# External Memory Algorithms and Data Structures: Dealing with Massive Data

JEFFREY SCOTT VITTER

*Duke University*

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 article we survey the state of the art in the design and analysis of external memory (or EM) algorithms and data structures, where the goal is to exploit locality in order to reduce the I/O costs. We consider a variety of EM paradigms for solving batched and online problems efficiently in external memory. For the batched problem of sorting and related problems such as permuting and fast Fourier transform, the key paradigms include distribution and merging. The paradigm of disk striping offers an elegant way to use multiple disks in parallel. For sorting, however,

Categories and Subject Descriptors: B.4.3 [**Input/Output and Data Communications**]: Interconnections—*parallel I/O*; E.1 [**Data Structures**]: *graphs and networks, trees*; E.5 [**Files**]: *sorting/searching*; F.1.1 [**Computation by Abstract Devices**]: Models of Computation—*bounded action devices, relations between models*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*computations on discrete structures, geometrical problems and computations, sorting and searching*; H.2.2 [**Database Management**]: Physical Design—*access methods*; H.2.8 [**Database Management**]: Database Applications—*spatial databases and GIS*; H.3.2 [**Information Storage and Retrieval**]: Information Storage—*file organization*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*information filtering, search process*

General Terms: Algorithms, Design, Experimentation, Performance, Theory

Additional Key Words and Phrases: Batched, block, B-tree, disk, dynamic, extendible hashing, external memory, hierarchical memory, I/O, multidimensional access methods, multilevel memory, online, out-of-core, secondary storage, sorting

---

This work was supported in part by Army Research Office MURI grant DAAH04-96-1-0013 and by National Science Foundation research grants CCR-9522047, EIA-9870734, and CCR-9877133.

Part of this work was done at BRICS, University of Aarhus, Aarhus, Denmark and at INRIA, Sophia Antipolis, France. Earlier, shorter versions of some of this material appeared in Vitter [1998; 1999a; 1999b; 1999c].

Author's address: Department of Computer Science, Levine Science Research Center, Duke University Box 90129, Durham, NC 27708-0129, e-mail: jsv@cs.duke.edu, URL: http://www.cs.duke.edu/~jsv.

disk striping can be nonoptimal with respect to I/O, so to gain further improvements we discuss distribution and merging techniques for using the disks independently. We also consider useful techniques for batched EM problems involving matrices (such as matrix multiplication and transposition), geometric data (such as finding intersections and constructing convex hulls), and graphs (such as list ranking, connected components, topological sorting, and shortest paths). In the online domain, canonical EM applications include dictionary lookup and range searching. The two important classes of indexed data structures are based upon extendible hashing and B-trees. The paradigms of filtering and bootstrapping provide a convenient means in online data structures to make effective use of the data accessed from disk. We also reexamine some of the above EM problems in slightly different settings, such as when the data items are moving, when the data items are variable-length (e.g., text strings), or when the allocated amount of internal memory can change dynamically. Programming tools and environments are available for simplifying the EM programming task. During the course of the survey, we report on some experiments in the domain of spatial databases using the TPIE system (transparent parallel I/O programming environment). 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 lowest level, 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 higher level, inexpensive 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. Programmers have a natural tendency 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 the 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), and accessing internal memory takes tens of nanoseconds, but the latency of accessing data from a disk is several milliseconds ($10^{-3}$ seconds), which is about one million times slower! In applications that process massive amounts of data, the input/output communication (or simply I/O) between levels of memory is often the bottleneck.

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 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 [Denning 1980]. 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 the cache and must be

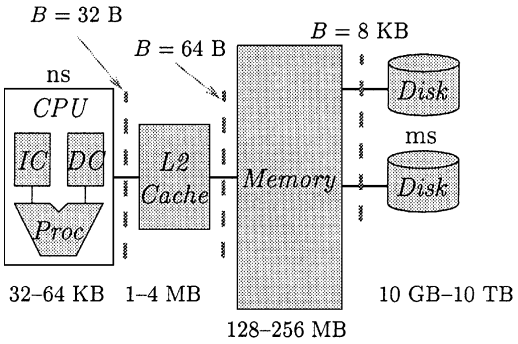


**Fig. 1**. The memory hierarchy of a typical uniprocessor system, including registers, instruction cache, data cache (level 1 cache), level 2 cache, internal memory, and disks. Below each memory level is the range of typical sizes for that memory level. Each value of $B$ at the top of the figure denotes the block transfer size between two adjacent levels of the hierarchy. All sizes are given in units of bytes (B), kilobytes (KB), megabytes (MB), gigabytes (GB), or terabytes (TB). (In the PDM model described in Section 2, we measure $B$ in units of items rather than in units of bytes.) In this figure, 8 KB is the indicated physical block transfer size between internal memory and the disks. However, in batched applications it is often more appropriate to use a substantially larger logical block transfer size.

retrieved by an I/O from a higher level of memory. For example, in a page fault, an I/O is needed to retrieve a disk page from disk and bring it into internal memory.

Caching and prefetching methods are typically designed to be general-purpose, and thus they cannot be expected to take full advantage of the locality present in every computation. Some computations themselves are inherently nonlocal, 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 survey 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 the disks.

### 1.2. Overview

In this article, we survey several paradigms for exploiting locality and thereby reducing I/O costs when solving problems 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 by streaming the data through the internal memory in one or more passes; and

(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 upon 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 the number of I/O operations, the disk space usage, and the CPU time. For reasons of brevity, we focus on the first two measures. Most of the algorithms we consider are also efficient in terms of CPU time. In Section 3, we list four fundamental I/O bounds that pertain to most of the problems considered here. In Section 4, we show why it is crucial for EM algorithms to exploit locality, and we

discuss an automatic load balancing technique called disk striping for using multiple disks in parallel.

In Section 5, 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. Sorting with a single disk is now well understood, so we concentrate on the more challenging task of using multiple (or parallel) disks, for which disk striping is nonoptimal. The challenge is to guarantee that the data in each I/O are spread evenly across the disks so that the disks can be used simultaneously. We also cover the fundamental lower bounds on the number of I/Os needed to perform sorting and related batched problems. In Section 6, we discuss grid and linear algebra batched computations.

For most problems, parallel disks can be utilized effectively by means of disk striping or the parallel disk techniques of Section 5, and hence we restrict ourselves starting in Section 7 to the conceptually simpler single-disk case. In Section 7, 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 (for 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), 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 8, we look at EM algorithms for combinatorial problems on graphs, such as list ranking, connected components, topological sorting, and finding shortest paths. One technique for constructing I/O-efficient EM algorithms is to simulate parallel algorithms; sorting is used between parallel steps in order to reblock the data for the simulation of the next parallel step.

In Sections 9 to 11, we consider data structures in the online setting. The dynamic dictionary operations of insert, delete, and lookup can be implemented by the well-known method of hashing. In Section 9, we examine hashing in external memory, in which extra care must be taken to pack data into blocks and to allow the number of items to vary dynamically. Lookups can be done generally with only one or two I/Os. Section 10 begins with a discussion of B-trees, the most widely used online EM data structure for dictionary operations and 1-D range queries. Weight-balanced B-trees provide a uniform mechanism for dynamically rebuilding substructures and are useful for a variety of online data structures. Level-balanced B-trees permit maintenance of parent pointers and support cut and concatenate operations, which are used in reachability queries on monotone subdivisions. 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. In Section 11, we discuss spatial data structures for multidimensional data, especially those that support online range search. Multidimensional extensions of the B-tree, such as the popular R-tree and its variants, use a linear amount of disk space and often perform well in practice, although their worst-case performance is poor. A nonlinear amount of disk space is required to perform 2-D orthogonal range queries efficiently in the worst case, but several important special cases of range searching can be done efficiently using only linear space. A useful paradigm for developing an efficient EM data structure is to "externalize" an efficient data structure designed for internal memory; a key component of how to make the structure I/O-efficient is to "bootstrap" a static EM data structure for small-sized problems into a fully dynamic data structure of arbitrary size. This paradigm provides optimal linear-space EM data structures for several variants of 2-D orthogonal range search.

In Section 12, we discuss some additional EM approaches useful for dynamic

**Table I.** Paradigms for I/O Efficiency Discussed in this Survey

| Paradigm | Reference |
|---|---|
| Batched dynamic processing | §10.4 |
| Batched filtering | §7 |
| Batched incremental construction | §7 |
| Bootstrapping | §11 |
| Disk striping | §4.2 |
| Distribution | §5.1 |
| Distribution Sweeping | §7 |
| Externalization | §11.3 |
| Fractional Cascading | §7 |
| Filtering | §11 |
| Lazy Updating | §10.4 |
| Load Balancing | §4 |
| Locality | §4 |
| Marriage-before-conquest | §7 |
| Merging | §5.2 |
| Parallel simulation | §8 |
| Persistence | §7 |
| Random sampling | §5.1 |
| Scanning (or streaming) | §2.2 |
| Sparsification | §8 |
| Time-forward processing | §10.4 |

data structures, and we also investigate kinetic data structures, in which the data items are moving. In Section 13, we focus on EM data structures for manipulating and searching text strings.

In Section 14, we discuss programming environments and tools that facilitate high-level development of efficient EM algorithms. We focus on the TPIE system (transparent parallel I/O environment), which we use in the various timing experiments in the article. In Section 15 we discuss EM algorithms that adapt optimally to dynamically changing internal memory allocations. We conclude with some final remarks and observations in Section 16. Table I lists several of the EM paradigms discussed in this survey.

## 2. PARALLEL DISK MODEL (PDM)

EM 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 on the platters in concentric circles 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 the correct track and then wait for the desired address to pass by. The seek time move from one random track to another is often on the order of 3 to 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 contiguous group of data items, called a *block*. Similar 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 and external memory performance. In fact the access gap is growing, since the latency and bandwidth of memory chips are improving more quickly than those of disks. Use of parallel processors further widens the gap. Storage systems such as RAID deploy multiple disks in order to get additional bandwidth [Chen et al. 1994; Hellerstein et al. 1997].

In the next section, we describe the high-level parallel disk model (PDM), which we use throughout this survey for the design and analysis of EM algorithms and data structures. In Section 2.2, we consider some practical modeling issues dealing with the sizes of blocks and tracks and the corresponding parameter values in PDM. In Section 2.3, we review the historical development of models of I/O and hierarchical memory.

### 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 [1994a]:

$N =$ problem size (in units of data items),

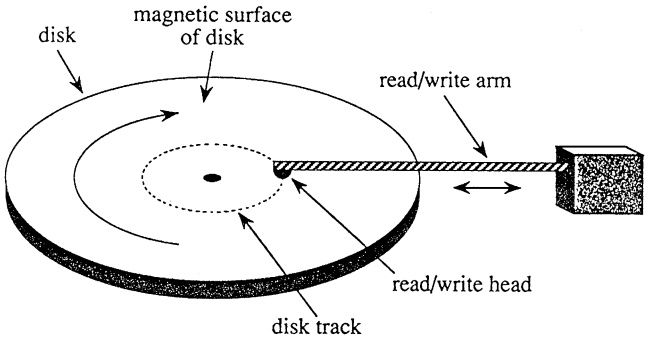


**Fig. 2**. Platter of a magnetic disk drive.

$M =$ internal memory size (in units of data items),
$B =$ block transfer size (in units of data items),
$D =$ number of independent disk drives, and
$P =$ number of CPUs,

where $M < N$, and $1 \leq DB \leq M/2$. The data items are assumed to be of fixed length. 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. For routing considerations, 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 $O((M/P)\log M)$ time.[1] The special cases of PDM for the case of a single processor ($P = 1$) and multiprocessors with one disk per processor ($P = D$) are pictured in Figure 3.

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 7, 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:

$Q =$ number of input queries (for a batched problem), and
$Z =$ query output size (in units of data items).

It is convenient to refer to some of the above PDM parameters in units of disk blocks rather than in units of data items; the resulting formulas are often simplified. We define the lowercase notation

$$n = \frac{N}{B}, \quad m = \frac{M}{B}, \quad q = \frac{Q}{B}, \quad z = \frac{Z}{B} \qquad (1)$$

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.

[1] We use the notation $\log n$ to denote the binary (base 2) logarithm $\log_2 n$. For bases other than 2, the base is specified explicitly.

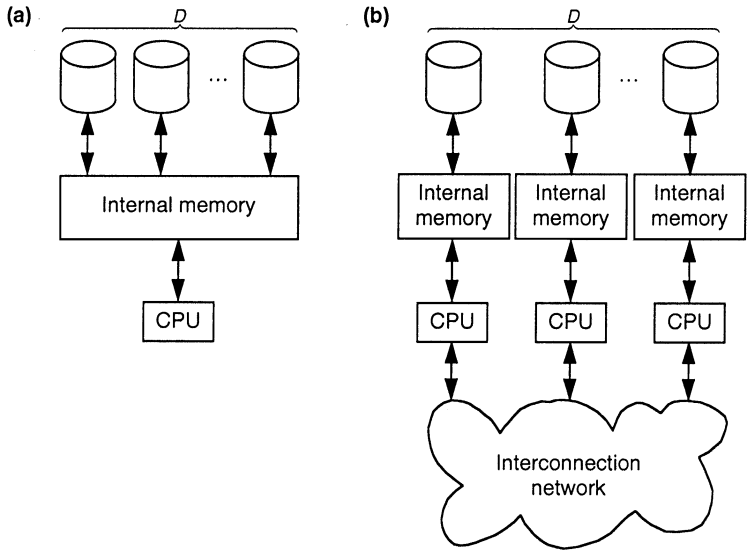


**Fig. 3**. Parallel disk model: (a) $P = 1$, in which the $D$ disks are connected to a common CPU; (b) $P = D$, in which each of the $D$ disks is connected to a separate processor.

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 (sequential or parallel) computation time.

For reasons of brevity in this survey we focus 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.

### 2.2. Practical Modeling Considerations

Track size is a fixed parameter of the disk hardware; for most disks it is in the range 50 to 200 KB. In reality, the track size for any given disk depends upon the radius of the track (cf. Figure 2). Sets of adjacent tracks are usually formatted to have the same track size, so there are typically only a small number of different track sizes for a given disk. A single disk can have a 3 : 2 variation in track size (and therefore bandwidth) between its outer and the inner tracks.

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 (and preferable in batched applications) to use logical blocks of larger size (sometimes called clusters) 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 PDM parameter $B$ to be five times larger than the track size, so that each logical block corresponds to five contiguous tracks, the time per I/O will correspond to five revolutions of the disk plus the (now relatively less significant) seek time and rotational latency. If the disk is smart enough, rotational latency can even be avoided altogether, since the block spans entire tracks and reading can begin as soon as the read head reaches the desired track. 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, especially when the data are streamed

| | $\mathcal{D}_0$ | $\mathcal{D}_1$ | $\mathcal{D}_2$ | $\mathcal{D}_3$ | $\mathcal{D}_4$ |
|---|---|---|---|---|---|
| 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 |

**Fig. 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 $\mathcal{D}_3$.

sequentially through internal memory, the block transfer size $B$ in PDM should be considered to be a fixed hardware parameter a little larger than the track size (say, on the order of 100 KB for most disks), and the time per I/O should be adjusted accordingly. For online applications that use pointer-based indexes, a smaller $B$ value such as 8 KB is appropriate, as in Figure 1. The particular block size that optimizes performance 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 14. More complex and precise disk models, such as the ones by Ruemmler and Wilkes [1994], Ganger [1995], Shriver et al. [1998], Barve et al. [1999], and Farach et al. [1998], distinguish between sequential reads and random reads and consider the effects 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. For example, algorithms for spatial join that access preexisting index structures (and thus do random I/O) can often be slower in practice than algorithms that access substantially more data but in a sequential order (as in streaming) [Arge et al. 2000]. It is thus helpful not only to consider the number of block transfers, but also to distinguish between the I/Os that are random versus those that are sequential. In some applications, automated dynamic block placement can improve disk locality and help reduce I/O time [Seltzer et al. 1995].

Another simplification of PDM is that the $D$ block transfers in each I/O are *synchronous*; they are assumed to take the same amount of time. This assumption makes it easier to design and analyze algorithms for multiple disks. In practice, however, if the disks are used independently, some block transfers will complete more quickly than others. We can often improve overall elapsed time if the I/O is done *asynchronously*, so that disks get utilized as soon as they become available. Buffer space in internal memory can be used to queue the read and write requests for each disk.

### 2.3. Related Memory Models, Hierarchical Memory, and Caching

The study of problem complexity and algorithm analysis when using EM devices began more than 40 years ago with Demuth's PhD on sorting [Demuth 1956; Knuth 1998]. In the early 1970s, Knuth [1998] did an extensive study of sorting using magnetic tapes and (to a lesser extent) magnetic disks. At about the same time, Floyd [1972] and Knuth [1998] considered a disk model akin to PDM for $D = 1$, $P = 1$, $B = M/2 = \Theta(N^c)$, for constant $c > 0$, and developed optimal upper and lower I/O bounds for sorting and matrix transposition. Hong and Kung [1981] developed a pebbling model of I/O for straightline computations, and Savage and Vitter [1987] extended the model to deal with block transfer. Aggarwal and Vitter [1988] generalized Floyd's I/O model to allow $D$ simultaneous block transfers, but the model was unrealistic in that the $D$ 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. In Section 5.3 we discuss a recent simulation technique due to Sanders et al. [2000]; the Aggarwal–Vitter model can be simulated probabilistically by PDM with only a constant factor more I/Os, thus making the two models theoretically equivalent

in the randomized sense. Deterministic simulations on the other hand require a factor of $\log(N/D)/\log\log(N/D)$ more I/Os [Armen 1996].

Surveys of I/O models, algorithms, and challenges appear in Arge [1997], Gibson et al. [1996], and Shriver and Nodine [1996]. Several versions of PDM have been developed for parallel computation [Dehne et al. 1999; Li et al. 1996; Sibeyn and Kaufmann 1997]. Models of "active disks" augmented with processing capabilities to reduce data traffic to the host, especially during streaming applications, are given in Acharya et al. [1998] and Riedel et al. [1998]. Models of microelectromechanical systems (MEMS) for mass storage appear in Griffin et al. [2000].

Some authors have studied problems that can be solved efficiently by making only one pass (or a small number of passes) over the data [Feigenbaum et al. 1999; Henzinger et al. 1999]. One approach to reduce the internal memory requirements is to require only an approximate answer to the problem; the more memory available, the better the approximation. A related approach to reducing I/O costs for a given problem is to use random sampling or data compression in order to construct a smaller version of the problem whose solution approximates the original. These approaches are highly problem-dependent and somewhat orthogonal to our focus in this survey.

The same type of bottleneck that occurs between internal memory (DRAM) and external disk storage can also occur at other levels of the memory hierarchy, such as between registers and level 1 cache, between level 1 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 [Aggarwal et al. 1987a,b; Vitter and Shriver 1994b; Vitter and Nodine 1993]. Conversely, the algorithms for hierarchical models can be run in the PDM setting, and in that setting many have the interesting property that they use no explicit knowledge of the PDM parameters like $M$ and $B$. Frigo et al. [1999] and Bender et al. [2000] develop cache-oblivious algorithms and data structures that require no knowledge of the storage parameters.

However, the match between theory and practice is harder to establish for hierarchical models and caches than for disks. The simpler hierarchical models are less accurate, and the more practical models are architecture-specific. The relative memory sizes and block sizes of the levels vary from computer to computer. Another issue is how blocks from one memory level are stored in the caches at a lower level. When a disk block is read into internal memory, it can be stored in any specified DRAM location. However, in level 1 and level 2 caches, each item can only be stored in certain cache locations, often determined by a hardware modulus computation on the item's memory address. The number of possible storage locations in cache for a given item is called the level of associativity. Some caches are direct-mapped (i.e., with associativity 1), and most caches have fairly low associativity (typically at most 4).

Another reason why the hierarchical models tend to be more architecture-specific is that the relative difference in speed between level 1 cache and level 2 cache or between level 2 cache and DRAM is orders of magnitude smaller than the relative difference in latencies between DRAM and the disks. Yet, it is apparent that good EM design principles are useful in developing cache-efficient algorithms. For example, sequential internal memory access is much faster than random access, by about a factor of 10, and the more we can build locality into an algorithm, the faster it will run in practice. By properly engineering the "inner loops," a programmer can often significantly speed up the overall running time. Tools such as simulation environments and system monitoring utilities [Knuth 1999; Rosenblum et al. 1997; Srivastava and Eustace 1994]

**Table II.** I/O Bounds for the Four Fundamental Operations. The PDM Parameters Are Defined in Section 2.1

| Operation | I/O Bound, $D = 1$ | I/O Bound, General $D \geq 1$ |
|---|---|---|
| $Scan(N)$ | $\Theta\left(\frac{N}{B}\right) = \Theta(n)$ | $\Theta\left(\frac{N}{DB}\right) = \Theta\left(\frac{n}{D}\right)$ |
| $Sort(N)$ | $\Theta\left(\frac{N}{B}\log_{M/B}\frac{N}{B}\right)$ $= \Theta(n\log_m n)$ | $\Theta\left(\frac{N}{DB}\log_{M/B}\frac{N}{B}\right)$ $= \Theta\left(\frac{n}{D}\log_m n\right)$ |
| $Search(N)$ | $\Theta(\log_B N)$ | $\Theta(\log_{DB} N)$ |
| $Output(Z)$ | $\Theta\left(\max\left\{1, \frac{Z}{B}\right\}\right)$ $= \Theta(\max\{1, z\})$ | $\Theta\left(\max\left\{1, \frac{Z}{DB}\right\}\right)$ $= \Theta\left(\max\left\{1, \frac{z}{D}\right\}\right)$ |

can provide sophisticated help in the optimization process.

For reasons of focus, we do not consider such hierarchical models and caching issues in this survey. We refer the reader to the following references. Aggarwal et al. [1987a] define an elegant hierarchical memory model, and Aggarwal et al. [1987b] augment it with block transfer capability. Alpern et al. [1994] model levels of memory in which the memory size, block size, and bandwidth grow at uniform rates. Vitter and Shriver [1994b] and Vitter and Nodine [1993] discuss parallel versions and variants of the hierarchical models. The parallel model of Li et al. [1996] also applies to hierarchical memory. Savage [1995] gives a hierarchical pebbling version of Savage and Vitter [1987]. Carter and Gatlin [1998] define pebbling models of nonassociative direct-mapped caches. Rahman and Raman [2000] and Sen and Chatterjee [2000] apply EM techniques to models of caches and translation lookaside buffers. Rao and Ross [1999; 2000] use B-tree techniques to exploit locality for the design of cache-conscious search trees.

## 3. FUNDAMENTAL I/O OPERATIONS AND BOUNDS

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

(1) *scanning* (a.k.a. *streaming* or *touching*) a file of $N$ data items, which involves the sequential reading or writing of the items in the file;

(2) *sorting* a file of $N$ data items, which puts the items into sorted order;

(3) *searching* online through $N$ sorted data items; and

(4) *outputting* the $Z$ answers to a query in a blocked "output-sensitive" fashion.

We give the I/O bounds for these operations in Table II. We single out the special case of a single disk ($D = 1$), since the formulas are simpler and many of the discussions in this survey are restricted to the single-disk case.

The first two of these I/O bounds–$Scan(N)$ and $Sort(N)$–apply to batched problems. The last two I/O bounds–$Search(N)$ and $Output(Z)$–apply to online problems and are typically combined into the form $Search(N) + Output(Z)$. As mentioned in Section 2.1, some batched problems also involve queries, in which case the I/O bound $Output(Z)$ may be relevant to them as well. In some pipelined contexts, the $Z$ answers to a query do not need to be output to disk but rather can be "piped" to another process, in which case there is no I/O cost for output. Relational database queries are often processed in such a pipeline fashion. For simplicity, in this article we explicitly consider the output cost for queries.

The I/O bound $Scan(N) = O(n/D)$, which is clearly required to read or write a file of $N$ items, represents 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 discuss later include permuting, transposing a matrix, list ranking, and several combinatorial graph problems. Many of these problems are equivalent in I/O complexity to permuting or sorting.

The linear I/O bounds for $Scan(N)$ and $Output(Z)$ are trivial. The algorithms and lower bounds for $Sort(N)$ and $Search(N)$ are relatively new and are discussed in later sections. As Table II indicates, the multiple-disk I/O bounds for $Scan(N)$, $Sort(N)$, and $Output(Z)$ are $D$ times smaller than the corresponding single-disk I/O bounds; such a speedup is clearly the best improvement possible with $D$ disks. For $Search(N)$, the speedup is less significant: the I/O bound $\Theta(\log_B N)$ for $D = 1$ becomes $\Theta(\log_{DB} N)$ for $D \geq 1$; the resulting speedup is only $\Theta((\log_B N)/\log_{DB} N) = \Theta((\log DB)/\log B) = \Theta(1 + (\log D)/\log B)$, which is typically less than 2.

In practice, the logarithmic terms $\log_m n$ in the $Sort(N)$ bound and $\log_{DB} N$ in the $Search(N)$ bound are small constants. For example, in units of items, we could have $N = 10^{10}$, $M = 10^7$, and $B = 10^4$, and thus we get $n = 10^6$, $m = 10^3$, and $\log_m n = 2$, in which case sorting can be done in a linear number of I/Os. If memory is shared with other processes, the $\log_m n$ term will be somewhat larger, but still bounded by a constant. In on-line applications, a smaller $B$ value, such as $B = 10^2$, is more appropriate, as explained in Section 2.2. The corresponding value of $\log_B N$ for the example is 5, so even with a single disk, online search can be done in a small constant number of I/Os.

