the problem of rectangle intersection, which is typically the first step in a spatial join computation. The first method, called Scalable Sweeping-Based Spatial Join (SSSJ) [**20**], is a robust new algorithm based upon the distribution sweep paradigm of Section 5. The other two methods are Partition-Based Spatial-Merge (QPBSM) used in Paradise [**108**] and a new modification called MPBSM that uses an improved dynamic data structure for intervals [**20**].

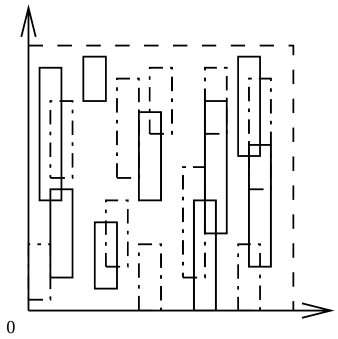
The algorithms were tested on several data sets. The timing results for the two data sets in Figures 8(a) and 8(b) are given in Figures 8(c) and 8(d), respectively. The first data set is the worst case for sweep line algorithms; a large fraction of the line segments in the file are active (i.e., they intersect the current sweep line). The second data set is a best case for sweep line algorithms. The two PBSM algorithms have the disadvantage of making extra copies. SSSJ shows considerable improvement over the PBSM-based methods. On more typical data, such as TIGER/line road data sets [**125**], experiments indicate that SSSJ and MPBSM run about 30% faster than QPBSM.

11.2. Batched Operations on R-trees. In the second experiment, three methods for building R-trees are evaluated in terms of their bulk loading time and the resulting query performance. The three methods tested are a newly developed buffer R-tree method [**17**] (labeled "buffer"), the naive sequential method for construction into R*-trees (labeled "naive"), and the best update algorithm for Hilbert R-trees (labeled "Hilbert") [**82**].

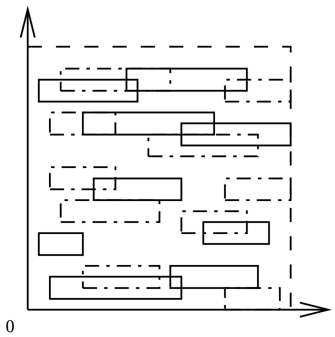
The experimental data came from TIGER/line road data sets from four U.S. states [**125**]. One experiment involved building an R-tree on the road data for each state and for each of four possible buffer sizes. The four buffer sizes were capable of storing 0, 600, 1,250, and 5,000 rectangles, respectively. The query performance of each resulting R-tree was measured by posing rectangle intersection queries, using rectangles taken from TIGER hydrographic data. The results, depicted in Figure 9, show that buffer R-trees, even with relatively small buffers, achieve a tremendous speedup in construction time without any worsening in query performance, compared with the naive method (which corresponds to a buffer size of 0).

In another experiment, a single R-tree was built for each of the four U.S. states, containing 75% of the road data objects for that state. Using each of the three algorithms, the remaining 25% of the objects were inserted into the R-tree, and the construction time was measured. Query performance was then tested as before. The results in Table 2 show that the buffer R-tree has faster construction time than the Hilbert R-tree (the previous best method for construction time) and similar or better query performance than repeated insertions (the previous best method for query performance).

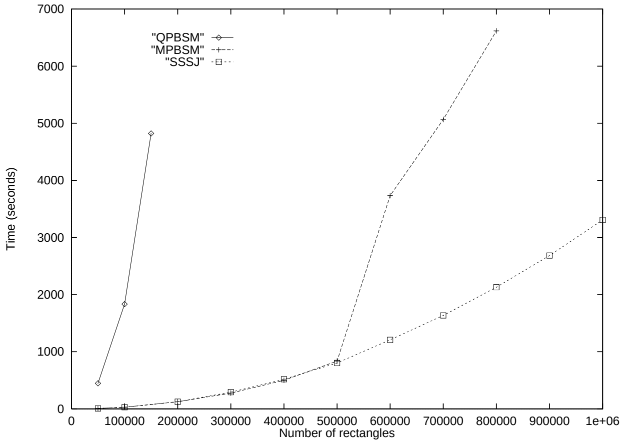
(a)



(b)



(c)



(d)

