Foundations and Trends® in
Theoretical Computer Science
2:4

Algorithms and Data Structures for External Memory

Jeffrey Scott Vitter

now
the essence of knowledge

Foundations and Trends® in
Theoretical Computer Science
Vol. 2, No. 4 (2006) 305–474

DOI: 10.1561/0400000014

now
the essence of knowledge

Algorithms and Data Structures for External Memory

Jeffrey Scott Vitter

Department of Computer Science, Purdue University, West Lafayette, Indiana, 47907–2107, USA, jsv@purdue.edu

Abstract

Data sets in large applications are often too massive to fit completely inside the computer's internal memory. The resulting input/output communication (or I/O) between fast internal memory and slower external memory (such as disks) can be a major performance bottleneck. In this manuscript, we survey the state of the art in the design and analysis of algorithms and data structures for *external memory* (or *EM* for short), where the goal is to exploit locality and parallelism 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 like 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, 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, geometric data, and graphs.

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 convenient means in online data structures to make effective use of the data accessed from disk. We also re-examine 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 such as character strings, when the data structure is compressed to save space, or when the allocated amount of internal memory can change dynamically.

Programming tools and environments are available for simplifying the EM programming task. 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 other methods used in practice.

Preface

I first became fascinated about the tradeoffs between computing and memory usage while a graduate student at Stanford University. Over the following years, this theme has influenced much of what I have done professionally, not only in the field of external memory algorithms, which this manuscript is about, but also on other topics such as data compression, data mining, databases, prefetching/caching, and random sampling.

The reality of the computer world is that no matter how fast computers are and no matter how much data storage they provide, there will always be a desire and need to push the envelope. The solution is not to wait for the next generation of computers, but rather to examine the fundamental constraints in order to understand the limits of what is possible and to translate that understanding into effective solutions.

In this manuscript you will consider a scenario that arises often in large computing applications, namely, that the relevant data sets are simply too massive to fit completely inside the computer's internal memory and must instead reside on disk. 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. This manuscript provides a detailed overview of the design and analysis of algorithms and data structures for *external memory* (or simply *EM*), where the goal is to exploit locality and parallelism in order to reduce the I/O costs. Along the way, you will learn a variety of EM paradigms for solving batched and online problems efficiently.

For the batched problem of sorting and related problems like permuting and fast Fourier transform, the two fundamental paradigms are distribution and merging. The paradigm of disk striping offers an elegant way to use multiple disks in parallel. For sorting, however, 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, including an elegant duality property that yields state-of-the-art algorithms. You will encounter other 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, which involves constructing data structures to answer queries, we discuss two canonical EM search applications: dictionary lookup and range searching. Two important paradigms for developing indexed data structures for these problems are hashing (including extendible hashing) and tree-based search (including B-trees). The paradigms of filtering and bootstrapping provide convenient means in online data structures to make effective use of the data accessed from disk. You will also be exposed to 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., strings of text), when the data structure is compressed to save space, and when the allocated amount of internal memory can change dynamically.

Programming tools and environments are available for simplifying the EM programming task. You will see some experimental results in the domain of spatial databases using the TPIE system, which stands for Transparent Parallel I/O programming Environment. The newly developed EM algorithms and data structures that incorporate the paradigms discussed in this manuscript are significantly faster than other methods used in practice.

I would like to thank my colleagues for several helpful comments, especially Pankaj Agarwal, Lars Arge, Ricardo Baeza-Yates, Adam Buchsbaum, Jeffrey Chase, Michael Goodrich, Wing-Kai Hon, David Hutchinson, Gonzalo Navarro, Vasilis Samoladas, Peter Sanders, Rahul Shah, Amin Vahdat, and Norbert Zeh. I also thank the referees and editors for their help and suggestions, as well as the many wonderful staff members I've had the privilege to work with. Figure 1.1 is a modified version of a figure by Darren Vengroff, and Figures 2.1 and 5.2 come from [118, 342]. Figures 5.4–5.8, 8.2–8.3, 10.1, 12.1, 12.2, 12.4, and 14.1 are modified versions of figures in [202, 47, 147, 210, 41, 50, 158], respectively.

This manuscript is an expanded and updated version of the article in *ACM Computing Surveys*, Vol. 33, No. 2, June 2001. I am very appreciative for the support provided by the National Science Foundation through research grants CCR–9522047, EIA–9870734, CCR–9877133, IIS–0415097, and CCF–0621457; by the Army Research Office through MURI grant DAAH04–96–1–0013; and by IBM Corporation. Part of this manuscript was done at Duke University, Durham, North Carolina; the University of Aarhus, Århus, Denmark; INRIA, Sophia Antipolis, France; and Purdue University, West Lafayette, Indiana.

I especially want to thank my wife Sharon and our three kids (or more accurately, young adults) Jillian, Scott, and Audrey for their ever-present love and support. I most gratefully dedicate this manuscript to them.

West Lafayette, Indiana — J. S. V.
March 2008

1

Introduction

The world is drowning in data! In recent years, we have been deluged by a torrent of data from a variety of increasingly data-intensive applications, including databases, scientific computations, graphics, entertainment, multimedia, sensors, web applications, and email. NASA's Earth Observing System project, the core part of the Earth Science Enterprise (formerly Mission to Planet Earth), produces petabytes (10^{15} bytes) of raster data per year [148]. A petabyte corresponds roughly to the amount of information in one billion graphically formatted books. The online databases of satellite images used by Microsoft TerraServer (part of MSN Virtual Earth) [325] and Google Earth [180] are multiple terabytes (10^{12} bytes) in size. Wal-Mart's sales data warehouse contains over a half petabyte (500 terabytes) of data. A major challenge is to develop mechanisms for processing the data, or else much of the data will be useless.

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

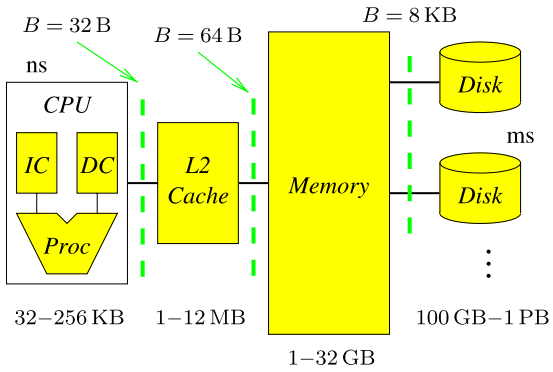

Fig. 1.1 The memory hierarchy of a typical uniprocessor system, including registers, instruction cache, data cache (level 1 cache), level 2 cache, internal memory, and disks. Some systems have in addition a level 3 cache, not shown here. Memory access latency ranges from less than one nanosecond (ns, 10^{-9} seconds) for registers and level 1 cache to several milliseconds (ms, 10^{-3} seconds) for disks. Typical memory sizes for each level of the hierarchy are shown at the bottom. Each value of B listed at the top of the figure denotes a typical block transfer size between two adjacent levels of the hierarchy. All sizes are given in units of bytes (B), kilobytes (KB, 10^3 B), megabytes (MB, 10^6 B), gigabytes (GB, 10^9 B), and petabytes (PB, 10^{15} B). (In the PDM model defined in Chapter 2, we measure the block size 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 we often use a substantially larger logical block transfer size.

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. These devices can be attached via a network fabric (e.g., Fibre Channel or iSCSI) to provide substantial external storage capacity. Figure 1.1 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 does not do 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 can take a fraction of a nanosecond (10^{-9} seconds), and accessing internal memory takes several nanoseconds, but the latency of accessing data on a disk is multiple 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 [139]. 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 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 manuscript 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.

1.1 Overview

In this manuscript, 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.
(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 chapter. 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 (parallel) I/O operations, the disk space usage,* and *the (parallel) 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 Chapter 3, we list four fundamental I/O bounds that pertain to most of the problems considered in this manuscript. In Chapter 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.

Our general goal is to design optimal algorithms and data structures, by which we mean that their performance measures are within a constant factor of the optimum or best possible.[1] In Chapter 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 — as well as the notion of duality that relates the two — 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 not optimal. 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. In Chapter 6, we cover the fundamental lower bounds on the number of I/Os needed to perform sorting and related batched problems. In Chapter 7, 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 Chapter 5, and hence we restrict ourselves starting in Chapter 8 to the conceptually simpler single-disk case. In Chapter 8, 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 visibility), 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 Chapter 9, 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.

[1] In this manuscript we generally use the term "optimum" to denote the absolute best possible and the term "optimal" to mean within a constant factor of the optimum.

In Chapters 10–12, 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 Chapter 10, 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. Chapter 11 begins with a discussion of B-trees, the most widely used online EM data structure for dictionary operations and one-dimensional 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 Chapter 12, 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 design paradigm for EM data structures 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 Chapter 13, we discuss some additional EM approaches useful for dynamic data structures, and we also investigate kinetic data structures, in which the data items are moving. In Chapter 14, we focus on EM data structures for manipulating and searching text strings. In many applications, especially those that operate on text strings, the data are highly compressible. Chapter 15 discusses ways to develop data structures that are themselves compressed, but still fast to query.

Table 1.1 Paradigms for I/O efficiency discussed in this manuscript.

Paradigm	Section
Batched dynamic processing	11.4
Batched filtering	8
Batched incremental construction	8
Bootstrapping	12
Buffer trees	11.4
B-trees	11, 12
Compression	15
Decomposable search	13.1
Disk striping	4.2
Distribution	5.1
Distribution sweeping	8
Duality	5.3
External hashing	10
Externalization	12.3
Fractional cascading	8
Filtering	12
Lazy updating	11.4
Load balancing	4
Locality	4.1
Marriage before conquest	8
Merging	5.2
Parallel block transfer	4.2
Parallel simulation	9
Persistence	11.1
Random sampling	5.1
R-trees	12.2
Scanning (or streaming)	2.2
Sparsification	9
Time-forward processing	11.4

In Chapter 16, we discuss EM algorithms that adapt optimally to dynamically changing internal memory allocations.

In Chapter 17, we discuss programming environments and tools that facilitate high-level development of efficient EM algorithms. We focus primarily on the TPIE system (Transparent Parallel I/O Environment), which we use in the various timing experiments in this manuscript. We conclude with some final remarks and observations in the Conclusions.

Table 1.1 lists several of the EM paradigms discussed in this manuscript.

2

Parallel Disk Model (PDM)

When a data set is too large to fit in internal memory, it is typically stored in external memory (EM) on one or more magnetic disks. EM algorithms explicitly control data placement and transfer, and thus it is important for algorithm designers to have a simple but reasonably accurate model of the memory system's characteristics.

A magnetic disk consists of one or more platters rotating at constant speed, with one read/write head per platter surface, as shown in Figure 2.1. The surfaces of the platters are covered with a magnetizable material capable of storing data in nonvolatile fashion. The read/write heads are held by arms that move in unison. When the arms are stationary, each read/write head traces out a concentric circle on its platter called a *track*. The vertically aligned tracks that correspond to a given arm position are called a *cylinder*. For engineering reasons, data to and from a given disk are typically transmitted using only one read/write head (i.e., only one track) at a time. Disks use a buffer for caching and staging data for I/O transfer to and from internal memory.

To store or retrieve a data item at a certain address on disk, the read/write heads must mechanically *seek* to the correct cylinder and then wait for the desired data to pass by on a particular track. The seek

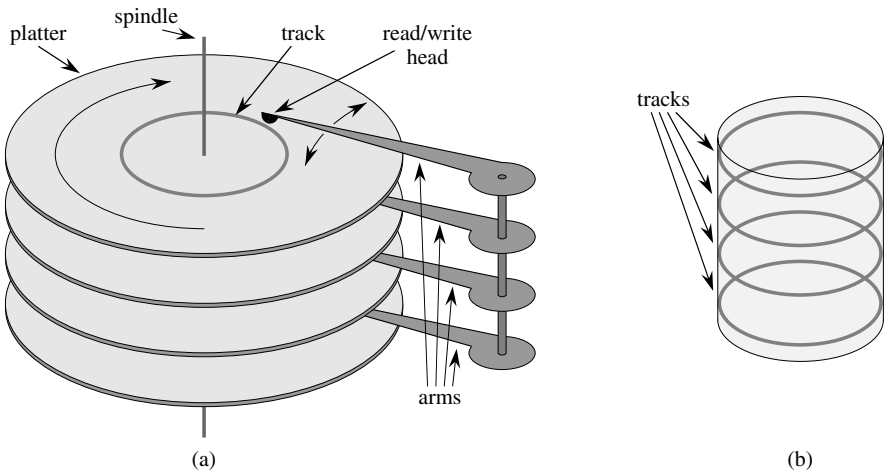

Fig. 2.1 Magnetic disk drive: (a) Data are stored on magnetized platters that rotate at a constant speed. Each platter surface is accessed by an arm that contains a read/write head, and data are stored on the platter in concentric circles called tracks. (b) The arms are physically connected so that they move in unison. The tracks (one per platter) that are addressable when the arms are in a fixed position are collectively referred to as a cylinder.

time to move from one random cylinder 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. Seek time can be avoided if the next access is on the current cylinder. The latency for accessing data, which is primarily a combination of seek time and rotational latency, is typically on the order of several milliseconds. In contrast, it can take less than one nanosecond to access CPU registers and cache memory — more than one million times faster than disk access!

Once the read/write head is positioned at the desired data location, subsequent bytes of data can be stored or retrieved as fast as the disk rotates, which might correspond to over 100 megabytes per second. We can thus amortize the relatively long initial delay by transferring a large *contiguous* group of data items at a time. We use the term *block* to refer to the amount of data transferred to or from one disk in a single I/O operation. Block sizes are typically on the order of several kilobytes and are often larger for batched applications. Other levels of the memory hierarchy have similar latency issues and as a result also

use block transfer. Figure 1.1 depicts typical memory sizes and block sizes for various levels of memory.

Because I/O is done in units of blocks, algorithms can run considerably faster when the pattern of memory accesses exhibit locality of reference as opposed to a uniformly random distribution. However, even if an application can structure its pattern of memory accesses and exploit locality, 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 (or multicores) further widens the gap. As a result, storage systems such as RAID deploy multiple disks that can be accessed in parallel in order to get additional bandwidth [101, 194].

In the next section, we describe the high-level parallel disk model (PDM), which we use throughout this manuscript 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 [345]. The two key mechanisms for efficient algorithm design in PDM are *locality of reference* (which takes advantage of block transfer) and *parallel disk access* (which takes advantage of multiple disks). In a single I/O, each of the D disks can simultaneously transfer a block of B contiguous data items.

PDM uses the following main parameters:

$$\begin{aligned} N &= \text{problem size (in units of data items);} \\ M &= \text{internal memory size (in units of data items);} \\ B &= \text{block transfer size (in units of data items);} \end{aligned}$$

$$\begin{aligned} D &= \text{number of independent disk drives;} \\ P &= \text{number of CPUs,} \end{aligned}$$

where $M < N$ and $1 \leq DB \leq M/2$. The N data items are assumed to be of fixed length. The ith block on each disk, for $i \geq 0$, consists of locations iB, $iB + 1$, ..., $(i + 1)B - 1$.

If $P \leq D$, each of the P processors (or cores) 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 or shared memory or combination of the two. For routing considerations, one desired property for the network is the capability to sort the M data items in the collective internal memories of the processors in parallel in optimal $O\big((M/P)\log M\big)$ 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 2.2.

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 Chapter 8, we are given a set of N points in the plane and a set of Q queries in the form of rectangles, and the problem is to report the points lying in each of the Q query rectangles. In both the batched and online settings, the number of items reported in response to each query may vary. We thus need to define two more performance parameters:

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

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

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

[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.

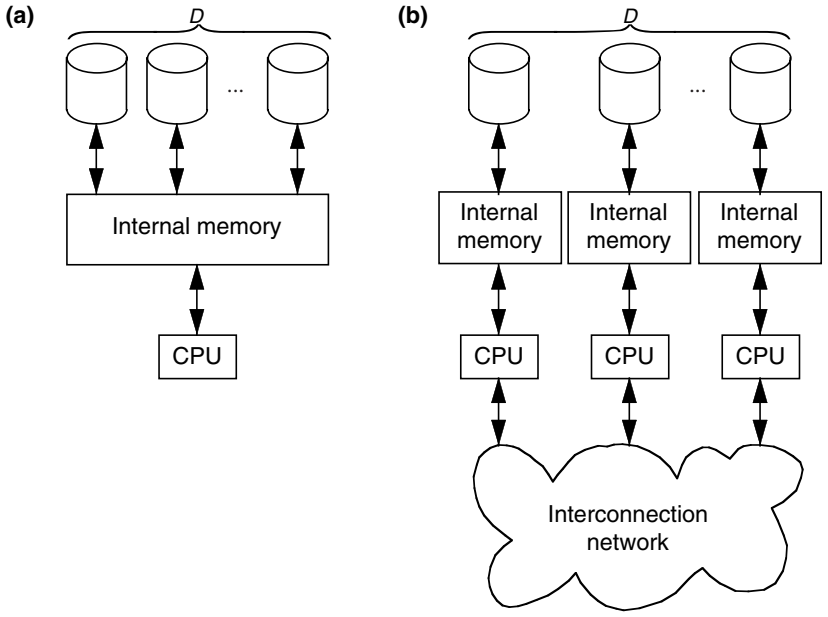

Fig. 2.2 Parallel disk model: (a) $P = 1$, in which the D disks are connected to a common CPU; (b) $P = D$, in which each of the D disks is connected to a separate processor.

to be the problem size, internal memory size, query specification size, and answer size, respectively, in units of disk blocks.

We assume that the data for the problem are initially "striped" across the D disks, in units of blocks, as illustrated in Figure 2.3, and we require the final data to be similarly striped. Striped format allows a file of N data items to be input or output in $O(N/DB) = O(n/D)$ I/Os, which is optimal.

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

Fig. 2.3 Initial data layout on the disks, for $D = 5$ disks and block size $B = 2$. The data items are initially striped block-by-block across the disks. For example, data items 6 and 7 are stored in block 0 (i.e., in stripe 0) of disk $\mathcal{D}_3$. Each stripe consists of DB data items, such as items 0–9 in stripe 0, and can be accessed in a single I/O.

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 manuscript 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. There are interesting issues associated with optimizing internal computation time in the presence of multiple disks, in which communication takes place over a particular interconnection network, but they are not the focus of this manuscript. 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 KB–2 MB. In reality, the track size for any given disk depends upon the radius of the track (cf. Figure 2.1). 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 tracks 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 1.1. 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 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 1.1. 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 Chapter 17. More complex and precise disk models, such as the ones by Ruemmler and Wilkes [295], Ganger [171], Shriver et al. [314], Barve et al. [70], Farach-Colton et al. [154], and Khandekar and Pandit [214], 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) [46]. 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 [310].

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 I/O requests for each disk [136].

2.3 Related Models, Hierarchical Memory, and Cache-Oblivious Algorithms

The study of problem complexity and algorithm analysis for EM devices began more than a half century ago with Demuth's PhD dissertation on sorting [138, 220]. In the early 1970s, Knuth [220] did an extensive study of sorting using magnetic tapes and (to a lesser extent) magnetic disks. At about the same time, Floyd [165, 220] considered a disk model akin to PDM for $D = 1, \quad P = 1, \quad B = M/2 = \Theta(N^c)$, where c is a constant in the range $0 < c < 1$. For those particular parameters, he developed optimal upper and lower I/O bounds for sorting and matrix transposition. Hong and Kung [199] developed a pebbling model of I/O for straightline computations, and Savage and Vitter [306] extended the model to deal with block transfer.

Aggarwal and Vitter [23] 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 5.4, we discuss a simulation technique due to Sanders et al. [304]; 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 [60].

Surveys of I/O models, algorithms, and challenges appear in [3, 31, 175, 257, 315]. Several versions of PDM have been developed for parallel computation [131, 132, 234, 319]. Models of "active disks" augmented with processing capabilities to reduce data traffic to the host, especially during streaming applications, are given in [4, 292]. Models of microelectromechanical systems (MEMS) for mass storage appear in [184].