It still makes sense to explicitly identify terms like $\log_m n$ and $\log_B N$ in the I/O bounds and not hide them within the big-oh or big-theta factors, since the terms can have a significant effect in practice. (Of course, it is equally important to consider any other constants hidden in big-oh and big-theta notations!) The nonlinear I/O bound $\Theta(n \log_m n)$ 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 suggested in Section 2.3, 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 $n$ I/Os will likely run at least an order of magnitude faster in hierarchical memory than would a multilevel algorithm generated from a PDM algorithm that does $n \log_m n$ I/Os [Vitter and Shriver 1994b].

## 4. EXPLOITING LOCALITY AND LOAD BALANCING

The key to achieving efficient I/O performance in EM applications is to design the application to access its data with a high degree of locality. Since each read I/O operation transfers a block of $B$ items, we make optimal use of that read operation when all $B$ items are needed by the application. A similar remark applies to write operations. An orthogonal form of locality more akin to load balancing arises when we use multiple disks, since we can transfer $D$ blocks in a single I/O only if the $D$ blocks reside on distinct disks.

An algorithm that does not exploit locality can be reasonably efficient when it is run on data sets that fit in internal memory, but it will perform miserably when deployed naively in an EM setting and virtual memory is used to handle page management. Examining such performance degradation is a good way to put the I/O bounds of Table II into perspective. In Section 4.1, we examine this phenomenon for the single-disk case, when $D = 1$.

In Section 4.2, we look at the multiple-disk case and discuss the important paradigm of *disk striping* [Kim 1986; Salem and Garcia-Molina 1986], for automatically converting a single-disk

algorithm into an algorithm for multiple disks. Disk striping can be used to get optimal multiple-disk I/O algorithms for three of the four fundamental operations in Table II. The only exception is sorting. The optimal multiple-disk algorithms for sorting require more sophisticated load balancing techniques, which we cover in Section 5.

### 4.1. Locality Issues with a Single Disk

A good way to appreciate the fundamental I/O bounds in Table II is to consider what happens when an algorithm does not exploit locality. For simplicity, we restrict ourselves in this section to the single-disk case $D = 1$. For many of the batched problems we look at in this survey, such as sorting, FFT, triangulation, and computing convex hulls, it is well known how to write programs to solve the corresponding internal memory versions of the problems in $O(N \log N)$ CPU time. But if we execute such a program on a data set that does not fit in internal memory, relying upon virtual memory to handle page management, the resulting number of I/Os may be $\Omega(N \log n)$, which represents a severe bottleneck. Similarly, in the online setting, many types of search queries, such as range search queries and stabbing queries, can be done using binary trees in $O(\log N + Z)$ query CPU time when the tree fits into internal memory, but the same data structure in an external memory setting may require $\Omega(\log N + Z)$ I/Os per query.

We would like instead to incorporate locality *directly* into the algorithm design and achieve the desired I/O bounds of $O(n \log_m n)$ for the batched problems and $O(\log_B N + z)$ for online search, in line with the fundamental bounds listed in Table II. At the risk of oversimplifying, we can paraphrase the goal of EM algorithm design for batched problems 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 $n$ and $z$, and so that the base of the logarithm terms is not 2 but instead $m$. For on-line problems, we want the base of the logarithm to be $B$ and to replace $Z$ by $z$. The relative speedup in I/O performance can be very significant, both theoretically and in practice. For example, for batched problems, the I/O performance improvement can be a factor of $(N \log n)/n \log_m n = B \log m$, which is extremely large. For on-line problems, the performance improvement can be a factor of $(\log N + Z)/(\log_B N + z)$; this value is always at least $(\log N)/\log_B N = \log B$, which is significant in practice, and can be as much as $Z/z = B$ for large $Z$.

### 4.2. Disk Striping for Multiple Disks

It is conceptually much simpler to program for the single-disk case ($D = 1$) than for the multiple-disk case ($D \geq 1$). *Disk striping* [Kim 1986; Salem and Garcia-Molina 1986] is a practical paradigm that can ease the programming task with multiple disks: I/Os are permitted only on entire stripes, one stripe at a time. For example, in the data layout in Figure 4, data items 20 to 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 $DB$.

We can thus apply the paradigm of disk striping to automatically convert an algorithm designed to use a single disk with block size $DB$ into an algorithm for use on $D$ disks each with block size $B$: in the single-disk algorithm, each I/O step transmits one block of size $DB$; in the $D$-disk algorithm, each I/O step consists of $D$ simultaneous block transfers of size $B$ each. The number of I/O steps in both algorithms is the same; in each I/O step, the $DB$ items transferred by the two algorithms are identical. Of course, in terms of wall clock time, the I/O step in the multiple-disk algorithm will be $\Theta(D)$ times faster than in the single-disk algorithm because of parallelism.

Disk striping can be used to get optimal multiple-disk algorithms for three of the four fundamental operations of Section 3–streaming, online search, and output reporting–but it is nonoptimal

for sorting. To see why, consider what happens if we use the technique of disk striping in conjunction with an optimal sorting algorithm for one disk, such as merge sort [Knuth 1998]. The optimal number of I/Os to sort using one disk with block size $B$ is

$$\Theta(n \log_m n) = \Theta\left(n \frac{\log n}{\log m}\right) = \Theta\left(\frac{N}{B} \frac{\log(N/B)}{\log(M/B)}\right). \quad (2)$$

With disk striping, the number of I/O steps is the same as if we use a block size of $DB$ in the single-disk algorithm, which corresponds to replacing each $B$ in (2) by $DB$, which gives the I/O bound

$$\Theta\left(\frac{N}{DB} \frac{\log(N/DB)}{\log(M/DB)}\right) = \Theta\left(\frac{n}{D} \frac{\log(n/D)}{\log(m/D)}\right). \quad (3)$$

On the other hand, the optimal bound for sorting is

$$\Theta\left(\frac{n}{D} \log_m n\right) = \Theta\left(\frac{n}{D} \frac{\log n}{\log m}\right). \quad (4)$$

The striping I/O bound (3) is larger than the optimal sorting bound (4) by a multiplicative factor of

$$\frac{\log(n/D)}{\log n} \frac{\log m}{\log(m/D)} \approx \frac{\log m}{\log(m/D)}. \quad (5)$$

When $D$ is on the order of $m$, the $\log(m/D)$ term in the denominator is small, and the resulting value of (5) is on the order of $\log m$, which can be significant in practice.

It follows that the only way theoretically to attain the optimal sorting bound (4) is to forsake disk striping and to allow the disks to be controlled *independently*, so that each disk can access a different stripe in the same I/O step. Actually, the only requirement for attaining the optimal bound is that either reading or writing is done independently. It suffices, for example, to do only read operations independently and to use disk striping for write operations. An advantage of using striping for write operations is that it facilitates the writing of parity information for error correction and recovery, which is a big concern in RAID systems. (We refer the reader to Chen et al. [1994] and Hellerstein et al. [1994] for a discussion of RAID and error correction issues.)

In practice, sorting via disk striping can be more efficient than complicated techniques that utilize independent disks, especially when $D$ is small, since the extra factor $(\log m)/\log(m/D)$ of I/Os due to disk striping may be less than the algorithmic and system overhead of using the disks independently [Vengroff and Vitter 1996b]. In the next section we discuss algorithms for sorting with multiple independent disks. The techniques that arise can be applied to many of the batched problems addressed later in the article. Two such sorting algorithms–distribution sort with randomized cycling and simple randomized merge sort–have relatively low overhead and will outperform disk-striped approaches.

## 5. 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 [Knuth 1998], and also because sorting is an important paradigm in the design of efficient EM algorithms, as we show in Section 8. 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 5.1 [AGGARWAL AND VITTER 1988; NODINE AND VITTER 1995]. *The average-case and worst-case number of I/Os required for sorting $N = nB$ data*

*items using D disks is*

$$Sort(N) = \Theta\left(\frac{n}{D}\log_m n\right). \qquad (6)$$

We saw in Section 4.2 how to construct efficient sorting algorithms for multiple disks by applying the disk striping paradigm to an efficient single-disk algorithm. But in the case of sorting, the resulting multiple-disk algorithm does not meet the optimal $Sort(N)$ bound of Theorem 5.1. In Sections 5.1 and 5.2, we discuss some recently developed external sorting algorithms that use disks independently. The algorithms are based upon the important *distribution* and *merge* paradigms, which are two generic approaches to sorting. The SRM method and its variants [Barve et al. 1997; Barve and Vitter 1999a; Sanders 2000], which are based upon a randomized merge technique, outperform disk striping in practice for reasonable values of $D$. All the algorithms use online load balancing strategies so that the data items accessed in an I/O operation are evenly distributed on the $D$ disks. The same techniques can be applied to many of the batched problems we discuss later in this survey.

All the methods we cover for parallel disks, with the exception of Greed Sort in Section 5.2, provide efficient support for writing redundant parity information onto the disks for purposes of error correction and recovery. For example, some of the methods access the $D$ disks independently during parallel read operations, but in a striped manner during parallel writes. As a result, if we write $D-1$ blocks at a time, the exclusive-OR of the $D-1$ blocks can be written onto the $D$th disk during the same write operation.

In Section 5.3, we show that, if we allow independent reads and writes, we can probabilistically simulate any algorithm written for the Aggarwal–Vitter model discussed in Section 2.3 by use of PDM with the same number of I/Os, up to a constant factor.

In Section 5.4, we consider the situation in which the items in the input file do not have unique keys. In Sections 5.5 and 5.6, we consider problems related to sorting, such as permuting, permutation networks, transposition, and fast Fourier transform. In Section 5.7, we give lower bounds for sorting and related problems.

### 5.1. Sorting by Distribution

*Distribution* sort [Knuth 1998] is a recursive process in which we use a set of $S-1$ partitioning elements to partition the items into $S$ disjoint buckets. All the items in one bucket precede all the items in the next bucket. We complete the sort by recursively sorting the individual buckets and concatenating them to form a single fully sorted list.

One requirement is that we choose the $S-1$ partitioning elements so that the buckets are of roughly equal size. When that is the case, the bucket sizes decrease from one level of recursion to the next by a relative factor of $\Theta(S)$, and thus there are $O(\log_S n)$ levels of recursion. During each level of recursion, we scan the data. As the items stream through internal memory, they are partitioned into $S$ buckets in an online manner. When a buffer of size $B$ fills for one of the buckets, its block is written to the disks in the next I/O, and another buffer is used to store the next set of incoming items for the bucket. 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 [Aggarwal and Vitter 1988; Nodine and Vitter 1993; Vitter and Shriver 1994a], which has the effect of doubling the number of levels of recursion. Probabilistic methods based upon random sampling can be found in Feller [1968]. A deterministic algorithm for the related problem of (exact) selection (i.e., given $k$, find the $k$th item in the file in sorted order) appears in Sibeyn [1999].

In order to meet the sorting bound (6), we must form the buckets at each level

of recursion using $O(n/D)$ 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 is 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 in Section 4.2, full striping (in which $C = D$) can be nonoptimal.

Vitter and Shriver [1994a] develop two randomized online techniques for the partitioning so that with high probability each bucket will be well balanced across the $D$ disks. In addition, they use partial striping in order to fit in internal memory the pointers needed to keep track of the layouts of the buckets on the disks. Their first partitioning technique applies 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 independent random order to a disk stripe, with all $D!$ orders equally likely. At the end of the partitioning, with high probability each bucket is evenly distributed among the disks. This situation is intuitively analogous to the *classical occupancy problem*, in which $b$ balls are inserted independently and uniformly at random into $d$ bins. It is well known that if the load factor $b/d$ grows asymptotically faster than $\log d$, the most densely populated bin contains $b/d$ balls asymptotically on the average, which corresponds to an even distribution. However if the load factor $b/d$ is 1, the largest bin contains $(\ln d)/\ln \ln d$ balls, whereas an average bin contains only one ball [Vitter and Flajolet 1990]. Intuitively, the blocks in a bucket act as balls and the disks act as bins. In our case, the parameters correspond to $b = \Omega(d \log d)$, which suggests that the blocks in the bucket should be evenly distributed among the disks.

By further analogy to the occupancy problem, if the number of blocks per bucket is not $\Omega(D \log D)$, then the technique breaks down and the distribution of each bucket among the disks tends to be uneven, causing a bottleneck for I/O operations. For these smaller values of $N$, Vitter and Shriver [1994a] use their second partitioning technique: the file is read in one pass, one memoryload at a time. Each memoryload is independently and 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. Vitter and Shriver [1994a] 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.

DeWitt et al. [1991] 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.

An even better way to do distribution sort, and deterministically at that, is the BalanceSort method developed by Nodine and Vitter [1993]. During the partitioning process, the algorithm keeps track of how evenly each bucket has been distributed so far among the disks. It maintains an invariant that guarantees good distribution across the disks for each bucket. For

each bucket $1 \le b \le S$ and disk $1 \le d \le D$, 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, $num_b = \sum_{1 \le d \le D} num_b(d)$. By application of matching techniques from graph theory, the BalanceSort algorithm is guaranteed to write at least half of any given memoryload to the disks in a blocked manner and still maintain the invariant for each bucket $b$ that the $\lfloor D/2 \rfloor$ largest values among $num_b(1)$, $num_b(2)$, ..., $num_b(D)$ differ by at most 1. As a result, each $num_b(d)$ is at most about twice the ideal value $num_b/D$, which implies that the number of I/Os needed to read a bucket into memory during the next level of recursion will be within a small constant factor of optimal.

The distribution sort methods that we mentioned above for parallel disks perform write operations in complete stripes, which makes it easy to write parity information for use in error correction and recovery. But since the blocks written in each stripe typically belong to multiple buckets, the buckets themselves will not be striped on the disks, and we must use the disks independently during read operations. In the write phase, each bucket must therefore keep track of the last block written to each disk so that the blocks for the bucket can be linked together.

An orthogonal approach is to stripe the contents of each bucket across the disks so that read operations can be done in a striped manner. As a result, the write operations must use disks independently, since during each write, multiple buckets will be writing to multiple stripes. Error correction and recovery can still be handled efficiently by devoting to each bucket one block-sized buffer in internal memory. The buffer is continuously updated to contain the exclusive-OR (parity) of the blocks written to the current stripe, and after $D - 1$ blocks have been written, the parity information in the buffer can be written to the final ($D$th) block in the stripe.

Under this new scenario, the basic loop of the distribution sort algorithm is, as before, to read one memoryload at a time and partition the items into $S$ buckets. However, unlike before, the blocks for each individual bucket will reside on the disks in contiguous stripes. Each block therefore has a predefined place where it must be written. If we choose the normal round-robin ordering for the stripes (namely, ..., 1, 2, 3, ..., $D$, 1, 2, 3, ..., $D$, ...), the blocks of different buckets may "collide," meaning that they need to be written to the same disk, and subsequent blocks in those same buckets will also tend to collide. Vitter and Hutchinson [2001] solve this problem by the technique of *randomized cycling*. For each of the $S$ buckets, they determine the ordering of the disks in the stripe for that bucket via a random permutation of $\{1, 2, \ldots, D\}$. The $S$ random permutations are chosen independently. If two blocks (from different buckets) happen to collide during a write to the same disk, one block is written to the disk and the other is kept on a write queue. With high probability, subsequent blocks in those two buckets will be written to different disks and thus will not collide. As long as there is a small pool of available buffer space to temporarily cache the blocks in the write queues, Vitter and Hutchinson show that with high probability the writing proceeds optimally.

We expect that the randomized cycling method or the related merge sort methods discussed at the end of Section 5.2 will be the methods of choice for sorting with parallel disks. Experiments are under way to evaluate their relative performance. Distribution sort algorithms may have an advantage over the merge approaches presented in Section 5.2 in that they typically make better use of lower levels of cache in the memory hierarchy of real systems, based upon analysis of distribution sort and merge sort algorithms on models of hierarchical memory, such as the RUMH model of Vitter and Nodine [1993].

### 5.2. Sorting by Merging

The *merge* paradigm is somewhat orthogonal to the distribution paradigm of the previous section. A typical merge sort

algorithm works as follows [Knuth 1998]. In the "run formation" phase, we scan the $n$ blocks of data, one memoryload at a time; we sort each memoryload into a single "run," which we then output onto a series of stripes on the disks. At the end of the run formation phase, there are $N/M = n/m$ (sorted) runs, each striped across the disks. (In actual implementations, we can use the "replacement-selection" technique to get runs of $2M$ data items, on the average, when $M \gg B$ [Knuth 1998].) After the initial runs are formed, the merging phase begins. In each pass of the merging phase, we merge groups of $R$ runs. For each merge, we scan the $R$ runs and merge the items in an online manner as they stream through internal memory. Double buffering is used to overlap I/O and computation. At most $R = \Theta(m)$ runs can be merged at a time, and the resulting number of passes is $O(\log_m n)$.

To achieve the optimal sorting bound (6), we must perform each merging pass 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 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.

For the binary merging case $R = 2$ we can devise a perfect solution, in which the next $D$ blocks needed for the merge are guaranteed to be on distinct disks, based upon the Gilbreath principle [Gardner 1977; Knuth 1998]: we stripe the first run into ascending order by disk number, and we stripe the other run into descending order. Regardless of how the items in the two runs interleave during the merge, it is always the case that we can access the next $D$ blocks needed for the output via a single I/O operation, and thus the amount of internal memory buffer space needed for binary merging is minimized. Unfortunately there is no analogue 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 [1995] 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, 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 $B$ 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 [Leighton 1985] suffices to restore total order; partial striping is employed to meet the memory constraints. One disadvantage of Greed Sort is that the block writes and block reads involve independent disks and are not done in a striped manner, thus making it difficult to write parity information for error correction and recovery.

Aggarwal and Plaxton [1994] developed an optimal deterministic merge sort based upon the Sharesort hypercube parallel sorting algorithm [Cypher and Plaxton 1993]. To guarantee even distribution during the merging, it employs two high-level merging schemes in which the scheduling is almost oblivious. Like Greed Sort, the Sharesort algorithm is theoretically optimal (i.e., within a constant factor of optimal), but the constant factor is larger than the distribution sort methods.

One of the most practical methods for sorting is based upon the *simple randomized merge sort* (SRM) algorithm of Barve et al. [1997] and Barve and Vitter [1999a], referred to as "randomized striping" by Knuth [1998]. Each run is striped

**Table III.** 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, Where $kD \approx M/2B$. The Figures Were Obtained by Simulation

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

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 enough room, the least needed blocks are "flushed" (without any I/Os required) to free up space. Barve et al. [1997] derive an asymptotic upper bound on the expected I/O performance, with no assumptions on the input distribution. A more precise analysis, which is related to the so-called *cyclic occupancy problem*, is an interesting open problem. The cyclic occupancy problem is similar to the classical occupancy problem we discussed in Section 5.1 in that there are $b$ balls distributed into $d$ bins. However, in the cyclical occupancy problem, the $b$ balls are grouped into $c$ chains of length $b_1$, $b_2, \ldots, b_c$, where $\sum_{1 \le i \le c} b_i = b$. Only the head of each chain is randomly inserted into a bin; the remaining balls of the chain are inserted into the successive bins in a cyclic manner (hence the name "cyclic occupancy"). It is conjectured that the expected maximum bin size in the cyclic occupancy problem is at most that of the classical occupancy problem [Barve et al. 1997; Knuth 1998, problem 5.4.9–28]. The bound has been established so far only in an asymptotic sense.

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 III. Experimental confirmation of the speedup was obtained on a 500 megahertz CPU with six fast disk drives, as reported by Barve and Vitter [1999a].

We can get further improvements in merge sort by a more careful prefetching schedule for the runs. Barve et al. [2000] and Kallahalla and Varman [1999, 2000] have developed competitive and optimal methods for prefetching blocks in parallel I/O systems. Hutchinson et al. [2001a, 2001b] have demonstrated a powerful duality between parallel writing and parallel prefetching, which gives an easy way to compute optimal prefetching and caching schedules for multiple disks. More significantly, they show that the same duality exists between distribution and merging, which they exploit to get a provably optimal and very practical parallel disk merge sort. Rather than use random starting points and round-robin stripes as in SRM, Hutchinson et al. [2001a, 2001b] order the stripes for each run independently, based upon the randomized cycling strategy discussed in Section 5.1 for distribution sort.

### 5.3. A General Simulation

Sanders et al. [2000] and Sanders [2001] give an elegant randomized technique to simulate the Aggarwal–Vitter model of Section 2.3, in which $D$ simultaneous block transfers are allowed regardless of where the blocks are located on the disks. On the average, the simulation realizes each I/O in the Aggarwal–Vitter model by only a constant number of I/Os in PDM. One property of the technique is that the read and write steps use the disks independently. Armen [1996] had earlier shown that deterministic simulations resulted in an increase in the number of I/Os by a multiplicative factor of $\log(N/D)/\log\log(N/D)$.

The technique of Sanders et al. [2000] consists of duplicating each disk block and storing the two copies on two independently and uniformly chosen disks (chosen by a hash function). In terms of the occupancy model, each ball (block) is duplicated and stored in two random bins (disks). Let us consider the problem of retrieving a specific set of $D$ blocks from the disks. For each block, there is a choice of two disks from which it can be read.

Regardless of which $D$ blocks are requested, Sanders et al. [2000] show how to choose the copies that permit the $D$ blocks to be retrieved with high probability in only two parallel I/Os. A natural application of this technique is to the layout of data on multimedia servers in order to support multiple stream requests, as in video-on-demand.

When writing blocks of data to the disks, each block must be written to both the disks where a copy is stored. Sanders et al. [2000] prove that with high probability $D$ blocks can be written in $O(1)$ I/O steps, assuming that there are $O(D)$ blocks of internal buffer space to serve as write queues. The read and write bounds can be improved with a corresponding trade-off in redundancy and internal memory space.

### 5.4. Handling Duplicates

Arge et al. [1993] describe a single-disk merge sort algorithm for the problem of *duplicate removal*, in which there are a total of $K$ distinct items among the $N$ items. It runs in $O(n \max\{1, \log_m(K/B)\})$ I/Os, which is optimal in the comparison model. The algorithm can be used to sort the file, assuming that a group of equal items can be represented by a single item and a count.

A harder instance of sorting called *bundle sorting* arises when we have $K$ distinct key values among the $N$ items, but all the items have different secondary information. Abello et al. [1998] and Matias et al. [2000] develop optimal distribution sort algorithms for bundle sorting using $BundleSort(N, K) = O(n \max\{1, \log_m \min\{K, n\}\})$ I/Os, and Matias et al. [2000] prove the matching lower bound. Matias et al. [2000] also show how to do bundle sorting (and sorting in general) *in place* (i.e., without extra disk space). In distribution sort, for example, the blocks for the subfiles can be allocated from the blocks freed up from the file being partitioned; the disadvantage is that the blocks in the individual subfiles are no longer consecutive on the disk. The algorithms can be adapted to run on $D$ disks with a speedup of $O(D)$ using the techniques described in Sections 5.1 and 5.2.

### 5.5. 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 5.2 [AGGARWAL AND VITTER 1998]. *The average-case and worst-case number of I/Os required for permuting $N$ data items using $D$ disks is*

$$\Theta\left(\min\left\{\frac{N}{D}, Sort(N)\right\}\right). \tag{7}$$

The I/O bound (7) for permuting can be realized by using one of the sorting algorithms from Section 5 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 nonblocked 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 Vitter and Shriver [1994a].

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 5.3 [AGGARWAL AND VITTER 1988]. *With $D$ disks, the number of I/Os required to transpose a $p \times q$ matrix from row-major order to column-major order is*

$$\Theta\left(\frac{n}{D} \log_m \min\{M, p, q, n\}\right), \tag{8}$$

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

When $B$ is relatively large (say, $\frac{1}{2}M$) and $N$ is $O(M^2)$, 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 $Ax \oplus c$, where $\oplus$ denotes bitwise exclusive-OR. 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. [1999] 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 5.4 [CORMEN ET AL. 1999]. *With $D$ disks, the number of I/Os required to perform the BMMC permutation defined by matrix $A$ and vector $c$ is*

$$\Theta\left(\frac{n}{D}\left(1 + \frac{\operatorname{rank}(\gamma)}{\log m}\right)\right), \qquad (9)$$

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

An interesting theoretical question is to determine the I/O cost for each individual permutation, as a function of some simple characterization of the permutation, such as number of inversions.

### 5.6. Fast Fourier Transform and Permutation Networks

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 an oblivious (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 [Wu and Feng 1981].

THEOREM 5.5 *With $D$ disks, the number of I/Os required for computing the $N$-input FFT digraph or an $N$-input permutation network is Sort$(N)$.*

Cormen and Nicol [1988] give some practical implementations for one-dimensional FFTs based upon the optimal PDM algorithm of Vitter and Shriver [1994a]. The algorithms for FFT are faster and simpler than for sorting because the computation is nonadaptive in nature, and thus the communication pattern is fixed in advance.

### 5.7. Lower Bounds on I/O

In this section we prove the lower bounds from Theorems 5.1 to 5.5 and mention some related I/O lower bounds for the batched problems in computational geometry and graphs that we cover later in Sections 7 and 8.

The most trivial batched problem is that of scanning (a.k.a. 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 (7) of Theorem 5.2 is due to Aggarwal and Vitter [1988]. The idea of the proof is to calculate, for each $t \geq 0$, the number of distinct orderings that are realizable by sequences 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 as a result of an I/O. In terms of increasing the number of realizable orderings, the effect of reading a disk block is considerably more than that of writing a disk block, so it suffices to consider only the effect of read operations. During a read operation, there are at most $B$ data items in the read block, and they can be interspersed 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.) There are at most $n + t \le N \log N$ ways to choose which disk block is involved in the $t$th 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 all possible sequences of $t$ I/Os is at most

$$(B!)^{N/B} \left( N(\log N) \binom{M}{B} \right)^t. \qquad (10)$$

Setting the expression in (10) to be at least $N!/2$, and simplifying by taking the logarithm, we get

$$N \log B + t \left( \log N + B \log \frac{M}{B} \right) = \Omega(N \log N). \qquad (11)$$

Solving for $t$, we get the matching lower bound $\Omega(n \log_m n)$ for permuting for the case $D = 1$. The general lower bound (7) of Theorem 5.2 follows by dividing by $D$.