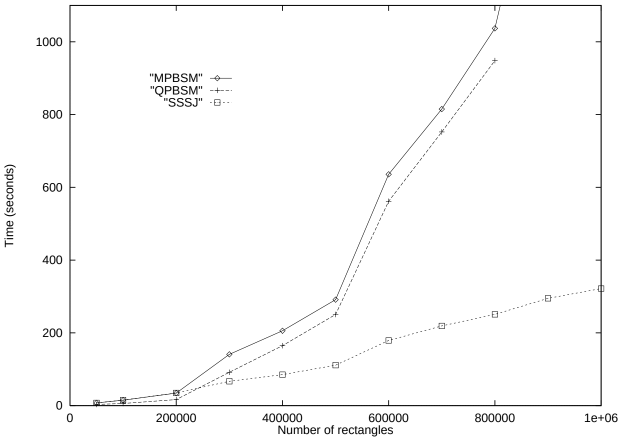


Figure 8. Comparison of Scalable Sweeping-Based Spatial Join (SSSJ) with the original PBSM (QPBSM) and a new variant (MPBSM) (a) Data set 1 consists of tall and skinny (vertically aligned) rectangles. (b) Data set 2 consists of short and wide (horizontally aligned) rectangles. (c) Running times on data set 1. (d) Running times on data set 2.

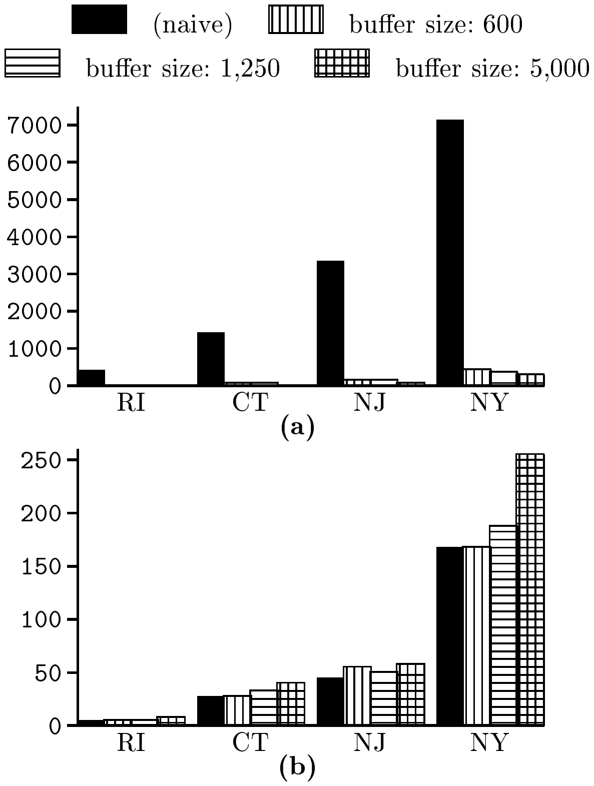


FIGURE 9. Costs for R-tree processing (in units of 1000 I/Os) using the naive repeated insertion method and the buffer R-tree for various buffer sizes: (a) cost for bulk-loading the R-tree, (b) query cost.

Data Set	Update Method	Update with 25% of the data Building	Querying	Packing
RI	naive	259,263	6,670	64%
	Hilbert	15,865	7,262	92%
	buffer	13,484	5,485	90%
CT	naive	805,749	40,910	66%
	Hilbert	51,086	40,593	92%
	buffer	42,774	37,798	90%
NJ	naive	1,777,570	70,830	66%
	Hilbert	120,034	69,798	92%
	buffer	101,017	65,898	91%
NY	naive	3,736,601	224,039	66%
	Hilbert	246,466	230,990	92%
	buffer	206,921	227,559	90%

TABLE 2. Summary of the costs (in number of I/Os) for R-tree updates and queries. Packing refers to the percentage storage utilization.

12. Dynamic Memory Allocation

The amount of memory allocated to a program may fluctuate during the course of execution because of demands placed on the system by other users and processes. EM algorithms must be able to adapt dynamically to whatever resources are available so as to preserve good performance [**107**]. The algorithms in the previous sections assume a fixed memory allocation; they must resort to virtual memory if the memory allocation is reduced, often causing a severe performance hit.