Some authors have studied problems that can be solved efficiently by making only one pass (or a small number of passes) over the

data [24, 155, 195, 265]. In such *data streaming* applications, one useful 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 problem-dependent and orthogonal to our focus in this manuscript; we refer the reader to the surveys in [24, 265].

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 [20, 21, 344, 346]. Conversely, the algorithms for hierarchical models can be run in the PDM setting.

Frigo et al. [168] introduce the important notion of *cache-oblivious algorithms*, which require no knowledge of the storage parameters, like M and B, nor special programming environments for implementation. It follows that, up to a constant factor, time-optimal and space-optimal algorithms in the cache-oblivious model are similarly optimal in the external memory model. Frigo et al. [168] develop optimal cache-oblivious algorithms for merge sort and distribution sort. Bender et al. [79] and Bender et al. [80] develop cache-oblivious versions of B-trees that offer speed advantages in practice. In recent years, there has been considerable research in the development of efficient cache-oblivious algorithms and data structures for a variety of problems. We refer the reader to [33] for a survey.

The match between theory and practice is harder to establish for hierarchical models and caches than for disks. Generally, the most significant speedups come from optimizing the I/O communication between internal memory and the 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 input 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 the 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 [221, 294, 322] can provide sophisticated help in the optimization process.

For reasons of focus, we do not consider hierarchical and cache models in this manuscript. We refer the reader to the previous references on cache-oblivious algorithms, as well to as the following references: Aggarwal et al. [20] define an elegant hierarchical memory model, and Aggarwal et al. [21] augment it with block transfer capability. Alpern et al. [29] model levels of memory in which the memory size, block size, and bandwidth grow at uniform rates. Vitter and Shriver [346] and Vitter and Nodine [344] discuss parallel versions and variants of the hierarchical models. The parallel model of Li et al. [234] also applies to hierarchical memory. Savage [305] gives a hierarchical pebbling version of [306]. Carter and Gatlin [96] define pebbling models of nonassociative direct-mapped caches. Rahman and Raman [287] and

Sen et al. [311] apply EM techniques to models of caches and translation lookaside buffers. Arge et al. [40] consider a combination of PDM and the Aggarwal–Vitter model (which allows simultaneous accesses to the same external memory module) to model multicore architectures, in which each core has a separate cache but the cores share the larger next-level memory. Ajwani et al. [26] look at the performance characteristics of flash memory storage devices.

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 these 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.
(4) *Outputting* the Z items of an answer to a query in a blocked "output-sensitive" fashion.

We give the I/O bounds for these four operations in Table 3.1. We single out the special case of a single disk ($D = 1$), since the formulas are simpler and many of the discussions in this manuscript will be restricted to the single-disk case.

We discuss the algorithms and lower bounds for $Sort(N)$ and $Search(N)$ in Chapters 5, 6, 10, and 11. The lower bounds for searching assume the comparison model of computation; searching via hashing can be done in $\Theta(1)$ I/Os on the average.

Table 3.1 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 $D \geq 1$
$Scan(N)$	$\Theta\left(\frac{N}{B}\right) = \Theta(n)$	$\Theta\left(\frac{N}{DB}\right) = \Theta\left(\frac{n}{D}\right)$
$Sort(N)$	$\Theta\left(\frac{N}{B}\log_{M/B}\frac{N}{B}\right)$ $= \Theta(n\log_m n)$	$\Theta\left(\frac{N}{DB}\log_{M/B}\frac{N}{B}\right)$ $= \Theta\left(\frac{n}{D}\log_m n\right)$
$Search(N)$	$\Theta(\log_B N)$	$\Theta(\log_{DB} N)$
$Output(Z)$	$\Theta\left(\max\left\{1, \frac{Z}{B}\right\}\right)$ $= \Theta(\max\{1, z\})$	$\Theta\left(\max\left\{1, \frac{Z}{DB}\right\}\right)$ $= \Theta\left(\max\left\{1, \frac{z}{D}\right\}\right)$

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 together 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 items in an answer to a query do not need to be output to the disks 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 manuscript 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.

As Table 3.1 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 $\Theta\big((\log_B N)/\log_{DB} N\big) = \Theta\big((\log DB)/\log B\big) = \Theta\big(1 + (\log D)/\log B\big)$, 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 $N = 10^{10}$, $M = 10^7$, $B = 10^4$, 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 online 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 relatively small constant number of I/Os.

It still makes sense to explicitly identify terms such as $\log_m n$ 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 problem 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 [346].

4

Exploiting Locality and Load Balancing

In order to achieve good I/O performance, an EM algorithm should exhibit locality of reference. Since each input I/O operation transfers a block of B items, we make optimal use of that input operation when all B items are needed by the application. A similar remark applies to output I/O 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 3.1 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* [216, 296], 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 3.1. The only

exception is sorting. The optimal multiple-disk algorithms for sorting require more sophisticated load balancing techniques, which we cover in Chapter 5.

4.1 Locality Issues with a Single Disk

A good way to appreciate the fundamental I/O bounds in Table 3.1 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 manuscript, 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 3.1. 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 online problems, we want the base of the logarithm to be B and to replace Z by z. The resulting 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 online 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 and Parallelism with 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* [216, 296] is a practical paradigm that can ease the programming task with multiple disks: When disk striping is used, I/Os are permitted only on entire stripes, one stripe at a time. The ith stripe, for $i \geq 0$, consists of block i from each of the D disks. For example, in the data layout in Figure 2.3, the DB data items 0–9 comprise stripe 0 and can be accessed in a single I/O step. The net effect of striping is that the D disks behave as a single logical disk, but with a larger logical block size DB corresponding to the size of a stripe.

We can thus apply the paradigm of disk striping automatically to 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 transmits one stripe, which consists of D simultaneous block transfers each of size B. 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 faster.

Disk striping can be used to get optimal multiple-disk algorithms for three of the four fundamental operations of Chapter 3 — streaming, online search, and answer 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 [220]. As given in Table 3.1, the optimal number of I/Os to sort using one disk with block size B is

$$\Theta\left(n \log_m n\right) = \Theta\left(n \frac{\log n}{\log m}\right) = \Theta\left(\frac{N}{B} \frac{\log(N/B)}{\log(M/B)}\right). \tag{4.1}$$

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 (4.1) by DB, which gives the I/O bound

$$\Theta\left(\frac{N}{DB}\frac{\log(N/DB)}{\log(M/DB)}\right) = \Theta\left(\frac{n}{D}\frac{\log(n/D)}{\log(m/D)}\right). \tag{4.2}$$

On the other hand, the optimal bound for sorting is

$$\Theta\left(\frac{n}{D}\log_m n\right) = \Theta\left(\frac{n}{D}\frac{\log n}{\log m}\right). \tag{4.3}$$

The striping I/O bound (4.2) is larger than the optimal sorting bound (4.3) by a multiplicative factor of

$$\frac{\log(n/D)}{\log n}\frac{\log m}{\log(m/D)} \approx \frac{\log m}{\log(m/D)}. \tag{4.4}$$

When D is on the order of m, the $\log(m/D)$ term in the denominator is small, and the resulting value of (4.4) 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 (4.3) 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 input or output is done independently. It suffices, for example, to do only input operations independently and to use disk striping for output operations. An advantage of using striping for output operations is that it facilitates the maintenance of parity information for error correction and recovery, which is a big concern in RAID systems. (We refer the reader to [101, 194] 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 [337]. In the next chapter, 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 this manuscript. Three such sorting algorithms we introduce in the next chapter — distribution sort and merge sort with randomized cycling (RCD and RCM) and simple randomized merge sort (SRM) — have low overhead and outperform algorithms that use disk striping.

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 [220], and also because sorting is an important paradigm in the design of efficient EM algorithms, as we show in Section 9.3. With some technical qualifications, many problems that can be solved easily in linear time in the (internal memory) RAM model, 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.

In this chapter, we discuss optimal EM algorithms for sorting. The following bound is the most fundamental one that arises in the study of EM algorithms:

Theorem 5.1 ([23, 274]). The average-case and worst-case number of I/Os required for sorting $N = nB$ data items using D disks is

$$Sort(N) = \Theta\left(\frac{n}{D}\log_m n\right). \tag{5.1}$$

The constant of proportionality in the lower bound for sorting is 2, as we shall see in Chapter 6, and we can come very close to that constant factor by some of the recently developed algorithms we discuss in this chapter.

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 (5.1) of Theorem 5.1.

In Sections 5.1–5.3, we discuss some recently developed external sorting algorithms that use disks independently and achieve bound (5.1). The algorithms are based upon the important *distribution* and *merge* paradigms, which are two generic approaches to sorting. They 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 manuscript.

The distribution sort and merge sort methods using randomized cycling (RCD and RCM) [136, 202] from Sections 5.1 and 5.3 and the simple randomized merge sort (SRM) [68, 72] of Section 5.2 are the methods of choice for external sorting. For reasonable values of M and D, they outperform disk striping in practice and achieve the I/O lower bound (5.1) with the lowest known constant of proportionality.

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 input operations, but in a striped manner during parallel output operations. As a result, if we output $D-1$ blocks at a time in an I/O, the exclusive-or of the $D-1$ blocks can be output onto the Dth disk during the same I/O operation.

In Section 5.3, we develop a powerful notion of duality that leads to improved new algorithms for prefetching, caching, and sorting. In Section 5.4, we show that if we allow independent inputs and output operations, 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 5.5, we consider the situation in which the items in the input file do not have unique keys. In Sections 5.6 and 5.7, we consider problems related to sorting, such as permuting, permutation networks, transposition, and fast Fourier transform. In Chapter 6, we give lower bounds for sorting and related problems.

5.1 Sorting by Distribution

Distribution sort [220] is a recursive process in which we use a set of $S-1$ partitioning elements $e_1, e_2, \ldots, e_{S-1}$ to partition the current set of items into S disjoint subfiles (or *buckets*), as shown in Figure 5.1 for the case $D = 1$. The ith bucket, for $1 \leq i \leq S$, consists of all items with key value in the interval $[e_{i-1}, e_i)$, where by convention we let $e_0 = -\infty$, $e_S = +\infty$. The important property of the partitioning is that all the items in one bucket precede all the items in the next bucket. Therefore, we can complete the sort by recursively sorting the individual buckets and concatenating them together to form a single fully sorted list.

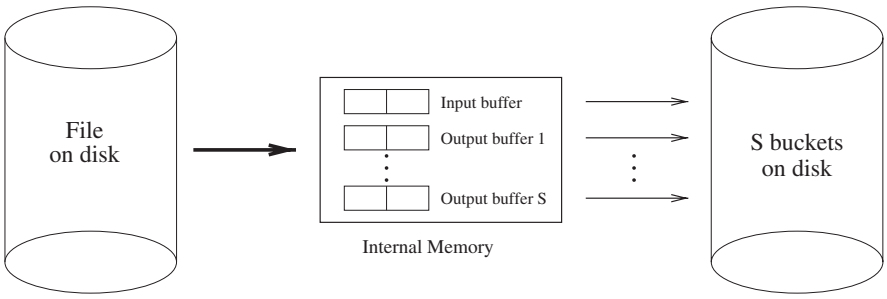

Fig. 5.1 Schematic illustration of a level of recursion of distribution sort for a single disk ($D = 1$). (For simplicity, the input and output operations use separate disks.) The file on the left represents the original unsorted file (in the case of the top level of recursion) or one of the buckets formed during the previous level of recursion. The algorithm streams the items from the file through internal memory and partitions them in an online fashion into S buckets based upon the key values of the $S-1$ partitioning elements. Each bucket has double buffers of total size at least $2B$ to allow the input from the disk on the left to be overlapped with the output of the buckets to the disk on the right.

5.1.1 Finding the Partitioning Elements

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 can be output to disk, and another buffer is used to store the next set of incoming items for the bucket. Therefore, the maximum number S of buckets (and partitioning elements) is $\Theta(M/B) = \Theta(m)$, and the resulting number of levels of recursion is $\Theta(\log_m n)$. In the last level of recursion, there is no point in having buckets of fewer than $\Theta(M)$ items, so we can limit S to be $O(N/M) = O(n/m)$. These two constraints suggest that the desired number S of partitioning elements is $\Theta\big(\min\{m, n/m\}\big)$.

It seems difficult to find $S = \Theta\big(\min\{m, n/m\}\big)$ partitioning elements deterministically 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 $S = \Theta\big(\min\{\sqrt{m}, n/m\}\big)$ partitioning elements [23, 273, 345], which has the effect of doubling the number of levels of recursion. A deterministic algorithm for the related problem of (exact) selection (i.e., given k, find the kth item in the file in sorted order) appears in [318].

Probabilistic methods for choosing partitioning elements based upon random sampling [156] are simpler and allow us to choose $S = O\big(\min\{m, n/m\}\big)$ partitioning elements in $o(n/D)$ I/Os: Let $d = O(\log S)$. We take a random sample of dS items, sort the sampled items, and then choose every dth item in the sorted sample to be a partitioning element. Each of the resulting buckets has the desired size of $O(N/S)$ items. The resulting number of I/Os needed to choose the partitioning elements is thus $O\big(dS + \mathit{Sort}(dS)\big)$. Since $S = O\big(\min\{m, n/m\}\big) = O(\sqrt{n})$, the I/O bound is $O(\sqrt{n}\log^2 n) = o(n)$ and therefore negligible.

5.1.2 Load Balancing Across the Disks

In order to meet the sorting I/O bound (5.1), 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. The challenge is in the more general multiple-disk case: Each input I/O step and each output I/O 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. In summary, the challenge in distribution sort is to output 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 input 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 output in "logical blocks" of size CB, one per cluster. Choosing $C = \sqrt{D}$ will not change the 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 [345] 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 output operation sends 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 on the average, whereas any individual bin contains an average of only one ball [341].[1] 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 use their second partitioning technique: The file is streamed through internal memory in one pass, one memoryload at a time. Each memoryload is independently and randomly permuted and output back to the disks in the new order. In a second pass, the file is input one memoryload at a time in a "diagonally striped" manner. Vitter and Shriver 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. [140] 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 [273]. 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

[1] We use the notation $\ln d$ to denote the natural (base e) logarithm $\log_e d$.

output 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 output 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 bring a bucket into memory during the next level of recursion will be within a small constant factor of the optimum.

5.1.3 Randomized Cycling Distribution Sort

The distribution sort methods that we mentioned above for parallel disks perform output operations in complete stripes, which make it easy to write parity information for use in error correction and recovery. But since the blocks that belong to a given stripe typically belong to multiple buckets, the buckets themselves will not be striped on the disks, and we must use the disks independently during the input operations in the next level of recursion. In the output phase, each bucket must therefore keep track of the last block output 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 input operations can be done in a striped manner. As a result, the output I/O operations must use disks independently, since during each output step, multiple buckets will be transmitting 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 output to the current stripe, and after $D - 1$ blocks have been output, the parity information in the buffer can be output to the final (Dth) block in the stripe.

Under this new scenario, the basic loop of the distribution sort algorithm is, as before, to stream the data items through internal memory and partition them into S buckets. However, unlike before, the blocks for each individual bucket will reside on the disks in stripes. Each block therefore has a predefined disk where it must be output. If we choose

the normal round-robin ordering of the disks for the stripes (namely, 1, 2, 3, ..., D, 1, 2, 3, ..., D, ...), then blocks of different buckets may "collide," meaning that they want to be output to the same disk at the same time, and since the buckets use the same round-robin ordering, subsequent blocks in those same buckets will also tend to collide.

Vitter and Hutchinson [342] 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. That is, each bucket has its own random permutation ordering, chosen independently from those of the other $S - 1$ buckets, and the blocks of each bucket are output to the disks in a round-robin manner using its permutation ordering. If two blocks (from different buckets) happen to collide during an output to the same disk, one block is output to the disk and the other is kept in an output buffer in internal memory. With high probability, subsequent blocks in those two buckets will be output to different disks and thus will not collide.

As long as there is a small pool of D/ε block-sized output buffers to temporarily cache the blocks, Vitter and Hutchinson [342] show analytically that with high probability the output proceeds optimally in $(1 + \varepsilon)n$ I/Os. We also need $3D$ blocks to buffer blocks waiting to enter the distribution process [220, problem 5.4.9–26]. There may be some blocks left in internal memory at the end of a distribution pass. In the pathological case, they may all belong to the same bucket. This situation can be used as an advantage by choosing the bucket to recursively process next to be the one with the most blocks in memory.

The resulting sorting algorithm, called *randomized cycling distribution sort* (RCD), provably achieves the optimal sorting I/O bound (5.1) on the average with extremely small constant factors. In particular, for any parameters $\varepsilon,\ \delta > 0$, assuming that $m \geq D(\ln 2 + \delta)/\varepsilon + 3D$, the average number of I/Os performed by RCD is

$$\left(2 + \varepsilon + O(e^{-\delta D})\right) \frac{n}{D} \left\lceil \log_{m-3D-D(\ln 2+\delta)/\varepsilon} \frac{n}{m} \right\rceil + 2\frac{n}{D} + o\left(\frac{n}{D}\right). \quad (5.2)$$

When $D = o(m)$, for any desired constant $0 < \alpha < 1$, we can choose ε and δ appropriately to bound (5.2) as follows with a constant of

proportionality of 2:

$$\sim 2\frac{n}{D}\lceil \log_{\alpha m} n \rceil. \tag{5.3}$$

The only differences between (5.3) and the leading term of the lower bound we derive in Chapter 6 are the presence of the ceiling around the logarithm term and the fact that the base of the logarithm is arbitrarily close to m but not exactly m.

RCD operates very fast in practice. Figure 5.2 shows a typical simulation [342] that indicates that RCD operates with small buffer memory requirements; the layout discipline associated with the SRM method discussed in Section 5.2.1 performs similarly.

Randomized cycling distribution sort and the related merge sort algorithms discussed in Sections 5.2.1 and 5.3.4 are the methods of

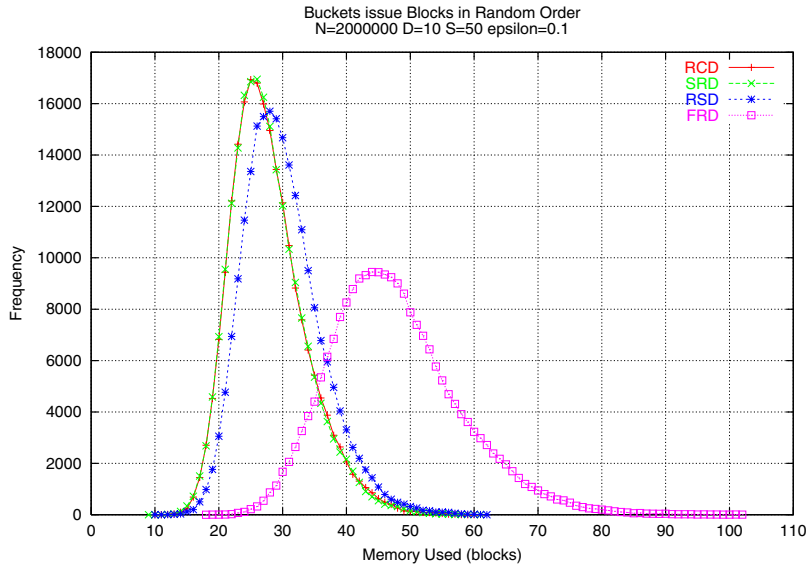

Fig. 5.2 Simulated distribution of memory usage during a distribution pass with $n = 2 \times 10^6$, $D = 10$, $S = 50$, $\varepsilon = 0.1$ for four methods: RCD (randomized cycling distribution), SRD (simple randomized distribution — striping with a random starting disk), RSD (randomized striping distribution — striping with a random starting disk for each stripe), and FRD (fully randomized distribution — each bucket is independently and randomly assigned to a disk). For these parameters, the performance of RCD and SRD are virtually identical.

choice for sorting with parallel disks. 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 [344]. On the other hand, the merge approaches can take advantage of replacement selection to start off with larger run sizes.