We get a stronger lower bound from a more refined argument that counts input operations separately from output operations [Hutchinson et al 2001c]. For the typical case in which $B \log m = \omega(\log N)$, the I/O lower bound, up to lower-order terms, is $2n \log_m n$. For the pathological in which $B \log m = o(\log N)$, the I/O lower bound, up to lower-order terms, is $N/D$.

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 (6) of Theorem 5.1 [Aggarwal and Vitter 1988]. 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 (fixed). Since the choice of disk block is fixed for each $t$, there is no $N \log N$ term as there is in (10), and correspondingly there is no additive $\log N$ term in the inner expression as there is in (11). Hence, when we solve for $t$, we get the lower bound (6) rather than (7). 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 FFT, since permutation networks can be formed from three FFTs in sequence. The transposition lower bound involves a potential argument based upon a togetherness relation [Aggarwal and Vitter 1988].

Arge et al. [1993] show for the comparison model that any problem with an $\Omega(N \log N)$ lower bound in the (internal memory) RAM model requires $\Omega(n \log_m n)$ I/Os in PDM for a single disk. Their argument leads to a matching lower bound of $\Omega(n \max\{1, \log_m(K/B)\})$ I/Os in the comparison model for duplicate removal with one disk.

For the problem of bundle sorting, in which the $N$ items have a total of $K$ distinct key values (but the secondary information of each item is different), Matias et al. [2000] derive the matching

lower bound $BundleSort(N, K) = \Omega(n \max\{1, \log_m \min\{K, n\}\})$. The proof consists of the following parts. The first part is a simple proof of the same lower bound as for duplicate removal, but without resorting to the comparison model (except for the pathological case $B \log m = o(\log n)$). It suffices to set (10) to be at least $N!/((N/K)!)^K$, which is the maximum number of permutations of $N$ numbers having $K$ distinct values. Solving for $t$ gives the lower bound $\Omega(n \max\{1, \log_m(K/B)\})$, which is equal to the desired lower bound for $BundleSort(N, K)$ when $K = B^{1+\Omega(1)}$ or $M = B^{1+\Omega(1)}$. Matias et al. [2000] derive the remaining case of the lower bound for $BundleSort(N, K)$ by a potential argument based upon the transposition lower bound. Dividing by $D$ gives the lower bound for $D$ disks.

Chiang et al. [1995], Arge [1995b], Arge and Miltersen [1999], and Munagala and Ranade [1999] give models and lower bound reductions for several computational geometry and graph problems. The geometry problems discussed in Section 7 are equivalent to sorting in both the internal memory and PDM models. Problems such as list ranking and expression tree evaluation have the same nonlinear I/O lower bound as permuting. Other problems such as connected components, biconnected components, and minimum spanning forests of sparse graphs with $E$ edges and $V$ vertices require as many I/Os as $E/V$ instances of permuting $V$ items. This situation is in contrast with the (internal memory) RAM model, in which the same problems can all be done in linear CPU time. (The known linear-time RAM algorithm for finding a minimum spanning tree is randomized.) In some cases there is a gap between the best-known upper and lower bounds, which we examine further in Section 8.

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 (6) remains valid even if the indivisibility assumption is lifted. However, for an artificial problem related to transposition, Adler [1996] showed that removing the indivisibility assumption can lead to faster algorithms. A similar result is shown by Arge and Miltersen [1999] for the decision problem of determining if $N$ data item values are distinct. Whether the conjecture is true is a challenging theoretical open problem.

## 6. 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 5.5. 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 6.1 [HONG AND KUNG 1981; SAVAGE AND VITTER 1987; VITTER AND SHRIVER 1994a; WOMBLE ET AL. 1993]. *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 $\Theta(K^3/\min\{K, \sqrt{M}\}DB)$.*

Hong and Kung [1981] and Nodine et al. [1991] give optimal EM algorithms for iterative grid computations, and Leiserson et al. [1993] reduce the number of I/Os of naive multigrid implementations by a $\Theta(M^{1/5})$ factor. Gupta et al. [1995] 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 5.3 when $B^2 \leq M$), transposition of sparse matrices is as hard as sorting.

THEOREM 6.2 *For a matrix stored in sparse format and containing $N_z$ nonzero elements, the number of I/Os required to convert the matrix from row-major order to column-major order, and vice versa, is* $\Theta(Sort(N_z))$.

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

For further discussion of numerical EM algorithms we refer the reader to the survey by Toledo [1999]. Some issues regarding programming environments are covered in Corbett et al. [1996] and Section 14.

## 7. BATCHED PROBLEMS IN COMPUTATIONAL GEOMETRY

Problems involving massive amounts of geometric data are ubiquitous in spatial databases [Laurini and Thompson 1992; Samet 1989a,b], geographic information systems (GIS) [Laurini and Thompson 1992; Somet 1989a; Van Kreveld et al. 1997], constraint logic programming [Kanellakis et al. 1990; 1996], object-oriented databases [Zdonik and Maier 1990], statistics, virtual reality systems, and computer graphics [Funkhouser et al. 1992]. NASA's Earth Observing System [1999] project, the core part of the Earth Science Enterprise (formerly Mission to Planet Earth), produces petabytes ($10^{15}$ bytes) of raster data per year. Microsoft's TerraServer [1998] online database of satellite images is over one terabyte in size. A major challenge is to develop mechanisms for processing the data, or else much of the data will be useless.[2]

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 6.3 *Certain batched problems involving $N = nB$ input items, $Q = qB$ queries, and $Z = zB$ output items can be solved using*

$$O((n + q)\log_m n + z) \tag{12}$$

*I/Os with a single disk:*

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

*The parameters $Q$ and $Z$ are set to* 0 *if they are not relevant for the particular problem.*

Goodrich et al. [1993], Zhu [1994], Arge et al. [1995; 1998b], and Crauser et al.

[2] For brevity, in the remainder of this survey 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 $D \geq 1$ by using the load balancing techniques of Section 5. In practice, disk striping (cf. Section 4.2) may be sufficient. For online problems, disk striping will convert optimal bounds for the case $D = 1$ into optimal bounds for $D \geq 1$.

[1998; 1999] develop EM algorithms for those problems using these EM paradigms for batched problems:

*Distribution sweeping*, a generalization of the distribution paradigm of Section 5 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 10.1), yielding a factor of $B$ improvement over the generic persistence techniques of Driscoll et al. [1989].

*Batched filtering*, a general method for performing simultaneous EM searches in data structures that can be modeled as planar layered directed acyclic graphs; it is useful for 3-D convex hulls and batched point location. Multisearch on parallel computers is considered in Dittrich et al. [1988].

*External fractional cascading*, an EM analogue to fractional cascading on a segment tree, in which the degree of the segment tree is $O(m^\alpha)$ for some constant $0 < \alpha \leq 1$. Batched queries can be performed efficiently using batched filtering; online queries can be supported efficiently by adapting the parallel algorithms of the work of Tamassia and Vitter [1996] to the I/O setting.

*External marriage-before-conquest*, an EM analogue to the technique of Kirkpatrick and Seidel [1986] for performing output-sensitive convex hull constructions.

*Batched incremental construction*, a localized version of the randomized incremental construction paradigm of Clarkson and Shor [1989], in which the updates to a simple dynamic data structure are done in a random order, with the goal of fast overall performance on the average. The data structure itself may have bad worst-case performance, but the randomization of the update order makes worst-case behavior unlikely. The key for the EM version so as to gain the factor of $B$ I/O speedup is to batch the incremental modifications.

We focus in the remainder of this section primarily on the distribution sweep paradigm [Goodrich et al. 1993], which is a combination of the distribution paradigm of Section 5.1 and the well-known sweeping paradigm from computational geometry [Preparata and Shamos 1985; de Berg et al. 1997]. As an example, let us consider computing the pairwise intersections of $N$ orthogonal segments in the plane by the following recursive distribution sweep. At each level of recursion, the region under consideration is partitioned into $\Theta(m)$ vertical *slabs*, each containing $\Theta(N/m)$ of the segments' endpoints. We sweep a horizontal line from top to bottom to process the $N$ segments. When the sweep line encounters a vertical segment, we insert the segment into the appropriate slab. When the sweep line encounters a horizontal segment $h$, as pictured in Figure 5, we report $h$'s intersections with all the "active" vertical segments in the slabs that are spanned *completely* by $h$. (A vertical segment is "active" if it intersects the current sweep line; vertical segments that are found to be no longer active are deleted from the slabs.) The remaining two end portions of $h$ (which "stick out" past a slab boundary) are passed recursively to the next level, along with the vertical segments. The downward sweep then proceeds. After the initial sorting (to get the segments with respect to the $y$-dimension), the sweep at each of the $O(\log_m n)$ levels of recursion requires $O(n)$ I/Os, yielding the desired bound (12). Some timing experiments on distribution sweeping appear in Chiang [1998]. Arge et al. [1998b] 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 sweep line algorithm for orthogonal segments

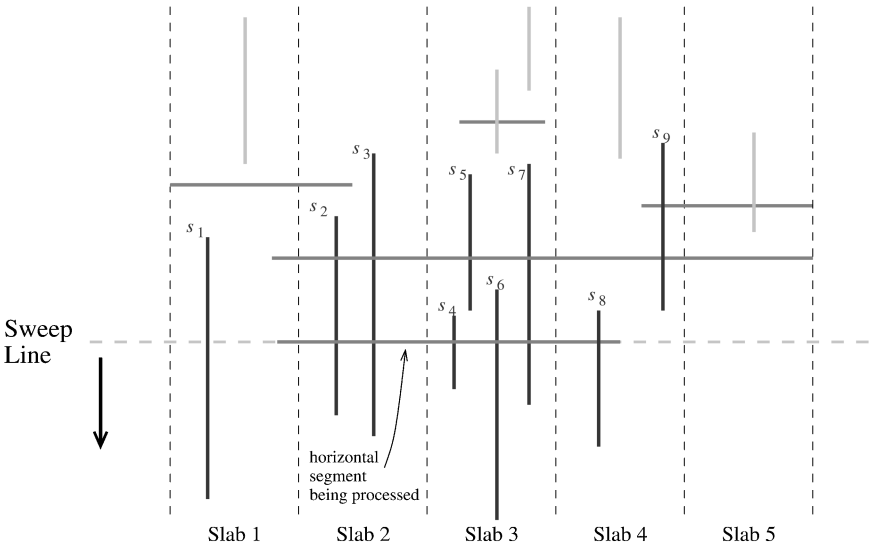


**Fig. 5**. Distribution sweep used for finding intersections among $N$ orthogonal segments. The vertical segments currently stored in the slabs are indicated in bold (namely, $s_1$, $s_2, \ldots, s_9$). Segments $s_5$ and $s_8$ are not active, but have not yet been deleted from the slabs. The sweep line has just advanced to a new horizontal segment that completely spans slabs 2 and 3, so slabs 2 and 3 are scanned and all the active vertical segments in slabs 2 and 3 (namely, $s_2, s_3, s_4, s_6, s_7$) are reported as intersecting the horizontal segment. In the process of scanning slab 3, segment $s_5$ is discovered to be no longer active and can be deleted from slab 3. The end portions of the horizontal segment that "stick out" into slabs 1 and 4 are handled by the lower levels of recursion, where the intersection with $s_8$ is eventually discovered.

with one for range searching. Arge et al. [1998b] take a more unified approach using distribution sweep, which is extendible to higher dimensions: the active objects that are stored in the data structure in this case are rectangles, not vertical segments. The authors choose the branching factor to be $\Theta(\sqrt{m})$. Each rectangle is associated with the largest contiguous range of vertical slabs that it spans. Each of the possible $\Theta(\binom{\sqrt{m}}{2}) = \Theta(m)$ contiguous ranges of slabs is called a *multislab*. The reason why the authors choose the branching factor to be $\Theta(\sqrt{m})$ rather than $\Theta(m)$ is so that the number of multislabs is $\Theta(m)$, and thus there is room in internal memory for a buffer for each multislab. The height of the tree remains $O(\log_m n)$.

The algorithm proceeds by sweeping a horizontal line from top to bottom to process the $N$ rectangles. When the sweep line first encounters a rectangle $R$, we consider the multislab lists for all the multislabs that $R$ intersects. We report all the active rectangles in those multislab lists, since they are guaranteed to intersect $R$. (Rectangles no longer active are discarded from the lists.) We then extract the left and right end portions of $R$ that partially "stick out" past slab boundaries, and we pass them down to process in the next lower level of recursion. We insert the remaining portion of $R$, which spans complete slabs, into the list for the appropriate multislab. The downward sweep then continues. After the initial sorting preprocessing, each of the $O(\log_m n)$ sweeps (one per level of recursion) takes $O(n)$ I/Os, yielding the desired bound (12).

The resulting algorithm, called scalable sweeping-based spatial join (SSSJ) [Arge et al. 1998a; 1998b], outperforms other techniques for rectangle intersection. It was tested against two other sweep line algorithms: the partition-based spatial-merge (QPBSM) used in Paradise

[Patel and Dewitt 1996] and a faster version called MPBSM that uses an improved dynamic data structure for intervals [Arge et al. 1998a]. The TPIE system described in Section 14 served as the common implementation platform. The algorithms were tested on several data sets. The timing results for the two data sets in Figures 6(a) and 6(b) are given in Figures 6(c) and 6(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, but the two PBSM algorithms have the disadvantage of making extra copies of the rectangles. In both cases, SSSJ shows considerable improvement over the PBSM-based methods. In other experiments done on more typical data, such as TIGER/line road data sets [TIGER 1992] SSSJ and MPBSM perform about 30% faster than does QPBSM. The conclusion we draw is that SSSJ is as fast as other known methods on typical data, but unlike other methods, it scales well even for worst-case data. If the rectangles are already stored in an index structure, such as the R-tree index structure we consider in Section 11.2, hybrid methods that combine distribution sweep with inorder traversal often perform best [Arge et al. 2000b].

For the problem of finding all intersections among $N$ line segments, Arge et al. [1995] give an efficient algorithm based upon distribution sort, but the output component of the I/O bound is slightly nonoptimal: $z \log_m n$ rather than $z$. Crauser et al. [1998; 1999] attain the optimal I/O bound (12) by constructing the trapezoidal decomposition for the intersecting segments using an incremental randomized construction. For I/O efficiency, they do the incremental updates in a series of batches, in which the batch size is geometrically increasing by a factor of $m$.

## 8. BATCHED PROBLEMS ON GRAPHS

The first work on EM graph algorithms was by Ullman and Yannakakis [1991] for the problem of transitive closure. Chiang et al. [1995] consider a variety of graph problems, several of which have upper and lower I/O bounds related to sorting and permuting. Abello et al. [1998] formalize a functional approach to EM graph problems, in which computation proceeds in a series of scan operations over the data; the scanning avoids side-effects and thus permits checkpointing to increase reliability. Kumar and Schwabe [1996], followed by Buchsbaum et al. [2000], develop graph algorithms based upon amortized data structures for binary heaps and tournament trees. Munagala and Ranade [1999] give improved graph algorithms for connectivity and undirected breadth-first search, and Arge et al. [2000a] extend the approach to compute the minimum spanning forest (MSF). Meyer [2001] provides some improvements for graphs of bounded degree. Arge [1995b] gives efficient algorithms for constructing ordered binary decision diagrams. Grossi and Italiano [1999] apply their multidimensional data structure to get dynamic EM algorithms for MSF and two-dimensional priority queues (in which the *delete_min* operation is replaced by $delete\_min_x$ and $delete\_min_y$). Techniques for storing graphs on disks for efficient traversal and shortest path queries are discussed in Agarwal et al. [1998b], Goldman et al. [1998], Hutchinson et al. [1999], and Nodine et al. [1996]. Computing wavelet decompositions and histograms [Vitter and Wang 1999; Vitter et al. 1998; Wang et al. 2001] is an EM graph problem related to transposition that arises in online analytical processing (OLAP). Wang et al. [1998] give an I/O-efficient algorithm for constructing classification trees for data mining.

Table IV gives the best known I/O bounds for several graph problems, as a function of the number $V = vB$ of vertices and the number $E = eB$ of edges. The best known I/O lower bound for these problems is $\Omega((E/V)Sort(V) = e \log_m v)$, as mentioned in Section 5.7. A sparsification technique [Eppstein et al. 1997] can often be applied to convert I/O bounds of the form $O(Sort(E))$ to the

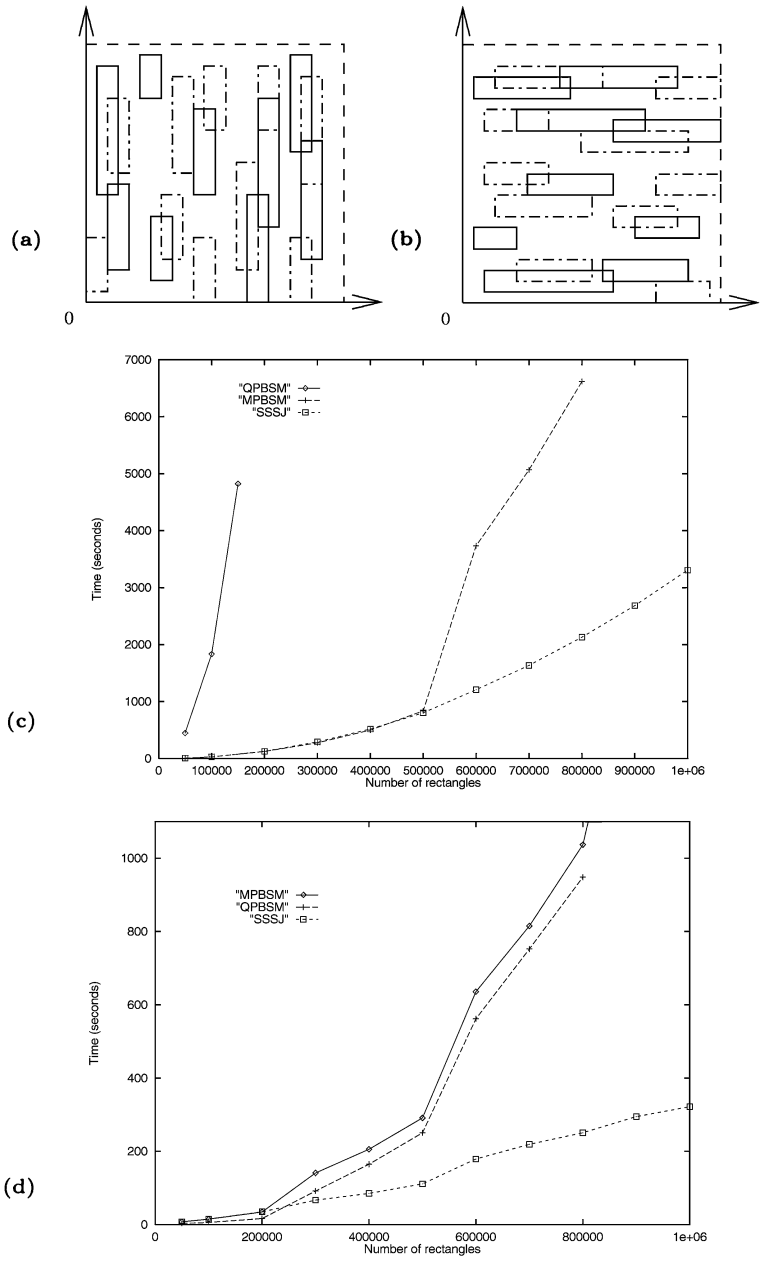


**Fig. 6**. Comparision of scalable sweeping-based spatial join (SSSJ) with the original PBSM (QPBSM) a new variant (MPBSM): (a) data set 1 consist of tall and skinny (verticlally aligned) rectangles; (b) data set 2 consist of short and wide (horizontally aligned) rectangles; (c) running times on data set 1; (d) running times on data set 2.

improved form $O((E/V)Sort(V))$. For example, the actual I/O bounds for connectivity and MSF derived by Munagala and Ranade [1999] and Arge et al. [2000a] are $O(\max\{1, \log\log(V/e)\}Sort(E))$. For the MSF problem, we can partition the edges of the graph into $E/V$ sparse subgraphs on $V$ vertices, and then apply the algorithm

**Table IV.** Best Known I/O Bounds for Batched Graph Problems for the Single-Disk Case $D = 1$. The Number of Vertices is Denoted by $V = vB$ and the Number of Edges by $E = eB$. The Terms *Sort(N)* and *BunSort(N,K)* are Defined in Sections 3 and 5.4. Lower Bounds are Discussed in Section 5.7

| Graph Problem | I/O Bound, $D = 1$ |
|---|---|
| List ranking,<br>Euler tour of a tree,<br>Centroid decomposition,<br>Expression tree evaluation | $\Theta(Sort(V))$ [Chiang et al. 1995] |
| Connected components,<br>Minimum spanning forest (MSF) | $O\left(\max\left\{1,\ \log\log\frac{V}{e}\right\}\frac{E}{V}Sort(V)\right)$<br>[Arge et al. 2000a; Eppstein et al. 1997; Munagala and Ranade 1999] (deterministic)<br>$\Theta\left(\frac{E}{V}Sort(V)\right)$ [Chiang et al. 1995] (randomized) |
| Bottleneck MSF,<br>Biconnected components | $O\left(\min\left\{V^2,\ \max\left\{1,\ \log\frac{V}{M}\right\}\frac{E}{V}Sort(V),\ (\log B)\frac{E}{V}Sort(V) + e\log V\right\}\right)$<br>[Abelo et al. 1998; Chiang et al. 1995; Eppstein et al. 1997; Kumar and Schwabe 1996] (deterministic)<br>$\Theta\left(\frac{E}{V}Sort(V)\right)$ [Chiang et al. 1995; Eppstein et al. 1997] (randomized) |
| Ear decomposition,<br>Maximal matching | $O\left(\min\left\{V^2,\ \max\left\{1,\ \log\frac{V}{M}\right\}Sort(E),\ (\log B)Sort(E) + e\log V\right\}\right)$<br>[Abello et al. 1998; Chiang et al. 1995; Kumar and Schwabe 1996] (deterministic)<br>$O(Sort(E))$ [Chiang et al. 1995] (randomized) |
| Undirected breadth-first search | $O(BundleSort(E, V) + V)$<br>[Munagala and Ranade 1999] |
| Undirected single-source shortest paths | $O(e\log e + V)$ [Kumar and Schwabe 1996] |
| Directed and undirected depth-first search,<br>Topological sorting,<br>Directed breadth-first search,<br>Directed single-source shortest paths | $O\left(\min\left\{\frac{ve}{m} + V,\ (V + e)\log v\right\}\right)$<br>[Buchsbaum et al. 2000; Chiang et al. 1995; Munagala and Schwabe 1996] |
| Transitive closure | $O\left(Vv\sqrt{\frac{e}{m}}\right)$ [Chiang et al. 1995] |

of Arge et al. [2000a] to each subproblem to create $E/V$ spanning forests in a total of $O(\max\{1, \log\log(V/e)\}(E/V)Sort(V))$ I/Os. We can then merge the $E/V$ spanning forests, two at a time, in a balanced binary merging procedure by repeatedly applying the algorithm of Arge et al. [2000a]. After the first level of binary merging, the spanning forests collectively have at most $E/2$ edges; after two levels, they have at most $E/4$ edges, and so on in a geometrically decreasing manner. The total

cost for the final spanning forest is thus $O(\max\{1, \log\log(V/e)\}(E/V)Sort(V))$ I/Os. The reason why sparsification works is that the spanning forest output by each binary merge is only $\Theta(V)$ in size, yet it preserves the necessary information needed for the next merge step. The same approach works for connectivity.

In the case of *semi-external graph problems* [Abello et al. 1998], in which the vertices fit in internal memory but not the edges (i.e., $V \leq M < E$), several of the problems in Table IV can be solved optimally in external memory. For example, finding connected components, biconnected components, and minimum spanning forests can be done in $O(e)$ I/Os when $V \leq M$. The I/O complexities of several problems in the general case remain open, including connected components, biconnected components, and minimum spanning forests in the deterministic case, as well as breadth-first search, topological sorting, shortest paths, depth-first search, and transitive closure. It may be that the I/O complexity for several of these problems is $\Theta((E/V)Sort(V) + V)$. For special cases, such as trees, planar graphs, outerplanar graphs, and graphs of bounded tree width, several of these problems can be solved substantially faster in $O(Sort(E))$ I/Os [Agarwal et al. 1998b; Chiang 1995; Maheshwari and Zeh 1999; 2001].

Chiang et al. [1995] exploit the key idea that efficient EM algorithms can often be developed by a sequential simulation of a parallel algorithm for the same problem. The intuition is that each step of a parallel algorithm specifies several operations and the data upon which they act. If we bring together the data arguments for each operation, which we can do by two applications of sorting, then the operations can be performed by a single linear scan through the data. After each simulation step, we sort again in order to reblock the data into the linear order required for the next simulation step. 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 of the simulation, which is $Sort(N) = \Theta(n \log_m n)$. The optimality of the EM algorithm given in Chiang et al. [1995] for list ranking assumes that $\sqrt{m} \log m = \Omega(\log n)$, which is usually true in practice. That assumption can be removed by use of the buffer tree data structure [Arge 1995a] (see Section 10.4). A practical randomized implementation of list ranking appears in Sibeyn [1997].