Barve and Vitter [**28**] discuss the design and analysis of EM algorithms that adapt gracefully to changing memory allocations. In their model, without loss of generality, a program $\mathcal{P}$ is allocated memory in phases: During the ith phase, $\mathcal{P}$ is allocated m_i blocks of internal memory, and this memory remains allocated to $\mathcal{P}$ until $\mathcal{P}$ completes $2m_i$ I/O operations, at which point the next phase begins. The process continues until $\mathcal{P}$ finishes execution. The model makes the reasonable assumption that the duration for each memory allocation phase is long enough to allow all the memory in that phase to be used by the program.

For sorting, the lower bound approach of (3.7) implies that

$$\sum_i 2m_i \log m_i = \Omega(n \log n).$$

We say that $\mathcal{P}$ is *dynamically optimal* for sorting if

$$\sum_i 2m_i \log m_i = O(n \log n)$$

for all possible sequences $m_1, m_2, \ldots$ of memory allocation. Intuitively, if $\mathcal{P}$ is dynamically optimal, no other program can perform more than a constant number of sorts in the worst-case for the same sequence of memory allocations.

Barve and Vitter [**28**] define the model in generality and give dynamically optimal strategies for sorting, matrix multiplication, and buffer trees operations. Their work represents the first theoretical model of dynamic allocation for EM algorithms. Pang et al. [**107**] and Zhang and Larson [**147**] give memory-adaptive merge sort algorithms, but their algorithms handle only special cases and can be made to perform poorly for certain patterns of memory allocation.

13. Conclusions

In this paper we have described several useful paradigms for the design and implementation of efficient external memory algorithms and data structures. The problem domains we have considered include sorting, permuting, FFT, scientific computing, computational geometry, graphs, databases, geographic information systems, and text and string processing. Interesting challenges remain in virtually all these problem domains. One difficult problem is to prove lower bounds for permuting and sorting without the indivisibility assumption. Another promising area is the design and analysis of algorithms for efficient use of multiple disks. Optimal bounds have not yet been determined for several basic graph problems like topological sorting, shortest paths, breadth-first and depth-first search, and connected components. There is an intriguing connection between problems that have good I/O speedups and problems that have fast and work-efficient parallel algorithms.

A continuing goal is to develop optimal EM algorithms and 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.

New architectures, such as networks of workstations and hierarchical storage devices, present many interesting challenges and opportunities. Work is beginning, for example, on extensions of TPIE to such domains and on applying the buffer management techniques of the SRM method in Section 3.2 to cache-friendly distribution sort algorithms. The ability to adapt in a reasonable way to changing memory allocations may be important for EM algorithms in practice.

Acknowledgements. The author wishes to thank the members of the Center for Geometric Computing at Duke University and the anonymous referees for very helpful comments and suggestions.

References

[1] J. Abello, A. Buchsbaum, and J. Westbrook. A functional approach to external memory graph algorithms. In *Proceedings of the 6th Annual European Symposium on Algorithms*, volume 1461 of *Lecture Notes in Computer Science*, 332–343, Venice, Italy, August 1998. Springer-Verlag.

[2] M. Adler. New coding techniques for improved bandwidth utilization. In *37th IEEE Symposium on Foundations of Computer Science*, 173–182, Burlington, VT, October 1996.

[3] 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.

[4] 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.

[5] 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.

[6] 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. American Mathematical Society Press, Providence, RI, 1999.

[7] A. Aggarwal, B. Alpern, A. K. Chandra, and M. Snir. A model for hierarchical memory. *Proceedings of the 19th ACM Symposium on Theory of Computation*, 305–314, 1987.

[8] A. Aggarwal, A. Chandra, and M. Snir. Hierarchical memory with block transfer. *Proceedings of the 28th Annual IEEE Symposium on Foundations of Computer Science*, 204–216, 1987.

[9] A. Aggarwal and C. G. Plaxton. Optimal parallel sorting in multi-level storage. *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, 659–668, 1994.

[10] A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9), 1116–1127, 1988.

[11] M. Ajtai, M. Fredman, and J. Komlos. Hash functions for priority queues. *Information and Control*, 63(3), 217–225, 1984.

[12] B. Alpern, L. Carter, E. Feig, and T. Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12(2-3), 72–109, 1994.

[13] 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.

[14] L. Arge. The I/O-complexity of ordered binary-decision diagram manipulation. In *Proceedings of the International Symposium on Algorithms and Computation*, volume 1004 of *Lecture Notes in Computer Science*, 82–91. Springer-Verlag, 1995.