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 [220]: 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 $M \gg B$ [136, 220].

After the initial runs are formed, the merging phase begins, as shown in Figure 5.3 for the case $D = 1$. In each pass of the merging phase, we merge groups of R runs. For each merge, we scan the R runs and

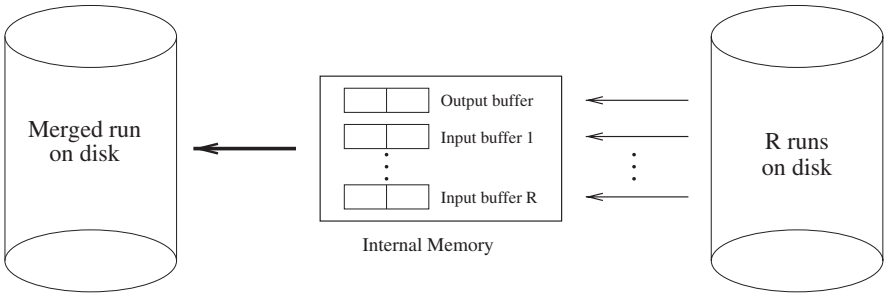

Fig. 5.3 Schematic illustration of a merge during the course of a single-disk ($D = 1$) merge sort. (For simplicity, the input and output use separate disks.) Each of R sorted runs on the disk on the right are streamed through internal memory and merged into a single sorted run that is output to the disk on the left. Each run has double buffers of total size at least $2B$ to allow the input from the runs to be overlapped with the output of the merged run.

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 (5.1), 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 input operation during the merging must on the average bring in the next $\Theta(D)$ blocks needed. The challenge is to ensure that those blocks reside on different disks so that they can be input in a single parallel 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 output 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 [172, 220]: 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 [274] 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 input 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 added to the output stripe, and the remaining B items are output back to the disks. The resulting run that is produced is only an "approximately" merged run, but its saving grace is that no

two inverted items are very far apart. A final application of Columnsort [232] suffices to restore total order; partial striping is employed to meet the memory constraints. One disadvantage of Greed Sort is that the input and output I/O operations involve independent disks and are not done in a striped manner, thus making it difficult to write parity information for error correction and recovery.

Chaudhry and Cormen [97] show experimentally that oblivious algorithms such as Columnsort work well in the context of cluster-based sorting.

Aggarwal and Plaxton [22] developed an optimal deterministic merge sort based upon the Sharesort hypercube parallel sorting algorithm [126]. 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 the optimum), but the constant factor is larger than for the distribution sort methods.

5.2.1 Simple Randomized Merge Sort

One approach to merge sort is to stripe each run across the disks and use the disk striping technique of Section 4.2. However, disk striping devotes too much internal memory (namely, $2RD$ blocks) to cache blocks not yet merged, and thus the effective order of the merge is reduced to $R = \Theta(m/D)$ (cf. (4.2)), which gives a nonoptimal result.

A better approach is the *simple randomized merge sort* (SRM) algorithm of Barve et al. [68, 72], referred to as "randomized striping" by Knuth [220]. It uses much less space in internal memory for caching blocks and thus allows R to be much larger. Each run is striped across the disks, but with a random starting point (the only place in the algorithm where randomness is utilized). During the merging process, the next block needed from each disk is input into memory, and if there is not enough room, the least needed blocks are "flushed" back to disk (without any I/Os required) to free up space.

Barve et al. [68] derive an asymptotic upper bound on the expected I/O performance, with no assumptions about the original distribution of items in the file. 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"). We conjecture that the expected maximum bin size in the cyclic occupancy problem is at most that of the classical occupancy problem [68, 220, problem 5.4.9–28]. The bound has been established so far only in an asymptotic sense [68].

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. Barve and Vitter [72] give experimental confirmation of the speedup with six fast disk drives and a 500 megahertz CPU, as shown in Table 5.1.

When the internal memory is large enough to provide $\Theta(D \log D)$ blocks of cache space and $3D$ blocks for output buffers, SRM provably achieves the optimal I/O bound (5.1). For any parameter $\varepsilon \to 0$, assuming that $m \ge D(\log D)/\varepsilon^2 + 3D$, the average number of I/Os performed by SRM is

$$(2+\varepsilon)\frac{n}{D}\left\lceil \log_{m-3D-D(\log D)/\varepsilon^2} \frac{n}{m} \right\rceil + 2\frac{n}{D} + o\left(\frac{n}{D}\right). \tag{5.4}$$

When $D = o(m/\log m)$, for any desired constant $0 < \alpha < 1$, we can choose ε to bound (5.4) as follows with a constant of proportionality of 2:

$$\sim 2\frac{n}{D}\lceil \log_{\alpha m} n \rceil. \tag{5.5}$$

Table 5.1 The ratio of the number of I/Os used by simple randomized merge sort (SRM) to the number of I/Os used by merge sort with disk striping, during a merge of kD runs, 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

In the next section, we show how to get further improvements in merge sort by a more careful prefetch schedule for the runs, combined with the randomized cycling strategy discussed in Section 5.1.

5.3 Prefetching, Caching, and Applications to Sorting

In this section, we consider the problem of *prefetch scheduling* for parallel disks: We are given a sequence of blocks

$$\Sigma = \langle b_1, b_2, \ldots, b_N \rangle.$$

The initial location of the blocks on the D disks is prespecified by an arbitrary function

$$disk : \Sigma \to \{1, 2, \ldots, D\}.$$

That is, block b_i is located on disk $disk(b_i)$. The object of prefetching is to schedule the fewest possible number of input I/O operations from the D disks so that the blocks can be read by an application program in the order given by Σ. When a block is given to the application, we say that it is "read" and can be removed from internal memory. (Note that "read" refers to the handover of the block from the scheduling algorithm to the application, whereas "input" refers to the prior I/O operation that brought the block into internal memory.) We use the m blocks of internal memory as *prefetch buffers* to store blocks that will soon be read.

If the blocks in Σ are distinct, we call the prefetching problem *read-once scheduling*. If some blocks are repeated in Σ (i.e., they must be read at more than one time by the application in internal memory), then it may be desirable to cache blocks in the prefetch buffers in order to avoid re-inputting them later, and we call the problem *read-many scheduling*.

One way to make more informed prefetching decisions is to use knowledge or prediction of future read requests, which we can informally call *lookahead*. Cao et al. [95], Kimbrel and Karlin [217], Barve et al. [69], Vitter and Krishnan [343], Curewitz et al. [125], Krishnan and Vitter [226], Kallahalla and Varman [204, 205], Hutchinson et al. [202], Albers and Büttner [28], Shah et al. [312, 313], and

Hon et al. [198] have developed competitive and optimal methods for prefetching blocks in parallel I/O systems.

We focus in the remainder of this section on prefetching with knowledge of future read requests. We use the general framework of Hutchinson et al. [202], who demonstrate a powerful duality between prefetch scheduling and the following problem "in the reverse direction," which we call *output scheduling*: We are given a sequence

$$\Sigma' = \langle b'_1, b'_2, \ldots, b'_N \rangle$$

of blocks that an application program produces (or "writes") in internal memory. A mapping

$$disk : \Sigma' \to \{1, 2, \ldots, D\}$$

specifies the desired final disk location for each block; that is, the target location for block b'_i is disk $disk(b'_i)$. The goal of output scheduling is to construct the shortest schedule of parallel I/O output operations so that each block b'_i is output to its proper target location disk $disk(b'_i)$. (Note that "write" refers to the handover of the block from the application to the scheduling algorithm, whereas "output" refers to the subsequent I/O operation that moves the block onto disk.) We use the m blocks of internal memory as *output buffers* to queue the blocks that have been written but not yet output to the disks.

If the blocks in Σ' are distinct, we call the output scheduling problem *write-once scheduling*. If some blocks are repeated in Σ', we call the problem *write-many scheduling*. If we are able to keep a block in an output buffer long enough until it is written again by the application program, then we need to output the block at most once rather than twice.

The output scheduling problem is generally easy to solve optimally. In Sections 5.3.2–5.3.4, we exploit a duality [202] of the output scheduling problems of write-once scheduling, write-many scheduling, and distribution in order to derive optimal algorithms for the dual prefetch problems of read-once scheduling, read-many scheduling, and merging. Figure 5.4 illustrates the duality and information flow for prefetch and output scheduling.

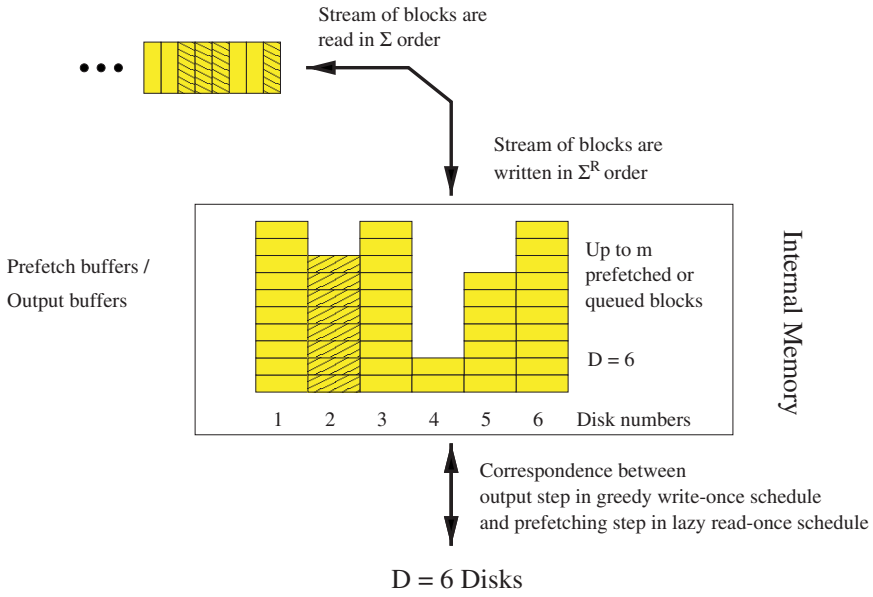

Fig. 5.4 Duality between prefetch scheduling and output scheduling. The prefetch scheduling problem for sequence Σ proceeds from bottom to top. Blocks are input from the disks, stored in the prefetch buffers, and ultimately read by the application program in the order specified by the sequence Σ. The output scheduling problem for the reverse sequence Σ^R proceeds from top to bottom. Blocks are written by the application program in the order specified by Σ^R, queued in the output buffers, and ultimately output to the disks. The hatched blocks illustrate how the blocks of disk 2 might be distributed.

5.3.1 Greedy Read-Once Scheduling

Before we discuss an optimum prefetching algorithm for read-once scheduling, we shall first look at the following natural approach adopted by SRM [68, 72] in Section 5.2.1, which unfortunately does not achieve the optimum schedule length. It uses a greedy approach: Suppose that blocks $b_1, b_2, \ldots, b_i$ of the sequence Σ have already been read in prior steps and are thus removed from the prefetch buffers. The current step consists of reading the next blocks of Σ that are already in the prefetch buffers. That is, suppose blocks $b_{i+1}, b_{i+2}, \ldots, b_j$ are in the prefetch buffers, but b_{j+1} is still on a disk. Then blocks $b_{i+1}, b_{i+2}, \ldots, b_j$ are read and removed from the prefetch buffers.

The second part of the current step involves input from the disks. For each of the D disks, consider its highest priority block not yet input,

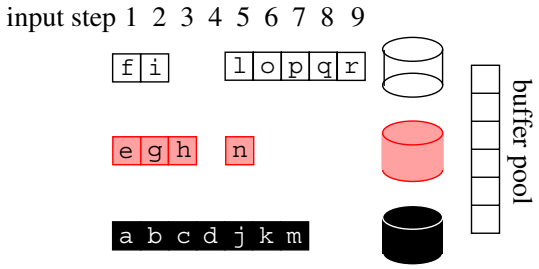

Fig. 5.5 Greedy prefetch schedule for sequence $\Sigma = \langle \mathtt{a}, \mathtt{b}, \mathtt{c}, \ldots, \mathtt{r} \rangle$ of $n = 18$ blocks, read using $D = 3$ disks and $m = 6$ prefetch buffers. The shading of each block indicates the disk it belongs on. The schedule uses $T = 9$ I/O steps, which is one step more than optimum.

according to the order specified by Σ. We use P to denote the set of such blocks. (It necessarily follows that b_{j+1} must be in P.) Let *Res* be the blocks already resident in the prefetch buffer, and let Res' be the m highest priority blocks in $P \cup Res$. We input the blocks of $Res' \cap P$ (i.e., the blocks of Res' that are not already in the prefetch buffer). At the end of the I/O step, the prefetch buffer contains the blocks of Res'. If a block b in *Res* does not remain in Res', then the algorithm has effectively "kicked out" b from the prefetch buffers, and it will have to be re-input in a later I/O step.

An alternative greedy policy is to not evict records from the prefetch buffers; instead we input the $m - |Res'|$ highest priority blocks of P.

Figure 5.5 shows an example of the greedy read schedule for the case of $m = 6$ prefetch buffers and $D = 3$ disks. (Both greedy versions described above give the same result in the example.) An optimum I/O schedule has $T = 8$ I/O steps, but the greedy schedule uses a nonoptimum $T = 9$ I/O steps. Records `g` and `h` are input in I/O steps 2 and 3 even though they are not read until much later, and as a result they take up space in the prefetch buffers that prevents block `l` (and thus blocks `o`, `p`, `q`, and `r`) from being input earlier.

5.3.2 Prefetching via Duality: Read-Once Scheduling

Hutchinson et al. [202] noticed a natural correspondence between a prefetch schedule for a read-once sequence Σ and an output schedule for the write-once sequence Σ^{R}, where Σ^{R} denotes the sequence of

blocks of Σ in reverse order. In the case of the write-once problem, the following natural greedy output algorithm is optimum: As the blocks of Σ^{R} are written, we put each block into an output buffer for its designated disk. There are m output buffers, each capable of storing one block, so the writing can proceed only as quickly as space is freed up in the write buffers. In each parallel I/O step, we free up space by outputting a queued block to each disk that has at least one block in an output buffer. The schedule of I/O output steps is called the output schedule, and it is easy to see that it is optimum in terms of the number of parallel I/O steps required. Figure 5.6, when read right to left, shows the output schedule for the reverse sequence Σ^{R}.

When we run any output schedule for Σ^{R} in reverse, we get a valid prefetch schedule for the original sequence Σ, and vice versa. Therefore, an optimum output schedule for Σ^{R}, which is easy to compute in a greedy fashion, can be reversed to get an optimum prefetch schedule for Σ. Figure 5.6, when considered from right to left, shows an optimum output schedule for sequence Σ^{R}. When looked at left to right, it shows an optimum prefetch schedule for sequence Σ.

The prefetch schedule of Figure 5.6 is "lazy" in the sense that the input I/Os seem to be artificially delayed. For example, cannot block `e` be input earlier than in step 4? Hutchinson et al. [202] give a *prudent prefetching* algorithm that guarantees the same optimum prefetch schedule length, but performs I/Os earlier when possible. It works by redefining the priority of a block to be the order it appears in the input step order of the lazy schedule (i.e., `a`, `b`, `f`, `c`, `i`, ... in Figure 5.6). Prefetch buffers are "reserved" for blocks in the same order of priority. In the current I/O step, using the language of Section 5.3.1, we input every block in P that has a reserved prefetch buffer. Figure 5.7 shows the prudent version of the lazy schedule of Figure 5.6. In fact, if we iterate the prudent algorithm, using as the block priorities the input step order of the prudent schedule of Figure 5.7, we get an even more prudent schedule, in which the `e` and `n` blocks move up one more I/O step than in Figure 5.7. Further iteration in this particular example will not yield additional improvement.

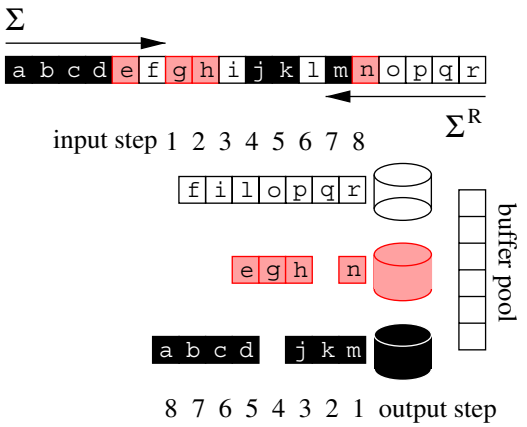

Fig. 5.6 Optimum dual schedules: read-once schedule for Σ via lazy prefetching and write-once schedule for Σ^{R} via greedy output. The read sequence $\Sigma = \langle \mathtt{a}, \mathtt{b}, \mathtt{c}, \ldots, \mathtt{r} \rangle$ of $n = 18$ blocks and its reverse sequence Σ^{R} are read/written using $D = 3$ disks and $m = 6$ prefetch/output buffers. The shading of each block indicates the disk it belongs on. The input steps are pictured left to right, and the output steps are pictured right to left. The schedule uses $T = 8$ I/O steps, which is optimum.

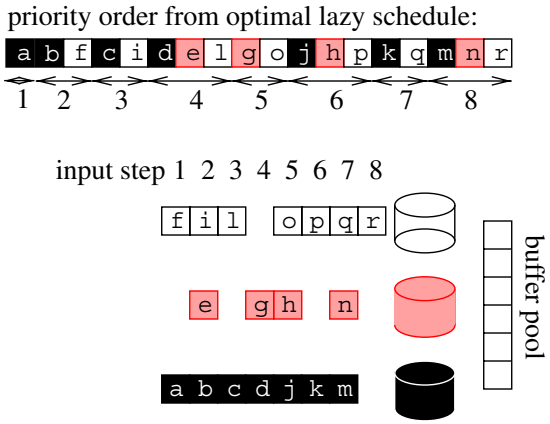

Fig. 5.7 Prudent prefetch schedule for the example in Figure 5.6. The blocks are prioritized using the input step order from the lazy prefetch schedule in Figure 5.6. The prefetch schedule remains optimum with $T = 8$ I/O steps, but some of the blocks are input earlier than in the lazy schedule.

To get an idea of how good optimum is, suppose that Σ is composed of S subsequences interlaced arbitrarily and that each of the S subsequences is striped in some manner on the disks.

Theorem 5.2 ([202]). If each of the S subsequences forming Σ is stored on the disks in a randomized cycling layout [342] (discussed in Section 5.1), the expected number of I/O steps for our optimum (lazy or prudent) prefetching method is

$$\left(1 + O\left(\frac{D}{m}\right)\right)\frac{n}{D} \;+\; \min\left\{S + \frac{n}{D},\; O\left(\frac{m}{D}\log m\right)\right\}. \tag{5.6}$$

If the internal memory size is large (i.e., $m > S(D-1)$) to guarantee outputting D blocks per I/O at each output step, then for both striped layouts and randomized cycling layouts, the number of I/Os is

$$\left\lfloor \frac{n}{D} \right\rfloor + S. \tag{5.7}$$

The second term in (5.7) can be reduced further for a randomized cycling layout.

In each of the expressions (5.6) and (5.7), the second of its two terms corresponds to the I/Os needed for initial loading of the prefetch buffers in the case of the lazy prefetch schedule (or equivalently, the I/Os needed to flush the output buffers in the case of the greedy output schedule).

5.3.3 Prefetching with Caching via Duality: Read-Many Scheduling

When any given block can appear multiple times in Σ, as in read-many scheduling, it can help to cache blocks in the prefetch buffers in order to avoid re-inputting them for later reads. For the corresponding write-once problem, we can construct an optimum output schedule via a modified greedy approach. In particular, at each output step and for each disk, we choose as the block to output to the disk the one whose next appearance in Σ^{R} is furthest into the future — a criteria akin to the least-recently-used heuristic developed by Belady [78] for cache

replacement. Intuitively, the "furthest" block will be of least use to keep in memory for a later write. Hutchinson et al. [202] show that the output schedule is provably optimum, and hence by duality, if we solve the write-many problem for Σ^{R}, then when we run the output schedule in reverse, we get an optimum read-many prefetch schedule for Σ.