Dehne et al. [1997; 1999] and Sibeyn and Kaufmann [1997] use a related approach and get efficient I/O bounds by simulating coarse-grained parallel algorithms in the BSP parallel model. Coarse-grained parallel algorithms may exhibit more locality than the fine-grained algorithms considered in Chiang et al. [1995], and as a result the simulation may require fewer sorting steps. Dehne et al. make certain assumptions, most notably that $\log_m n \leq c$ for some small constant $c$ (or equivalently that $M^c < NB$), so that the periodic sortings can each be done in a linear number of I/Os. Since the BSP literature is well developed, their simulation technique provides efficient single-processor and multiprocessor EM algorithms for a wide variety of problems.

In order for the simulation techniques to be reasonably efficient, the parallel algorithm being simulated must run in $O((\log N)^c)$ time using $N$ processors. Unfortunately, the best known polylog-time algorithms for problems such as depth-first search and shortest paths use a polynomial number of processors, not a linear number. P-complete problems such as lexicographically first depth-first search are unlikely to have polylogarithmic-time algorithms even with 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. It may thus be possible to show by reduction that

certain groups of problems are "equally hard" to solve efficiently in terms of I/O and are thus unlikely to have solutions as fast as sorting.

## 9. EXTERNAL HASHING FOR ONLINE DICTIONARY SEARCH

We now turn our attention to online data structures for supporting the dictionary operations of insert, delete, and lookup. Given a value $x$, the lookup operation returns the item(s), if any, in the structure with key value $x$. The two main types of EM dictionaries are hashing, which we discuss in this section, and tree-based approaches, which we defer until Section 10. The advantage of hashing is that the expected number of probes per operation is a constant, regardless of the number $N$ of items. The common element of all EM hashing algorithms is a predefined hash function

$$hash : \{\text{all possible keys}\} \rightarrow \{0, 1, 2, \ldots, K-1\}$$

that assigns the $N$ items to $K$ address locations in a uniform manner. Hashing algorithms differ from each other in how they resolve the *collision* that results when there is no room to store an item at its assigned location.

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

EM hashing methods fall into one of two categories: *directory* methods and *directoryless* methods. Fagin et al. [1979] proposed a directory scheme called *extendible hashing*. Let us assume that the size $K$ of the range of the hash function *hash* is sufficiently large. The directory, for a given $d \geq 0$, consists of a table (array) of $2^d$ pointers. Each item is assigned to the table location corresponding to the $d$ least significant bits of its hash address. The value of $d$, called the *global depth*, is set to the smallest value for which each table location has at most $B$ items assigned to it. Each table location contains a pointer to a block where its items are stored. Thus, a lookup takes two I/Os: one to access the directory and one to access the block storing the item. If the directory fits in internal memory, only one I/O is needed.

Several table locations may have many fewer than $B$ assigned items, and for purposes of minimizing storage utilization, they can share the same disk block for storing their items. A table location shares a disk block with all the other table locations having the same $k$ least significant bits in their address, where the *local depth* $k$ is chosen to be as small as possible so that the pooled items fit into a single disk block. Each disk block has its own local depth. An example is given in Figure 7.

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

More specifically, let $hash_d$ be the hash function corresponding to the $d$ least significant bits of *hash*; that is, $hash_d(x) = hash(x) \bmod 2^d$. Initially a single disk block is created to store the data items, and all the slots in the directory are initialized to point to the block. The local depth $k$ of the block is set to 0.

When an item with key value $x$ is inserted, it is stored in the disk block pointed to by directory slot $hash_d(x)$. If as a result the block (call it $b$) overflows, then block $b$ splits into two blocks—the original block $b$ and a new block $b'$—and its

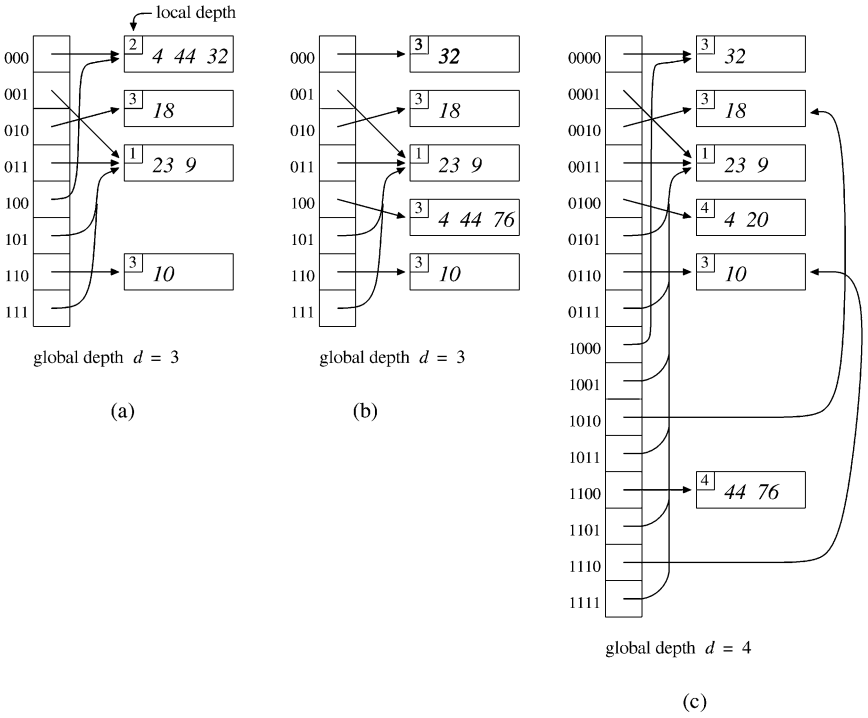


**Fig. 7**. Extendible hashing with block size $B = 3$. The keys are indicated in italics; the hash address of a key consists of its binary representation. For example, the hash address of key *4* is "...000100" and the hash address of key *44* is "...0101100." (a) The hash table after insertion of the keys *4*, *23*, *18*, *10*, *44*, *32*, *9*. (b) Insertion of the key *76* into table location 100 causes the block with local depth 2 to split into two blocks with local depth 3. (c) Insertion of the key *20* into table location 100 causes a block with local depth 3 to split into two blocks with local depth 4. The directory doubles in size and the global depth $d$ is incremented from 3 to 4.

items are redistributed based upon the $(b.k + 1)$st least significant bit of $hash(x)$. (Here $b.k$ refers to $b$'s local depth $k$.) We increment $b.k$ by 1 and store that value also in $b'.k$. In the unlikely event that $b$ or $b'$ is still overfull, we continue the splitting procedure and increment the local depths appropriately. At this point, some of the data items originally stored in block $b$ have been moved to other blocks, based upon their hash addresses. If $b.k \leq d$, we simply update those directory pointers originally pointing to $b$ that need changing, as shown in Figure 7(b). Otherwise, the directory isn't large enough to accommodate hash addresses with $b.k$ bits, so we repeatedly double the directory size and increment the global depth $d$ by 1 until $d$ becomes equal to $b.k$, as shown in Figure 7(c). The pointers in the new directory are initialized to point to the appropriate disk blocks. As noted before, the disk blocks do not need to be modified during doubling, except for the block that overflows.

Extendible hashing can handle deletions in a similar way. When two blocks with the same local depth $k$ contain items whose hash addresses share the same $k - 1$ least significant bits and can fit into a single block, then their items are merged into a single block with a decremented value of $k$. The combined size of the blocks being merged must be sufficiently less than $B$ to prevent immediate splitting after a subsequent insertion. The directory shrinks by half (and the global depth $d$ is decremented by 1) when all the local depths are less than the current value of $d$.

The expected number of disk blocks required to store the data items is asymptotically $n/\ln 2 \approx n/0.69$; that is, the blocks tend to be about 69% full [Mendelson 1982]. At least $\Omega(n/B)$ blocks are needed to store the directory. Flajolet [1983] showed that on average the directory uses $\Theta(N^{1/B}n/B) = \Theta(N^{1+1/B}/B^2)$ blocks, which can be superlinear in $N$ asymptotically! However, for practical values of $N$ and $B$, the $N^{1/B}$ term is a small constant, typically less than 2, and directory size is within a constant factor of optimal.

The resulting directory is equivalent to the leaves of a perfectly balanced trie [Knuth 1998], in which the search path for each item is determined by its hash address, except that hashing allows the leaves of the trie to be accessed directly in a single I/O. Any item can thus be retrieved in a total of two I/Os. If the directory fits in internal memory, only one I/O is needed.

A disadvantage of directory schemes is that two I/Os rather than one I/O are required when the directory is stored in external memory. Litwin [1980] and Larson [1982] developed a directoryless method called *linear hashing* that expands the number of data blocks in a controlled regular fashion. For example, suppose that the disk blocks currently allocated are blocks $0, 1, 2, \ldots, 2^d + p - 1$, for some $0 \le p < 2^d$. When $N$ grows sufficiently larger (say, by $0.8B$ items), block $p$ is split by allocating a new block $2^d + p$. Some of the data items from block $p$ are redistributed to block $2^d + p$, based upon the value of $hash_{d+1}$, and $p$ is incremented by 1. When $p$ reaches $2^d$, it is reset to 0 and the global depth $d$ is incremented by 1. To search for an item with key value $x$, the hash address $hash_d(x)$ is used if it is $p$ or larger; otherwise if the address is less than $p$, then the corresponding block has already been split, so $hash_{d+1}(x)$ is used instead as the hash address.

In contrast to directory schemes, the blocks in directoryless methods are chosen for splitting in a predefined order. Thus the block that splits is usually not the block that has overflowed, so some of the blocks may require auxiliary overflow lists to store items assigned to them. On the other hand, directoryless methods have the advantage that there is no need for access to a directory structure, and thus searches often require only one I/O. A related technique called spiral storage (or spiral hashing) [Martin 1979; Mullin 1985] combines constrained bucket splitting and overflowing buckets. More detailed surveys and analysis of methods for dynamic hashing appear in Baeza-Yates and Soza-Pollman [1998] and Enbody and Du [1988].

The above hashing schemes and their many variants work very well for dictionary applications in the average case, but have poor worst-case performance. They also do not support sequential search, such as retrieving all the items with key value in a specified range. Some clever work has been done on order-preserving hash functions, in which items with sequential keys are stored in the same block or in adjacent blocks, but the search performance is less robust and tends to deteriorate because of unwanted collisions. (See Gaede and Günther [1998] for a survey, plus recent work in Indyk et al. [1997]). A more effective approach for sequential search is to use multiway trees, which we explore next.

## 10. MULTIWAY TREE DATA STRUCTURES

An advantage of search trees over hashing methods is that the data items in a tree are sorted, and thus the tree can be used readily for one-dimensional range search. The items in a range $[x, y]$ can be found by searching for $x$ in the tree, and then performing an inorder traversal in the tree from $x$ to $y$. In this section we explore some important search-tree data structures in external memory.

### 10.1. B-Trees and Variants

Tree-based data structures arise naturally in the online setting, in which the data can be updated and queries must be processed immediately. Binary trees have a host of applications in the (internal memory) 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.

The well-known balanced multiway *B-tree* due to Bayer and McCreight [1972], Comer [1979], and Knuth [1998] is the most widely used nontrivial EM data structure. The degree of each node in the B-tree (with the exception of the root) is required to be $\Theta(B)$, which guarantees that the height of a B-tree storing $N$ items is roughly $\log_B N$. B-trees support dynamic dictionary operations and one-dimensional range search optimally in linear space, $O(\log_B N)$ I/Os per insert or delete, and $O(\log_B N + z)$ I/Os per query, where $Z = zB$ is the number of items output. When a node overflows during an insertion, it splits into two half-full nodes, and if the splitting causes the parent node to overflow, the parent node splits, and so on. Splittings can thus propagate up to the root, which is how the tree grows in height. Deletions are handled in a symmetric way by merging nodes.

In the *$B^+$-tree* variant, pictured in Figure 8, all the items are stored in the leaves, and the leaves are linked together in symmetric order to facilitate range queries and sequential access. The internal nodes store only key values and pointers and thus can have a higher branching factor. In the most popular variant of $B^+$-trees, called *B*-trees*, splitting can usually be postponed when a node overflows, by "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 the number of times new nodes must be created and thus increases the storage utilization. And since there are fewer nodes in the tree, search I/O costs are lower. When no sharing is done (as in $B^+$-trees), Yao [1978] 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 $2\ln(3/2) \approx 81\%$ [Baeza-Yates 1989; Küspert 1983]. Storage utilization can be increased further by sharing among several siblings, at the cost of more complicated insertions and deletions. Some helpful space-saving techniques borrowed from hashing are partial expansions [Baeza-Yates and Larson 1989] and use of overflow nodes [Srinivasan 1991].

A cross between B-trees and hashing, where each subtree rooted at a certain level of the B-tree is instead organized as an external hash table, was developed by Litwin and Lomet [1987] and further studied in Baeza-Yates [1996] and Lomet [1988]. O'Neil [1992] proposed a B-tree variant called the SB-tree that clusters together on the disk symmetrically ordered nodes from the same level so as to optimize range queries and sequential access. Rao and Ross [1999; 2000] use similar ideas to exploit locality and optimize search tree performance in internal memory. Reducing the number of pointers allows a higher branching factor and thus faster search.

Partially persistent versions of B-trees have been developed by Becker et al. [1996] and Varman and Verma [1997]. By persistent data structure, we mean that searches can be done with respect to any timestamp $y$ [Driscoll et al. 1989; Easton 1986]. In a partially persistent data structure, only the most recent version of the data structure can be updated. In a fully persistent data structure, any update done with timestamp $y$ affects all future queries for any time after $y$. An interesting open problem is whether B-trees can be made fully persistent. Salzberg and Tsotras [1999] survey work done on persistent access methods and other techniques for time-evolving data. Lehman and Yao [1981], Mohan [1990], and Lomet and Salzberg [1997] explore mechanisms to add concurrency and recovery to B-trees.

### 10.2. Weight-Balanced B-Trees

Arge and Vitter [1996] introduce a powerful variant of B-trees called *weight-balanced B-trees*, with the property that the weight of any subtree at level $h$ (i.e.,

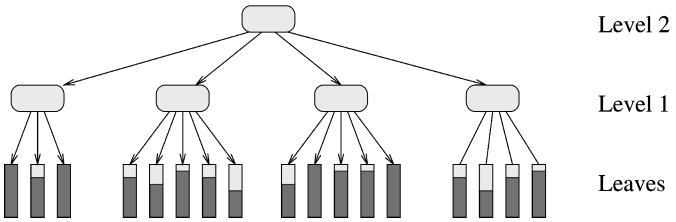


**Fig. 8**. $B^+$-tree multiway search tree. Each internal and leaf node corresponds to a disk block. All the items are stored in the leaves; the darker portion of each leaf block indicates its relative fullness. The internal nodes store only key values and pointers, $\Theta(B)$ of them per node. Although not indicated here, the leaf blocks are linked together sequentially.

the number of nodes in the subtree rooted at a node 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 $h$ of a weight-balanced B-tree gets rebalanced, no further rebalancing is needed until its subtree is updated $\Omega(a^h)$ times. Weight-balanced B-trees support a wide array of applications in which the I/O cost to rebalance a node of weight $w$ is $O(w)$; the rebalancings can be scheduled in an amortized (and often worst-case) way with only $O(1)$ I/Os. Such applications are very common when the nodes have secondary structures, as in multidimensional search trees, or when rebuilding is expensive. Agarwal et al. [2001a] apply weight-balanced B-trees to convert partition trees such as $k$d-trees, BBD trees, and BAR trees, which were designed for internal memory, into efficient EM data structures.

Weight-balanced trees called BB[$\alpha$]-trees [Blum and Mehlhorn 1980; Nievergelt and Reingold 1973] have been designed for internal memory; they maintain balance via rotations, which is appropriate for binary trees, but not for the multiway trees needed for external memory. In contrast, weight-balanced B-trees maintain balance via splits and merges.

Weight-balanced B-trees were originally conceived as part of an optimal dynamic EM interval tree structure for stabbing queries and a related EM segment tree structure. We discuss their use for stabbing queries and other types of range queries in Sections 11.3 to 11.5. They also have applications in the (internal memory) RAM model [Arge and Vitter 1996; Grossi and Italiano 1997] where they offer a simpler alternative to BB[$\alpha$]-trees. For example, by setting $a$ to a constant in the EM interval tree based upon weight-balanced B-trees, we get a simple worst-case implementation of interval trees [Edelsbrunner 1983a; 1983b] in internal memory. Weight-balanced B-trees are also preferable to BB[$\alpha$]-trees for purposes of augmenting one-dimensional data structures with range restriction capabilities [Willard and Lueker 1985].

### 10.3. Parent Pointers and Level-Balanced B-Trees

It is sometimes useful to augment B-trees with parent pointers. For example, if we represent a total order via the leaves in a B-tree, we can answer order queries such as, "Is $x < y$ in the total order?" by walking upwards in the B-tree from the leaves for $x$ and $y$ until we reach their common ancestor. Order queries arise in online algorithms for planar point location and for determining reachability in monotone subdivisions [Agarwal et al. 1999]. If we augment a conventional B-tree with parent pointers, then each split operation costs $\Theta(B)$ I/Os to update parent pointers, although the I/O cost is only $O(1)$ when amortized over the updates to the node. However, this amortized bound does not apply if the B-tree needs to support cut

and concatenate operations, in which case the B-tree is cut into contiguous pieces and the pieces are rearranged arbitrarily. For example, reachability queries in a monotone subdivision are processed by maintaining two total orders, called the leftist and rightist orders, each of which is represented by a B-tree. When an edge is inserted or deleted, the tree representing each order is cut into four consecutive pieces, and the four pieces are rearranged via concatenate operations into a new total order. Doing cuts and concatenation via conventional B-trees augmented with parent pointers will require $\Theta(B)$ I/Os per level in the worst case. Node splits can occur with each operation (unlike the case where there are only inserts and deletes), and thus there is no convenient amortization argument that can be applied.

Agarwal et al. [1999] describe an interesting variant of B-trees called *level-balanced B-trees* for handling parent pointers and operations like cut and concatenate. The balancing condition is "global": the data structure represents a forest of B-trees in which the number of nodes on level $h$ in the forest is allowed to be at most $N_h = 2N/(b/3)^h$, where $b$ is some fixed parameter in the range $4 < b < B/2$. It immediately follows that the total height of the forest is roughly $\log_b N$.

Unlike previous variants of B-trees, the degrees of individual nodes of level-balanced B-trees can be arbitrarily small, and for storage purposes, nodes are packed together into disk blocks. Each node in the forest is stored as a node record (which points to the parent's node record) and a doubly linked list of child records (which point to the node records of the children). There are also pointers between the node record and the list of child records. Every disk block stores only node records or only child records, but all the child records for a given node must be stored in the same block (possibly with child records for other nodes). The advantage of this extra level of indirection is that cuts and concatenates can usually be done in only $O(1)$ I/Os per level of the forest. For example, during a cut, a node record gets split into two, and its list of child nodes is chopped into two separate lists. The parent node must therefore get a new child record to point to the new node. These updates require $O(1)$ I/Os except when there is not enough space in the disk block of the parent's child records, in which case the block must be split into two, and extra I/Os are needed to update the pointers to the moved child records. The amortized I/O cost, however, is only $O(1)$ per level, since each update creates at most one node record and child record at each level. The other dynamic update operations can be handled similarly.

All that remains is to reestablish the global level invariant when a level gets too many nodes as a result of an update. If level $h$ is the lowest such level out of balance, then level $h$ and all the levels above it are reconstructed via a postorder traversal in $O(N_h)$ I/Os so that the new nodes get degree $\Theta(b)$ and the invariant is restored. The final trick is to construct the new parent pointers that point from the $\Theta(N_{h-1}) = \Theta(bN_h)$ node records on level $h-1$ to the $\Theta(N_h)$ level-$h$ nodes. The parent pointers can be accessed in a blocked manner with respect to the new ordering of the nodes on level $h$. By sorting, the pointers can be rearranged to correspond to the ordering of the nodes on level $h-1$, after which the parent pointer values can be written via a linear scan. The resulting I/O cost is $O(N_h + Sort(bN_h) + Scan(bN_h))$, which can be amortized against the $\Theta(N_h)$ updates that occurred since the last time the level-$h$ invariant was violated, yielding an amortized update cost of $O(1 + (b/B)\log_m n)$ I/Os per level.

Order queries such as "Does leaf $x$ precede leaf $y$ in the total order represented by the tree?" can be answered using $O(\log_B N)$ I/Os by following parent pointers starting at $x$ and $y$. The update operations insert, delete, cut, and concatenate can be done in $O((1 + (b/B)\log_m n)\log_b N)$ I/Os amortized, for any $2 \le b \le B/2$, which is never worse than $O((\log_B N)^2)$ by appropriate choice of $b$.

Using the multislab decomposition we discuss in Section 11.3, Agarwal et al.

[1999] apply level-balanced B-trees in a data structure for point location in monotone subdivisions, which supports queries and (amortized) updates in $O((\log_B N)^2)$ I/Os. They also use it to dynamically maintain planar *st*-graphs using $O((1+(b/B)(\log_m n)\log_b N)$ I/Os (amortized) per update, so that reachability queries can be answered in $O(\log_B N)$ I/Os (worst-case). (Planar *st*-graphs are planar directed acyclic graphs with a single source and a single sink.) An interesting open 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.

### 10.4. Buffer Trees

An important paradigm for constructing algorithms for batched problems in an internal memory setting is to use a dynamic data structure to process a sequence of updates. For example, we can sort $N$ items by inserting them one by one into a priority queue, followed by a sequence of $N$ *delete_min* operations. Similarly, many batched problems in computational geometry can be solved by dynamic plane sweep techniques. For example, in Section 7 we showed how to compute orthogonal segment intersections by dynamically keeping track of the active vertical segments (i.e., those hit by the horizontal sweep line); we mentioned a similar algorithm for orthogonal rectangle intersections.

However, if we use this paradigm naively in an EM setting, with a B-tree as the dynamic data structure, the resulting I/O performance will be highly nonoptimal. For example, if we use a B-tree as the priority queue in sorting or to store the active vertical segments hit by the sweep line, each update and query operation will take $O(\log_B N)$ I/Os, resulting in a total of $O(N \log_B N)$ I/Os, which is larger than the optimal bound $Sort(N)$ by a substantial factor of roughly $B$. One solution suggested in Vitter [1991] is to use a binary tree data structure 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 $O(\log n)$, yielding a total I/O bound of $O(n \log n)$, which is still nonoptimal by a significant $\log m$ factor.

Arge [1995a] developed the elegant *buffer tree* data structure to support *batched dynamic* operations, 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 $\Theta(m)$ rather than degree $\Theta(B)$, except possibly for the root. Its key distinguishing feature is that each node has a buffer that can store $\Theta(M)$ items (i.e., $\Theta(m)$ blocks of items). Items in a node are pushed down to the children when the buffer fills. Emptying a full buffer requires $\Theta(m)$ I/Os, which amortizes the cost of distributing the $M$ items to the $\Theta(m)$ children. Each item thus incurs an amortized cost of $O(m/M) = O(1/B)$ I/Os per level, and the resulting cost for queries and updates is $O((1/B)\log_m n)$ I/Os amortized.

Buffer trees have an ever-expanding list of applications. They can be used as a subroutine in the standard sweep line algorithm in order to get an optimal EM algorithm for orthogonal segment intersection. Arge showed how to extend buffer trees to implement segment trees [Bentley 1980] in external memory in a batched dynamic setting by reducing the node degrees to $\Theta(\sqrt{m})$ and by introducing *multislabs* in each node, which were explained in Section 7 for the related batched problem of intersecting rectangles. Buffer trees provide a natural amortized implementation of priority queues for *time-forward processing* applications such as discrete event simulation, sweeping, and list ranking [Chiang et al. 1995]. Govindrajan et al. [2000] use time-forward processing to construct a well-separated pair decomposition of $N$ points in $d$ dimensions in $O(Sort(N))$ I/Os, and they apply it to the problems of finding the $K$ nearest neighbors for each point and the $K$ closest pairs. Brodal and Katajainen [1998] 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. Practical implementations of priority queues based upon these ideas are examined in Brengel et al. [1999] and Sanders [1999]. In Section 11.2 we report on some timing experiments involving buffer trees for use in bulk loading of R-trees. Further experiments on buffer trees appear in Hutchinson et al. [1997].

## 11. SPATIAL DATA STRUCTURES AND RANGE SEARCH

In this section we consider online EM data structures for storing and querying spatial data. A fundamental database primitive in spatial databases and geographic information systems (GIS) is range search, which includes dictionary lookup as a special case. An orthogonal range query, for a given $d$-dimensional rectangle, returns all the points in the interior of the rectangle. In this section we use range searching (especially for the orthogonal 2-D case when $d = 2$) as the canonical query operation on spatial data. Other types of spatial queries include point location, ray shooting, nearest neighbor, and intersection queries, but for brevity we restrict our attention primarily to range searching.

There are two types of spatial data structures: data-driven and space-driven. R-trees and $k$d-trees are data-driven since they are based upon a partitioning of the data items themselves, whereas space-driven methods such as quad trees and grid files are organized by a partitioning of the embedding space, akin to order-preserving hash functions. In this section we discuss primarily data-driven data structures.

Multidimensional range search is a fundamental primitive in several online geometric applications, and it provides indexing support for constraint and object-oriented data models. (See Kanellakis et al. [1996] for background.) We have already discussed multidimensional range searching in a batched setting in Section 7. In this section we concentrate on data structures for the online case.

For many types of range searching problems, it is very difficult to develop theoretically optimal algorithms and data structures. Many open problems remain. The primary design criteria are to achieve the same performance we get using B-trees for one-dimensional range search:

(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 (namely, $O(n)$ blocks) of disk storage space, and
(3) to support dynamic updates in $O(\log_B N)$ I/Os (in the case of dynamic data structures).