[15] L. Arge. External-memory algorithms with applications in geographic information systems. 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.

[16] L. Arge, P. Ferragina, R. Grossi, and J. Vitter. On sorting strings in external memory. In *Proceedings of the ACM Symposium on Theory of Computation*, 540–548, 1997.

[17] 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.

[18] L. Arge, M. Knudsen, and K. Larsen. A general lower bound on the I/O-complexity of comparison-based algorithms. In *Proceedings of the 3rd Workshop on Algorithms and Data Structures*, volume 709 of *Lecture Notes in Computer Science*, 83–94. Springer-Verlag, 1993.

[19] L. Arge and P. Miltersen. On showing lower bounds for external-memory computational geometry problems. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*. American Mathematical Society Press, Providence, RI, this volume.

[20] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Scalable sweeping-based spatial join. In *Proceedings of the 24th International Conference on Very Large Databases*, 570–581, New York, August 1998.

[21] 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 *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 685–694, 1998.

[22] 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.

[23] L. Arge, D. E. Vengroff, and J. S. Vitter. External-memory algorithms for processing line segments in geographic information systems. *Algorithmica*, to appear. Special issue on cartography and geographic information systems. An earlier version appeared in *Proceedings of the Third European Symposium on Algorithms*, volume 979 of *Lecture Notes in Computer Science*, 295–310, Springer-Verlag, September 1995.

[24] 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.

[25] R. A. Baeza-Yates. Expected behaviour of B^+-trees under random insertions. *Acta Informatica*, 26(5), 439–472, 1989.

[26] R. D. Barve, E. F. Grove, and J. S. Vitter. Simple randomized mergesort on parallel disks. *Parallel Computing*, 23(4), 601–631, 1997.

[27] 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. In *Joint International Conference on Measurement and Modeling of Computer Systems*, Atlanta, GA, May 1999.

[28] 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.

[29] R. Bayer and E. McCreight. Organization of large ordered indexes. *Acta Inform.*, 1, 173–189, 1972.

[30] 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.

[31] 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.

[32] J. L. Bentley. Multidimensional divide and conquer. *Communications of the ACM*, 23(6), 214–229, 1980.

[33] 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.

[34] 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.

[35] 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.

[36] L. Carter and K. S. Gatlin. Towards an optimal bit-reversal permutation program. In *Proceedings of the IEEE Symposium on Foundations of Comp. Sci.*, Palo Alto, CA, November 1998.

[37] B. Chazelle. Filtering search: a new approach to query-answering. *SIAM Journal on Computing*, 15, 703–724, 1986.

[38] B. Chazelle. Lower bounds for orthogonal range searching: I. the reporting case. *Journal of the ACM*, 37(2), 200–212, April 1990.

[39] B. Chazelle and H. Edelsbrunner. Linear space data structures for two types of range search. *Discrete & Computational Geometry*, 2, 113–126, 1987.

[40] 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.

[41] Y.-J. Chiang. Experiments on the practical I/O efficiency of geometric algorithms: Distribution sweep vs. plane sweep. *Computational Geometry: Theory and Applications*, 8(4), 211–236, 1998.

[42] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 139–149, January 1995.

[43] 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, this volume. American Mathematical Society Press.

[44] D. R. Clark and J. I. Munro. Efficient suffix trees on secondary storage. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 383–391, Atlanta, GA, June 1996.

[45] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *Discrete and Computational Geometry*, 4, 387–421, 1989.

[46] D. Comer. The ubiquitous B-tree. *Comput. Surveys*, 11(2), 121–137, 1979.

[47] P. Corbett, D. Feitelson, S. Fineberg, Y. Hsu, B. Nitzberg, J.-P. Prost, M. Snir, B. Traversat, and P. Wong. Overview of the MPI-IO parallel I/O interface. In R. Jain, J. Werth, and J. C. Browne, editors, *Input/Output in Parallel and Distributed Computer Systems*, volume 362 of *The Kluwer International Series in Engineering and Computer Science*, chapter 5, 127–146. Kluwer Academic Publishers, 1996.

[48] T. H. Cormen and D. M. Nicol. Performing out-of-core FFTs on parallel disk systems. *Parallel Computing*, 24(1), 5–20, January 1998.