5.3.4 Duality between Merge Sort and Distribution Sort

In the case of merging R runs together, if the number R of runs is small enough so that we have RD block-sized prefetch buffers in internal memory, then it is easy to see that the merge can proceed in an optimum manner. However, this constraint limits the size of R, as in disk striping, which can be suboptimal for sorting. The challenge is to make use of substantially fewer prefetch buffers so that we can increase R to be as large as possible. The larger R is, the faster we can do merge sort, or equivalently, the larger the files that we can sort in a given number of passes. We saw in Section 5.2.1 that $\Theta(D \log D)$ prefetch buffers suffice for SRM to achieve optimal merge sort performance.

A tempting approach is duality: We know from Section 5.1.3 that we need only $\Theta(D)$ output buffers to do a distribution pass if we lay out the buckets on the disks in a randomized cycling (RCD) pattern. If we can establish duality, then we can merge runs using $\Theta(D)$ prefetch buffers, assuming the runs are stored on the disks using randomized cycling. Figure 5.8 illustrates the duality between merging and distribution.

However, one issue must first be resolved before we can legitimately apply duality. In each merge pass of merge sort, we merge R runs at a time into a single run. In order to apply duality, which deals with read and write sequences, we need to predetermine the read order Σ for the merge. That is, if we can specify the proper read order Σ of the blocks, then we can legitimately apply Theorem 5.2 to the write problem on Σ^{R}.

The solution to determining Σ is to partition internal memory so that not only does it consist of several prefetch buffers but it also includes R merging buffers, where R is the number of runs. Each merging buffer stores a (partially filled) block from a run that is participating in the merge. We say that a block is *read* when it is moved from

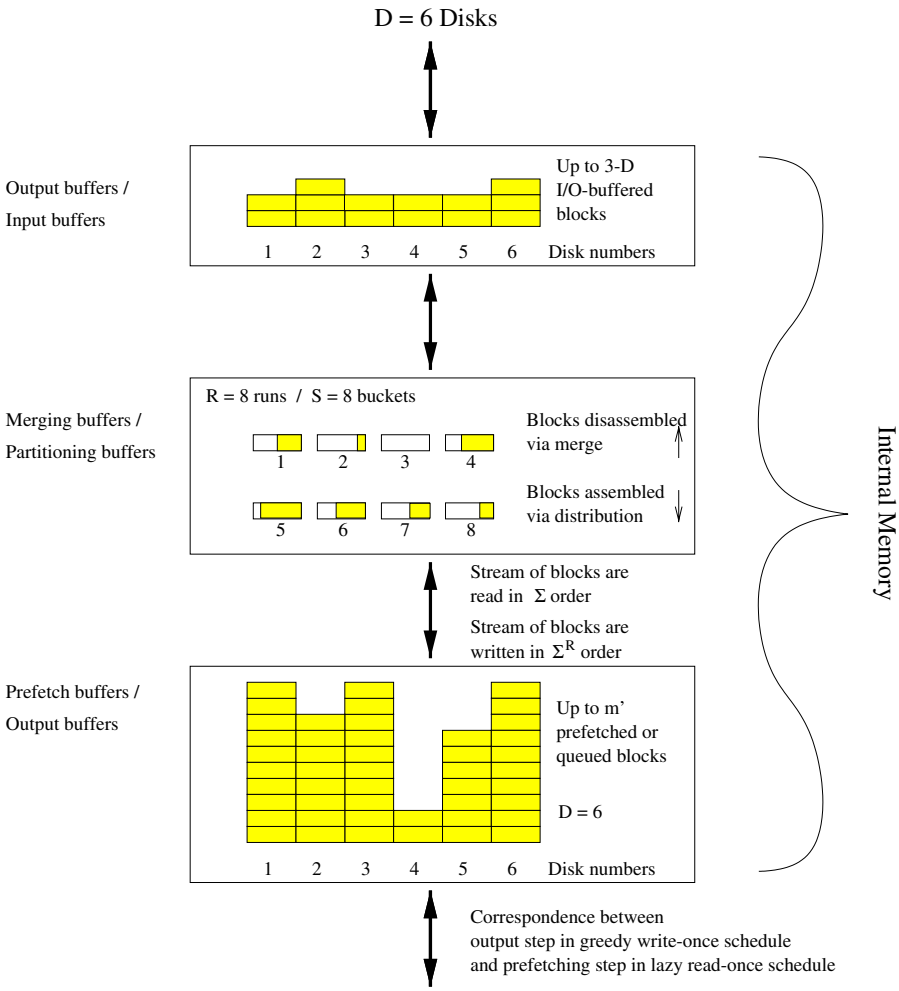

Fig. 5.8 Duality between merging with $R = 8$ runs and distribution with $S = 8$ buckets, using $D = 6$ disks. The merge of the R runs proceeds from bottom to top. Blocks are input from the disks, stored in the prefetch buffers, and ultimately read into the merging buffers. The blocks of the merged run are moved to the output buffers and then output to the disks. The order in which blocks enter the merging buffers determines the sequence Σ, which can be predetermined by ordering the blocks based upon their smallest key values. The distribution into S buckets proceeds from top to bottom. Blocks are input from the disks into input buffers and moved to the partitioning buffers. The blocks of the resulting buckets are written in the order Σ^R to the output buffers and then output to the appropriate disks.

a prefetch buffer to a merging buffer, where it stays until its items are exhausted by the merging process. When a block expires, it is replaced by the next block in the read sequence Σ (unless Σ has expired) before the merging is allowed to resume. The first moment that a block absolutely must be read and moved to the merging buffer is when its smallest key value enters into the merging process. We therefore define the read priority of a block b to be its *smallest key value*. We can sort the smallest key values (one per block) to form the read order Σ. Computing the read sequence Σ is fast to do because sorting N/B key values is a considerably smaller problem than sorting the entire file of N records.

A subtle point is to show that this Σ ordering actually "works," namely, that at each step of the merging process, the item r with the smallest key value not yet in the merged run will be added next to the merged run. It may be, for example, that the R merging buffers contain multiple blocks from one run but none from another. However, at the time when item r should be added to the merged run, there can be at most one other nonempty run in a merging buffer from each of the other $R - 1$ runs. Therefore, since there are R merging buffers and since the merging proceeds only when all R merging buffers are nonempty (unless Σ is expired), it will always be the case that the block containing r will be resident in one of the merging buffers before the merging proceeds.

We need to use a third partition of internal memory to serve as output buffers so that we can output the merged run in a striped fashion to the D disks. Knuth [220, problem 5.4.9–26] has shown that we may need as many output buffers as prefetch buffers, but about $3D$ output buffers typically suffice. So the remaining $m' = m - R - 3D$ blocks of internal memory are used as prefetch buffers.

We get an optimum merge schedule for read sequence Σ by computing the greedy output schedule for the reverse sequence Σ^{R}. Figure 5.8 shows the flow through the various components in internal memory.

When the runs are stored on the disks using randomized cycling, the length of the greedy output schedule corresponds to the performance of a distribution pass in RCD, which is optimal. We call the resulting merge sort *randomized cycling merge sort* (RCM). It has the identical I/O performance bound (5.2) as does RCD, except that each level of

merging requires some extra overhead to fill the prefetch buffers to start the merge, corresponding to the additive terms in Theorem 5.2. For any parameters $\varepsilon,\ \delta > 0$, assuming that $m \geq D(\ln 2 + \delta)/\varepsilon + 3D$, the average number of I/Os for RCM is

$$\left(2 + \varepsilon + O(e^{-\delta D})\right) \frac{n}{D} \left\lceil \log_{m-3D-D(\ln 2+\delta)/\varepsilon} \frac{n}{m} \right\rceil + 2\frac{n}{D} + \min\left\{ \frac{n}{D}, O\left(\frac{\log m}{\varepsilon}\right) \right\} + o\left(\frac{n}{D}\right). \tag{5.8}$$

When $D = o(m) = o(n/\log n)$, for any desired constant $\alpha > 0$, we can choose ε and δ appropriately to bound (5.8) as follows with a constant of proportionality of 2:

$$\sim 2\frac{n}{D}\left\lceil \log_{\alpha m} n \right\rceil. \tag{5.9}$$

Dementiev and Sanders [136] show how to overlap computation effectively with I/O in the RCM method. We can apply the duality approach to other methods as well. For example, we could get a simple randomized distribution sort that is dual to the SRM method of Section 5.2.1.

5.4 A General Simulation for Parallel Disks

Sanders et al. [304] and Sanders [303] 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 input and output I/O steps use the disks independently. Armen [60] 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. 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 $k =$

$O(D)$ blocks from the disks. For each block, there is a choice of two disks from which it can be input. Regardless of which k blocks are requested, Sanders et al. show that with high probability $\lceil k/D \rceil$ or $\lceil k/D \rceil + 1$ I/Os suffice to retrieve all k blocks. They also give a simple linear-time greedy algorithm that achieves optimal input schedules with high probability. 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 outputting blocks of data to the disks, each block must be output to both the disks where a copy is stored. Sanders et al. prove that with high probability D blocks can be output in $O(1)$ I/O steps, assuming that there are $O(D)$ blocks of internal buffer space to serve as output buffers. The I/O bounds can be improved with a corresponding tradeoff in redundancy and internal memory space.

5.5 Handling Duplicates: Bundle Sorting

Arge et al. [42] 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. When duplicates get grouped together during a merge, they are replaced by a single copy of the item and a count of the occurrences. The algorithm runs in $O\big(n \max\{1, \log_m(K/B)\}\big)$ 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 that must be maintained, and therefore items cannot be aggregated with a count. Abello et al. [2] and Matias et al. [249] develop optimal distribution sort algorithms for bundle sorting using

$$\mathit{BundleSort}(N,K) = O\big(n \cdot \max\{1, \log_m \min\{K,n\}\}\big) \tag{5.10}$$

I/Os, and Matias et al. [249] prove a matching lower bound. Matias et al. [249] 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.

5.6 Permuting

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 5.3 ([345]). The average-case and worst-case number of I/Os required for permuting N data items using D disks is

$$\Theta\left(\min\left\{\frac{N}{D}, \mathit{Sort}(N)\right\}\right). \tag{5.11}$$

The I/O bound (5.11) for permuting can be realized by one of the optimal external sorting algorithms except in the extreme case for which $B \log m = o(\log n)$, when 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 [345].

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. We examine special classes of permutations having to do with matrices, such as matrix transposition, in Chapter 7.

5.7 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 that can realize any of the $N!$ possible

permutations; data items can only be reordered when they are in internal memory. A permutation network can be constructed by a series of three FFTs [358].

Theorem 5.4 ([23]). 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 [119] give some practical implementations for one-dimensional FFTs based upon the optimal PDM algorithm of Vitter and Shriver [345]. 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.

6

Lower Bounds on I/O

In this chapter, we prove the lower bounds from Theorems 5.1–5.4, including a careful derivation of the constants of proportionality in the permuting and sorting lower bounds. We also mention some related I/O lower bounds for the batched problems in computational geometry and graphs that we cover later in Chapters 8 and 9.

6.1 Permuting

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 if we assume that the N items are indivisible and must be transferred as individual entities, permuting requires a nonlinear number of I/Os in PDM because of the locality constraints imposed by the block parameter B.

Our main result for parallel disk sorting is that we close the gap between the upper and lower bounds up to lower order terms. The lower bound from [23] left open the nature of the constant factor of

proportionality of the leading term; in particular, it was not clear what happens if the number of output steps and input steps differ.

Theorem 6.1 ([202]). Assuming that $m = M/B$ is an increasing function, the number of I/Os required to sort or permute n indivisible items, up to lower-order terms, is at least

$$\frac{2N}{D}\frac{\log n}{B\log m + 2\log N} \sim \begin{cases} \dfrac{2n}{D}\log_m n & \text{if } B\log m = \omega(\log N); \\ \dfrac{N}{D} & \text{if } B\log m = o(\log N). \end{cases} \tag{6.1}$$

The main case in Theorem 6.1 is the first one, and this theorem shows that the constant of proportionality in the $Sort(N)$ bound (5.1) of Theorem 5.1 is at least 2.

The second case in the theorem is the pathological case in which the block size B and internal memory size M are so small that the optimum way to permute the items is to move them one at a time in the naive manner, not making use of blocking.

We devote the rest of this section to a proof of Theorem 6.1. For the lower bound calculation, we can assume without loss of generality that there is only one disk, namely, $D = 1$. The I/O lower bound for general D follows by dividing the lower bound for one disk by D.

We call an input operation *simple* if each item that is transferred from the disk gets removed from the disk and deposited into an empty location in internal memory. Similarly, an output is *simple* if the transferred items are removed from internal memory and deposited into empty locations on disk.

Lemma 6.2 ([23]). For each computation that implements a permutation of the N items, there is a corresponding computation strategy involving only simple I/Os such that the total number of I/Os is no greater.

The lemma can be demonstrated easily by starting with a valid permutation computation and working backwards. At each I/O step,

in backwards order, we cancel the transfer of an item if its transfer is not needed for the final result; if it is needed, we make the transfer simple. The resulting I/O strategy has only simple I/Os.

For the lower bound, we use the basic approach of Aggarwal and Vitter [23] and bound the maximum number of permutations that can be produced by at most t I/Os. If we take the value of t for which the bound first reaches $N!$, we get a lower bound on the worst-case number of I/Os. In a similar way, we can get a lower bound on the average case by computing the value of t for which the bound first reaches $N!/2$.

In particular, we say that a permutation $\langle p_1, p_2, \ldots, p_N \rangle$ of the N items can be *produced after t_I input operations and t_O output operations* if there is some intermixed sequence of t_I input operations and t_O output operations so that the items end up in the permuted order $\langle p_1, p_2, \ldots, p_N \rangle$ in extended memory. (By extended memory we mean the memory locations of internal memory followed by the memory locations on disk, in sequential order.) The items do not have to be in contiguous positions in internal memory or on disk; there can be arbitrarily many empty locations between adjacent items.

As mentioned above, we can assume that I/Os are simple. Each I/O causes the transfer of exactly B items, although some of the items may be **nil**. In the PDM model, the I/Os obey block boundaries, in that all the non-**nil** items in a given I/O come from or go to the same block on disk.

Initially, before any I/Os are performed and the items reside on disk, the number of producible permutations is 1. Let us consider the effect of an output. There can be at most $N/B + o - 1$ nonempty blocks before the oth output operation, and thus the items in the oth output can go into one of $N/B + o$ places relative to the other blocks. Hence, the oth output boosts the number of producible permutations by a factor of at most $N/B + o$, which can be bounded trivially by

$$N(1 + \log N). \tag{6.2}$$

For the case of an input operation, we first consider an input I/O from a specific block on disk. If the b items involved in the input I/O were together in internal memory at some previous time (e.g., if the block was created by an earlier output operation), then the items could

have been arranged in an arbitrary order by the algorithm while they were in internal memory. Thus, the $b!$ possible orderings of the b input items relative to themselves could already have been produced before the input operation. This implies in a subtle way that rearranging the newly input items among the other $M - b$ items in internal memory can boost the number of producible permutations by a factor of at most $\binom{M}{b}$, which is the number of ways to intersperse b indistinguishable items within a group of size M.

The above analysis applies to input from a specific block. If the input was preceded by a total of o output operations, there are at most $N/B + o \leq N(1 + \log N)$ blocks to choose from for the I/O, so the number of producible permutations is boosted further by at most $N(1 + \log N)$. Therefore, assuming that at some prior time the b input items were together in internal memory, an input operation can boost the number of producible permutations by at most

$$N(1 + \log N)\binom{M}{b}. \tag{6.3}$$

Now let us consider an input operation in which some of the input items were not together previously in internal memory (e.g., the first time a block is input). By rearranging the relative order of the items in internal memory, we can increase the number of producible permutations by a factor of $B!$. Given that there are N/B full blocks initially, we get the maximum increase when all N/B blocks are input in full, which boosts the number of producible permutations by a factor of

$$(B!)^{N/B}. \tag{6.4}$$

Let I be the total number of input I/O operations. In the ith input operation, let b_i be the number of items brought into internal memory. By the simplicity property, some of the items in the block being accessed may not be brought into internal memory, but rather may be left on disk. In this case, b_i counts only the number of items that are removed from disk and put into internal memory. In particular, we have $0 \leq b_i \leq B$.

By the simplicity property, we need to make room in internal memory for the new items that arrive, and in the end all items are stored

back on disk. Therefore, we get the following lower bound on the number O of output operations:

$$O \geq \frac{1}{B}\Big(\sum_{1\leq i\leq I} b_i\Big). \tag{6.5}$$

Combining (6.2), (6.3), and (6.4), we find that

$$\big(N(1+\log N)\big)^{I+O} \prod_{1\leq i\leq I} \binom{M}{b_i} \geq \frac{N!}{(B!)^{N/B}}, \tag{6.6}$$

where O satisfies (6.5).

Let $\tilde{B} \leq B$ be the average number of items input during the I input operations. By a convexity argument, the left-hand side of (6.6) is maximized when each b_i has the same value, namely, $\tilde{B}$. We can rewrite (6.5) as $O \geq I\tilde{B}/B$, and thus we get $I \leq (I+O)/(1+\tilde{B}/B)$. Combining these facts with (6.6), we get

$$\big(N(1+\log N)\big)^{I+O} \binom{M}{\tilde{B}}^{I} \geq \frac{N!}{(B!)^{N/B}}; \tag{6.7}$$

$$\big(N(1+\log N)\big)^{I+O} \binom{M}{\tilde{B}}^{(I+O)/(1+\tilde{B}/B)} \geq \frac{N!}{(B!)^{N/B}}. \tag{6.8}$$

By assumption that M/B is an increasing function, the left-hand side of (6.8) is maximized when $\tilde{B} = B$, so we get

$$\big(N(1+\log N)\big)^{I+O} \binom{M}{B}^{(I+O)/2} \geq \frac{N!}{(B!)^{N/B}}. \tag{6.9}$$

The lower bound on $I+O$ for $D=1$ follows by taking logarithms of both sides of (6.9) and solving for $I+O$ using Stirling's formula. We get the general lower bound of Theorem 6.1 for D disks by dividing the result by D.

6.2 Lower Bounds for Sorting and Other Problems

Permuting is a special case of sorting, and hence, the permuting lower bound of Theorem 6.1 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 (5.1) of Theorem 5.1 [23].

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 constant of proportionality of 2 in the lower bound (6.1) of Theorem 6.1 is nearly realized by randomized cycling distribution sort (RCD) in Section 5.1.3, simple randomized merge sort (SRM) in Section 5.2.1, and the dual methods of Section 5.3.4.

The derivation of the permuting lower bound in Section 6.1 also works for permutation networks, in which the communication pattern is oblivious (fixed). Since the choice of disk block is fixed for each I/O step, there is no $N(1 + \log N)$ term as there is in (6.6), and correspondingly there is no additive $2 \log N$ term as there is in the denominator of the left-hand side of Theorem 6.1. Hence, when we solve for $I + O$, we get the lower bound (5.1) rather than (5.11). 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.

Arge et al. [42] 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 $\Omega\big(n \max\{1, \log_m(K/B)\}\big)$ I/Os in the comparison model for duplicate removal with one disk. Erickson [150] extends the sorting and element distinctness lower bound to the more general algebraic decision tree model.

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. [249] derive the matching lower bound $\mathit{BundleSort}(N, K) = \Omega\big(n \max\{1, \log_m \min\{K, n\}\}\big)$. 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 replace the right-hand side of (6.9)

by $N!/\big((N/K)!\big)^K$, which is the maximum number of permutations of N numbers having K distinct values. Solving for $I + O$ gives the lower bound $\Omega\big(n \max\{1, \log_m(K/B)\}\big)$, which is equal to the desired lower bound for $\mathit{BundleSort}(N, K)$ when $K = B^{1+\Omega(1)}$ or $M = B^{1+\Omega(1)}$. Matias et al. [249] derive the remaining case of the lower bound for $\mathit{BundleSort}(N, K)$ by a potential argument based upon the derivation of the transposition lower bound (Theorem 7.2). Dividing by D gives the lower bound for D disks.