Criterion 1 combines the I/O cost $Search(N) = O(\log_B N)$ of the search component of queries with the I/O cost $Output(Z) = O(\lceil z \rceil)$ for reporting the $Z$ output items. Combining the costs has the advantage that when one cost is much larger than the other, the query algorithm has the extra freedom to follow a *filtering* paradigm [Chazelle 1986], in which both the search component and the output reporting are allowed to use the larger number of I/Os. For example, to do queries optimally when $Output(Z)$ is large with respect to $Search(N)$, the search component can afford to be somewhat sloppy as long as it doesn't use more than $O(z)$ I/Os, and when $Output(Z)$ is relatively small, the $Z$ output items do not need to reside compactly in only $O(\lceil z \rceil)$ blocks. Filtering is an important design paradigm for many of the algorithms we discuss in this section.

We find in Section 11.7 under a fairly general computational model for general 2-D orthogonal queries, as pictured in Figure 9(d), it is impossible to satisfy Criteria 1 and 2 simultaneously. At least $\Omega(n(\log n)/\log(\log_B N + 1))$ blocks of disk space must be used to achieve a query bound of $O((\log_B N)^c + z)$ I/Os per query, for any constant $c$ [Subramanian and Ramaswamy 1995]. Three natural questions arise.

—What sort of performance can be achieved when using only a linear amount of disk space? In Sections 11.1 and 11.2, we discuss some of the

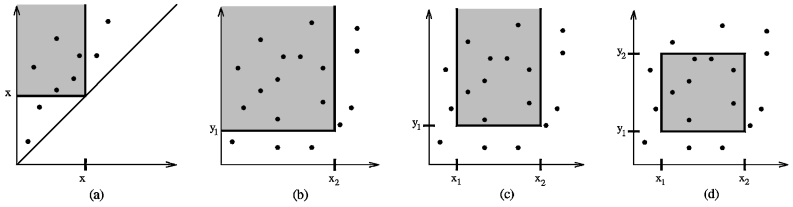


**Fig. 9**. Different types of 2-D orthogonal range queries: (a) diagonal corner two-sided 2-D query (equivalent to a stabbing query; cf. Section 11.3); (b) two-sided 2-D query; (c) three-sided 2-D query; (d) general four-sided 2-D query.

linear-space data structures used extensively in practice. None of them come close to satisfying Criteria 1 and 3 for range search in the worst case, but in typical-case scenarios they often perform well. We devote Section 11.2 to R-trees and their variants, which are the most popular general-purpose spatial structures developed to date.

—Since the lower bound applies only to general 2-D rectangular queries, are there any data structures that meet Criteria 1 to 3 for the important special cases of 2-D range searching pictured in Figures 9(a) through (c)? Fortunately the answer is yes. We show in Sections 11.3 and 11.4 how to use a bootstrapping paradigm to achieve optimal search and update performance.

—Can we meet Criteria 1 and 2 for general four-sided range searching if the disk space allowance is increased to $O(n(\log n)/\log(\log_B N + 1))$ blocks? Yes again! In Section 11.5, we show how to adapt the optimal structure for three-sided searching in order to handle general four-sided searching in optimal search cost. The update cost, however, is not known to be optimal.

In Section 11.6, we discuss other scenarios of range search dealing with three dimensions and nonorthogonal queries. We discuss the lower bounds for 2-D range searching in Section 11.7.

### 11.1. Linear-Space Spatial Structures

Grossi and Italiano [1999] construct an elegant multidimensional version of the B-tree called the *cross tree*. Using linear space, it combines the data-driven partitioning of weight-balanced B-trees (cf. Section 10.2) 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 O-tree of Kanth and Singh [1999] provides similar bounds. Cross trees also support the dynamic operations of cut and concatenate in $O(n^{1-1/d})$ I/Os. In some restricted models for linear-space data structures, the 2-D range search query performance of cross trees and O-trees can be considered to be optimal, although it is much larger than the logarithmic bound of Criterion 1.

One way to get multidimensional EM data structures is to augment known internal memory structures, such as quad trees and $k$d-trees, with block-access capabilities. Examples include *kd-B-trees* [Robinson 1981], *buddy trees* [Seeger and Kriegel 1990], and *hB-trees* [Evangelides et al. 1997; Lomet and Salzberg 1990]. *Grid files* [Hinrichs 1985; Krishnamurthy and Wang 1985; Nievergelt et al. 1984] are a flattened data structure for storing the cells of a two-dimensional grid in disk blocks. 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 [Gargantini 1982; Kamel and Faloutsas 1994; Orenstein and Merrett 1984]. Linearization can also be used to represent nonpoint data, in which the data items are partitioned into one or more multidimensional rectangular regions [Abel

1984; Orenstein 1989]. 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, such as grid files, queries can require $\Theta(n)$ I/Os, even if there are no points satisfying the query. We refer the reader to Agarwal and Erickson [1999], Gaede and Günther [1998], and Nievergelt and Widmayer [1997] for a broad survey of these and other interesting methods. Space-filling curves arise again in connection with R-trees, which we describe next.

### 11.2. R-Trees

The *R-tree* of Guttman [1984] and its many variants are a practical multidimensional generalization of the B-tree for storing a variety of geometric objects, such as points, segments, polygons, and polyhedra, using linear disk space. Internal nodes have degree $\Theta(B)$ (except possibly the root), and leaves store $\Theta(B)$ items. Each node in the tree has associated with it a bounding box (or bounding polygon) of all the items 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 to 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.

In the dynamic setting, there are several popular heuristics for where to insert new items into an R-tree and how to rebalance it; see Agarwal and Erickson [1999], Gaede and Günther [1998], and Greene [1989] for a survey. The *R*-tree* variant of Beckmann et al. [1990] seems to give best overall query performance. To insert an item, we start at the root and recursively insert the item into the subtree whose bounding box would expand the least in order to accommodate the item. In case of a tie (e.g., if the item already fits inside the bounding boxes of two or more subtrees), we choose the subtree with the smallest resulting bounding box. In the normal R-tree algorithm, if a leaf node gets too many items or if an internal node gets too many children, we split it, as in B-trees. Instead, in the R*-tree algorithm, we remove a certain percentage of the items from the overflowing node and reinsert them into the tree. The items we choose to reinsert are the ones whose centroids are farthest from the center of the node's bounding box. This *forced reinsertion* tends to improve global organization and reduce query time. If the node still overflows after the forced reinsertion, we split it. The splitting heuristics try to partition the items into nodes so as to minimize intuitive measures such as coverage, overlap, or perimeter. During deletion, in both the normal R-tree and R*-tree algorithms, if a leaf node has too few items or if an internal node has too few children, we delete the node and reinsert all its items back into the tree by forced reinsertion.

The rebalancing heuristics perform well in many practical scenarios, especially in low dimensions, but they result in poor worst-case query bounds. An interesting open problem is whether nontrivial query 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 Section 11.1. New R-tree partitioning methods by de Berg et al. [2000] and Agarwal et al. [2001b] provide some provable bounds on overlap and query performance.

In the static setting, in which there are no updates, constructing the R*-tree by repeated insertions, one by one, is extremely slow. A faster alternative to the dynamic R-tree construction algorithms mentioned above is to bulk-load the R-tree in a bottom-up fashion [Abel 1984; Kamel and Faloutsos 1993; Orenstein 1989]. Such methods use some heuristic for grouping the items into leaf nodes of the R-tree, and then recursively build the nonleaf nodes from bottom to top. As an example, in the so-called Hilbert R-tree of Kamel and Faloutsos [1993], each item is labeled with the position of its centroid on the Peano–Hilbert space-filling curve, and a $B^+$-tree

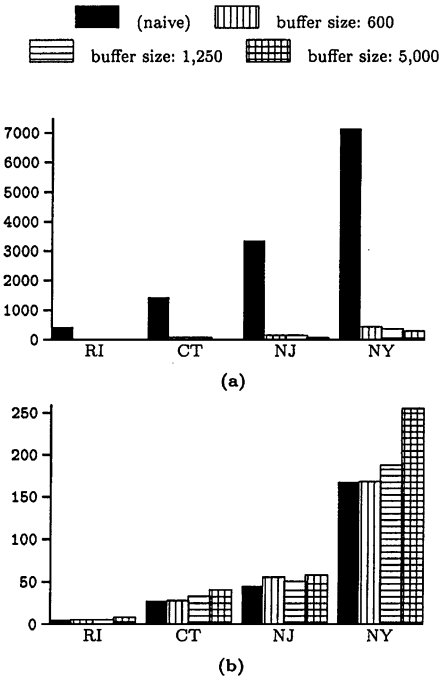


**Fig. 10**. Costs for R-tree processing (in units of 1000 I/Os) using the naive repeated insertion method and the buffer R-tree for various buffer sizes: (a) cost for bulk-loading the R-tree; (b) query cost.

**Table V.** Summary of the Costs (in Number of I/Os) for R-Tree Updates and Queries. Packing Refers to the Percentage Storage Utilization

| Data Set | Update Method | Update with 50% 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% |

is built upon the totally ordered labels in a bottom-up manner. Bulk loading a Hilbert R-tree is therefore easy to do once the centroid points are presorted. These static construction method algorithms are very different in spirit from the dynamic insertion methods. The dynamic methods explicitly try to reduce the coverage, overlap, or perimeter of the bounding boxes of the R-tree nodes, and as a result, they usually achieve good query performance. The static construction methods do not consider the bounding box information at all. Instead, the hope is that the improved storage utilization (up to 100%) of these packing methods compensates for a higher degree of node overlap. A dynamic insertion method related to Kamel and Faloutsos [1993] was presented in Kamel and Faloutsos [1994]. The quality of the Hilbert R-tree in terms of query performance is generally not as good as that of an R*-tree, especially for higher-dimensional data [Berchtold et al. 1998; Kamel et al. 1996].

In order to get the best of both worlds—the query performance of R*-trees and the bulk construction efficiency of Hilbert R-trees—Arge et al. [1999a] and van den Bercken et al. [1997] independently devised fast bulk loading methods based upon buffer trees that do top-down construction in $O(n \log_m n)$ I/Os, which matches the performance of the bottom-up methods within a constant factor. The former method is especially efficient and supports dynamic batched updates and queries. In Figure 10 and Table V, we report on some experiments that test the construction, update, and query performance of various R-tree methods. The experimental data came from TIGER/line data sets from four US states [TIGER 1992]; the implementations were done using the TPIE system, described in Section 14.

Figure 10 compares the construction cost for building R-trees and the resulting query performance in terms of I/Os for the naive sequential method for construction into R*-trees (labeled "naive") and the newly developed buffer R*-tree method [Arge et al. 1995a] (labeled "buffer"). An R-tree was constructed on the TIGER road data for each state and for each of four possible buffer sizes. The four buffer sizes were capable of storing 0, 600, 1250, and 5000 rectangles, respectively; buffer size 0 corresponds to the naive method and the

larger buffers correspond to the buffer method. 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 10, show that buffer R*-trees, even with relatively small buffers, achieve a tremendous speedup in number of I/Os for construction without any worsening in query performance, compared with the naive method. The CPU costs of the two methods are comparable. The storage utilization of buffer R*-trees tends to be in the 90% range, as opposed to roughly 70% for the naive method.

Bottom-up methods can build R-trees even more quickly and more compactly, but they generally do not support bulk dynamic operations, which is a big advantage of the buffer tree approach. Kamel et al. [1996] develop a way to do bulk updates with Hilbert R-trees, but at a cost in terms of query performance. Table V compares dynamic update methods for the naive method, for buffer R-trees, and for Hilbert R-trees [Kamel et al. 1996] (labeled "Hilbert"). A single R-tree was built for each of the four US states, containing 50% of the road data objects for that state. Using each of the three algorithms, the remaining 50% 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 V indicate that the buffer R*-tree and the Hilbert R-tree achieve a similar degree of packing, but the buffer R*-tree provides better update and query performance.

### 11.3. Bootstrapping for 2-D Diagonal Corner and Stabbing Queries

An obvious paradigm for developing an efficient dynamic EM data structure, given an existing data structure that works well when the problem fits into internal memory, is to "externalize" the internal memory data structure. If the internal memory data structure uses a binary tree, then a multiway tree such as a B-tree must be used instead. However, when searching a B-tree, it can be difficult to report the outputs in an output-sensitive manner. For example, in certain searching applications, each of the $\Theta(B)$ subtrees of a given node in a B-tree may contribute one item to the query output, and as a result each subtree may need to be explored (costing several I/Os) just to report a single output item.

Fortunately, we can sometimes achieve output-sensitive reporting by augmenting the data structure with a set of filtering substructures, each of which is a data structure for a smaller version of the same problem. We refer to this approach, which we explain shortly in more detail, as the *bootstrapping* paradigm. Each substructure typically needs to store only $O(B^2)$ items and to answer queries in $O(\log_B B^2 + Z'/B) = O(\lceil Z'/B \rceil)$ I/Os, where $Z'$ is the number of items reported. A substructure can even be static if it can be constructed in $O(B)$ I/Os, since we can keep updates in a separate buffer and do a global rebuilding in $O(B)$ I/Os whenever there are $\Theta(B)$ updates. Such a rebuilding costs $O(1)$ I/Os (amortized) per update. We can often remove the amortization make it worst-case using the weight-balanced B-trees of Section 10.2 as the underlying B-tree structure.

Arge and Vitter [1996] first uncovered the bootstrapping paradigm while designing an optimal dynamic EM data structure for diagonal corner two-sided 2-D queries (see Figure 9(a)) that meets all three design criteria listed in Section 11. Diagonal corner two-sided queries are equivalent to stabbing queries, which have the form: "Given a set of one-dimensional intervals, report all the intervals 'stabbed' by the query value $x$." (That is, report all intervals that contain $x$.) A diagonal corner query $x$ on a set of 2-D points $\{(a_1, b_2), (a_2, b_2), \ldots\}$ is equivalent to a stabbing query $x$ on the set of closed intervals $\{[a_1, b_2], [a_2, b_2], \ldots\}$.

The EM data structure for stabbing queries is a multiway version of the well-known interval tree data structure [Edelsbrunner 1983a; 1983b] for internal memory, which supports stabbing queries in $O(\log N + Z)$ CPU time and updates

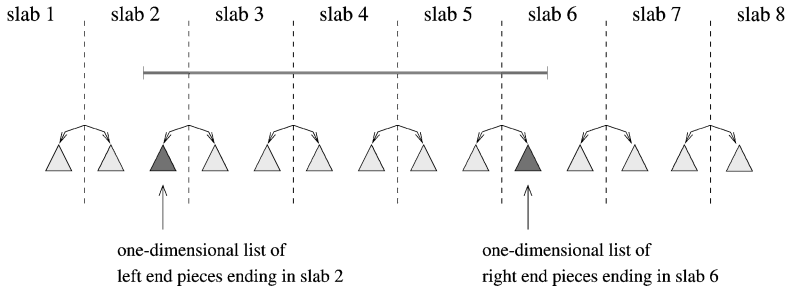


**Fig. 11**. Internal node $v$ of the EM priority search tree, for $B = 64$ with $\sqrt{B} = 8$ slabs. Node $v$ is the lowest node in the tree completely containing the indicated interval. The middle piece of the interval is stored in the multislab list corresponding to slabs 3 to 5. (The multislab lists are not pictured.) The left and right end pieces of the interval are stored in the left-ordered list of slab 2 and the right-ordered list of slab 6, respectively.

in $O(\log N)$ CPU time and uses $O(N)$ space. We can externalize it by using a weight-balanced B-tree as the underlying base tree, where the nodes have degree $\Theta(\sqrt{B})$. Each node in the base tree corresponds in a natural way to a one-dimensional range of $x$-values; its $\Theta(\sqrt{B})$ children correspond to subranges called slabs, and the $\Theta(\sqrt{B}^2) = \Theta(B)$ contiguous sets of slabs are called *multislabs*, as in Section 7 for a similar batched problem. Each input interval is stored in the lowest node $v$ in the base tree whose range completely contains the interval. The interval is decomposed by $v$'s $\Theta(\sqrt{B})$ slabs into at most three pieces: the middle piece that completely spans one or more slabs of $v$, the left end piece that partially protrudes into a slab of $v$, and the right end piece that partially protrudes into another slab of $v$, as shown in Figure 11. The three pieces are stored in substructures of $v$. In the example in Figure 11, the middle piece is stored in a list associated with the multislab it spans (corresponding to the contiguous range of slabs 3 to 5), the left end piece is stored in a one-dimensional list for slab 2 ordered by left endpoint, and the right end piece is stored in a one-dimensional list for slab 6 ordered by right endpoint.

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

The big problem with this approach is that we have to spend at least one I/O per multislab containing $x$, regardless of how many intervals are in the multislab lists. For example, there may be $\Theta(B)$ such multislab lists, with each list containing only a few stabbed intervals (or worse yet, none at all). The resulting query performance will be highly nonoptimal. The solution, according to the bootstrapping paradigm, is to use a substructure in each node consisting of an optimal static data structure for a smaller version of the same problem; a good choice is the corner data structure developed by Kanellakis et al. [1996]. The corner substructure in this case is used to store all the intervals from the "sparse" multislab lists, namely, those that contain fewer than $B$ intervals, and thus the substructure contains only $O(B^2)$ intervals. When visiting node $v$, we access only $v$'s non-sparse multislab lists, each of which contributes $Z' \geq B$ intervals to the output, at an output-sensitive cost of $O(Z'/B)$ I/Os, for some $Z'$. The remaining $Z''$ stabbed intervals stored in $v$ can be found by a single query to $v$'s corner substructure, at a cost of $O(\log_B B^2 + Z''/B) = O(\lceil Z''/B \rceil)$ I/Os. Since there are $O(\log_B N)$ nodes

along the search path in the base tree, the total collection of $Z$ stabbed intervals is reported in $O(\log_B N + z)$ I/Os, which is optimal. Using a weight-balanced B-tree as the underlying base tree allows the static substructures to be rebuilt in worst-case optimal I/O bounds.

Stabbing queries are important because, when combined with one-dimensional range queries, they provide a solution to *dynamic interval management*, in which one-dimensional intervals can be inserted and deleted, and intersection queries can be performed. These operations support indexing of one-dimensional constraints in constraint databases. Other applications of stabbing queries arise in graphics and GIS. For example, Chiang and Silva [1999] 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 Agarwal et al. [1998b]. The above bootstrapping approach also yields dynamic EM segment trees with optimal query and update bound and $O(n \log_B N)$-block space usage.

### 11.4. Bootstrapping for Three-Sided Orthogonal 2-D Range Search

Arge et al. [1999b] provide another example of the bootstrapping paradigm by developing an optimal dynamic EM data structure for three-sided orthogonal 2-D range searching (see Figure 9(c)) that meets all three design criteria. In internal memory, the optimal structure is the priority search tree [McCreight 1985], which answers three-sided range queries in $O(\log N + Z)$ CPU time, does updates in $O(\log N)$ CPU time, and uses $O(N)$ space. The EM structure of Arge et al. [1999b] is an externalization of the priority search tree, using a weight-balanced B-tree as the underlying base tree. Each node in the base tree corresponds to a one-dimensional range of $x$-values, and its $\Theta(B)$ children correspond to subranges consisting of vertical slabs. Each node $v$ contains a small substructure called a *child cache* that supports three-sided queries. Its child cache stores the "Y-set" $Y(w)$ for each of the $\Theta(B)$ children $w$ of $v$. The Y-set $Y(w)$ for child $w$ consists of the highest $\Theta(B)$ points in $w$'s slab that are not already stored in the child cache of some ancestor of $v$. There are thus a total of $\Theta(B^2)$ points stored in $v$'s child cache.

We can answer a three-sided query of the form $[x_1, x_2] \times [y_1, +\infty)$ by visiting a set of nodes in the base tree, starting with the root. For each visited node $v$, we pose the query $[x_1, x_2] \times [y_1, +\infty)$ to $v$'s child cache and output the results. The following rules are used to determine which of $v$'s children to visit. We visit $v$'s child $w$ if either

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

(See Figure 12.) There are $O(\log_B N)$ nodes $w$ that are visited because of Rule 1. When Rule 1 is not satisfied, Rule 2 provides an effective filtering mechanism to guarantee output-sensitive reporting. The I/O cost for initially accessing a child node $w$ can be charged to the $\Theta(B)$ points of $Y(w)$ reported from $v$'s child cache; conversely, if not all of $Y(w)$ is reported, then the points stored in $w$'s subtree will be too low to satisfy the query, and there is no need to visit $w$. (See Figure 12(b).) Provided that each child cache can be queried in $O(1)$ I/Os plus the output-sensitive cost to output the points satisfying the query, the resulting overall query time is $O(\log_B N + z)$, as desired.

All that remains is to show how to query a child cache in a constant number of I/Os, plus the output-sensitive cost. Arge et al. [1999b] provide an elegant and optimal static data structure for three-sided range search, which can be used in the EM priority search tree described above to implement the child caches of size $O(B^2)$. The static structure is a persistent B-tree optimized for batched construction. When used for $O(B^2)$ points, it occupies $O(B)$

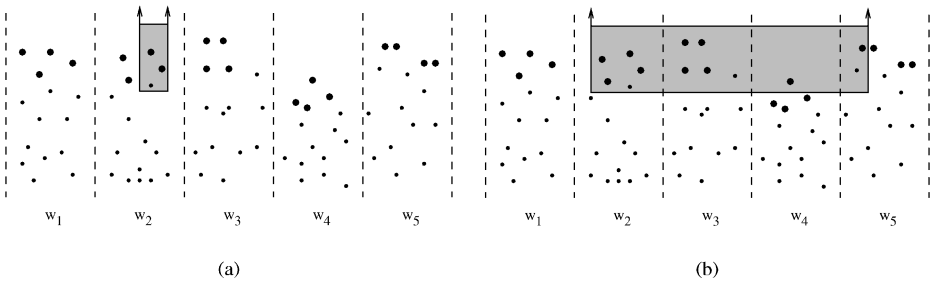


**Fig. 12**. Internal node $v$ of the EM priority search tree, with slabs (children) $w_1, w_2, \ldots, w_5$. The Y-sets of each child, which are stored collectively in $v$'s child cache, are indicated by the bold points. (a) The three-sided query is completely contained in the $x$-range of $w_2$. The relevant (bold) points are reported from $v$'s child cache, and the query is recursively answered in $w_2$. (b) The three-sided query spans several slabs. The relevant (bold) points are reported from $v$'s child cache, and the query is recursively answered in $w_2$, $w_3$, and $w_5$. The query is *not* extended to $w_4$ in this case because not all of its Y-set $Y(w_4)$ (stored in $v$'s child cache) satisfies the query, and as a result none of the points stored in $w_4$'s subtree can satisfy the query.

blocks, can be built in $O(B)$ I/Os, and supports three-sided queries in $O(\lceil Z'/B \rceil)$ I/Os per query, where $Z'$ is the number of points reported. The static structure is so simple that it may be useful in practice on its own.

Both the three-sided structure developed by Arge et al. [1999b] and the structure for two-sided diagonal queries discussed in Section 11.3 satisfy Criteria 1 to 3 of Section 11. So in a sense, the three-sided query structure subsumes the diagonal two-sided structure, since three-sided queries are more general. However, diagonal two-sided structures may prove to be faster in practice, because in each of its corner substructures, the data accessed during a query are always in contiguous blocks, whereas the static substructures used in three-sided search do not guarantee block contiguity. Empirical work is ongoing to evaluate the performance of these data structures.

On a historical note, earlier work on two- and three-sided queries was done by Ramaswamy and Subramanian [1994] using the notion of *path caching*; their structure met Criterion 1 but had higher storage overheads and amortized and/or nonoptimal update bounds. Subramanian and Ramaswamy [1995] subsequently developed the *p-range tree* data structure for three-sided queries, with optimal linear disk space and nearly optimal query and amortized update bounds.

### 11.5. General Orthogonal 2-D Range Search

The dynamic data structure for three-sided range searching can be generalized using the filtering technique of Chazelle [1986] to handle general four-sided queries with optimal I/O query bound $O(\log_B N + z)$ and optimal disk space usage $O(n(\log n)/\log(\log_B N + 1))$ [Arge et al. 1999b]. The update bound becomes $O((\log_B N)(\log n)/\log(\log_B N + 1))$, which may not be optimal.

The outer level of the structure is a balanced $(\log_B N + 1)$-way 1-D search tree with $\Theta(n)$ leaves, oriented, say, along the $x$-dimension. It therefore has about $(\log n)/\log(\log_B N + 1)$ levels. At each level of the tree, each input point is stored in four substructures (described below) that are associated with the particular tree node at that level that spans the $x$-value of the point. The space and update bounds quoted above follow from the fact that the substructures use linear space and can be updated in $O(\log_B N)$ I/Os.

To search for the points in a four-sided query rectangle $[x_1, x_2] \times [y_1, y_2]$, we decompose the four-sided query in the following natural way into two three-sided queries, a stabbing query, and $\log_B N - 1$ list traversals. We find the

lowest node $v$ in the tree whose $x$-range contains $[x_1, x_2]$. If $v$ is a leaf, we can answer the query in a single I/O. Otherwise we query the substructures stored in those children of $v$ whose $x$-ranges intersect $[x_1, x_2]$. Let $2 \leq k \leq \log_B N + 1$ be the number of such children. The range query when restricted to the leftmost such child of $v$ is a three-sided query of the form $[x_1, +\infty] \times [y_1, y_2]$, and when restricted to the rightmost such child of $v$, the range query is a three-sided query of the form $[-\infty, x_2] \times [y_1, y_2]$. Two of the substructures at each node are devoted to three-sided queries of these types; using the linear-sized data structures of Arge et al. [1999b] in Section 11.4, each such query can be done in $O(\log_B N + z)$ I/Os.