[49] T. H. Cormen, T. Sundquist, and L. F. Wisniewski. Asymptotically tight bounds for performing BMMC permutations on parallel disk systems. *SIAM Journal on Computing*, 28(1), 105–136, 1999.

[50] A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, and E. Ramos. Randomized external-memory algorithms for geometric problems. In *Proceedings of the 14th ACM Symposium on Computational Geometry*, June 1998.

[51] A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, and E. Ramos. I/O-optimal computation of segment intersections. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*. American Mathematical Society Press, Providence, RI, this volume.

[52] R. Cypher and G. Plaxton. Deterministic sorting in nearly logarithmic time on the hypercube and related computers. *Journal of Computer and System Sciences*, 47(3), 501–548, 1993.

[53] F. Dehne, D. Hutchinson, and A. Maheshwari. Reducing I/O complexity by simulating coarse grained parallel algorithms. In *Proceedings of the International Parallel Processing Symmposium*, April 1999.

[54] H. B. Demuth. *Electronic Data Sorting*. Ph.d., Stanford University, 1956. A shortened version appears in *IEEE Transactions on Computing*, C-34(4), 296–310, April 1985, special issue on sorting, E. E. Lindstrom, C. K. Wong, and J. S. Vitter, editors.

[55] P. J. Denning. Working sets past and present. *IEEE Transactions on Software Engineering,*, SE-6, 64–84, 1980.

[56] D. J. DeWitt, J. F. Naughton, and D. A. Schneider. Parallel sorting on a shared-nothing architecture using probabilistic splitting. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, 280–291, December 1991.

[57] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38, 86–124, 1989.

[58] NASA's Earth Observing System (EOS) web page, NASA Goddard Space Flight Center, `http://eospso.gsfc.nasa.gov/`.

[59] G. Evangelidis, D. B. Lomet, and B. Salzberg. The hB^{Π}-tree: A multi-attribute index supporting concurrency, recovery and node consolidation. *VLDB Journal*, 6, 1–25, 1997.

[60] M. Farach, P. Ferragina, and S. Muthukrishnan. Overcoming the memory bottleneck in suffix tree construction. In *Proceedings of the IEEE Symposium on Foundations of Comp. Sci.*, Palo Alto, CA, November 1998.
[61] W. Feller. *An Introduction to Probability Theory and its Applications*, volume 1. John Wiley & Sons, New York, third edition, 1968.
[62] P. Ferragina and R. Grossi. Fast string searching in secondary storage: Theoretical developments and experimental results. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 373–382, Atlanta, June 1996.
[63] P. Ferragina and R. Grossi. The String B-tree: A new data structure for string search in external memory and its applications. *Journal of the ACM*, to appear. An earlier version appeared in *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*, 693–702, Las Vegas, NV, May 1995.
[64] R. W. Floyd. Permuting information in idealized two-level storage. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, 105–109. Plenum, 1972.
[65] T. A. Funkhouser, C. H. Sequin, and S. J. Teller. Management of large amounts of data in interactive building walkthroughs. In *Proceedings of the 1992 ACM SIGGRAPH Symposium on Interactive 3D Graphics*, 11–20, Boston, March 1992.
[66] V. Gaede and O. Günther. Multidimensional access methods. *Computing Surveys*, 30(2), 170–231, June 1998.
[67] M. Gardner. *Magic Show*, chapter 7. Knopf, New York, 1977.
[68] 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.
[69] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *IEEE Foundations of Computer Science*, 714–723, Palo Alto, CA, November 1993.
[70] 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.
[71] 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.
[72] 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*. American Mathematical Society Press, Providence, RI, this volume.
[73] S. K. S. Gupta, Z. Li, and J. H. Reif. Generating efficient programs for two-level memories from tensor-products. In *Proceedings of the Seventh IASTED/ISMM International Conference on Parallel and Distributed Computing and Systems*, 510–513, Washington, D.C., October 1995.
[74] 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.
[75] 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.
[76] 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.
[77] K. H. Hinrichs. *The grid file system: Implementation and case studies of applications.* PhD thesis, Dept. Information Science, ETH, Zürich, 1985.
[78] J. W. Hong and H. T. Kung. I/O complexity: The red-blue pebble game. *Proceedings of the 13th Annual ACM Symposium on Theory of Computation*, 326–333, May 1981.
[79] D. Hutchinson, A. Maheshwari, J.-R. Sack, and R. Velicescu. Early experiences in implementing the buffer tree. Workshop on Algorithm Engineering, 1997.
[80] I. Kamel and C. Faloutsos. On packing R-trees. In *Proceedings of the 2nd International Conference on Information and Knowledge Management*, 490–499, 1993.
[81] 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.
[82] 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.