Chiang et al. [105], Arge [30], Arge and Miltersen [45], Munagala and Ranade [264], and Erickson [150] give models and lower bound reductions for several computational geometry and graph problems. The geometry problems discussed in Chapter 8 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 forest 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. In some cases there is a gap between the best known upper and lower bounds, which we examine further in Chapter 9.

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 (5.1) remains valid even if the indivisibility assumption is lifted. However, for an artificial problem related to transposition, Adler [5] showed that removing the indivisibility assumption can lead to faster algorithms. A similar result is shown by Arge and Miltersen [45] for the decision problem of determining if N data item values are distinct. Whether the conjecture is true is a challenging theoretical open problem.

7

Matrix and Grid Computations

7.1 Matrix Operations

Dense matrices are generally represented in memory in row-major or column-major order. 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 7.1 ([199, 306, 345, 357]). 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

$$\Theta\left(\frac{K^3}{\min\{K, \sqrt{M}\}DB}\right).$$

The lower bound follows from the related pebbling lower bound by Savage and Vitter [306] for the case $D = 1$ and then dividing by D.

Hong and Kung [199] and Nodine et al. [272] give optimal EM algorithms for iterative grid computations, and Leiserson et al. [233] reduce

the number of I/Os of naive multigrid implementations by a $\Theta(M^{1/5})$ factor. Gupta et al. [188] 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 $(i, j, A_{i,j})$. Vengroff and Vitter [337] report on algorithms and benchmarks for dense and sparse matrix operations.

For further discussion of numerical EM algorithms we refer the reader to the surveys by Toledo [328] and Kowarschik and Weiß [225]. Some issues regarding programming environments are covered in [115] and Chapter 17.

7.2 Matrix Transposition

Matrix transposition is the special case of permuting that involves conversion of a matrix from row-major order to column-major order.

Theorem 7.2 ([23]). 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

$$\Theta\left(\frac{n}{D}\log_m \min\{M, p, q, n\}\right), \tag{7.1}$$

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 single pass by transposing the submatrices one at a time in internal memory.

The transposition lower bound involves a potential argument based upon a togetherness relation [23], an elaboration of an approach first developed by Floyd [165, 220] for a special case of transposition.

When the matrices are stored using a sparse representation, transposition is always as hard as sorting, unlike the $B^2 \leq M$ case for dense matrix transposition (cf. Theorem 7.2).

Theorem 7.3 ([23]). For a matrix stored in sparse format and containing N_z nonzero elements, the number of I/Os required to transpose the matrix from row-major order to column-major order, and vice-versa, is $\Theta\big(Sort(N_z)\big)$.

Sorting suffices to perform the transposition. The lower bound follows by reduction from sorting: If the ith item to sort has key value $x \neq 0$, there is a nonzero element in matrix position (i, x).

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. [120] 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 7.4 ([120]). With D disks, the number of I/Os required to perform the BMMC permutation defined by matrix A and vector c is

$$\Theta\left(\frac{n}{D}\left(1 + \frac{\operatorname{rank}(\gamma)}{\log m}\right)\right), \tag{7.2}$$

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

8

Batched Problems in Computational Geometry

For brevity, in the remainder of this manuscript 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 Chapter 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 $D \geq 1$.

Problems involving massive amounts of geometric data are ubiquitous in spatial databases [230, 299, 300], geographic information systems (GIS) [16, 230, 299, 334], constraint logic programming [209, 210], object-oriented databases [361], statistics, virtual reality systems, and computer graphics [169].

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

been developed for solving these problems in external memory, as we illustrate in the following theorem:

Theorem 8.1. Certain batched problems involving $N = nB$ items, $Q = qB$ queries, and total answer size $Z = zB$ can be solved using

$$O\big((n+q)\log_m n + z\big) \tag{8.1}$$

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 non-intersecting red line segments and N non-intersecting 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) Constructing the Voronoi diagram of N points in the plane;
(6) Constructing a triangulation of N points in the plane;
(7) Performing Q point location queries in a planar subdivision of size N;
(8) Finding all nearest neighbors for a set of N points in the plane;
(9) Finding the pairwise intersections of N orthogonal rectangles in the plane;
(10) Computing the measure of the union of N orthogonal rectangles in the plane;
(11) Computing the visibility of N segments in the plane from a point; and
(12) 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. [179], Zhu [364], Arge et al. [57], Arge et al. [48], and Crauser et al. [122, 123] develop EM algorithms for those batched problems using the following EM paradigms:

Distribution sweeping, a generalization of the distribution paradigm of Section 5.1 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 11.1), yielding a factor of B improvement over the generic persistence techniques of Driscoll et al. [142].

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 [141].

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 Tamassia and Vitter [324] to the I/O setting.

External marriage-before-conquest, an EM analogue to the technique of Kirkpatrick and Seidel [218] for performing output-sensitive convex hull constructions.

Batched incremental construction, a localized version of the randomized incremental construction paradigm of Clarkson and Shor [111], 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 together the incremental modifications.

8.1 Distribution Sweep

We focus in the remainder of this section primarily on the distribution sweep paradigm [179], which is a combination of the distribution

paradigm of Section 5.1 and the well-known sweeping paradigm from computational geometry [129, 284]. 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 8.1, 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

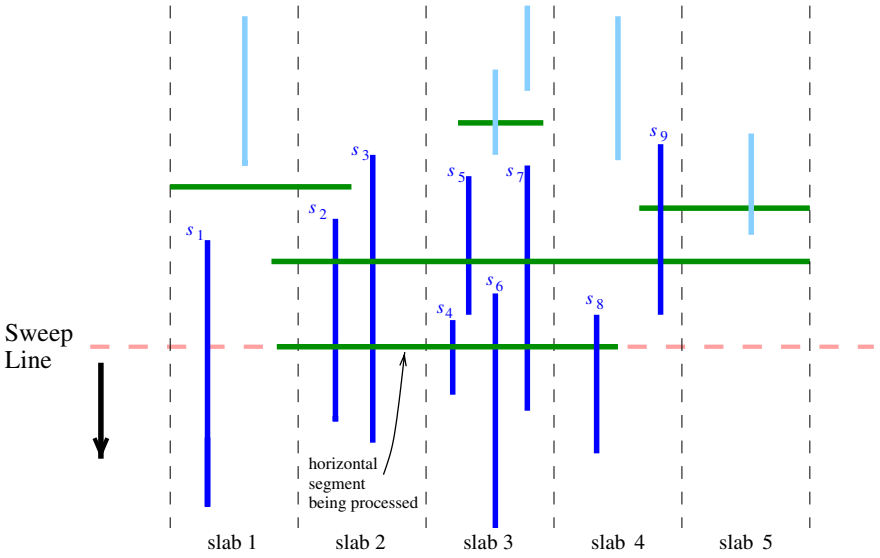

Fig. 8.1 Distribution sweep used for finding intersections among N orthogonal segments. The vertical segments currently stored in the slabs are indicated in bold (namely, s_1, s_2, ..., s_9). Segments s_5 and s_9 are not active, but have not yet been deleted from the slabs. The sweep line has just advanced 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.

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 of recursion, along with the vertical segments. The downward sweep then continues. After an initial one-time sorting (to order 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 (8.1). Some timing experiments on distribution sweeping appear in [104]. Arge et al. [48] 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 with one for range searching. Arge et al. [48] 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 $\Theta\left(\binom{\sqrt{m}}{2}\right) = \Theta(m)$ 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 sort preprocessing, each of the $O(\log_m n)$ sweeps (one per level of recursion) takes $O(n)$ I/Os, yielding the desired bound (8.1).

The resulting algorithm, called scalable sweeping-based spatial join (SSSJ) [47, 48], outperforms other techniques for rectangle intersection. It was tested against two other sweep line algorithms: the partition-based spatial merge (QPBSM) used in Paradise [282] and a faster version called MPBSM that uses an improved dynamic data structure for intervals [47]. The TPIE system described in Chapter 17 served as the common implementation platform. The algorithms were tested on several data sets. The timing results for the two data sets in Figures 8.2(a) and 8.2(b) are given in Figures 8.3(a) and 8.3(b), 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 the 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 [327], 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

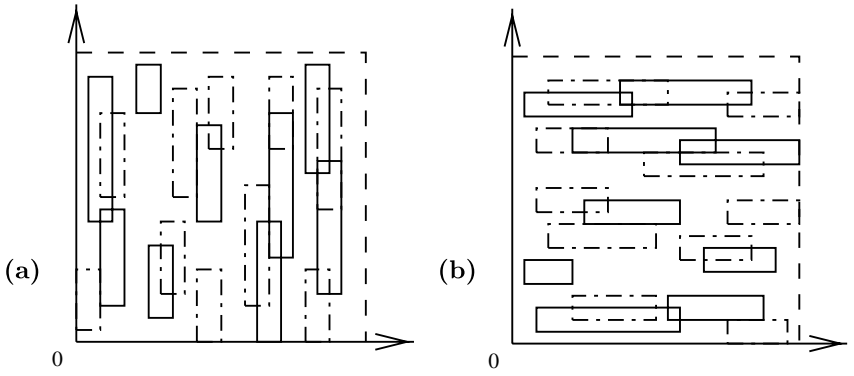

Fig. 8.2 Comparison of Scalable Sweeping-Based Spatial Join (SSSJ) with the original PBSM (QPBSM) and a new variant (MPBSM): (a) Data set 1 consists of tall and skinny (vertically aligned) rectangles; (b) Data set 2 consists of short and wide (horizontally aligned) rectangles.

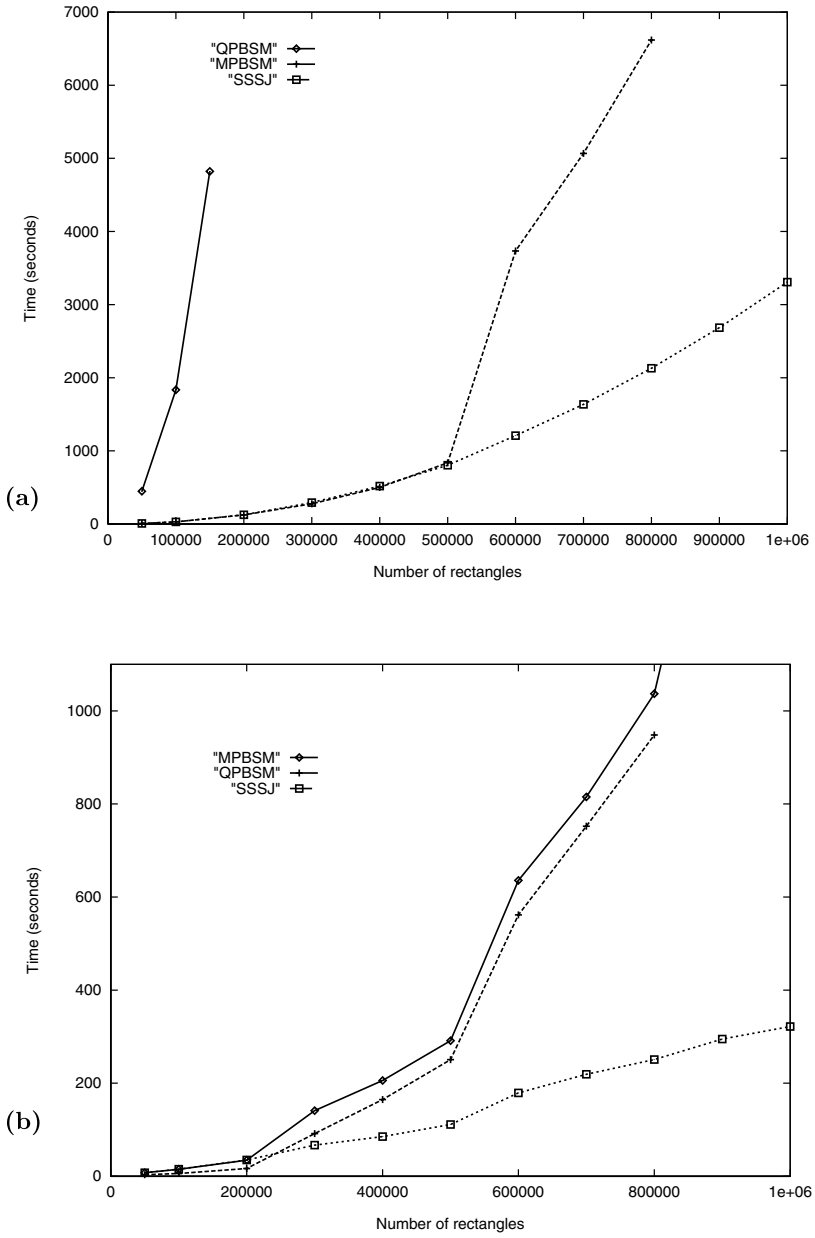

Fig. 8.3 Comparison of Scalable Sweeping-Based Spatial Join (SSSJ) with the original PBSM (QPBSM) and a new variant (MPBSM): (a) Running times on data set 1; (b) Running times on data set 2.

data. If the rectangles are already stored in an index structure, such as the R-tree index structure we consider in Section 12.2, hybrid methods that combine distribution sweep with inorder traversal often perform best [46].

For the problem of finding all intersections among N line segments, Arge et al. [57] give an efficient algorithm based upon distribution sort, but the answer component of the I/O bound is slightly nonoptimal: $z \log_m n$ rather than z. Crauser et al. [122, 123] attain the optimal I/O bound (8.1) 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.

8.2 Other Batched Geometric Problems

Other batched geometric problems studied in the PDM model include range counting queries [240], constrained Delauney triangulation [14], and a host of problems on terrains and grid-based GIS models [7, 10, 16, 35, 36, 54, 192]. Breimann and Vahrenhold [89] survey several EM problems in computational geometry.

9

Batched Problems on Graphs

The problem instance for graph problems includes an encoding of the graph in question. We adopt the convention that the edges of the graph, each of the form (u, v) for some vertices u and v, are given in list form in arbitrary order. We denote the number of vertices by V and the number of edges by E. For simplicity of notation, we assume that $E \geq V$; in those cases where there are fewer edges than vertices, we set E to be V. We also adopt the lower case notation $v = V/B$ and $e = E/B$ to denote the number of blocks of vertices and edges. We can convert the graph to adjacency list format via a sort operation in $Sort(E)$ I/Os.

Tables 9.1 and 9.2 give the best known I/O bounds (with appropriate corrections made for errors in the literature) for several graph problems. The problems in Table 9.1 have sorting-like bounds, whereas the problems in Table 9.2 seem to be inherently sequential in that they do not take advantage of block I/O. As mentioned in Chapter 6, the best known I/O lower bound for these problems is $\Omega\big((E/V)Sort(V) = e \log_m v\big)$.

The first work on EM graph algorithms was by Ullman and Yannakakis [331] for the problem of transitive closure. Chiang et al. [105] consider a variety of graph problems, several of which have upper

Table 9.1 Best known I/O bounds for batched graph problems with sorting-like bounds. We show only the single-disk case $D = 1$. The number of vertices is denoted by $V = vB$ and the number of edges by $E = eB$; for simplicity, we assume that $E \geq V$. The term $Sort(N)$ is the I/O bound for sorting defined in Chapter 3. Lower bounds are discussed in Chapter 6.

Graph Problem	I/O Bound, $D = 1$	
List ranking, Euler tour of a tree, Centroid decomposition, Expression tree evaluation	$\Theta\big(Sort(V)\big)$	[105]
Connected components, Biconnected components, Minimum spanning forest (MSF), Bottleneck MSF, Ear decomposition	$O\left(\min\left\{Sort(V^2),\ \max\left\{1, \log\frac{V}{M}\right\}\frac{E}{V}Sort(V),\ \max\left\{1, \log\log\frac{V}{e}\right\}Sort(E),\ (\log\log B)\frac{E}{V}Sort(V)\right\}\right)$ (deterministic)	[2, 34, 105, 149, 227, 264]
	$\Theta\left(\frac{E}{V}Sort(V)\right)$ (randomized)	[105, 149]
Maximal independent sets, Triangle enumeration	$\Theta\big(Sort(E)\big)$	[362]
Maximal matching	$O\big(Sort(E)\big)$ (deterministic)	[362]
	$\Theta\left(\frac{E}{V}Sort(V)\right)$ (randomized)	[105]

and lower I/O bounds related to sorting and permuting. Abello et al. [2] 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 [227], followed by Buchsbaum et al. [93], develop EM graph algorithms based upon amortized data structures for binary heaps and tournament trees. Munagala and Ranade [264] give improved graph algorithms for connectivity and undirected breadth-first search (BFS). Their approach is extended by Arge et al. [34] to compute the minimum spanning forest

Table 9.2 Best known I/O bounds for batched graph problems that appear to be substantially harder than sorting, for the single-disk case $D = 1$. The number of vertices is denoted by $V = vB$ and the number of edges by $E = eB$; for simplicity, we assume that $E \geq V$. The term $Sort(N)$ is the I/O bound for sorting defined in Chapter 3. The terms $SF(V, E)$ and $MSF(V, E)$ represent the I/O bounds for finding a spanning forest and minimum spanning forest, respectively. We use w and W to denote the minimum and maximum weights in a weighted graph. Lower bounds are discussed in Chapter 6.

Graph Problem	I/O Bound, $D = 1$	
Undirected breadth-first search	$O\big(\sqrt{Ve} + Sort(E) + SF(V,E)\big)$	[252]
Undirected single-source shortest paths	$O\left(\min\left\{V + e\log V,\ \sqrt{Ve}\log V + MSF(V,E),\ \sqrt{Ve\log\left(1+\frac{W}{w}\right)} + MSF(V,E)\right\}\right)$	[227, 258, 259]
Directed and undirected depth-first search, Topological sorting, Directed breadth-first search, Directed single-source shortest paths	$O\left(\min\left\{V + Sort(E) + \frac{ve}{m},\ (V+e)\log V\right\}\right)$	[93, 105, 227]
Transitive closure	$O\left(Vv\sqrt{\frac{e}{m}}\right)$	[105]
Undirected all-pairs shortest paths	$O\big(V\sqrt{Ve} + Ve\log e\big)$	[108]
Diameter, Undirected unweighted all-pairs shortest paths	$O\big(VSort(E)\big)$	[43, 108]

(MSF) and by Mehlhorn and Meyer [252] for BFS. Ajwani et al. [27] give improved practical implementations of EM algorithms for BFS. Meyer [255] gives an alternate EM algorithm for undirected BFS for sparse graphs, including the dynamic case. Meyer [256] provides new results on approximating the diameter of a graph. Dementiev et al. [137] implement practical EM algorithms for MSF, and Dementiev [133] gives practical EM implementations for approximate graph coloring, maximal independent set, and triangle enumeration. Khuller et al. [215] present approximation algorithms for data migration, a problem related to coloring that involves converting one layout of blocks to another.

Arge [30] gives efficient algorithms for constructing ordered binary decision diagrams.

Techniques for storing graphs on disks for efficient traversal and shortest path queries are discussed in [10, 53, 177, 201, 271]. Computing wavelet decompositions and histograms [347, 348, 350] is an EM graph problem related to transposition that arises in online analytical processing (OLAP). Wang et al. [349] give an I/O-efficient algorithm for constructing classification trees for data mining. Further surveys of EM graph algorithms appear in [213, 243].

9.1 Sparsification

We can often apply sparsification [149] to convert I/O bounds of the form $O\big(Sort(E)\big)$ to the improved form $O\big((E/V)Sort(V)\big)$. For example, the I/O bound for minimum spanning forest (MSF) actually derived by Arge et al. [34] is $O\big(\max\{1, \log\log(V/e)\}Sort(E)\big)$. For the MSF problem, we can partition the edges of the graph into E/V sparse subgraphs, each with V edges on the V vertices, and then apply the algorithm of [34] to each subproblem to create E/V spanning forests in a total of $O\big(\max\{1, \log\log(V/v)\}(E/V)Sort(V)\big) = O\big((\log\log B)(E/V)Sort(V)\big)$ I/Os. We then merge the E/V spanning forests, two at a time, in a balanced binary merging procedure by repeatedly applying the algorithm of [34]. 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 I/O cost for the final spanning forest is thus $O\big(\max\{1, \log\log B\}(E/V)Sort(V)\big)$ I/Os.