For the $k-2$ intermediate children of $v$, their $x$-ranges are completely contained inside the $x$-range of the query rectangle, and thus we need only do $k-2$ list traversals in $y$-order and retrieve the points whose $y$-values are in the range $[y_1, y_2]$. If we store the points in each node in $y$-order (in the third type of substructure), the $Z'$ output points from a node can be found in $O(\lceil Z'/B \rceil)$ I/Os, once a starting point in the linear list is found. We can find all $k-2$ starting points via a single query to a stabbing query substructure $S$ associated with $v$. (This structure is the fourth type of substructure.) For each two $y$-consecutive points $(a_i, b_i)$ and $(a_{i+1}, b_{i+1})$ associated with a child of $v$, we store the $y$-interval $[b_i, b_{i+1}]$ in $S$. Note that $S$ contains intervals contributed by each of the $\log_B N + 1$ children of $v$. By a single stabbing query with query value $y_1$, we can thus identify the $k-2$ starting points in only $O(\log_B N)$ I/Os [Arge and Vitter 1996], as described in Section 11.3. (We actually get starting points for all the children of $v$, not just the $k-2$ ones of interest, but we can discard the starting points we don't need.) The total number of I/Os to answer the range query is thus $O(\log_B N + z)$, which is optimal.

### 11.6. Other Types of Range Search

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

Vengroff and Vitter [1996a] develop the first theoretically near-optimal EM data structure for static three-dimensional orthogonal range searching. They create a hierarchical partitioning in which all the points that dominate a query point are densely contained in a set of blocks. Compression techniques are needed to minimize disk storage. With some recent modifications [Vitter and Vengroff 1999], queries can be done in $O(\log_B N + z)$ I/Os, which is optimal, and the space usage is $O(n(\log n)^{k+1}/(\log(\log_B N + 1))^k)$ disk blocks to support $(3+k)$-sided 3-D range queries, in which $k$ of the dimensions ($0 \leq k \leq 3$) have finite ranges. The result also provides optimal $O(\log N + Z)$-time query performance for three-sided 3-D queries in the (internal memory) RAM model, but using $O(N \log N)$ space.

By the reduction in Chazelle and Edelsbrunner [1987], a data structure for three-sided 3-D queries also applies to *2-D homothetic range search*, in which the queries correspond to scaled and translated (but not rotated) transformations of an arbitrary fixed polygon. A interesting special case is "fat" orthogonal 2-D range search, where the query rectangles are required to have bounded aspect ratio. For example, every rectangle with bounded aspect ratio can be covered by two overlapping squares. An interesting open problem is to develop linear-sized optimal data structures for fat orthogonal 2-D range search. By the reduction, one possible approach would be to develop optimal linear-sized data structures for three-sided 3-D range search.

Agarwal et al. [1998a] consider halfspace range searching, in which a query is specified by a hyperplane and a bit indicating one of its two sides, and the output of the query consists of all the points on that side of the hyperplane. They give various data structures for halfspace range searching in two, three, and higher dimensions, including one that works for simplex (polygon) queries in two dimensions,

but with a higher query I/O cost. They have subsequently improved the storage bounds for halfspace range queries in two dimensions to obtain an optimal static data structure satisfying Criteria 1 and 2 of Section 11.

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

Callahan et al. [1995] develop dynamic EM data structures for several online problems in $d$ dimensions. For any fixed $\epsilon > 0$, they can find an approximately nearest neighbor of a query point (within a $1+\epsilon$ factor of optimal) in $O(\log_B N)$ I/Os; insertions and deletions can also be done in $O(\log_B N)$ I/Os. They use a related approach to maintain the closest pair of points; each update costs $O(\log_B N)$ I/Os. Govindarajan et al. [2000] achieve the same bounds for closest pair by maintaining a well-separated pairs decomposition. For finding nearest neighbors and approximate nearest neighbors, two other approaches are partition trees [Agarwal et al. 1998a; 2000] and locality-sensitive hashing [Gionis et al. 1999]. Numerous other data structures have been developed for range queries and related problems on spatial data. We refer to Agarwal and Erickson [1999], Gaede and Günther [1998], and Nievergelt and Widmayer [1997] for a broad survey.

### 11.7. Lower Bounds for Orthogonal Range Search

We mentioned in Section 11 that Subramanian and Ramaswamy [1995] prove that no EM data structure for 2-D range searching can achieve design Criterion 1 using less than $O(n(\log n)/\log(\log_B N + 1))$ disk blocks, even if we relax the criterion to allow $O((\log_B N)^c + z)$ 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 [1990] for the internal memory model.

Hellerstein et al. [1997] consider a generalization of the layout-based lower bound argument of Kanellakis et al. [1996] for studying the tradeoff between disk space usage and query performance. They develop a model for *indexability*, in which an "efficient" data structure is expected to contain the $Z$ output points to a query compactly within $O(\lceil Z/B \rceil) = O(\lceil z \rceil)$ blocks. One shortcoming of the model is that it considers only data layout and ignores the search component of queries, and thus it rules out the important filtering paradigm discussed earlier in Section 11. For example, it is reasonable for any query algorithm to perform at least $\log_B N$ I/Os, so if the output size $Z$ is at most $B$, an algorithm may still be able to satisfy Criterion 1 even if the output is contained within $O(\log_B N)$ blocks rather than $O(z) = O(1)$ blocks. Arge et al. [1999b] modify the model to rederive the same nonlinear space lower bound $O(n(\log n)/\log(\log_B N + 1))$ of Subramanian and Ramaswamy [1995] for 2-D range searching by considering only output sizes $Z$ larger than $(\log_B N)^c B$, for which the number of blocks allowed to hold the outputs is $Z/B = O((\log_B N)^c + z)$. This approach ignores the complexity of how to find the relevant blocks, but as mentioned in Section 11.5 the authors separately provide an optimal 2-D range search data structure that uses the same amount of disk space and does queries in the optimal $O(\log_B N + z)$ I/Os. Thus, despite its shortcomings, the indexability model is elegant and can provide much insight into the complexity of blocking data in external memory. Further results in this model appear in Koutsoupias and Taylor [1998] and Samoladas and Miranker [1998].

One intuition from the indexability model is that less disk space is needed to efficiently answer 2-D queries when the queries have bounded aspect ratio (i.e., when the ratio of the longest side

length to the shortest side length of the query rectangle is bounded). An interesting question is whether R-trees and the linear-space structures of Sections 11.1 and 11.2 can be shown to perform provably well for such queries. Another interesting scenario is where the queries correspond to snapshots of the continuous movement of a sliding rectangle.

When the data structure is restricted to contain only a single copy of each point, Kanth and Singh [1999] 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 [Grossi and Italiano 1999] mentioned in Section 11.1, which in addition supports the operations of cut and concatenate.

## 12. DYNAMIC AND KINETIC DATA STRUCTURES

In this section we consider two scenarios where data items change: *dynamic* (in which items are inserted and deleted) and *kinetic* (in which the data items move continuously along specified trajectories). In both cases, queries can be done at any time. It is often useful for kinetic data structures to allow insertions and deletions; for example, if the trajectory of an item changes, we must delete the old trajectory and insert the new one.

### 12.1. Logarithmic Method for Decomposable Search Problems

In Sections 9 to 11 we've already encountered several dynamic data structures for the problems of dictionary lookup and range search. In Section 11, we saw how to develop optimal EM range search data structures by externalizing some known internal memory data structures. The key idea was to use the bootstrapping paradigm, together with weight-balanced B-trees as the underlying data structure, in order to consolidate several static data structures for small instances of range searching into one dynamic data structure for the full problem. The bootstrapping technique is specific to the particular data structures involved. In this section we look at another technique that is based upon the properties of the problem itself rather than upon that of the data structure.

We call a problem *decomposable* if we can answer a query by querying individual subsets of the problem data and then computing the final result from the solutions to each subset. Dictionary search and range searching are obvious examples of decomposable problems. Bentley developed the *logarithmic method* [Bentley and Saxe 1980; Overmars 1983] to convert efficient static data structures for decomposable problems into general dynamic ones. In the internal memory setting, the logarithmic method consists of maintaining a series of static substructures, at most one each of size 1, 2, 4, 8, .... When a new item is inserted, it is initialized in a substructure of size 1. If a substructure of size 1 already exists, the two substructures are combined into a single substructure of size 2. If there is already a substructure of size 2, they in turn are combined into a single substructure of size 4, and so on. For the current value of $N$, it is easy to see that the $k$th substructure (i.e., of size $2^k$) is present exactly when the $k$th bit in the binary representation of $N$ is 1. Since there are at most $\log N$ substructures, the search time bound is $\log N$ times the search time per substructure. As the number of items increases from 1 to $N$, the $k$th structure is built a total of $N/2^k$ times (assuming $N$ is a power of 2). If it can be built in $O(2^k)$ time, the total time for all insertions and all substructures is thus $O(N \log N)$, making the amortized insertion time $O(\log N)$. If we use up to three substructures of size $2^k$ at a time, we can do the reconstructions in advance and convert the amortized update bounds to worst-case [Overmars 1983].

In the EM setting, in order to eliminate the dependence upon the binary logarithm in the I/O bounds, the number of substructures must be reduced from $\log N$ to $\log_B N$, and thus the maximum size of the

$k$th substructure must be increased from $2^k$ to $B^k$. As the number of items increases from 1 to $N$, the $k$th substructure has to be built $NB/B^k$ times (when $N$ is a power of $B$), each time taking $O(B^k(\log_B N)/B)$ I/Os. The key point is that the extra factor of $B$ in the numerator of the first term is cancelled by the factor of $B$ in the denominator of the second term, and thus the resulting total insertion time over all $N$ insertions and all $\log_B N$ structures is $O(N(\log_B N)^2)$ I/Os, which is $O((\log_B N)^2)$ I/Os amortized per insertion. By global rebuilding we can do deletions in $O(\log_B N)$ I/Os amortized. As in the internal memory case, the amortized updates can typically be made worst-case.

Arge and Vahrenhold [2000] obtain I/O bounds for dynamic point location in general planar subdivisions similar to those of Agarwal et al. [1999], but without use of level-balanced trees. Their method uses a weight-balanced base structure at the outer level and a multislab structure for storing segments similar to that of Arge and Vitter [1996] described in Section 11.3. They use the logarithmic method to construct a data structure to answer vertical rayshooting queries in the multislab structures. The method relies upon a total ordering of the segments, but such an ordering can be changed drastically by a single insertion. However, each substructure in the logarithmic method is static (until it is combined with another substructure), and thus a static total ordering can be used for each substructure. The authors show by a type of fractional cascading that the queries in the $\log_B N$ substructures do not have to be done independently, which saves a factor of $\log_B N$ in the I/O cost, but at the cost of making the data structures amortized instead of worst case.

Agarwal et al. [2001a] apply the logarithmic method (in both the binary form and $B$-way variant) to get EM versions of $k$d-trees, BBD trees, and BAR trees.

### 12.2. Continuously Moving Items

Early work on temporal data generally concentrated on time-series or multiversion data [Salzberg and Tsotras 1999]. A question of growing interest in this mobile age is how to store and index continuously moving items, such as mobile telephones, cars, and airplanes (e.g., see Jensen and Theodoridis [2000], Saltenis et al. [2000], and Wolfson et al. [1999]). There are two main approaches to storing moving items. The first technique is to use the same sort of data structure as for nonmoving data, but to update it whenever items move sufficiently far so as to trigger important combinatorial events that are relevant to the application at hand [Basch et al. 1999]. For example, an event relevant for range search might be triggered when two items move to the same horizontal displacement (which happens when the $x$-ordering of the two items is about to switch). A different approach is to store each item's location and speed trajectory, so that no updating is needed as long as the item's trajectory plan does not change. Such an approach requires fewer updates, but the representation for each item generally has higher dimension, and the search strategies are therefore less efficient.

Kollios et al. [1999] developed a linear-space indexing scheme for moving points along a (one-dimensional) line, based upon the notion of partition trees. Their structure supports a variety of range search and approximate nearest neighbor queries. For example, given a range and time, the points in that range at the indicated time can be retrieved in $O(n^{1/2+\epsilon}+k)$ I/Os, for arbitrarily small $\epsilon > 0$. Updates require $O((\log n)^2)$ I/Os. Agarwal et al. [2000] extend the approach to handle range searches in two dimensions, and they improve the update bound to $O((\log_B n)^2)$ I/Os. They also propose an event-driven data structure with the same query times as the range search data structure of Arge et al. [1999b] discussed in Section 11.5, but with the potential need to do many updates. A hybrid data structure combining the two approaches permits a tradeoff between query performance and update frequency.

R-trees offer a practical generic mechanism for storing multidimensional points

and are thus a natural alternative for storing mobile items. One approach is to represent time as a separate dimension and to cluster trajectories using the R-tree heuristics. However, the orthogonal nature of the R-tree does not lend itself well to diagonal trajectories. For the case of points moving along linear trajectories, Šaltenis et al. [2000] build the R-tree upon only the spatial dimensions, but parameterize the bounding box coordinates to account for the movement of the items stored within. They maintain an outer approximation of the true bounding box, which they periodically update to refine the approximation. Agarwal and Har-Peled [2001] show how to maintain a provably good approximation of the minimum bounding box with need for only a constant number of refinement events.

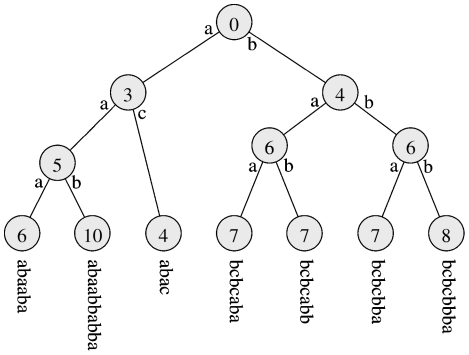


**Fig. 13**. Patricia trie representation of a single node of an SB-tree, with branching factor $B = 8$. The seven strings used for partitioning are pictured at the leaves; in the actual data structure, pointers to the strings, not the strings themselves, are stored at the leaves. The pointers to the $B$ children of the SB-tree node are also stored at the leaves.

## 13. STRING PROCESSING

In this section we survey methods used to process strings in external memory, such as inverted files, search trees, suffix trees and suffix arrays, and sorting, with particular attention to more recent developments.

### 13.1. Inverted Files

The simplest and most commonly used method to index text in large documents or collections of documents is the *inverted file*, which is analogous to the index at the back of a book. The words of interest in the text are sorted alphabetically, and each item in the sorted list has a list of pointers to the occurrences of that word in the text. In an EM setting, a hybrid approach makes sense, in which the text is divided into large chunks (consisting of one or more blocks) and an inverted file is used to specify the chunks containing each word; the search within a chunk can be carried out by using a fast sequential method, such as the Knuth–Morris–Pratt [1977] or Boyer–Moore [1977] methods. This particular hybrid method was introduced as the basis of the widely used GLIMPSE search tool [Manber and Wu 1994]. Another way to index text is to use hashing to get small signatures for portions of text. The reader is referred to Frakes and Baeza-Yates [1992] and Baeza-Yates and Ribeiro-Neto [1999] for more background on the above methods.

### 13.2. String B-Trees

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 into one or two blocks. However, if the keys are variable-sized text strings, the keys can be arbitrarily long, and 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. In order to save space in each node, Bayer and Unterauer [1977] investigated the use of prefix representations of keys. Ferragina and Grossi [1996; 1999] recently developed an elegant generalization of the B-tree called the *String B-tree* or simply *SB-tree* (not to be confused with the SB-tree [O'Neil 1992] mentioned in Section 10.1). An SB-tree differs from a conventional B-tree in the way that each $\Theta(B)$-way branching node is represented.

An individual node of Ferragina and Grossi's SB-tree is pictured in Figure 13.

It is based upon a variant of the *Patricia trie* character-based data structure [Knuth 1998; Morrison 1968] along the lines of Ajtai et al. [1984]. It achieves $B$-way branching with a total storage of $O(B)$ characters, which fit in $O(1)$ blocks. 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 Figure 13 the right child of the root has index 4 and its outgoing edges have character labels "a" and "b", which means that the node's left subtrie consists of strings whose position 4 (fifth character) is "a", and its right subtrie consists of strings whose position 4 (fifth character) is "b". The first 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 trie 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 13 traverses the far-right path of the Patricia trie, examining character positions 0, 4, and 6.

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 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 (position 3), and therefore the search string is lexicographically smaller than the entire right subtrie of the root. It thus fits in between the leaves "abac" and "bcbcaba".

Searching each 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 $\ell$ characters is therefore $O(\log_B N + \ell/B)$, which is optimal. Insertions and deletions can be done in the same I/O bound. Ferragina and Grossi [1996; 1999] apply SB-trees to the problems of string matching, prefix search, and substring search. Ferragina and Luccio [1998] apply SB-trees to get new results for dynamic dictionary matching; their structure even provides a simpler approach for the (internal memory) RAM model.

### 13.3. Suffix Trees and Suffix Arrays

Tries and Patricia tries are commonly used as internal memory data structures for storing sets of strings. One particularly interesting application of Patricia tries is to store the set of suffixes of a text string. The resulting data structure, called a *suffix tree* [McCreight 1976; Weiner 1973], can be built in linear time and supports search for an arbitrary substring of the text in time linear in the size of the substring. A more compact (but static) representation of a suffix tree, called a *suffix array* [Manber and Myers 1993], consisting of the leaves of the suffix tree in symmetric traversal order, can also be used for fast searching. (See Gusfield [1997] for general background.) Farach et al. [1998] show how to construct SB-trees, suffix trees, and suffix arrays on strings of total length $N$ using $O(n \log_m n)$ I/Os, which is optimal. Clark and Munro [1996] give a practical implementation of dynamic suffix trees that use about five bytes per indexed suffix. Crauser and Ferragina [1999] present an extensive set of experiments on various text collections in which they compare the practical performance of some novel and known suffix array construction algorithms.

### 13.4. Sorting Strings

Arge et al. [1997] 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*. The problem can be solved in the (internal memory) RAM model 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 $O(k \log_m k + n)$, where $k = \max\{1, K/B\}$. 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 13.1 [ARGE ET AL. 1997]. *The number of I/Os needed to sort $K$ strings of total length $N$, where there are $K_1$ short strings of total length $N_1$ and $K_2$ long strings of total length $N_2$ (i.e., $N = N_1 + N_2$ and $K = K_1 + K_2$), is*

$$O\left(\min\left\{\frac{N_1}{B}\log_m\left(\frac{N_1}{B}+1\right), K_1 \log_M(K_1+1)\right\} + K_2 \log_M(K_2+1) + \frac{N}{B}\right). \quad (13)$$

Lower bounds for various models of how strings can be manipulated are given in Arge et al. [1997]. There are gaps in some cases between the upper and lower bounds for sorting.

## 14. THE TPIE EXTERNAL MEMORY PROGRAMMING ENVIRONMENT

In this section we describe the TPIE (transparent parallel I/O environment)[3] [Arge et al. 1999a; TPIE 1999; Vengroff and Vitter 1996b], which serves as the implementation platform for the experiments described in Sections 7 and 11.2 as well as in several of the referenced papers. TPIE is a comprehensive set of C++ templates for EM paradigms and a run-time library. Its goal is to help programmers develop high-level, portable, and efficient implementations of EM algorithms.

There are three basic approaches to supporting development of I/O-efficient code, which we call *access-*, *array-* and *framework-oriented*. TPIE falls primarily into the third category with some elements of the first category. Access-oriented systems preserve the programmer abstraction of explicitly requesting data transfer. They often extend the read–write interface to include data type specifications and collective specification of multiple transfers, sometimes involving the memories of multiple processing nodes. Examples of access-oriented systems include the UNIX file system at the lowest level, higher-level parallel file systems such as Whiptail [Shriver and Wisniewski 1995], Vesta [Corbett and Feitelson 1996], PIOUS [Moyer and Sunderam 1996], and the High Performance Storage System [Watson and Coyne 1995], and I/O libraries MPI-IO [Corbett et al. 1996], and LEDA-SM [Crauser and Mehlhorn 1999].

Array-oriented systems access data stored in external memory primarily by means of compiler-recognized data types (typically arrays) and operations on those data types. The external computation is directly specified via iterative loops or explicitly data-parallel operations, and the system manages the explicit I/O transfers. Array-oriented systems are effective for scientific computations that make regular strides through arrays of data and can deliver high-performance parallel I/O in applications such as computational fluid dynamics, molecular dynamics, and weapon system design and simulation. Array-oriented systems are generally ill-suited to irregular or combinatorial computations. Examples of array-oriented systems include PASSION [Thakur et al. 1996], Panda [Seamons and Winslett 1996] (which also has aspects of access orientation), PI/OT [Parsons et al. 1997], and ViC* [Colvin and Cormen 1998].

TPIE [Arge et al. 1999a; TPIE 1999; Vengroff and Vitter 1996b] provides a framework-oriented interface for batched computation as well as an access-oriented

[3] The TPIE software distribution is available free of charge at http://www.cs.duke.edu/TPIE/ on the World Wide Web.

interface for online computation. Instead of viewing batched computation as an enterprise in which code reads data, operates on them, and writes results, a framework-oriented system 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. TPIE 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 feed the computation.

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 (or streaming), multiway distribution, and merging, which TPIE supports as access mechanisms. Batched programs in TPIE thus consist primarily of a call to one or more of these standard access mechanisms. For example, a distribution sort can be programmed by using the access mechanism for multiway distribution. The programmer has to specify the details as to how the partitioning elements are formed and how the buckets are defined. Then the multiway distribution is invoked, during which TPIE automatically forms the buckets and writes them to disk using double buffering.

For online data structures such as hashing, B-trees, and R-trees, TPIE supports more traditional block access like the access-oriented systems.

## 15. DYNAMIC MEMORY ALLOCATION

The amount of internal 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 [Pang et al. 1993a]. 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 degradation in performance.

Barve and Vitter [1999b] discuss the design and analysis of EM algorithms that adapt gracefully to changing memory allocations. In their model, without loss of generality, an algorithm (or program) $\mathcal{P}$ is allocated internal memory in phases. During the $i$th 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 algorithm.

For sorting, the lower bound approach of (10) 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 algorithm can perform more than a constant number of sorts in the worst case for the same sequence of memory allocations.

Barve and Vitter [1999b] define the model in generality and give dynamically optimal strategies for sorting, matrix multiplication, and buffer tree operations. Their work represents the first theoretical model of dynamic allocation and the first algorithms that can be considered dynamically optimal. Previous work was done on memory-adaptive algorithms for merge sort [Pang et al. 1993a; Zhang and Larson 1997] and hash join [Pang et al. 1993b], but the algorithms handle only special cases and can be made to perform nonoptimally for certain patterns of memory allocation.

## 16. CONCLUSIONS

In this survey 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 EM algorithms for efficient use of multiple disks. Optimal bounds have not yet been determined for several basic EM graph problems such as topological sorting, shortest paths, breadth 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. Several problems remain open in the dynamic and kinetic settings, such as range searching, ray shooting, point location, and finding nearest neighbors.

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. In practice, algorithms cannot assume a static internal memory allocation; they must adapt in a robust way when the memory allocation changes.

Many interesting challenges and opportunities in algorithm design and analysis arise from new architectures being developed, such as networks of workstations, hierarchical storage devices, disk drives with processing capabilities, and storage devices based upon microelectromechanical systems (MEMS). Active (or intelligent) disks, in which disk drives have some processing capability and can filter information sent to the host, have recently been proposed to further reduce the I/O bottleneck, especially in large database applications [Acharya et al. 1998; Riedel et al. 1998]. MEMS-based nonvolatile storage has the potential to serve as an intermediate level in the memory hierarchy between DRAM and disks. It could ultimately provide better latency and bandwidth than disks, at less cost per bit than DRAM [Schlosser et al. 2000; Vettiger et al. 2000].

## ACKNOWLEDGMENTS

The author wishes to thank Pankaj Agarwal, Lars Arge, Ricardo Baeza-Yates, Adam Buchsbaum, Jeff Chase, David Hutchinson, Vasilis Samoladas, Amin Vahdat, the members of the Center for Geometric Computing at Duke University, and the referees for several helpful comments and suggestions. Figure 1 is a modified version of a figure by Darren Vengroff, and Figure 2 comes from Cormen et al. [1990]. Figures 6, 7, 9, 10, 12, and 13 are modified versions of figures in Arge et al. [1998a] Enbody and Du [1998], Kanellakis et al. [1996], Arge et al. [1999a,b], and Ferragina and Grossi [1999], respectively.

## REFERENCES

ABEL, D. J. 1984. A $B^+$-tree structure for large quadtrees. *Computer Vision, Graphics, and Image Processing 27*, 1 (July), 19–31.

ABELLO, J., BUCHSBAUM, A., AND WESTBROOK, J. 1998. A functional approach to external graph

algorithms. In *Proceedings of the European Symposium on Algorithms*, Vol. 1461 of *Lecture Notes in Computer Science* (Venice, August). Springer-Verlag, 332–343.

ACHARYA, A., UYSAL, M., AND SALTZ, J. 1998. Active disks: Programming model, algorithms and evaluation. *ACM SIGPLAN Notices 33*, 11 (Nov), 81–91.

ADLER, M. 1996. New coding techniques for improved bandwidth utilization. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, Vol. 37 (Burlington, VT, Oct.), 173–182.

AGARWAL, P. K. AND ERICKSON, J. 1999. Geometric range searching and its relatives. In B. Chazelle, J. E. Goodman, and R. Pollack, Eds, *Advances in Discrete and Computational Geometry*, Vol. 23 of *Contemporary Mathematics*, American Mathematical Society, Providence, RI, 1–56.