[83] M. V. Kameshwar and A. Ranade. I/O-complexity of graph algorithms. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, Baltimore, MD, January 1999.

[84] P. C. Kanellakis, G. M. Kuper, and P. Z. Revesz. Constraint query languages. *Proceedings of the 9th ACM Conference on Principles of Database Systems*, 299–313, 1990.

[85] 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.

[86] 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.

[87] D. G. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm? *SIAM Journal on Computing*, 15, 287–299, 1986.

[88] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading MA, second edition, 1998.

[89] 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.

[90] R. Krishnamurthy and K.-Y. Wang. Multilevel grid files. Tech. report, IBM T. J. Watson Center, Yorktown Heights, NY, November 1985.

[91] V. Kumar and E. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing*, 169–176, October 1996.

[92] K. Küspert. Storage utilization in B*-trees with a generalized overflow technique. *Acta Informatica*, 19, 35–55, 1983.

[93] R. Laurini and D. Thompson. *Fundamentals of Spatial Information Systems*. Academic Press, 1992.

[94] F. T. Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Transactions on Computers*, C-34(4), 344–354, April 1985. Special issue on sorting, E. E. Lindstrom and C. K. Wong and J. S. Vitter, editors.

[95] C. E. Leiserson, S. Rao, and S. Toledo. Efficient out-of-core algorithms for linear relaxation using blocking covers. In *Proceedings of the IEEE Symposium on Foundations of Comp. Sci.*, 704–713, 1993.

[96] Z. Li, P. H. Mills, and J. H. Reif. Models and resource metrics for parallel and distributed computation. *Parallel Algorithms and Applications*, 8, 35–59, 1996.

[97] 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.

[98] D. B. Lomet and B. Salzberg. Concurrency and recovery for index trees. *The VLDB Journal*, 6(3), 224–240, 1997.

[99] Y. Matias, E. Segal, and J. S. Vitter. Efficient bundle sorting, 1999. Manuscript.

[100] D. R. Morrison. Patricia: Practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15, 514–534, 1968.

[101] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: An adaptable, symmetric multi-key file structure. *ACM Trans. Database Syst.*, 9, 38–71, 1984.

[102] 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.

[103] M. H. Nodine, M. T. Goodrich, and J. S. Vitter. Blocking for external graph searching. *Algorithmica*, 16(2), 181–214, August 1996.

[104] M. H. Nodine, D. P. Lopresti, and J. S. Vitter. I/O overhead and parallel VLSI architectures for lattice computations. *IEEE Transactions on Computers*, 40(7), 843–852, July 1991.

[105] M. H. Nodine and J. S. Vitter. Deterministic distribution sort in shared and distributed memory multiprocessors. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, 120–129, Velen, Germany, June–July 1993.

[106] M. H. Nodine and J. S. Vitter. Greed Sort: An optimal sorting algorithm for multiple disks. *Journal of the ACM*, 42(4), 919–933, July 1995.

[107] H. Pang, M. Carey, and M. Livny. Memory-adaptive external sorts. *Proceedings of the 19th Conference on Very Large Data Bases*, 1993.

[108] J. M. Patel and D. J. DeWitt. Partition based spatial-merge join. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 259–270, June 1996.

[109] 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.

[110] 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.

[111] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 17–28, March 1994.

[112] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley, 1989.

[113] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1989.

[114] 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.

[115] J. E. Savage. Extending the Hong-Kung model to memory hierarchies. In *Proceedings of the 1st Annual International Conference on Computing and Combinatorics*, volume 959 of *Lecture Notes in Computer Science*, 270–281. Springer-Verlag, August 1995.

[116] J. E. Savage and J. S. Vitter. Parallelism in space-time tradeoffs. In F. P. Preparata, editor, *Advances in Computing Research, Volume 4*, 117–146. JAI Press, 1987.

[117] 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.

[118] 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.

[119] E. A. M. Shriver and M. H. Nodine. An introduction to parallel I/O models and algorithms. In R. Jain, J. Werth, and J. C. Browne, editors, *Input/Output in Parallel and Distributed Computer Systems*, chapter 2, 31–68. Kluwer Academic Publishers, 1996.