The reason why sparsification works is that the spanning forest created by each binary merge is only $\Theta(V)$ in size, yet it preserves the necessary information needed for the next merge step. That is, the MSF of the merge of two graphs G and G' is the MSF of the merge of the MSFs of G and G'.

The same sparsification approach can be applied to connectivity, biconnectivity, and maximal matching. For example, to apply sparsification to finding biconnected components, we modify the merging process by first replacing each biconnected component by a cycle

that contains the vertices in the biconnected component. The resulting graph has $O(V)$ size and contains the necessary information for computing the biconnected components of the merged graph.

9.2 Special Cases

In the case of *semi-external graph problems* [2], in which the vertices fit into internal memory but the edges do not (i.e., $V \leq M < E$), several of the problems in Table 9.1 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 latter problems is $\Theta\big((E/V)Sort(V) + V\big)$. 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 $O\big(Sort(E)\big)$ I/Os [55, 105, 241, 242, 244, 329]. Other EM algorithms for planar, near-planar, and bounded-degree graphs appear in [10, 44, 51, 52, 59, 191, 254].

9.3 Sequential Simulation of Parallel Algorithms

Chiang et al. [105] 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 [105] 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 [32] discussed in Section 11.4. A practical randomized implementation of list ranking appears in [317].

Dehne et al. [131, 132] and Sibeyn and Kaufmann [319] 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 [105], 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 $O\big((\log N)^c\big)$ 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.

10

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 chapter, and tree-based approaches, which we defer until Chapter 11 and succeeding chapters.

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 pre-defined hash function

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

that assigns the N items to K address locations in a uniform manner. Hashing algorithms differ from each another 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 $O\big(Output(Z)\big) = O\big(\lceil z \rceil\big)$ I/Os per lookup (where $Z = zB$ is the number of items in the answer), $O(1)$ 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.

10.1 Extendible Hashing

EM dynamic hashing methods fall into one of two categories: *directory* methods and *directoryless* methods. Fagin et al. [153] 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 10.1.

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.

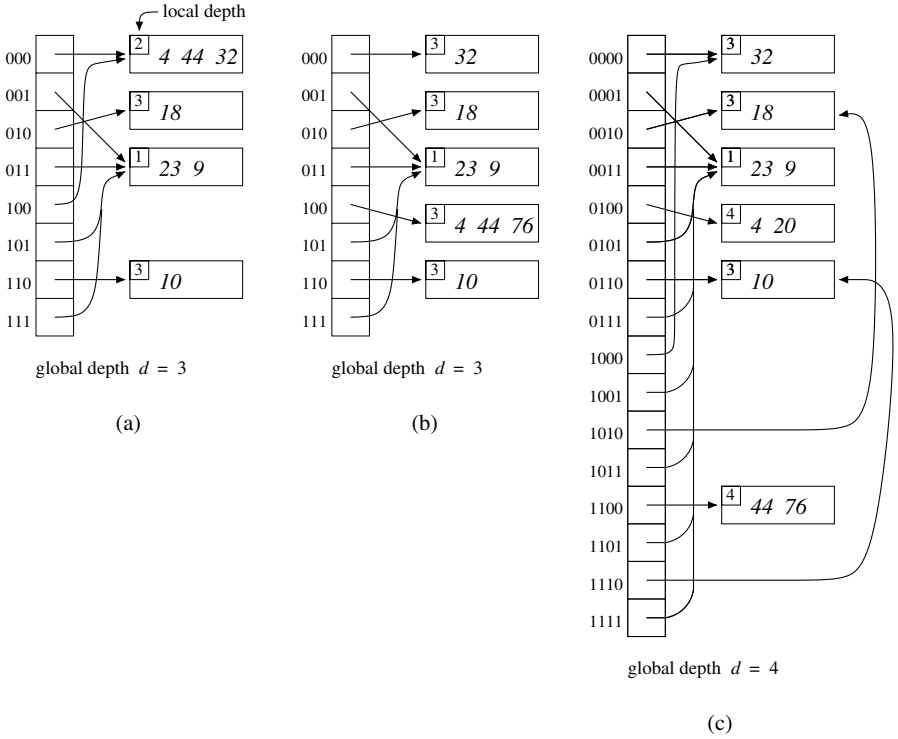

Fig. 10.1 Extendible hashing with block size $B = 3$. The keys are indicated in italics; the hash address of a key consists of its binary representation. For example, the hash address of key *4* is "...000100" and the hash address of key *44* is "...0101100". (a) The hash table after insertion of the 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.

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 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 10.1(b). Otherwise, the directory is not 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 10.1(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 together 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 [253]. At least $\Omega(n/B)$ blocks are needed to store the directory. Flajolet [164] showed on the average that the directory uses $\Theta(N^{1/B}n/B) = \Theta(N^{1+1/B}/B^2)$ blocks, which can be superlinear in N asymptotically! However, for practical values of N and B, the $N^{1/B}$ term is a small constant, typically less than 2, and the directory size is within a constant factor of the optimum.

The resulting directory is equivalent to the leaves of a perfectly balanced trie [220], 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.

10.2 Directoryless Methods

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 [235] and Larson [229] 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, ..., $2^d + p - 1$, for some $0 \leq p < 2^d$. 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. Further analysis appears in [67].

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) [248, 263] combines constrained bucket splitting and overflowing buckets. A more detailed survey of EM dynamic hashing methods appears in [147].

10.3 Additional Perspectives

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 range search (retrieving all the

items with key value in a specified range). Some clever work in support of range search has been done on order-preserving hash functions, in which items with consecutive key values are stored in the same block or in adjacent blocks. However, the search performance is less robust and tends to deteriorate because of unwanted collisions. See [170] for a survey of multidimensional hashing methods and in addition more recent work in [85, 203]. In the next two chapters, we explore a more natural approach for range search with good worst-case performance using multiway trees.

11

Multiway Tree Data Structures

In this chapter, we explore some important search-tree data structures in external memory. 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.

11.1 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. One primary application is search. Some binary search trees, for example, support lookup queries for a dictionary of N items in $O(\log N)$ time, which is optimal in the comparison model of computation. The generalization to external memory, in which data are transferred in blocks of B items and B comparisons can be done per I/O, provides an $\Omega(\log_B N)$ I/O lower bound. These lower bounds depend heavily

upon the model; we saw in the last chapter that hashing can support dictionary lookup in a constant number of I/Os.

In order to exploit block transfer, trees in external memory generally represent each node by a block that can store $\Theta(B)$ pointers and data values and can thus achieve $\Theta(B)$-way branching. The well-known balanced multiway *B-tree* due to Bayer and McCreight [74, 114, 220], 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 using $O(\log_B N)$ 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 have too many children and 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. Franceschini et al. [167] show how to achieve the same I/O bounds without space for pointers.

In the *B^+-tree* variant, pictured in Figure 11.1, all the items are stored in the leaves, and the leaves are linked together in symmetric order to facilitate range queries and sequential access. The internal nodes store only key values and pointers and thus can have a higher branching factor. In the most popular variant of B^+-trees, called *B^*-trees*, splitting can usually be postponed when a node overflows, by

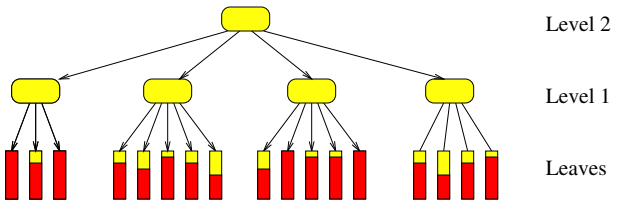

Fig. 11.1 B^+-tree multiway search tree. Each internal and leaf node corresponds to a disk block. All the items are stored in the leaves; the darker portion of each leaf block indicates its relative fullness. The internal nodes store only key values and pointers, $\Theta(B)$ of them per node. Although not indicated here, the leaf blocks are linked together sequentially.

"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 [360] 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 $2\ln(3/2) \approx 81\%$ [63, 228]. 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 [65] and use of overflow nodes [321].

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 [236] and further studied in [64, 237]. O'Neil [275] 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 [290, 291] use similar ideas to exploit locality and optimize search tree performance in the RAM model. 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. [76], Varman and Verma [335], and Arge et al. [37]. By persistent data structure, we mean that searches can be done with respect to any time stamp y [142, 143]. 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 time stamp y affects all future queries for any time after y. Batched versions of partially persistent B-trees provide an elegant solution to problems of point location and visibility discussed in Chapter 8.

An interesting open problem is whether B-trees can be made fully persistent. Salzberg and Tsotras [298] survey work done on persistent access methods and other techniques for time-evolving data. Lehman and Yao [231], Mohan [260], Lomet and Salzberg [239], and Bender

et al. [81] explore mechanisms to add concurrency and recovery to B-trees.

11.2 Weight-Balanced B-trees

Arge and Vitter [58] 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., 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 the nodes have secondary structures, as in multidimensional search trees, or when rebuilding is expensive. Agarwal et al. [11] apply weight-balanced B-trees to convert partition trees such as kd-trees, BBD trees, and BAR trees, which were designed for internal memory, into efficient EM data structures.

Weight-balanced trees called BB[α]-trees [87, 269] 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 12.3–12.5. They also have applications in the (internal memory) RAM model [58, 187], where they offer a simpler alternative to BB[α]-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 [144, 145] in the RAM model. Weight-balanced B-trees are also

preferable to BB[α]-trees for purposes of augmenting one-dimensional data structures with range restriction capabilities [354].

11.3 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 [6]. 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. [6] describe an interesting variant of B-trees called *level-balanced B-trees* for handling parent pointers and operations such as 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 $O\big(N_h + \mathit{Sort}(bN_h) + \mathit{Scan}(bN_h)\big)$, which can be amortized against the $\Theta(N_h)$ updates that have occurred since the last time the level-h invariant was violated, yielding an amortized update cost of $O\big(1 + (b/B)\log_m n\big)$ 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 $O\big(\big(1 + (b/B)\log_m n\big)\log_b N\big)$ I/Os amortized, for any $2 \le b \le B/2$, which is never worse than $O\big((\log_B N)^2\big)$ by appropriate choice of b.

Using the multislab decomposition we discuss in Section 12.3, Agarwal et al. [6] apply level-balanced B-trees in a data structure for point location in monotone subdivisions, which supports queries and (amortized) updates in $O\big((\log_B N)^2\big)$ I/Os. They also use it to dynamically maintain planar st-graphs using $O\big(1 + (b/B)(\log_m n)\log_b N\big)$ I/Os (amortized) per update, so that reachability queries can be answered in $O(\log_B N)$ I/Os (worst-case). (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.

11.4 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. In Section 8.1, 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 $Sort(N)$ bound (5.1) by a substantial factor of roughly B.

One solution suggested in [339] 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 [32] 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 $O\big((1/B)\log_m n\big)$ 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 [82] 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 8.1 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 [105]. Govindarajan et al. [182] use time-forward processing to construct a well-separated pair decomposition of N points in d dimensions in $O\big(Sort(N)\big)$ 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 [92] 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 [90, 302]. Brodal and Fagerberg [91] examine I/O tradeoffs

between update and search for comparison-based EM dictionaries. Matching upper bounds for several cases can be achieved with a truncated version of the buffer tree.

In Section 12.2, we report on some timing experiments involving buffer trees for use in bulk loading of R-trees. Further experiments on buffer trees appear in [200].

12

Spatial Data Structures and Range Search

In this chapter, we consider online tree-based 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. We shall 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, which we discuss briefly in Section 12.6.

There are two types of spatial data structures: data-driven and space-driven. R-trees and kd-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 chapter, we focus primarily on 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 [210] for background). We have already discussed multidimensional range searching in a batched

setting in Chapter 8. In this chapter, 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 answer 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 $\mathit{Search}(N) = O(\log_B N)$ of the search component of queries with the I/O cost $\mathit{Output}(Z) = O\big(\lceil z \rceil\big)$ for reporting the Z items in the answer to the query. 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 [98], in which both the search component and the answer reporting are allowed to use the larger number of I/Os. For example, to do queries optimally when $\mathit{Output}(Z)$ is large with respect to $\mathit{Search}(N)$, the search component can afford to be somewhat sloppy as long as it does not use more than $O(z)$ I/Os, and when $\mathit{Output}(Z)$ is relatively small, the Z items in the answer do not need to reside compactly in only $O\big(\lceil z \rceil\big)$ blocks. Filtering is an important design paradigm for many of the algorithms we discuss in this chapter.

We find in Section 12.7 that under a fairly general computational model for general 2-D orthogonal queries, as pictured in Figure 12.1(d), it is impossible to satisfy Criteria 1 and 2 simultaneously. At least $\Omega\big(n(\log n)/\log(\log_B N + 1)\big)$ blocks of disk space must be used to achieve a query bound of $O\big((\log_B N)^c + z\big)$ I/Os per query, for any constant c [323]. Three natural questions arise:

- What sort of performance can be achieved when using only a linear amount of disk space? In Sections 12.1 and 12.2,

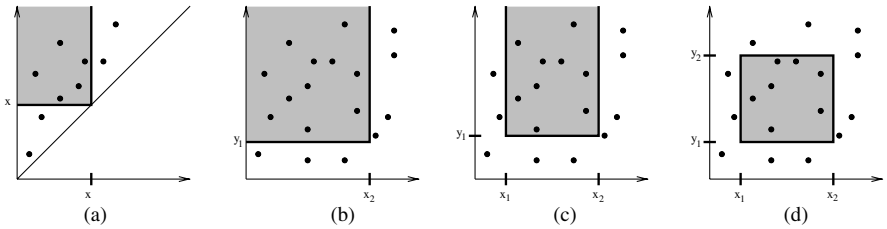

Fig. 12.1 Different types of 2-D orthogonal range queries: (a) Diagonal corner two-sided 2-D query (equivalent to a stabbing query, cf. Section 12.3); (b) Two-sided 2-D query; (c) Three-sided 2-D query; (d) General four-sided 2-D query.

we discuss some of the 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 12.2 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 1–3 for the important special cases of 2-D range searching pictured in Figures 12.1(a), 12.1(b), and 12.1(c)? Fortunately the answer is yes. We show in Sections 12.3 and 12.4 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 $O\big(n(\log n)/\log(\log_B N + 1)\big)$ blocks? Yes again! In Section 12.5, 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 12.6, 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 12.7.

12.1 Linear-Space Spatial Structures

Grossi and Italiano [186] 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 11.2) at the upper levels of the tree with the space-driven partitioning of methods such as quad trees at the lower levels of the tree. Cross trees can be used to construct dynamic EM algorithms for MSF and 2-D priority queues (in which the *delete_min* operation is replaced by $delete\_min_x$ and $delete\_min_y$). 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 [211] 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 kd-trees, with block-access capabilities. Examples include *kd-B-trees* [293], *buddy trees* [309], *hB-trees* [151, 238], and *Bkd-trees* [285]. *Grid files* [196, 268, 353] 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 [173, 207, 277]. Linearization can also be used to represent nonpoint data, in which the data items are partitioned into one or more multidimensional rectangular regions [1, 276]. 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 [18, 170, 270] for a broad survey of these and other interesting methods. Space-filling curves arise again in connection with R-trees, which we describe next.

12.2 R-trees

The *R-tree* of Guttman [190] 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 [18, 170, 183] for a survey. The *R*-tree* variant of Beckmann et al. [77] 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 furthest 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 12.1. New R-tree partitioning methods by de Berg et al. [128], Agarwal et al. [17], and Arge et al. [38] 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 [1, 206, 276]. 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 [206], each item is labeled with the position of its centroid on the Peano-Hilbert space-filling curve, and a B^+-tree 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 methods 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 [206] was presented in [207]. 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 [84, 208].

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. [41] and van den Bercken et al. [333] 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

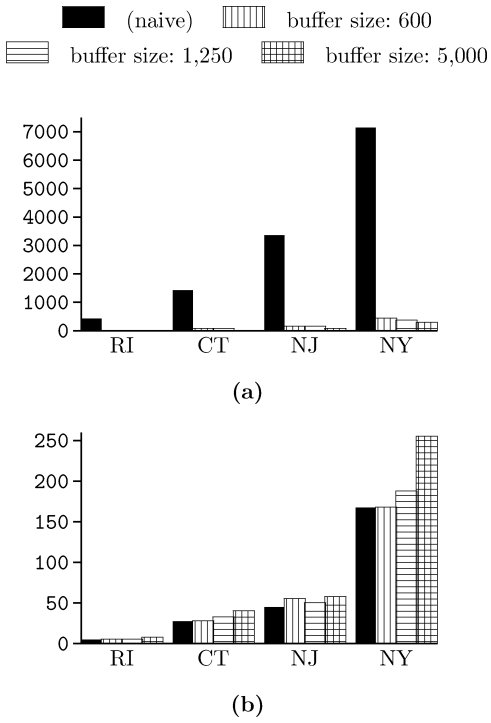

Fig. 12.2 Costs for R-tree processing (in units of 1000 I/Os) using the naive repeated insertion method and the buffer R-tree for various buffer sizes: (a) Cost for bulk-loading the R-tree; (b) Query cost.

the bottom-up methods within a constant factor. The former method is especially efficient and supports dynamic batched updates and queries. In Figure 12.2 and Table 12.1, 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 [327]; the implementations were done using the TPIE system, described in Chapter 17.

Figure 12.2 compares the construction cost for building R-trees and the resulting query performance in terms of I/Os for the naive sequential method for constructing R*-trees (labeled "naive") and the newly developed buffer R*-tree method [41] (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,

Table 12.1 Summary of the costs (in number of I/Os) for R-tree updates and queries. Packing refers to the percentage storage utilization.

		Update with 50% of the data		
Data set	Update method	Building	Querying	Packing (%)
	Naive	259,263	6,670	64
RI	Hilbert	15,865	7,262	92
	Buffer	13,484	5,485	90
	Naive	805,749	40,910	66
CT	Hilbert	51,086	40,593	92
	Buffer	42,774	37,798	90
	Naive	1,777,570	70,830	66
NJ	Hilbert	120,034	69,798	92
	Buffer	101,017	65,898	91
	Naive	3,736,601	224,039	66
NY	Hilbert	246,466	230,990	92
	Buffer	206,921	227,559	90

600, 1,250, and 5,000 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 12.2, 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. [208] develop a way to do bulk updates with Hilbert R-trees, but at a cost in terms of query performance. Table 12.1 compares dynamic update methods for the naive method, for buffer R-trees, and for Hilbert R-trees [208] (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 12.1 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.

12.3 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 all the items in the answer to the query 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 answer, and as a result each subtree may need to be explored (costing several I/Os) just to report a single item of the answer.

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 $O(\log_B B^2 + Z'/B) = O\big(\lceil Z'/B \rceil\big)$ 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 and make it worst-case using the weight-balanced B-trees of Section 11.2 as the underlying B-tree structure.