AGARWAL, P. K. AND HAR-PELED, S. 2000. Maintaining the approximate extent measures of moving points. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, Vol. 12 (Washington, Jan.), 148–157.

AGARWAL, P. K., ARGE, L., AND ERICKSON, J. 2000. Indexing moving points. In *Proceedings of the ACM Symposium on Principles of Database Systems*, Vol. 19, 175–186.

AGARWAL, P. K., ARGE, L., BRODAL, G. S., AND VITTER, J. S. 1999. I/O-efficient dynamic point location in monotone planar subdivisions. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, Vol. 10, 11–20.

AGARWAL, P. K., ARGE, L., ERICKSON, J., FRANCIOSA, P. G., AND VITTER, J. S. 1998a. Efficient searching with linear constraints. In *Proceedings of the ACM Symposium on Principles of Database Systems*, Vol. 17, 169–178.

AGARWAL, P. K., ARGE, L., MURALI, T. M., VARADARAJAN, K., AND VITTER, J. S. 1998b. I/O-efficient algorithms for contour line extraction and planar graph blocking. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, Vol. 9, 117–126.

AGARWAL, P. K., ARGE, L., PROCOPIUC, O., AND VITTER, J. S. 2001a. A framework for index dynamization. In *Proceedings of the International Colloquium on Automata, Languages, and Programming* (Crete, Greece, July), Vol. 2076 of *Lecture Notes in Computer Science*, Springer-Verlag.

AGARWAL, P. K., DE BERG, M., GUDMUNDSSON, J., HAMMAR, M., AND HAVERKORT, H. J. 2001b. Constructing box trees and R-trees with low stabbing number. In *Proceedings of the ACM Symposium on Computational Geometry* (Medford, MA, June), Vol. 17.

AGGARWAL, A. AND PLAXTON, C. G. 1994. Optimal parallel sorting in multi-level storage. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, Vol. 5, 659–668.

AGGARWAL, A. AND VITTER, J. S. 1988. The input/output complexity of sorting and related problems. *Communications of the ACM 31*, 9, 1116–1127.

AGGARWAL, A., ALPERN, B., CHANDRA, A. K., AND SNIR, M. 1987a. A model for hierarchical memory. In *Proceedings of the ACM Symposium on Theory of Computing*, Vol. 19 (New York), 305–314.

AGGARWAL, A., CHANDRA, A., AND SNIR, M. 1987b. Hierarchical memory with block transfer. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, Vol. 28 (Los Angeles), 204–216.

AJTAI, M., FREDMAN, M., AND KOMLOS, J. 1984. Hash functions for priority queues. *Information and Control*, *63*, 3, 217–225.

ALPERN, B., CARTER, L., FEIG, E., AND SELKER, T. 1994. The uniform memory hierarchy model of computation. *Algorithmica 12*, 2–3, 72–109.

ARGE, L. 1995a. The buffer tree: A new technique for optimal I/O-algorithms. In *Proceedings of the Workshop on Algorithms and Data Structures*, Vol. 955 of *Lecture Notes in Computer Science*, Springer-Verlag, 334–345. A complete version appears as BRICS Technical Report RS–96–28, University of Aarhus.

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

ARGE, L. 1997. External-memory algorithms with applications in geographic information systems. In M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer, eds, *Algorithmic Foundations of GIS*, Vol. 1340 of *Lecture Notes in Computer Science*, Springer-Verlag, 213–254.

ARGE, L. AND MILTERSEN, P. 1999. On showing lower bounds for external-memory computational geometry problems. In J. Abello and J. S., Vitter, Eds., *External Memory Algorithms and Visualization*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, Providence, RI, 139–159.

ARGE, L. AND VAHRENHOLD, J. 2000. I/O-efficient dynamic planar point location. In *Proceedings of the ACM Symposium on Computational Geometry* (June), Vol. 9, 191–200.

ARGE, L. AND VITTER, J. S. 1996. Optimal dynamic interval management in external memory. In *Proceedings of the IEEE Symposium on Foundations of Computer Science* (Burlington, VT, Oct.), Vol. 37, 560–569.

ARGE, L., BRODAL, G. S., AND TOMA, L. 2000a. On external memory MST, SSSP and multi-way planar graph separation. In *Proceedings of the Scandinavian Workshop on Algorithmic Theory* (July) Vol. 1851 of *Lecture Notes in Computer Science*, Springer-Verlag.

ARGE, L., FERRAGINA, P., GROSSI, R., AND VITTER, J. 1997. On sorting strings in external memory.

In *Proceedings of the ACM Symposium on Theory of Computing*, Vol. 29, 540–548.

ARGE, L., HINRICHS, K. H., VAHRENHOLD, J., AND VITTER, J. S. 1999a. Efficient bulk operations on dynamic R-trees. In *Workshop on Algorithm Engineering and Experimentation*, Vol. 1619 of *Lecture Notes in Computer Science* (Baltimore, Jan.) Springer-Verlag, 328–348.

ARGE, L., KNUDSEN, M., AND LARSEN, K. 1993. A general lower bound on the I/O-complexity of comparison-based algorithms. In *Proceedings of the Workshop on Algorithms and Data Structures*, Vol. 709 of *Lecture Notes in Computer Science*, Springer-Verlag, 83–94.

ARGE, L., PROCOPIUC, O., RAMASWAMY, S., SUEL, T., VAHRENHOLD, J., AND VITTER, J. S. 2000b. A unified approach for indexed and non-indexed spatial joins. In *Proceedings of the International Conference on Extending Database Technology* (Konstanz, Germany, March), Vol. 7.

ARGE, L., PROCOPIUC, O., RAMASWAMY, S., SUEL, T., AND VITTER, J. S. 1998a. Scalable sweeping-based spatial join. In *Proceedings of the International Conference on Very Large Databases* (New York, August) Vol. 24, 570–581.

ARGE, L., PROCOPIUC, O., RAMASWAMY, S., SUEL, T., AND VITTER, J. S. 1998b. Theory and practice of I/O-efficient algorithms for multidimensional batched searching problems. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, Vol. 9, 685–694.

ARGE, L., SAMOLADAS, V., AND VITTER, J. S. 1999b. Two-dimensional indexability and optimal range search indexing. In *Proceedings of the ACM Conference Principles of Database Systems* (Philadelphia, May–June), Vol. 18, 346–357.

ARGE, L., VENGROFF, D. E., AND VITTER, J. S. 1995. 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*, (Sept.), Vol. 979 of *Lecture Notes in Computer Science*, Springer-Verlag 295–310.

ARMEN, C. 1996. Bounds on the separation of two parallel disk models. In *Proceedings of the Workshop on Input/Output in Parallel and Distributed Systems* (Philadelphia, May), Vol. 4, 122–127.

BAEZA-YATES, R. 1996. Bounded disorder: The effect of the index. *Theoretical Computer Science 168*, 21–38.

BAEZA-YATES, R. AND RIBEIRO-NETO, B. Eds. 1999. *Modern Information Retrieval*. Addison-Wesley Longman: Chapter 8.

BAEZA-YATES, R. AND SOZA-POLLMAN, H. 1998. Analysis of linear hashing revisited. *Nordic Journal of Computing 5*, 70–85.

BAEZA-YATES, R. A. 1989. Expected behaviour of $B^+$-trees under random insertions. *Acta Informatica 26*, 5, 439–472.

BAEZA-YATES, R. A. AND LARSON, P.-A. 1989. Performance of $B^+$-trees with partial expansions. *IEEE Transactions on Knowledge and Data Engineering 1*, 2 (June), 248–257.

BARVE, R. D. AND VITTER, J. S. 1999a. A simple and efficient parallel disk mergesort. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures* (St. Malo, France, June), Vol. 11, 232–241.

BARVE, R. D. AND VITTER, J. S. 1999b. A theoretical framework for memory-adaptive algorithms. In *Proceedings of the IEEE Symposium on Foundations of Computer Science* (New York, Oct.), Vol. 40, 273–284.

BARVE, R. D., GROVE, E. F., AND VITTER, J. S. 1997. Simple randomized mergesort on parallel disks. *Parallel Computing 23*, 4, 601–631.

BARVE, R. D., KALLAHALLA, M., VARMAN, P. J., AND VITTER, J. S. 2000. Competitive analysis of buffer management algorithms. *Journal of Algorithms 36* (August).

BARVE, R. D., SHRIVER, E. A. M., GIBBONS, P. B., HILLYER, B. K., MATIAS, Y., AND VITTER, J. S. 1999. Modeling and optimizing I/O throughput of multiple disks on a bus. In *Procedings of ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems* (Atlanta, GA, May), 83–92.

BASCH, J., GUIBAS, L. J., AND HERSHBERGER, J. 1999. Data structures for mobile data. *Journal of Algorithms 31*, 1–28.

BAYER, R. AND MCCREIGHT, E. 1972. Organization of large ordered indexes. *Acta Informatica 1*, 173–189.

BAYER, R. AND UNTERAUER, K. 1977. Prefix B-trees. *ACM Transactions on Database Systems 2*, 1 (March), 11–26.

BECKER, B., GSCHWIND, S., OHLER, T., SEEGER, B., AND WIDMAYER, P. 1996. An asymptotically optimal multiversion B-tree. *VLDB Journal 5*, 4 (Dec.), 264–275.

BECKMANN, N., KRIEGEL, H.-P., SCHNEIDER, R., AND SEEGER, B. 1990. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 322–331.

BENDER, M. A., DEMAINE, E. D., AND FARACH-COLTON, M. 2000. Cache-oblivious B-trees. In *Proceedings of the IEEE Symposium on Foundations of Computer Science* (Redondo Beach, Cal., Nov.), Vol. 41, 12–14.

BENTLEY, J. L. 1980. Multidimensional divide and conquer. *Communications of the ACM 23*, 6, 214–229.

BENTLEY, J. L. AND SAXE, J. B. 1980. Decomposable searching problems I: Static-to-dynamic transformations. *Journal of Algorithms 1*, 4 (Dec.), 301–358.

BERCHTOLD, S., BÖHM, C., AND KRIEGEL, H.-P. 1998. Improving the query performance of

high-dimensional index structures by bulk load operations. In *Proceedings of the International Conference on Extending Database Technology*, Vol. 1377 of *Lecture Notes in Computer Science*, Springer-Verlag, 216–230.

BLUM, N. AND MEHLHORN, K. 1980. On the average number of rebalancing operations in weight-balanced trees. *Theoretical Computer Science 11*, 3 (July), 303–320.

BOYER, R. S. AND MOORE, J. S. 1977. A fast string searching algorithm. *Communications of the ACM 20*, 10 (Oct.), 762–772.

BRENGEL, K., CRAUSER, A., FERRAGINA, P., AND MEYER, U. 1999. An experimental study of priority queues. In J. S. Vitter and C. Zaroliagis, Ed., *Proceedings of the Workshop on Algorithm Engineering*, Vol. 1668 of *Lecture Notes in Computer Science* (London, July), Springer-Verlag, 345–359.

BRODAL, G. S. AND KATAJAINEN, J. 1998. Worst-case efficient external-memory priority queues. In *Proceedings of the Scandinavian Workshop on Algorithmic Theory*, Vol. 1432 of *Lecture Notes in Computer Science* (Stockholm, July), Springer-Verlag, 107–118.

BUCHSBAUM, A. L., GOLDWASSER, M., VENKATASUBRAMANIAN, S., AND WESTBROOK, J. R. 2000. On external memory graph traversal. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms* (Jan.), Vol. 11.

CALLAHAN, P., GOODRICH, M. T., AND RAMAIYER, K. 1995. Topology B-trees and their applications. In *Proceedings of the Workshop on Algorithms and Data Structures*, Vol. 955 of *Lecture Notes in Computer Science*, Springer-Verlag, 381–392.

CARTER, L. AND GATLIN, K. S. 1998. Towards an optimal bit-reversal permutation program. In *Proceedings of the IEEE Symposium on Foundations of Computer Science* (Palo Alto, Nov.), Vol. 39, 544–553.

CHAZELLE, B. 1986. Filtering search: A new approach to query-answering. *SIAM Journal on Computing 15*, 703–724.

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

CHAZELLE, B. AND EDELSBRUNNER, H. 1987. Linear space data structures for two types of range search. *Discrete and Computational Geometry 2*, 113–126.

CHEN, P. M., LEE, E. K., GIBSON, G. A., KATZ, R. H., AND PATTERSON, D. A. 1994. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys 26*, 2 (June), 145–185.

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

CHIANG, Y.-J., GOODRICH, M. T., GROVE, E. F., TAMASSIA, R., VENGROFF, D. E., AND VITTER, J. S. 1995. External-memory graph algorithms. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms* (Jan.), Vol. 6, 139–149.

CHIANG, Y.-J. AND SILVA, C. T. 1999. External memory techniques for isosurface extraction in scientific visualization. In J. Abello and J. S. Vitter, Eds., *External Memory Algorithms and Visualization*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, Providence, RI. 247–277.

CLARK, D. R. AND MUNRO, J. I. 1996. Efficient suffix trees on secondary storage. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms* (Atlanta, June), Vol. 7, 383–391.

CLARKSON, K. L. AND SHOR, P. W. 1989. Applications of random sampling in computational geometry, II. *Discrete and Computational Geometry 4*, 387–421.

COLVIN, A. AND CORMEN, T. H. 1998. ViC*: A compiler for virtual-memory c*. In *Proceedings of the International Workshop on High-Level Programming Models and Supportive Environments*, Vol. 3.

COMER, D. 1979. The ubiquitous B-tree. *ACM Computing Surveys 11*, 2, 121–137.

CORBETT, P., FEITELSON, D., FINEBERG, S., HSU, Y., NITZBERG, B., PROST, J.-P., SNIR, M., TRAVERSAT, B., AND WONG, P. 1996. Overview of the MPI-IO parallel I/O interface. In R., Jain, J., Werth, and J. C. Browne, Eds., *Input/Output in Parallel and Distributed Computer Systems*, Vol. 362 of *The Kluwer International Series in Engineering and Computer Science*, Kluwer Academic, Chapter 5, 127–146.

CORBETT, P. F. AND FEITELSON, D. G. 1996. The Vesta parallel file system. *ACM Transactions on Computer Systems 14*, 3 (August), 225–264.

CORMEN, T. H. AND NICOL, D. M. 1998. Performing out-of-core FFTs on parallel disk systems. *Parallel Computing 24*, 1 (Jan.), 5–20.

CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. 1990. *Introduction to Algorithms*. MIT Press, Cambridge, MA.

CORMEN, T. H., SUNDQUIST, T., AND WISNIEWSKI, L. F. 1999. Asymptotically tight bounds for performing BMMC permutations on parallel disk systems. *SIAM Journal on Computing 28*, 1, 105–136.

CRAUSER, A. AND FERRAGINA, P. 1999. On constructing suffix arrays in external memory. In *Proceedings of the European Symposium on Algorithms*, Vol. 1643 of *Lecture Notes in Computer Science*, Springer-Verlag.

CRAUSER, A. AND MEHLHORN, K. 1999. LEDA-SM: Extending LEDA to secondary memory. In J. S., Vitter, and C., Zaroliagis, Eds., *Proceedings of the European Symposium on Algorithms*, Vol. 1643 of *Lecture Notes in Computer Science* (London, July), Springer-Verlag, 228–242.

CRAUSER, A., FERRAGINA, P., MEHLHORN, K., MEYER, U., AND RAMOS, E. A. 1998. Randomized external-memory algorithms for geometric problems. In

*Proceedings of the ACM Symposium on Computational Geometry* (June), Vol. 14, 259–268.

CRAUSER, A., FERRAGINA, P., MEHLHORN, K., MEYER, U., AND RAMOS, E. A. 1999. I/O-optimal computation of segment intersections. In J., Abello, and J. S. Vitter, Eds., *External Memory Algorithms and Visualization*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, Providence, RI, 131–138.

CYPHER, R. AND PLAXTON, G. 1993. Deterministic sorting in nearly logarithmic time on the hypercube and related computers. *Journal of Computer and System Sciences 47*, 3, 501–548.

DE BERG, M., GUDMUNDSSON, J., HAMMAR, M., AND OVERMARS, M. 2000. On R-trees with low stabbing number. In *Proceedings of the European Symposium on Algorithms*, Vol. 1879 of *Lecture Notes in Computer Science* (Saarbrücken, Germany, Sept.), Springer-Verlag, 167–178.

DE BERG, M., VAN KREVELD, M., OVERMARS, M., AND SCHWARZKOPF, O. 1997. *Computational Geometry Algorithms and Applications*. Springer-Verlag, Berlin.

DEHNE, F., DITTRICH, W., AND HUTCHINSON, D. 1997. Efficient external memory algorithms by simulating coarse-grained parallel algorithms. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures* (June), Vol. 9, 106–115.

DEHNE, F., HUTCHINSON, D., AND MAHESHWARI, A. 1999. Reducing I/O complexity by simulating coarse grained parallel algorithms. In *Proceedings of the International Parallel Processing Symmposium* (April), Vol. 13, 14–20.

DEMUTH, H. B. 1956. *Electronic data sorting*. Ph.D., Stanford University. A shortened version appears in *IEEE Transactions on Computing C-34*, 4, (April), 296–310, 1985, Special Issue on Sorting, E. E. Lindstrom, C. K. Wong, and J. S. Vitter, Eds.

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

DEWITT, D. J., NAUGHTON, J. F., AND SCHNEIDER, D. A. 1991. Parallel sorting on a shared-nothing architecture using probabilistic splitting. In *Proceedings of the International Conference on Parallel and Distributed Information Systems* (Dec.), Vol. 1, 280–291.

DITTRICH, W., HUTCHINSON, D., AND MAHESHWARI, A. 1998. Blocking in parallel multisearch problems. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, Vol. 10, 98–107.

DRISCOLL, J. R., SARNAK, N., SLEATOR, D. D., AND TARJAN, R. E. 1989. Making data structures persistent. *Journal of Computer and System Sciences 38*, 86–124.

EASTON, M. C. 1986. Key-sequence data sets on indelible storage. *IBM Journal of Research and Development 30*, 230–241.

EDELSBRUNNER, H. 1983a. A new approach to rectangle intersections, part I. *International Journal of Computer Mathematics 13*, 209–219.

EDELSBRUNNER, H. 1983b. A new approach to rectangle intersections, part II. *International Journal of Computer Mathematics 13*, 221–229.

ENBODY, R. J. AND DU, H. C. 1988. Dynamic hashing schemes. *ACM Computing Surveys 20*, 2 (June), 85–113.

EPPSTEIN, D., GALIL, Z., ITALIANO, G. F., AND NISSENZWEIG, A. 1997. Sparsification—A technique for speeding up dynamic graph algorithms. *Journal of the ACM 44*, 5, 669–96.

EVANGELIDIS, G., LOMET, D. B., AND SALZBERG, B. 1997. The $hB^{\Pi}$-tree: A multi-attribute index supporting concurrency, recovery and node consolidation. *VLDB Journal 6*, 1–25.

FAGIN, R., NIEVERGELT, J., PIPPINGER, N., AND STRONG, H. R. 1979. Extendible hashing—A fast access method for dynamic files. *ACM Transactions on Database Systems 4*, 3, 315–344.

FARACH, M., FERRAGINA, P., AND MUTHUKRISHNAN, S. 1998. Overcoming the memory bottleneck in suffix tree construction. In *Proceedings of the IEEE Symposium on Foundations of Computer Science* (Palo Alto, Nov.), Vol. 39, 174–183.

FEIGENBAUM, J., KANNAN, S., STRAUSS, M., AND VISWANATHAN, M. 1999. An approximate l1-difference algorithm for massive data streams. In *Proceedings of the IEEE Symposium on Foundations of Computer Science* (New York, Oct.), Vol. 40, 501–511.

FELLER, W. 1968. *An Introduction to Probability Theory and its Applications*, Vol. 1, 3rd edition. John Wiley, New York.

FERRAGINA, P. AND GROSSI, R. 1996. Fast string searching in secondary storage: Theoretical developments and experimental results. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms* (Atlanta, June), Vol. 7, 373–382.

FERRAGINA, P. AND GROSSI, R. 1999. The string B-tree: A new data structure for string search in external memory and its applications. *Journal of the ACM 46*, 2 (March), 236–280.

FERRAGINA, P. AND LUCCIO, F. 1998. Dynamic dictionary matching in external memory. *Information and Computation 146*, 2 (Nov.), 85–99.

FLAJOLET, P. 1983. On the performance evaluation of extendible hashing and trie searching. *Acta Informatica 20*, 4, 345–369.

FLOYD, R. W. 1972. Permuting information in idealized two-level storage. In R. Miller and J. Thatcher, Eds., *Complexity of Computer Computations*. Plenum, New York, 105–109.

FRAKES, W. AND BAEZA-YATES, R. Eds. 1992. *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall.

FRIGO, M., LEISERSON, C. E., PROKOP, H., AND RAMACHANDRAN, S. 1999. Cache-oblivious algorithms. In *Proceedings of the IEEE*

*Symposium on Foundations of Computer Science*, Vol. 40.

FUNKHOUSER, T. A., SEQUIN, C. H., AND TELLER, S. J. 1992. Management of large amounts of data in interactive building walkthroughs. In *Proceedings of the ACM SIGGRAPH Conference on Computer Graphics* (Boston, March), 11–20.

GAEDE, V. AND GÜNTHER, O. 1998. Multidimensional access methods. *ACM Computing Surveys 30*, 2 (June), 170–231.

GANGER, G. R. 1995. Generating representative synthetic workloads: An unsolved problem. In *Proceedings of the Computer Measurement Group Conference* (Dec.), Vol. 21, 1263–1269.

GARDNER, M. 1977. *Magic Show*. Knopf, New York, Chapter 7.

GARGANTINI, I. 1982. An effective way to represent quadtrees. *Communications of the ACM 25*, 12 (Dec.), 905–910.

GIBSON, G. A., VITTER, J. S., AND WILKES, J. 1996. Report of the working group on storage I/O issues in large-scale computing. *ACM Computing Surveys 28*, 4 (Dec.), 779–793.

GIONIS, A., INDYK, P., AND MOTWANI, R. 1999. Similarity search in high dimensions via hashing. In *Proceedings of the International Conference on Very Large Databases* (Edinburgh), Vol. 25, Morgan Kaufmann, San Mateo, CA, 78–89.

GOLDMAN, R., SHIVAKUMAR, N., VENKATASUBRAMANIAN, S., AND GARCIA-MOLINA, H. 1998. Proximity search in databases. In *Proceedings of the International Conference on Very Large Databases* (August), Vol. 24, 26–37.

GOODRICH, M. T., TSAY, J.-J., VENGROFF, D. E., AND VITTER, J. S. 1993. External-memory computational geometry. In *Proceedings of the IEEE Symposium on Foundations of Computer Science* Palo Alto (Nov.), Vol. 34, 714–723.

GOVINDARAJAN, S., LUKOVSZKI, T., MAHESHARI, A., AND ZEH, N. 2000. I/O-efficient well-separated pair decomposition and its applications. In *Proceedings of the European Symposium on Algorithms* (Saarbrücken, Germany, Sept.), Vol. 1879 of *Lecture Notes in Computer Science*, Springer-Verlag.

GREENE, D. 1989. An implementation and performance analysis of spatial data access methods. In *Proceedings of IEEE International Conference on Data Engineering*, Vol. 5, 606–615.

GRIFFIN, J. L., SCHLOSSER, S. W., GANGER, G. R., AND NAGLE, D. F. 2000. Modeling and performance of MEMS-based storage devices. In *Procedings of ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems* (Santa Clara, CA, June).

GROSSI, R. AND ITALIANO, G. F. 1999. Efficient cross-trees for external memory. In J. Abello and J. S. Vitter, Eds., *External Memory Algorithms and Visualization*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, Providence, RI, 87–106.

GROSSI, R. AND ITALIANO, G. F. 1997. Efficient splitting and merging algorithms for order decomposable problems. *Information and Computation* (in press). An earlier version appears in *Proceedings of the International Colloquium on Automata, Languages and Programming*, Vol. 1256 of *Lecture Notes in Computer Science*, Springer-Verlag, 605–615.

GUPTA, S. K. S., LI, Z., AND REIF, J. H. 1995. Generating efficient programs for two-level memories from tensor-products. In *Proceedings of the IASTED/ISMM International Conference on Parallel and Distributed Computing and Systems* (Washington, DC, Oct.), Vol. 7, 510–513.

GUSFIELD, D. 1997. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, Cambridge, UK.

GUTTMAN, A. 1984. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 47–57.

HELLERSTEIN, J. M., KOUTSOUPIAS, E., AND PAPADIMITRIOU, C. H. 1997. On the analysis of indexing schemes. In *Proceedings of the ACM Symposium on Principles of Database Systems* (Tucson, AZ, May), Vol. 16, 249–256.

HELLERSTEIN, L., GIBSON, G., KARP, R. M., KATZ, R. H., AND PATTERSON, D. A. 1994. Coding techniques for handling failures in large disk arrays. *Algorithmica 12*, 2–3, 182–208.

HENZINGER, M. R., RAGHAVAN, P., AND RAJAGOPALAN, S. 1999. Computing on data streams. In J. Abello and J. S. Vitter, Eds., *External Memory Algorithms and Visualization*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, Providence, RI, 107–118.

HINRICHS, K. H. 1985. The grid file system: Implementation and case studies of applications. PhD Thesis, Dept. Information Science, ETH, Zürich.

HONG, J. W. AND KUNG, H. T. 1981. I/O complexity: The red-blue pebble game. In *Proceedings of the ACM Symposium on Theory of Computing* (May), Vol. 13, 326–333.