[120] J. F. Sibeyn. From parallel to external list ranking. Technical Report MPI–I–97–1–021, Max-Planck-Institut, September 1997.

[121] J. F. Sibeyn and M. Kaufmann. BSP-like external-memory computation. In *Proceedings of the 3rd Italian Conference on Algorithms and Complexity*, 229–240, 1997.

[122] 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.

[123] R. Tamassia and J. S. Vitter. Optimal cooperative search in fractional cascaded data structures. *Algorithmica*, 15(2), 154–171, February 1996.

[124] Microsoft's TerraServer online database of satellite images, available on the World-Wide Web at `http://terraserver.microsoft.com/`.

[125] TIGER/Line (tm). 1992 technical documentation. Technical report, U. S. Bureau of the Census, 1992.

[126] S. Toledo. A survey of out-of-core algorithms in numerical linear algebra. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*. American Mathematical Society Press, Providence, RI, this volume.

[127] J. D. Ullman and M. Yannakakis. The input/output complexity of transitive closure. *Annals of Mathematics and Artificial Intellegence*, 3, 331–360, 1991.

[128] 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.

[129] M. van Kreveld, J. Nievergelt, T. Roos, and P. W. (Eds.). *Algorithmic Foundations of GIS*, volume 1340 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[130] 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.

[131] D. E. Vengroff. *TPIE User Manual and Reference*. Duke University, 1997. The manual and software distribution are available on the web at `http://www.cs.duke.edu/TPIE/`.

[132] 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.

[133] D. E. Vengroff and J. S. Vitter. I/O-efficient scientific computation using TPIE. In *Proceedings of the Fifth NASA Goddard conference on Mass Storage Systems*, II, 553–570, September 1996.

[134] 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.

[135] J. S. Vitter and P. Flajolet. Average-case analysis of algorithms and data structures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, chapter 9, 431–524. North-Holland, 1990.

[136] J. S. Vitter and M. H. Nodine. Large-scale sorting in uniform memory hierarchies. *Journal of Parallel and Distributed Computing*, 17, 107–114, 1993.

[137] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2–3), 110–147, 1994.

[138] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory II: Hierarchical multilevel memories. *Algorithmica*, 12(2–3), 148–169, 1994.

[139] J. S. Vitter and M. Wang. Approximate computation of multidimensional aggregates of sparse data using wavelets. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Philadelphia, PA, June 1999.

[140] J. S. Vitter, M. Wang, and B. Iyer. Data cube approximation and histograms via wavelets. In *Proceedings of the Seventh International Conference on Information and Knowledge Management*, 96–104, Washington, November 1998.

[141] M. Wang, J. S. Vitter, and B. R. Iyer. Scalable mining for classification rules in relational databases. In *Proceedings of the International Database Engineering & Application Symposium*, 58–67, Cardiff, Wales, July 1998.

[142] D. Willard and G. Lueker. Adding range restriction capability to dynamic data structures. *Journal of the ACM*, 32(3), 597–617, 1985.

[143] D. Womble, D. Greenberg, S. Wheat, and R. Riesen. Beyond core: Making parallel computer I/O practical. In *Proceedings of the 1993 DAGS/PC Symposium*, 56–63, Hanover, NH, June 1993. Dartmouth Institute for Advanced Graduate Studies.

[144] C. Wu and T. Feng. The universality of the shuffle-exchange network. *IEEE Transactions on Computers*, C-30, 324–332, May 1981.

[145] A. C. Yao. On random 2-3 trees. *Acta Informatica*, 9, 159–170, 1978.

[146] S. B. Zdonik and D. Maier, editors. *Readings in Object-Oriented Database Systems*. Morgan Kauffman, 1990.

[147] W. Zhang and P.-A. Larson. Dynamic memory adjustment for external mergesort. *Proceedings of the Twenty-third International Conference on Very Large Data Bases*, 1997.

[148] B. Zhu. Further computational geometry in secondary memory. In *Proceedings of the International Symposium on Algorithms and Computation*, 1994.

Center for Geometric Computing, Department of Computer Science, Duke University, Durham, NC 27708–0129, USA

E-mail address: `jsv@cs.duke.edu`

URL: `http://www.cs.duke.edu/~jsv/`