Arge and Vitter [58] first uncovered the bootstrapping paradigm while designing an optimal dynamic EM data structure for diagonal corner two-sided 2-D queries (see Figure 12.1(a)) that meets all three design criteria listed in Chapter 12. Diagonal corner two-sided

queries are equivalent to stabbing queries, which have the following 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 [144, 145] for internal memory, which supports stabbing queries in $O(\log N + Z)$ CPU time and updates 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 8.1 for a similar batched problem. Each interval in the problem instance 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 12.3. The three pieces

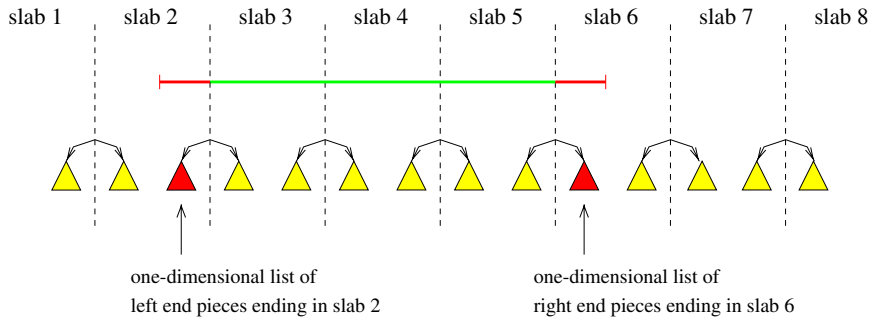

Fig. 12.3 Internal node v of the EM priority search tree, for $B = 64$ with $\sqrt{B} = 8$ slabs. Node v is the lowest node in the tree completely containing the indicated interval. The middle piece of the interval is stored in the multislab list corresponding to slabs 3–5. (The multislab lists are not 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.

are stored in substructures of v. In the example in Figure 12.3, the middle piece is stored in a list associated with the multislab it spans (corresponding to the contiguous range of slabs 3–5), 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 list and left-ordered list for the slab of v that contains x, reporting intervals in an output-sensitive way.

The big problem with this approach is that we have to 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. [210]. 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 nonsparse multislab lists, each of which contributes $Z' \geq B$ intervals to the answer, 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 $O(\log_B B^2 + Z''/B) = O\big(\lceil Z''/B \rceil\big)$ 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.

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.

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 [106] 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 [10]. Range-max and stabbing-max queries are studied in [13, 15].

12.4 Bootstrapping for Three-Sided Orthogonal 2-D Range Search

Arge et al. [50] 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 12.1(c)) that meets all three design criteria. In internal memory, the optimal structure is the priority search tree [251], 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. [50] 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 12.4.) 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 12.4(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. [50]

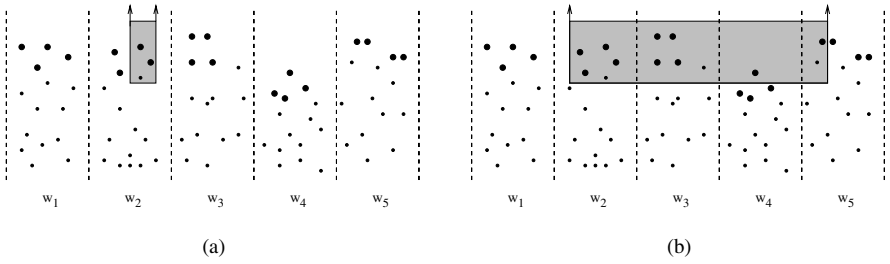

Fig. 12.4 Internal node v of the EM priority search tree, with slabs (children) w_1, w_2, ..., w_5. The Y-sets of each child, which are stored collectively in v's child cache, are indicated by the bold points. (a) The three-sided query is completely contained in the x-range of w_2. The relevant (bold) 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.

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 $O(B)$ blocks, can be built in $O(B)$ I/Os, and supports three-sided queries in $O\big(\lceil Z'/B \rceil\big)$ 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. [50] and the structure for two-sided diagonal queries discussed in Section 12.3 satisfy Criteria 1–3 of Chapter 12. 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 structure 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.

On a historical note, earlier work on two-sided and three-sided queries was done by Ramaswamy and Subramanian [289] 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 [323] 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.

12.5 General Orthogonal 2-D Range Search

The dynamic data structure for three-sided range searching can be generalized using the filtering technique of Chazelle [98] to handle general four-sided queries with optimal I/O query bound $O(\log_B N + z)$ and optimal disk space usage $O\big(n(\log n)/\log(\log_B N + 1)\big)$ [50]. The update bound becomes $O\big((\log_B N)(\log n)/\log(\log_B N + 1)\big)$, 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 point of the problem instance 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 for three-sided queries of these types; using the linear-sized data structures of Arge et al. [50] in Section 12.4, 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 $O\big(\lceil Z'/B \rceil\big)$ 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 [58], as described in Section 12.3. (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 that we do not need.) The total number of I/Os to answer the range query is thus $O(\log_B N + z)$, which is optimal.

Atallah and Prabhakar [62] and Bhatia et al. [86] consider the problem of how to tile a multidimensional array of blocks onto parallel disks so that range queries on a range queries can be answered in near-optimal time.

12.6 Other Types of Range Search

Govindarajan et al. [181] develop data structures for 2-D range-count and range-sum queries. For other types of range searching, such as in higher dimensions and for nonorthogonal queries, different filtering techniques are needed. So far, relatively little work has been done, and many open problems remain.

Vengroff and Vitter [336] 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. By using $(B \log n)$-approximate boundaries rather than B-approximate boundaries [340], queries can be done in $O(\log_B N + z)$ I/Os, which is optimal, and the space usage is $O\big(n(\log n)^{k+1}/(\log(\log_B N + 1))^k\big)$ disk blocks to support $(3 + k)$-sided 3-D range queries, in which k of the dimensions $(0 \leq k \leq 3)$ 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 [100], 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. An interesting special case is "fat" orthogonal 2-D range search, where the query rectangles are required to 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). For example, every rectangle with bounded aspect ratio can be covered by a constant number of 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. [9] consider halfspace range searching, in which a query is specified by a hyperplane and a bit indicating one of its two sides, and the answer to 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 Chapter 12.

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.

Cheng et al. [102, 103] develop efficient indexes for range queries and join queries over data with uncertainty, in which each data point has an estimated distribution of possible locations. Chien et al. [107] derive a duality relation that links the number of I/O steps and the space bound for range searching to the corresponding measures for text indexing.

Callahan et al. [94] develop dynamic EM data structures for several online problems in d dimensions. For any fixed $\varepsilon > 0$, they can find an approximately nearest neighbor of a query point (within a $1 + \varepsilon$ factor of optimum) 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. [182] achieve the same bounds for closest pair by maintaining a well-separated pair decomposition. For finding nearest neighbors and approximate nearest neighbors, two other approaches are partition trees [8, 9] and locality-sensitive hashing [176]. Planar point location is studied in [37, 332], and the dual problem of planar

point enclosure is studied in [51]. Numerous other data structures have been developed for range queries and related problems on spatial data. We refer to [18, 31, 170, 270] for a broad survey.

12.7 Lower Bounds for Orthogonal Range Search

We mentioned in the introduction to the chapter that Subramanian and Ramaswamy [323] prove that no EM data structure for 2-D range searching can achieve design Criterion 1 using less than $O\big(n(\log n)/\log(\log_B N + 1)\big)$ disk blocks, even if we relax the criterion to allow $O\big((\log_B N)^c + z\big)$ I/Os per query, for any constant c. The result holds for an EM version of the pointer machine model, based upon the approach of Chazelle [99] for the (internal memory) RAM model.

Hellerstein et al. [193] consider a generalization of the layout-based lower bound argument of Kanellakis et al. [210] for studying the trade-off 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 points in the answer to a query compactly within $O\big(\lceil Z/B \rceil\big) = O\big(\lceil z \rceil\big)$ blocks. One shortcoming of the model is that it considers only data layout and ignores the search component of queries, and thus it rules out the important filtering paradigm discussed earlier in Chapter 12. For example, it is reasonable for any query algorithm to perform at least $\log_B N$ I/Os, so if the answer size Z is at most B, an algorithm may still be able to satisfy Criterion 1 even if the items in the answer are contained within $O(\log_B N)$ blocks rather than $O(z) = O(1)$ blocks. Arge et al. [50] modify the model to rederive the same nonlinear-space lower bound $O\big(n(\log n)/\log(\log_B N + 1)\big)$ of Subramanian and Ramaswamy [323] for 2-D range searching by considering only answer sizes Z larger than $(\log_B N)^c B$, for which the number of blocks allowed to hold the items in the answer is $Z/B = O\big((\log_B N)^c + z\big)$. This approach ignores the complexity of how to find the relevant blocks, but as mentioned in Section 12.5 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 [224, 301].

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. An interesting question is whether R-trees and the linear-space structures of Sections 12.1 and 12.2 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 [211] 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 [186] mentioned in Section 12.1, which in addition supports the operations of cut and concatenate.

13

Dynamic and Kinetic Data Structures

In this chapter, 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.

13.1 Dynamic Methods for Decomposable Search Problems

In Chapters 10–12, we have already encountered several dynamic data structures for the problems of dictionary lookup and range search. In Chapter 12, 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 a 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* [83, 278] 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 sizes 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 kth substructure (i.e., of size 2^k) is present exactly when the kth 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 kth 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 [278].

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 kth substructure must be increased from 2^k to B^k. As the number of items increases from 1 to N, the kth substructure has to be built NB/B^k times (when N is a power of B), each time taking $O\big(B^k(\log_B N)/B\big)$ 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 $O\big(N(\log_B N)^2\big)$ I/Os, which is

$O\big((\log_B N)^2\big)$ 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 [56] obtain I/O bounds for dynamic point location in general planar subdivisions similar to those of [6], 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 [58] described in Section 12.3. 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. [11] apply the logarithmic method (in both the binary form and B-way variant) to get EM versions of kd-trees, BBD trees, and BAR trees.

13.2 Continuously Moving Items

Early work on temporal data generally concentrated on time-series or multiversion data [298]. 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 [283, 297, 356]). 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 [73]. 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. [223] 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 $O(n^{1/2+\varepsilon} + k)$ I/Os, for arbitrarily small $\varepsilon > 0$. Updates require $O\big((\log n)^2\big)$ I/Os. Agarwal et al. [8] extend the approach to handle range searches in two dimensions, and they improve the update bound to $O\big((\log_B n)^2\big)$ I/Os. They also propose an event-driven data structure with the same query times as the range search data structure of Arge and Vitter [50] discussed in Section 12.5, 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. [297] build an R-tree (called the TPR-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 [19] show how to maintain a provably good approximation of the minimum bounding box with need for only a constant number of refinement events. Agarwal et al. [12] develop persistent data structures where query time degrades in proportion to how far the time frame of the query is from the current time. Xia et al. [359] develop change-tolerant versions of R-trees with fast update capabilities in practice.

14

String Processing

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

For the case of strings we make the following modifications to our standard notation:

$$\begin{aligned} K &= \text{number of strings;} \\ N &= \text{total length of all strings (in units of characters);} \\ M &= \text{internal memory size (in units of characters);} \\ B &= \text{block transfer size (in units of characters).} \end{aligned}$$

where $M < N$, and $1 \leq B \leq M/2$. The characters are assumed to come from an alphabet Σ of length $|\Sigma|$; that is, each character is represented in $\log|\Sigma|$ bits.

14.1 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, it makes sense to use a hybrid approach, 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 [222] or Boyer–Moore [88] methods. This particular hybrid method was introduced as the basis of the widely used GLIMPSE search tool [247]. Another way to index text is to use hashing to get small signatures for portions of text. The reader is referred to [66, 166] for more background on the above methods.

14.2 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 [75] investigated the use of prefix representations of keys.

Ferragina and Grossi [157, 158] 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 [275] mentioned in Section 11.1). 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 the SB-tree is represented by a digital tree (or *trie* for short), as pictured in Figure 14.1. The $\Theta(B)$ keys of the SB-tree node form the leaves of the trie. The particular variant of trie is the compressed *Patricia trie* data structure [220, 261], in which all the internal nodes are non-unary branching nodes, and as a result the entire Patricia trie has size $\Theta(B)$ and can fit into a single disk block.

Each of its internal Patricia nodes stores an index (a number from 0 to L, where L is the maximum length of a leaf) and a one-character

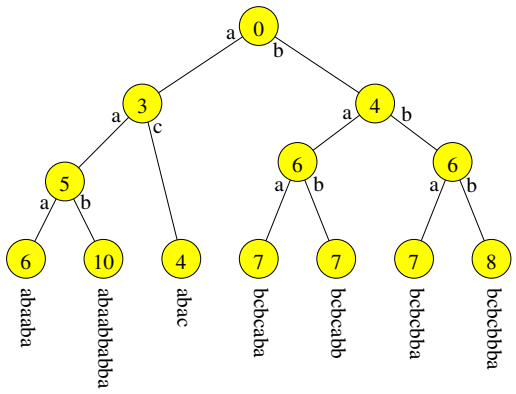

Fig. 14.1 Patricia trie representation of a single node of an SB-tree, with branching factor $B = 8$. The seven strings used for partitioning are pictured at the leaves; in the actual data structure, pointers to the strings, not the strings themselves, are stored at the leaves. The pointers to the B children of the SB-tree node are also stored at the leaves.

label for each of its outgoing edges. Navigation from root to leaf in the Patricia trie is done using the bit representation of the strings. For example, suppose we want to search for the leaf "`abac`." We start at the root, which has index 0; the index indicates that we should examine character 0 of the search string "`abac`" (namely, "`a`"), which leads us to follow the branch labeled "`a`" (left branch). The next node we encounter has index 3, and so we examine character 3 (namely, "`c`"), follow the branch labeled "`c`" (right branch), and arrive at the leaf "`abac`."

Searching for a text string that does not match one of the leaves is more complicated and exploits the full power of the data structure, using an amortization technique of Ajtai et al. [25]. Suppose we want to search for "`bcbabcba`." Starting at the root, with index 0, we examine character 0 of "`bcbabcba`" (namely, "`b`") and follow the branch labeled "`b`" (right branch). We continue searching in this manner, which leads along the rightmost path, examining indexes 4 and 6, and eventually we arrive at the far-right leaf "`bcbcbbba`." However, the search string "`bcbabcba`" does not match the leaf string "`bcbcbbba`." The problem is that they differ at index 3, but only indexes 0, 4, and 6 were examined in the traversal, and thus the difference was not detected.

In order to determine efficiently whether or not there is a match, we go back and sequentially compare the characters of the search string

with those of the leaf, and if they differ, we find the first index where they differ. In the example, the search string "bcbabcba" differs from "bcbcbbba" in index 3; that is, character 3 of the search string ("a") comes before character 3 of the leaf string ("c"). Index 3 corresponds to a location in the Patricia trie between the root and its right child, 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 the sequential scan does not have to be read again for lower levels of the SB-tree, so the I/Os for the sequential scan can be "charged" to the blocks of the search string. The resulting query time to search in an SB-tree for a string of ℓ characters is therefore $O(\log_B N + \ell/B)$, which is optimal. Insertions and deletions can be done in the same I/O bound. The SB-tree uses linear (optimal) space.

Bender et al. [80] show that cache-oblivious SB-tree data structures are competitive with those developed with explicit knowledge of the parameters in the PDM model.

Ciriani et al. [109] construct a randomized EM data structure that exhibits static optimality, in a similar way as splay trees do in the (internal memory) RAM model. In particular, they show that Q search queries on a set of K strings $s_1, s_2, \ldots, s_K$ of total length N can be done in

$$O\left(\frac{N}{B} + \sum_{1 \le i \le K} f_i \log_B \frac{Q}{f_i}\right) \tag{14.1}$$

I/Os, where f_i is the number of times s_i is queried. Insertion or deletion of a string can be done in the same bounds as given for SB-trees.

Ferragina and Grossi [157, 158] apply SB-trees to the problems of string matching, prefix search, and substring search. Ferragina and Luccio [161] apply SB-trees to get new results for dynamic dictionary matching; their structure even provides a simpler approach for the (internal memory) RAM model. Hon et al. [197] use SB-trees to externalize approximate string indexes. Eltabakh et al. [146] use SB-trees

and three-sided structures to develop a dynamic data structure that indexes strings compressed by run-length encoding, discussed further in Section 15.2.

14.3 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* [250, 352], 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. Ghanem et al. [174] use buffer techniques to index suffix trees and unbalanced search trees. Clark and Munro [110] give a practical implementation of dynamic suffix trees that use about five bytes per indexed suffix.

A more compact (but static) representation of a suffix tree, called a *suffix array* [246], consisting of the leaves of the suffix tree in symmetric traversal order, can also be used for fast searching (see [189] for general background). Farach-Colton et al. [154] show how to construct SB-trees, suffix trees, and suffix arrays on strings of total length N characters using $O(n \log_m n)$ I/Os, which is optimal. Crauser and Ferragina [121] and Dementiev et al. [134] present an extensive set of experiments on various text collections that compare the practical performance of suffix array construction algorithms. Sinha et al. [320] give a practical implementation of suffix arrays with fast query performance. Puglisi et al. [286] study relative merits of suffix arrays and inverted files.

Ferragina et al. [160] give algorithms for two-dimensional indexing. Kärkkäinen and Rao [212] survey several aspects of EM text indexing. Chien et al. [107] demonstrate a duality between text indexing and range searching and use it to derive improved EM algorithms and stronger lower bounds for text indexing.

14.4 Sorting Strings

Arge et al. [39] 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 14.1 ([39]). The number of I/Os needed to sort K strings of total length N characters, 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

$$O\left(\min\left\{\frac{N_1}{B}\log_m\left(\frac{N_1}{B}+1\right), K_1 \log_M(K_1+1)\right\} + K_2 \log_M(K_2+1) + \frac{N}{B}\right). \tag{14.2}$$

Further work appears in [152]. Lower bounds for various models of how strings can be manipulated are given in [39]. There are gaps in some cases between the upper and lower bounds for sorting.

15

Compressed Data Structures

The focus in the previous chapters has been to develop external memory algorithms and data structures as a means of dealing with massive amounts of data. In particular, the goal has been to exploit locality and parallelism in order to reduce the I/O communication bottleneck.

Another approach to handle massive data is to compress the data. *Compressed data structures* reduce the amount of space needed to store the data, and therefore there is less to process. Moreover, if the reduced footprint of the data allows the data to be located in a faster level of the memory hierarchy, the latency to access the data is improved as well.

The challenge is to design clever mechanisms that allow the compressed data to be processed directly and efficiently rather than require costly decompress and recompress operations. The ultimate goal is to develop data structures that require significantly less space than uncompressed versions and perform operations just as fast.

The area of compressed data structures has largely been investigated in the (internal memory) RAM model. However, if we consider the always expanding sizes of datasets that we want to process, locality of reference is very important. The study of compressed data structures in external memory is thus likely to increase in importance.

15.1 Data Representations and Compression Models

Sometimes the standard data structures to solve a problem are asymptotically larger than the size of the problem instance. Examples include the suffix tree and suffix array data structures for text indexing, which require $\Theta(N)$ pointers to process a text of N characters; pointer size grows logarithmically with N, whereas characters may have constant size. We can achieve substantial reduction in size by using an approach that avoids so many pointers. Some authors refer to such data structures as "succinct."

While succinct data structures are in a sense compressed, we can often do much better and construct data structures that are *sublinear* in the problem size. The minimum space required for a data structure should relate in some sense to the entropy of the underlying data. If the underlying data are totally random, then it is generally impossible to discern a meaningful way to represent the data in compressed form. Said differently, the level of compression in the data structure should be dependent upon the statistics of the data in the problem instance.

In this chapter, we consider two generic data types:

- *Set data*, where the data consist of a subset S containing N items from a universe U of size $|U|$.
- *Text data*, consisting of an ordered sequence of N characters of text from a finite alphabet Σ. Sometimes the N characters are grouped into a set S of K variable-length subsequences, which we call *strings*; the K strings form a set and are thus unordered with respect to one another, but within each string the characters of the string are ordered.

There does not seem to be one single unifying model that captures all the desired compression properties, so we consider various approaches.

15.1.1 Compression Models for Set Data Representations

There are two standard representations for a set $S = \{s_1, s_2, \ldots, s_N\}$ of N items from a universe U:

- (*String representation*) Represent each s_i in binary format by a string of $\lceil \log |U| \rceil$ bits.
- (*Bitvector representation*) Create a bitvector of length $|U|$ bits, indicating each s_i by placing a 1 in the s_ith position of the bitvector, with all other positions set to 0.

One *information-theoretic model* for representing the set S is simply to count the space required to indicate which subset of the universe U is S. For N items, there are $\binom{|U|}{N}$ possible subsets, thus requiring $\log \binom{|U|}{N} \approx N \log\bigl(|U|/N\bigr)$ bits to represent S. When S is sparse (i.e., when $N \ll |U|$), this encoding is substantially shorter than the $|U|$ bits of the bitvector representation.