HUTCHINSON, D., MAHESHWARI, A., SACK, J.-R., AND VELICESCU, R. 1997. Early experiences in implementing the buffer tree. In *Proceedings of the Workshop on Algorithm Engineering*, Vol. 1.

HUTCHINSON, D., MAHESHWARI, A., AND ZEH, N. 1999. An external memory data structure for shortest path queries. In *Proceedings of the International Conference on Computing and Combinatorics* (July), Vol. 1627 of *Lecture Notes in Computer Science*, Springer-Verlag, 51–60.

HUTCHINSON, D. A., SANDERS, P., AND VITTER, J. S. 2001a. Duality between prefetching and queued writing with applications to integrated caching and prefetching and to external sorting. In *Proceedings of the European Symposium on Algorithms* (Århus, Denmark, August),

Vol. 2161 of Lecture Notes in Computer Science, Springer-Verlag.

HUTCHINSON, D. A., SANDERS, P., AND VITTER, J. S. 2001b. The power of duality for prefetching and sorting with parallel disks. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures* (Crete, Greece, July), Vol. 2076 of Lecture Notes in Computer Science, Springer-Verlag.

HUTCHINSON, D. A., SANDERS, P., AND VITTER, J. S. 2001c. Notes.

INDYK, P., MOTWANI, P. R., RAGHAVAN, P., AND VEMPALA, S. 1997. Locality-preserving hashing in multidimensional spaces. In *Proceedings of the ACM Symposium on Theory of Computing* (El Paso, TX, May), Vol. 29, 618–625.

JENSEN, D. P. C. S. AND THEODORIDIS, Y. 2000. Novel approaches to the indexing of moving object trajectories. In *Proceedings of the International Conference on Very Large Databases* (Cairo), Vol. 26, 395–406.

KALLAHALLA, M. AND VARMAN, P. J. 1999. Optimal read-once parallel disk scheduling. In *Proceedings of the Workshop on Input/Output in Parallel and Distributed Systems* (Atlanta, GA, May), Vol. 6, 68–77, ACM Press, New York.

KALLAHALLA, M. AND VARMAN, P. J. 2001. Optimal prefetching and caching for parallel i/o systems. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures* (Crete, July), Vol. 13.

KAMEL, I. AND FALOUTSOS, C. 1993. On packing R-trees. In *Proceedings of the International ACM Conference on Information and Knowledge Management*, Vol. 2, 490–499.

KAMEL, I., AND FALOUTSOS, C. 1994. Hilbert R-tree: An improved R-tree using fractals. In *Proceedings of the International Conference on Very Large Databases*, Vol. 20, 500–509.

KAMEL, I., KHALIL, M., AND KOURAMAJIAN, V. 1996. Bulk insertion in dynamic R-trees. In *Proceedings of the International Symposium on Spatial Data Handling*, Vol. 4, 3B, 31–42.

KANELLAKIS, P. C., KUPER, G. M., AND REVESZ, P. Z. 1990. Constraint query languages. In *Proceedings of the ACM Conference Principles of Database Systems*, Vol. 9, 299–313.

KANELLAKIS, P. C., RAMASWAMY, S., VENGROFF, D. E., AND VITTER, J. S. 1996. Indexing for data models with constraints and classes. *Journal of Computer and System Sciences 52*, 3, 589–612.

KANTH, K. V. R. AND SINGH, A. K. 1999. Optimal dynamic range searching in non-replicating index structures. In *Proceedings of the International Conference on Database Theory* (Jan.), Vol. 1540 of *Lecture Notes in Computer Science*, Springer-Verlag, 257–276.

KIM, M. Y. 1986. Synchronized disk interleaving. *IEEE Transactions on Computers 35*, 11 (Nov.), 978–988.

KIRKPATRICK, D. G. AND SEIDEL, R. 1986. The ultimate planar convex hull algorithm? *SIAM Journal on Computing 15*, 287–299.

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

KNUTH, D. E. 1999. *MMIXware*. Springer, Berlin.

KNUTH, D. E., MORRIS, J. H., AND PRATT, V. R. 1977. Fast pattern matching in strings. *SIAM Journal on Computing 6*, 323–350.

KOLLIOS, G., GUNOPULOS, D., AND TSOTRAS, V. J. 1999. On indexing mobile objects. In *Proceedings of the ACM Symposium on Principles of Database Systems*, Vol. 18, 261–272.

KOUTSOUPIAS, E. AND TAYLOR, D. S. 1998. Tight bounds for 2-dimensional indexing schemes. In *Proceedings of the ACM Symposium on Principles of Database Systems* (Seattle, June), Vol. 17, 52–58.

KRISHNAMURTHY, R. AND WANG, K.-Y. 1985. Multilevel grid files. Technical Report, IBM T. J. Watson Center, Yorktown Heights, NY, November.

KUMAR, V. AND SCHWABE, E. 1996. Improved algorithms and data structures for solving graph problems in external memory. In *Proceedings of the IEEE Symposium on Parallel and Distributed Processing* (Oct.), Vol. 8, 169–176.

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

LARSON, P.-A. 1982. Performance analysis of linear hashing with partial expansions. *ACM Transactions on Database Systems 7*, 4 (Dec.), 566–587.

LAURINI, R. AND THOMPSON, D. 1992. *Fundamentals of Spatial Information Systems*. Academic Press.

LEHMAN, P. L. AND YAO, S. B. 1981. Efficient locking for concurrent operations on B-Trees. *ACM Transactions on Database Systems 6*, 4 (Dec.), 650–570.

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

LEISERSON, C. E., RAO, S., AND TOLEDO, S. 1993. Efficient out-of-core algorithms for linear relaxation using blocking covers. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, Vol. 34, 704–713.

LI, Z., MILLS, P. H., AND REIF, J. H. 1996. Models and resource metrics for parallel and distributed computation. *Parallel Algorithms and Applications 8*, 35–59.

LITWIN, W. 1980. Linear hashing: A new tool for files and tables addressing. In *Proceedings of the International Conference on Very Large Databases* (Montreal, Oct.), Vol. 6, 212–223.

LITWIN, W. AND LOMET, D. 1987. A new method for fast data searches with keys. *IEEE Software 4*, 2 (March), 16–24.

LOMET, D. 1988. A simple bounded disorder file organization with good performance. *ACM Transactions on Database Systems 13*, 4, 525–551.

LOMET, D. B. AND SALZBERG, B. 1990. The hB-tree: A multiattribute indexing method with good guaranteed performance. *ACM Transactions on Database Systems 15*, 4, 625–658.

LOMET, D. B. AND SALZBERG, B. 1997. Concurrency and recovery for index trees. *VLDB Journal 6*, 3, 224–240.

MAHESHWARI, A. AND ZEH, N. External memory algorithms for outerplanar graphs. In *Proceedings of the International Conference on Computing and Combinatorics* (July), Vol. 1627 of *Lecture Notes in Computer Science*, Springer-Verlag, 51–60.

MAHESHWARI, A. AND ZEH, N. 2001. I/O-efficient algorithms for bounded treewidth graphs. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms* (Washington, DC, Jan.), Vol. 12.

MANBER, U. AND MYERS, G. 1993. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing 22*, 5 (Oct.), 935–948.

MANBER, U. AND WU, S. 1994. GLIMPSE: A tool to search through entire file systems. In USENIX Association, Ed., *Proceedings of the Winter USENIX Conference* (San Francisco, Jan.), 23–32.

MARTIN, G. N. N. 1979. Spiral storage: Incrementally augmentable hash addressed storage. Technical Report CS-RR-027, University of Warwick, March.

MATIAS, Y., SEGAL, E., AND VITTER, J. S. 2000. Efficient bundle sorting. In *Proceedings of the ACM–SIAM Symposium on Discrete Algorithms* (San Francisco, Jan.), Vol. 11, 839–848.

MCCREIGHT, E. M. 1976. A space-economical suffix tree construction algorithm. *Journal of the ACM 23*, 2, 262–272.

MCCREIGHT, E. M. 1985. Priority search trees. *SIAM Journal on Computing 14*, 2 (May), 257–276.

MENDELSON, H. 1982. Analysis of extendible hashing. *IEEE Transactions on Software Engineering SE–8* (Nov.), 611–619.

MEYER, U. 2001. External memory BFS on undirected graphs with bounded degree. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms* (Washington, DC, Jan), Vol. 12.

Microsoft 1998. TerraServer online database of satellite images, available on the World-Wide Web at http://terraserver.microsoft.com/.

MOHAN, C. 1990 ARIES/KVL: A key-value locking method for concurrency control of multiaction transactions on B-tree indices. In *Proceedings of the International Conference on Very Large Databases* (Brisbane, August), Vol. 16, 392.

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

MOYER, S. A. AND SUNDERAM, V. 1996. Characterizing concurrency control performance for the PIOUS parallel file system. *Journal of Parallel and Distributed Computing 38*, 1 (Oct.), 81–91.

MULLIN, J. K. 1985. Spiral storage: Efficient dynamic hashing with constant performance. *The Computer Journal 28*, 3 (July), 330–334.

MUNAGALA, K. AND RANADE, A. 1999. I/O-complexity of graph algorithms. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms* (Baltimore, Jan.), Vol. 10, 687–694.

NASA 1999. Earth Observing System (EOS) web page, NASA Goddard Space Flight Center, http://eospso.gsfc.nasa.gov/.

NIEVERGELT, J. AND REINGOLD, E. M. 1973. Binary search tree of bounded balance. *SIAM Journal on Computing 2*, 1.

NIEVERGELT, J. AND WIDMAYER, P. 1997. Spatial data structures: Concepts and design choices. In M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer, Eds. *Algorithmic Foundations of GIS*, Vol. 1340 of *Lecture Notes in Computer Science*, Springer-Verlag, 153 ff.

NIEVERGELT, J., HINTERBERGER, H., AND SEVCIK, K. C. 1984. The grid file: An adaptable, symmetric multi-key file structure. *ACM Transactions on Database Systems 9*, 38–71.

NODINE, M. H. AND VITTER, J. S. 1993. Deterministic distribution sort in shared and distributed memory multiprocessors. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures* (Velen, Germany, June–July), Vol. 5, 120–129.

NODINE, M. H. AND VITTER, J. S. 1995. Greed Sort: An optimal sorting algorithm for multiple disks. *Journal of the ACM 42*, 4 (July), 919–933.

NODINE, M. H., GOODRICH, M. T., AND VITTER, J. S. 1996. Blocking for external graph searching. *Algorithmica 16*, 2 (August), 181–214.

NODINE, M. H., LOPRESTI, D. P., AND VITTER, J. S. 1991. I/O overhead and parallel VLSI architectures for lattice computations. *IEEE Transactions on Communications 40*, 7 (July), 843–852.

O'NEIL, P. E. 1992. The SB-tree. An index-sequential structure for high-performance sequential access. *Acta Informatica 29*, 3 (June), 241–265.

ORENSTEIN, J. A. 1989. Redundancy in spatial databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Portland, OR, June), 294–305.

ORENSTEIN, J. A. AND MERRETT, T. H. 1984. A class of data structures for associative searching. In *Proceedings of the ACM Conference Principles of Database Systems*, Vol. 3, 181–190.

OVERMARS, M. H. 1983. *The Design of Dynamic Data Structures*. Lecture Notes in Computer Science. Springer-Verlag.

PANG, H., CAREY, M., AND LIVNY, M. 1993a. Memory-adaptive external sorts. In *Proceedings of the*

*International Conference on Very Large Databases* (Dublin), Vol. 19, 618–629.

PANG, H., CAREY, M. J., AND LIVNY, M. 1993b. Partially preemptive hash joins. In P. Buneman and S. Jajodia, Eds. *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Washington, DC, May), 59–68.

PARSONS, I., UNRAU, R., SCHAEFFER, J., AND SZAFRON, D. 1997. PI/OT: Parallel I/O templates. *Parallel Computing 23*, 4 (June), 543–570.

PATEL, J. M. AND DEWITT, D. J. 1996. Partition based spatial-merge join. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (June), 259–270.

PREPARATA, F. P. AND SHAMOS, M. I. 1985. *Computational Geometry*. Springer-Verlag.

RAHMAN, N., AND RAMAN, R. 2000. Adapting radix sort to the memory hierarchy. In *Workshop on Algorithm Engineering and Experimentation* (Jan.), Vol. 1982 of *Lecture Notes in Computer Science*. Springer-Verlag.

RAMASWAMY, S. AND SUBRAMANIAN, S. 1994. Path caching: A technique for optimal external searching. In *Proceedings of the ACM Conference Principles of Database Systems* (Minneapolis, MN), Vol. 13, 25–35.

RAO, J. AND ROSS, K. 1999. Cache conscious indexing for decision-support in main memory. In M. Atkinson et al., Eds. *Proceedings of the International Conference on Very Large Databases* (Los Altos, CA), Vol. 25, 78–89. Morgan Kaufmann san Mateor, CA.

RAO, J. AND ROSS, K. A. 2000. Making $B^+$-trees cache conscious in main memory. In W. Chen, J. Naughton, and P. A. Bernstein, Eds, *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Dallas). 475–486.

RIEDEL, E., GIBSON, G. A., AND FALOUTSOS, C. 1998. Active storage for large-scale data mining and multimedia. In *Proceedings of the International Conference on Very Large Databases* (August), Vol. 22, 62–73.

ROBINSON, J. T. 1981. The $k$-d-b-tree: A search structure for large multidimensional dynamic indexes. In *Proceedings of the ACM Conference Principles of Database Systems*, Vol. 1, 10–18.

ROSENBLUM, M., BUGNION, E., DEVINE, S., AND HERROD, S. A. 1997. Using the SimOS machine simulator to study complex computer systems. *ACM Transactions on Modeling and Computer Simulation 7*, 1 (Jan.), 78–103.

RUEMMLER, C. AND WILKES, J. 1994. An introduction to disk drive modeling. *IEEE Computer* (March), 17–28.

SALEM, K. AND GARCIA-MOLINA, H. 1986. Disk striping. In *Proceedings of IEEE International Conference on Data Engineering* (Los Angeles), Vol. 2, 336–242.

ŠALTENIS, S., JENSEN, C. S., LEUTENEGGER, S. T., AND LOPEZ, M. A. 2000. Indexing the positions of continuously moving objects. In W. Chen, J. Naughton, and P. A. Bernstein, Eds. *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Dallas), 331–342.

SALZBERG, B. AND TSOTRAS, V. J. 1999. Comparison of access methods for time-evolving data. *ACM Computing Surveys 31* (June), 158–221.

SAMET, H. 1989a. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley.

SAMET, H. 1989b. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley.

SAMOLADAS, V. AND MIRANKER, D. 1998. A lower bound theorem for indexing schemes and its application to multidimensional range queries. In *Proceedings of the ACM Symposium on Principles of Database Systems* (Seattle, June), Vol. 17, 44–51.

SANDERS, P. 1999. Fast priority queues for cached memory. In *Workshop on Algorithm Engineering and Experimentation*, Vol. 1619 of *Lecture Notes in Computer Science*, Springer-Verlag (Jan.), 312–327.

SANDERS, P. 2000. Personal communication.

SANDERS, P. 2001. Reconciling simplicity and realism in parallel disk models. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms* (Washington, Jan), Vol. 12.

SANDERS, P., EGNER, S., AND KORST, J. 2000. Fast concurrent access to parallel disks. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms* (San Francisco, Jan.), Vol. 11, 849–858.

SAVAGE, J. E. 1995. Extending the Hong-Kung model to memory hierarchies. In *Proceedings of the International Conference on Computing and Combinatorics* (August), Vol. 959 of *Lecture Notes in Computer Science*, Springer-Verlag, 270–281.

SAVAGE, J. E. AND Vitter, J. S. 1987. Parallelism in space-time tradeoffs. In F. P. Preparata, Ed., *Advances in Computing Research*, Vol. 4, JAI Press, Grreenwich, CT, 117–146.

SCHLOSSER, S. W., GRIFFIN, J. L., NAGLE, D. F., AND GANGER, G. R. 2000. Designing computer systems with MEMS-based storage. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (Nov.), Vol. 9.

SEAMONS, K. E. AND WINSLETT, M. 1996. Multidimensional array I/O in Panda 1.0. *Journal of Supercomputing 10*, 2, 191–211.

SEEGER, B. AND KRIEGEL, H.-P. 1990 The buddy-tree: An efficient and robust access method for spatial data base systems. In *Proceedings of the International Conference on Very Large Databases*, Vol. 16, 590–601.

SELTZER, M., SMITH, K. A., BALAKRISHNAN, H., CHANG, J., MCMAINS, S., AND PADMANABHAN, V. 1995. File system logging versus clustering: A performance comparison. In *Proceedings of the Annual*

*USENIX Technical Conference* (New Orleans), 249–264.

SEN, S. AND CHATTERJEE, S. 2000. Towards a theory of cache-efficient algorithms. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms* (San Francisco, Jan.), Vol. 11.

SHRIVER, E., MERCHANT, A., AND WILKES, J. 1998. An analytic behavior model for disk drives with readahead caches and request reordering. In *Proceedings of ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems* (June), 182–191.

SHRIVER, E. A. M. AND NODINE, M. H. 1996. An introduction to parallel I/O models and algorithms. In R. Jain, J. Werth, and J. C. Browne, Eds. *Input/Output in Parallel and Distributed Computer Systems*, Kluwer Academic, Chapter 2, 31–68.

SHRIVER, E. A. M. AND WISNIEWSKI, L. F. 1995. An API for choreographing data accesses. Technical Report PCS-TR95-267, Dept. of Computer Science, Dartmouth College, November.

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

SIBEYN, J. F. 1999. External selection. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, Vol. 1563 of *Lecture Notes in Computer Science*, Springer-Verlag, 291–301.

SIBEYN, J. F. AND KAUFMANN, M. 1997. BSP-like external-memory computation. In *Proceedings of the Italian Conference on Algorithms and Complexity*, Vol. 3, 229–240.

SRINIVASAN, B. 1991. An adaptive overflow technique to defer splitting in b-trees. *The Computer Journal 34,* 5, 397–405.

SRIVASTAVA, A. AND EUSTACE, A. 1994. ATOM: A system for building customized program analysis tools. *ACM SIGPLAN Notices 29,* 6 (June), 196–205.

SUBRAMANIAN, S. AND RAMASWAMY, S. 1995. The P-range tree: A new data structure for range searching in secondary memory. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, Vol. 6, 378–387.

TAMASSIA, R. AND VITTER, J. S. 1996. Optimal cooperative search in fractional cascaded data structures. *Algorithmica 15,* 2 (Feb.), 154–171.

THAKUR, R., CHOUDHARY, A., BORDAWEKAR, R., MORE, S., AND KUDITIPUDI, S. 1996. Passion: Optimized I/O for parallel applications. *IEEE Computer 29,* 6 (June), 70–78.

TIGER/Line (tm). 1992. Technical documentation. Technical Report, US Bureau of the Census.

TOLEDO, S. 1999. A survey of out-of-core algorithms in numerical linear algebra. In J. Abello and J. S. Vitter, Eds. *External Memory Algorithms and Visualization*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, Providence, RI, 161–179.

TPIE. 1999. User manual and reference, The manual and software distribution are available on the web at http://www.cs.duke.edu/TPIE/.

ULLMAN, J. D. AND YANNAKAKIS, M. 1991. The input/output complexity of transitive closure. *Annals of Mathematics and Artificial Intelligence 3*, 331–360.

VAN DEN BERCKEN, J., SEEGER, B., AND WIDMAYER, P. 1997. A generic approach to bulk loading multidimensional index structures. In *Proceedings of the International Conference on Very Large Databases*, Vol. 23, 406–415.

VAN KREVELD, M., NIEVERGELT, J., ROOS, T., AND WINDMAYER, P. Eds. 1997. *Algorithmic Foundations of GIS*, Vol. 1340 of *Lecture Notes in Computer Science*, Springer-Verlag.

VARMAN, P. J. AND VERMA, R. M. 1997. An efficient multiversion access structure. *IEEE Transactions on Knowledge and Data Engineering 9*, 3 (May–June), 391–409.

VENGROFF, D. E. AND VITTER, J. S. 1996a. Efficient 3-d range searching in external memory. In *Proceedings of the ACM Symposium on Theory of Computing* (Philadelphia, May), Vol. 28, 192–201.

VENGROFF, D. E. AND VITTER, J. S. 1996b. I/O-efficient scientific computation using TPIE. In *Proceedings of NASA Goddard Conference on Mass Storage Systems* (Sept.), Vol. 5, II, 553–570.

VETTIGER, P., DESPONT, M., DRECHSLER, U., DÜRIG, U., HÄBERLE, W., LUTWYCHE, M. I., ROTHUIZEN, E., STUTZ, R., WIDMER, R., AND BINNIG, G. K. 2000. The "Millipede"—More than one thousand tips for future AFM data storage. *IBM Journal of Research and Development 44*, 3, 323–340.

VITTER, J. S. 1991. Efficient memory access in large-scale computation. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, invited paper, Vol. 480 of *Lecture Notes in Computer Science*, Invited Paper, Springer-Verlag, 26–41.

VITTER, J. S. 1998. External memory algorithms. In *Proceedings of the European Symposium on Algorithms* (Venice, Italy, August), Vol. 1461 of *Lecture Notes in Computer Science*, invited paper, Springer-Verlag.

VITTER, J. S. 1999a. External memory algorithms and data structures. In J. Abello and J. S. Vitter, Eds. *External Memory Algorithms and Visualization*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, Providence, RI.

VITTER, J. S. 1999b. Online data structures in external memory. In *Proceedings of the International Colloquium on Automata, Languages, and Programming* (Prague, August). Vol. 1644 of Lecture Notes in Computer Science, invited paper, Springer-Verlag, 119–133.

VITTER, J. S. 1999c. Online data structures in external memory. In *Proceedings of the Workshop on Algorithms and Data Structures* (Vancouver,

August), Vol. 1668 of *Lecture Notes in Computer Science*, Invited paper. Springer-Verlag.

VITTER, J. S. AND FLAJOLET, P. 1990. Average-case analysis of algorithms and data structures. In J. van Leeuwen, Ed., *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, North-Holland, Chapter 9, 431–524.

VITTER, J. S. AND HUTCHINSON, D. A. 2001. Distribution sort with randomized cycling. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms* (Washington, January), Vol. 12.

VITTER, J. S., AND NODINE, M. H. 1993. Large-scale sorting in uniform memory hierarchies. *Journal of Parallel and Distributed Computing 17*, 107–114.

VITTER, J. S. AND SHRIVER, E. A. M. 1994a. Algorithms for parallel memory I: Two-level memories. *Algorithmica* 12, 2–3, 110–147.

VITTER, J. S. AND SHRIVER, E. A. M. 1994b. Algorithms for parallel memory II: Hierarchical multilevel memories. *Algorithmica 12*, 2–3, 148–169.

VITTER, J. S. AND VENGROFF, D. E. 1999. Notes.

VITTER, J. S. AND WANG, M. 1999. Approximate computation of multidimensional aggregates of sparse data using wavelets. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Philadelphia, June), 193–204.

VITTER, J. S., WANG, M., AND IYER, B. 1998. Data cube approximation and histograms via wavelets. In *Proceedings of the International ACM Conference on Information and Knowledge Management* (Washington, Nov.), Vol. 7, 96–104.

WANG, M., IYER, B., AND VITTER, J. S. 1998. Scalable mining for classification rules in relational databases. In *Proceedings of the International Database Engineering & Application Symposium* (Cardiff, July), 58–67.

WANG, M., VITTER, J. S., LIM, L., AND PADMANABHAN, S. 2001. Wavelet-based cost estimation for spatial queries, July.

WATSON, R. W. AND COYNE, R. A. 1995. The parallel I/O architecture of the high-performance storage system (HPSS). In *Proceedings of the IEEE Symposium on Mass Storage Systems* (Sept.), Vol. 14, 27–44.

WEINER, P. 1973. Linear pattern matching algorithm. In *Proceedings of the IEEE Symposium on Switching and Automata Theory* (Washington, DC), Vol. 14, 1–11.

WILLARD, D. AND LUEKER, G. 1985. Adding range restriction capability to dynamic data structures. *Journal of the ACM, 32*, 3, 597–617.

WOLFSON, O., SISTLA, P., XU, B., ZHOU, J., AND CHAMBERLAIN, S. 1999. DOMINO: Databases fOr MovINg Objects tracking. In A. Delis, C. Faloutsos, and S. Ghandeharizadeh, Eds., *Proceedings of the ACM SIGMOD International Conference on Management of Data* May, 547–549.

WOMBLE, D., GREENBERG, D., WHEAT, S., AND RIESEN, R. 1993. Beyond core: Making parallel computer I/O practical. In *Proceedings of the DAGS Symposium on Parallel Computation* (Hanover, NH, June), 56–63. Vol. 2, Dartmouth Institute for Advanced Graduate Studies.

WU, C., AND FENG, T. 1981. The universality of the shuffle-exchange network. *IEEE Transactions on Computers C-30* (May), 324–332.

YAO, A. C. 1978. On random 2-3 trees. *Acta Informatica 9*, 159–170.

ZDONIK, S. B. AND MAIER, D. Eds. 1990. *Readings in Object-Oriented Database Systems*. Morgan Kauffman, San Mateo, CA.

ZHANG, W. AND LARSON, P.-A. Dynamic memory adjustment for external mergesort. In *Proceedings of the International Conference on Very Large Databases* (Athens), Vol. 23, 376–385.

ZHU, B. 1994. Further computational geometry in secondary memory. In *Proceedings of the International Symposium on Algorithms and Computation*, Vol. 834 of *Lecture Notes in Computer Science*, Springer-Verlag, 514 ff.

Received May 1999; revised March 2000; accepted September 2000