Another way to represent the set S is the *gap model*, often used in inverted indexes to store an increasing sequence of document numbers or positions within a file (see [355]). Let us suppose for convenience that the items $s_1, s_2, \ldots, s_N$ of the set S are in sorted order; that is, $i < j$ implies $s_i < s_j$. The ith gap $g_i = s_i - s_{i-1}$ denotes the distance between two consecutive items from S. We can represent the set S by the value N and the stream $G = \langle g_1, \ldots, g_N \rangle$, where $g_1 = s_1$. We then have to encode these values in some fashion. We can store them naively using roughly

$$gap = \sum_{i=1}^{N} \log(g_i + 1) \approx \sum_{i=1}^{N} \log g_i, \tag{15.1}$$

bits, not counting the encoding of N and some extra overhead. If the gaps are all roughly $|U|/N$ in size, then after summing up we require about $N \log\bigl(|U|/N\bigr)$ bits. We can achieve the same space with the information-theoretic model as well. However, if the gap values are at all skewed, the space will be even less than that of the information-theoretic bound. No matter what the distribution of gaps, this model requires at least N bits. The naive gap method is not very useful as an actual encoding method, since it takes $O(i)$ addition operations to reconstruct s_i from the gaps, but it gives an idea of the possible space savings.

Another popular compression model for S, in the general class of front-coding schemes [355], is the *prefix omission method* (POM) [219].

We consider each of the N items in S as a string of up to $\log|U|$ bits. Let us assume that the items $s_1, s_2, \ldots, s_N$ are in sorted order lexicographically. We represent each item with respect to its preceding item. We encode each item by two components: the first part indicates the number of bits at the trailing end of s_i that are different from s_{i-1}, and the second part is the trailing end r. For instance, if $s_{i-1} = 00101001$ and $s_i = 00110010$, then s_i is represented as $(5, 10010)$. Alternatively, we can represent S as the leaves of a trie (digital tree), and we can encode it by the cardinal tree encoding of the trie along with the labels on the arcs. Further improvement is possible by collapsing nodes of outdegree 1 to obtain a Patricia trie of $N-1$ internal nodes and N leaves.

15.1.2 Compression Models for Text Data Representations

We represent a text as an ordered sequence of N characters from an alphabet Σ, which in uncompressed format requires $N\log|\Sigma|$ bits. The alphabet size for ascii format is $|\Sigma| = 256$, so each character requires $\log 256 = 8$ bits in uncompressed form. For DNA sequences, we have $|\Sigma| = 4$, and hence $\log 4 = 2$ bits per character suffice. In practice, English text is often compressible by a factor of 3 or 4. A text T is compressible if each of the N characters in the text can be represented, on average, in fewer than $\log|\Sigma|$ bits.

We can measure the randomness (or the *entropy*) of a text T by the *0th-order empirical entropy* $H_0(T)$, given by

$$H_0(T) = \sum_{y\in\Sigma} \text{Prob}[y] \cdot \log \frac{1}{\text{Prob}[y]}, \tag{15.2}$$

where the sum ranges over the characters of the alphabet Σ. In other words, an efficient coding would encode each character of the text based upon its frequency, rather than simply using $\log|\Sigma|$ bits.

Consider the example of standard English text. The letter "e" appears roughly 13% of the time, as opposed to the letter "q," which is over 100 times less frequent. The 0th-order entropy measure would encode each "e" in about $\log(1/13\%) \approx 3$ bits, far fewer than $\log|\Sigma|$ bits. The letter "q" would be encoded in more than the $\log|\Sigma|$ bits

from the naive representation, but this over-costing is more than made up for by the savings from the encodings of the many instances of "e."

Higher-order empirical entropy captures the dependence of characters upon their *context*, made up of the h previous characters in the text. For a given text T and $h > 0$, we define the *hth-order empirical entropy* $H_h(T)$ as

$$H_h(T) = \sum_{x \in \Sigma^h} \sum_{y \in \Sigma} \text{Prob}[y,x] \cdot \log \frac{1}{\text{Prob}[y \mid x]}, \tag{15.3}$$

where $\text{Prob}[y,x]$ represents the empirical joint probability that a random sequence of $h+1$ characters from the text consists of the h-character sequence x followed by the character y, and $\text{Prob}[y \mid x]$ represents the empirical conditional probability that the character y occurs next, given that the preceding h characters are x.

The expression (15.3) is similar to expression (15.2) for the 0th-order entropy, except that we partition the probability space in (15.3) according to context, to capture statistically significant patterns from the text. We always have $H_h(T) \leq H_0(T)$. To continue the English text example, with context length $h = 1$, the letter "h" would be encoded in very few bits for context "t," since "h" often follows "t." On the other hand, "h" would be encoded in a large number of bits in context "b," since "h" rarely follows a "b."

15.2 External Memory Compressed Data Structures

Some of the basic primitives we might want to perform on a set S of N items from universe U are the following dictionary operations:

- *Member*(P): determine whether P occurs in S;
- *Rank*(P): count the number of items s in S such that $s \leq P$;
- *Select*(i): return item s_i, the ith smallest item in S.

Raman et al. [288] have shown for the (internal) RAM model how to represent S in $\log \binom{|U|}{N} + O\big(|U|\big(\log\log|U|\big)/\log|U|\big)$ bits so that each of the three primitive operations can be performed in constant time.

When storing a set S of K variable-length text strings containing a total of N characters, we use the lexicographic ordering to compare strings. We may want to support an additional primitive:

- *Prefix_Range*(P): return all strings in S that have P as a prefix.

To handle this scenario, Ferragina et al. [159] discuss some edge linearizations of the classic trie dictionary data structure that are simultaneously cache-friendly and compressed. The front-coding scheme is one example of linearization; it is at the core of prefix B-trees [75] and many other disk-conscious compressed indexes for string collections. However, it is largely thought of as a space-effective heuristic without efficient search support. Ferragina et al. introduce new insights on front-coding and other novel linearizations, and they study how close their space occupancy is to the information-theoretic minimum L, which is given by the minimum encoding needed for a labeled binary cardinal tree. They adapt these linearizations along with an EM version of centroid decomposition to design a novel cache-oblivious (and therefore EM) dictionary that offers competitive I/O search time and achieves nearly optimal space in the information-theoretic sense. In particular, the data structures in [159] have the following properties:

- Space usage of $\big(1 + o(1)\big)L + 4K$ bits in blocked format; Query bounds of $O\big(|P|/B + \log K\big)$ I/Os for primitives *Member*(P) and *Rank*(P), $O\big(|s_i|/B + \log K\big)$ I/Os for *Select*(i), and $O\big(|P|/B + \log K + Z/B\big)$ I/Os for *Prefix_Range*(P);
- Space usage of $(1 + \varepsilon)L + O(K)$ bits in blocked format; Query bounds of $O\big(\big(|P| + |succ(P)|\big)/\varepsilon B + \log_B K\big)$ I/Os for *Member*(P) and *Rank*(P), $O\big(|s_i|/\varepsilon B + \log_B K\big)$ I/Os for *Select*(i), and $O\big(\big(|P| + |succ(P)|\big)/\varepsilon B + \log_B K + Z/B\big)$ I/Os for *Prefix_Range*(P),

where Z is the number of occurrences, $succ(P)$ denotes the successor of P in S, and $0 < \varepsilon < 1$ is a user-defined parameter. The data structures can also be tuned optimally to handle any particular distribution of queries.

Eltabakh et al. [146] consider sets of variable-length strings encoded using run-length coding in order to exploit space savings when there are repeated characters. They adapt string B-trees [157, 158] (see Section 14.2) with the EM priority search data structure for three-sided range searching [50] (see Section 12.4) to develop a dynamic compressed data structure for the run-length encoded data. The data structure supports substring matching, prefix matching, and range search queries. The data structure uses $O(\hat{N}/B)$ blocks of space, where $\hat{N}$ is the total length of the compressed strings. Insertions and deletions of t run-length encoded suffixes take $O\big(t \log_B(\hat{N} + t)\big)$ I/Os. Queries for a pattern P can be performed in $O\big(\log_B \hat{N} + \big(|\hat{P}| + Z\big)/B\big)$ I/Os, where $|\hat{P}|$ is the size of the search pattern in run-length encoded format.

One of the major advances in text indexing in the last decade has been the development of entropy-compressed data structures. Until fairly recently, the best general-purpose data structures for pattern matching queries were the suffix tree and its more space-efficient version, the suffix array, which we studied in Section 14.3. However, the suffix array requires several times more space than the text being indexed. The basic reason is that suffix arrays store an array of pointers, each requiring at least $\log N$ bits, whereas the original text being indexed consists of N characters, each of size $\log|\Sigma|$ bits. For a terabyte of ascii text (i.e., $N = 2^{40}$), each text character occupies 8 bits. The suffix array, on the other hand, consists of N pointers to the text, each pointer requiring $\log N = 40$ bits, which is five times larger.[1]

For reasonable values of h, the compressed suffix array of Grossi et al. [185] requires an amount of space in bits per character only as large as the hth-order entropy $H_h(T)$ of the original text, plus lower-order terms. In addition, the compressed suffix array is *self-indexing*, in that it encodes the original text and provides random access to the characters of the original text. Therefore, the original text does not need to be stored, and we can delete it. The net result is a big improvement over conventional suffix arrays: Rather than having to store both the original

[1] Imagine going to a library and finding that the card catalog is stored in an annex that is five times larger than the main library! Suffix arrays support general pattern matching, which card catalogs do not, but it is still counterintuitive for an index to require so much more space than the text it is indexing.

text and a suffix array that is several times larger, we can instead get fast lookup using only a compressed suffix array, whose size is just a fraction of the original text size. Similar results are obtained by Ferragina and Manzini [162] and Ferragina et al. [163] using the Burrows–Wheeler transform. In fact, the two transforms of [185] and [162] are in a sense inverses of one another. A more detailed survey of compressed text indexes in the internal memory setting appears in [267].

In the external memory setting, however, the approaches of [162, 185] have the disadvantage that the algorithms exhibit little locality and thus do not achieve the desired I/O bounds. Chien et al. [107] introduce a new variant of the Burrows–Wheeler transform called the Geometric Burrows–Wheeler Transform (GBWT). Unlike BWT, which merely permutes the text, GBWT converts the text into a set of points in two-dimensional geometry. There is a corresponding equivalence between text pattern matching and geometric range searching. Using this transform, we can answer several open questions about compressed text indexing:

(1) Can compressed data structures be designed in external memory with similar I/O performance as their uncompressed counterparts?
(2) Can compressed data structures be designed for position-restricted pattern matching, in which the answers to a query must be located in a specified range in the text?

The data structure of Chien et al. [107] has size $O\big(N \log |\Sigma|\big)$ bits and can be stored in fully blocked format; pattern matching queries for a pattern P can be done in $O\big(|P|/B + (\log_{|\Sigma|} N)(\log_B N) + Z \log_B N\big)$ I/Os, where Z is the number of occurrences.

The size of the Chien et al. data structure [107] is on the order of the size of the problem instance. An important open problem is to design a pattern matching data structure whose size corresponds to a higher-order compression of the original text, as exists for the (internal memory) RAM model. Another open problem is to reduce the second-order terms in the I/O bound.

Arroyuelo and Navarro [61] propose an EM version of a self-index based upon the Lempel–Ziv compression method [266, 365]. It uses

$8NH_h(T) + o\big(N \log |\Sigma|\big)$ bits of space in blocked format, but does not provide provably good I/O bounds. In practice, it uses about double the space occupied by the original text and has reasonable I/O performance. Mäkinen et al. [245] achieve space bounds of $O\big(N(H_0 + 1)\big)$ bits and $O\big(N\big(H_h(\log |\Sigma| + \log\log N\big) + 1)\big)$ bits in blocked format and a pattern matching query bound of $O\big(|P| \log_B N + Z\big)$ I/Os. The index of González and Navarro [178] uses $O\big(R(\log N) \log(1 + N/R)\big)$ bits in blocked format, where $R \leq \min\big\{N, NH_h + |\Sigma|^h\big\}$, and does pattern matching queries in $O\big(|P| + Z/B\big)$ I/Os.

Compressed data structures have also been explored for graphs, but in a less formal sense. The object is to represent the graph succinctly and still provide fast support for the basic primitives, such as accessing the vertices and edges of the graph. For example, consider a large graph that represents a subset of the World Wide Web, in which each web page corresponds to a vertex and each link from one web page to another corresponds to an edge. Such graphs can often be compressed by a factor of 3–5 by exploiting certain characteristics. For example, the indegrees and outdegrees of the vertices tend to be distributed according to a power law; the probability that a web page has i links is proportional to $1/i^\theta$, where $\theta \approx 2.1$ for incoming links and $\theta \approx 2.72$ for outgoing links. Most of the links from a web page tend to point to another page on the same site with a nearby address, and thus gap methods can give a compressed representation. Adjacent web pages often share the same outgoing links, and thus the adjacency lists can be compressed relative to one another. We refer the reader to [112] for a survey of graph models and initial work in the EM setting.

16

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 [279]. The algorithms in the previous chapters 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 [71] 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 ith phase, $\mathcal{P}$ is allocated m_i blocks of internal memory, and this memory remains allocated to $\mathcal{P}$ until $\mathcal{P}$ completes $2m_i$ I/O operations, at which point the next phase begins. The process continues until $\mathcal{P}$ finishes execution. The model makes the reasonable assumption that the duration for each memory allocation phase is long enough to allow all the memory in that phase to be used by the algorithm.

For sorting, the lower bound approach of Theorem 6.1 implies that

$$\sum_i 2m_i \log m_i = \Omega(n \log n). \tag{16.1}$$

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

$$\sum_i 2m_i \log m_i = O(n \log n) \tag{16.2}$$

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 [71] 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 [279, 363] and hash join [280], but the algorithms handle only special cases and can be made to perform nonoptimally for certain patterns of memory allocation.

17

External Memory Programming Environments

There are three basic approaches to supporting development of I/O-efficient code, which we call access-oriented, array-oriented, and framework-oriented.

Access-oriented systems preserve the programmer abstraction of explicitly requesting data transfer. They often extend the I/O 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 [316], Vesta [116], PIOUS [262], and the High Performance Storage System [351], and I/O libraries MPI-IO [115] and LEDA-SM [124].

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 [326], Panda [308] (which also has aspects of access orientation), PI/OT [281], and ViC* [113].

Instead of viewing batched computation as an enterprise in which code reads data, operates on it, 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 (Transparent Parallel I/O Environment) [41, 49, 330, 337] provides a framework-oriented interface for batched computation, as well as an access-oriented interface for online computation. For batched computations, 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.[1]

TPIE [41, 49, 330, 337], which serves as the implementation platform for the experiments described in Sections 8.1 and 12.2, as well as in several of the references, is a comprehensive set of C++ templated classes and functions for EM paradigms and a run-time library. Its goal is to help programmers develop high-level, portable, and efficient implementations of EM algorithms. 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 internal 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.

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

We have seen in the previous chapter 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 such as 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 outputs 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.

STXXL (Standard Template Library for XXL Data Sets) [135] supports all three basic approaches: access-oriented via a block management layer, array-oriented via the vector class, and framework-oriented via pipelining and iteration. STXXL's support for pipelined scanning and sorting, in which the output of one computation is fed directly into the input of a subsequent computation, can save a factor of about 2 in the number of I/Os for some batched problems. STXXL also supports a library of standard data structures, such as stacks, queues, search trees, and priority queues.

The FG programming environment [117, 127] combines elements of both access-oriented and framework-oriented systems. The programmer creates software pipelines to mitigate data accesses that exhibit high latency, such as disk accesses or interprocessor communication. Buffers traverse each pipeline; each pipeline stage repeatedly accepts a buffer from its predecessor stage, performs some action on the buffer, and conveys the buffer to its successor stage. FG maps each stage to its own thread, and thus multiple stages can overlap. Programmers can implement many batched EM algorithms efficiently — in terms of both I/O complexity and programmer effort — by structuring each pipeline to implement a single pass of a PDM algorithm, matching the buffer

size to the block size, and running a copy of the pipeline on each node of a cluster.

Google's MapReduce [130] is a framework-oriented system that supports a simple functional style of batched programming. The input data are assumed to be in the form of a list of key-value pairs. The programmer specifies a Map function and a Reduce function. The system applies Map to each key-value pair, which produces a set of intermediate key-value pairs. For each k, the system groups together all the intermediate key-value pairs that have the same key k and passes them to the Reduce function; Reduce merges together those key-value pairs and forms a possibly smaller set of values for key k. The system handles the details of data routing, parallel scheduling, and buffer management. This framework is useful for a variety of massive data applications on computer clusters, such as pattern matching, counting access frequencies of web pages, constructing inverted indexes, and distribution sort.

Conclusions

In this manuscript, we have described several useful paradigms for the design and implementation of efficient external memory (EM) 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, as well as for related models such as data streaming and cache-oblivious algorithms. 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 I/O bounds have not yet been determined for several basic EM graph problems such as topological sorting, shortest paths, breadth-first search, depth-first search, and connected components. Several problems remain open in the dynamic and kinetic settings, such as range searching, ray shooting, point location, and finding nearest neighbors.

With the growing power of multicore processors, consideration of parallel CPU time will become increasingly more important. There is an

intriguing connection between problems that have good I/O speedups and problems that have fast and work-efficient parallel algorithms.

A continuing goal is to develop optimal EM algorithms and to translate theoretical gains into observable improvements in practice. For some of the problems that can be solved optimally up to a constant factor, the constant overhead is too large for the algorithm to be of practical use, and simpler approaches are needed. In practice, algorithms cannot assume a static internal memory allocation; they must adapt in a robust way when the memory allocation changes.

Many interesting algorithmic challenges arise from newly developed architectures. The EM concepts and techniques discussed in the book may well apply. Examples of new architectures include computer clusters, multicore processors, hierarchical storage devices, wearable computers, small mobile devices and sensors, disk drives with processing capabilities, and storage devices based upon microelectromechanical systems (MEMS). For example, active (or intelligent) disks, in which disk drives have some processing capability and can filter information sent to the host, have been proposed to further reduce the I/O bottleneck, especially in large database applications [4, 292]. 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 [307, 338].

Notations and Acronyms

Several of the external memory (EM) paradigms discussed in this manuscript are listed in Table 1.1 at the end of Chapter 1.

The parameters of the parallel disk model (PDM) are defined in Section 2.1:

$$
\begin{aligned}
N &= \text{problem size (in units of data items);} \\
M &= \text{internal memory size (in units of data items);} \\
B &= \text{block transfer size (in units of data items);} \\
D &= \text{number of independent disk drives;} \\
P &= \text{number of CPUs;} \\
Q &= \text{number of queries (for a batched problem);} \\
Z &= \text{answer size (in units of data items).}
\end{aligned}
$$

The parameter values satisfy $M < N$ 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.

For simplicity, we often refer to some of the above PDM parameters in units of disk blocks rather than in units of data items; the following

lowercase notations

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

represent the problem size, internal memory size, query specification size, and answer size, respectively, in units of disk blocks.

In Chapter 9, we use some modified notation to describe the problem sizes of graph problems:

$$V = \text{number of vertices}; \quad v = \frac{V}{B};$$
$$E = \text{number of edges}; \quad e = \frac{E}{B}.$$

For simplicity, we assume that $E \geq V$; in those cases where there are fewer edges than vertices, we set E to be V.

In the notations below, k and ℓ denote nonnegative integers and x and y denote real-valued expressions:

$Scan(N)$	number of I/Os to scan a file of N items (Chapter 3).
$Sort(N)$	number of I/Os to sort a file of N items (Chapter 3).
$Search(N)$	number of I/Os to do a dictionary lookup in a data structure of N items (Chapter 3).
$Output(Z)$	number of I/Os to output the Z items of the answer (Chapter 3).
$BundleSort(N, K)$	number of I/Os to sort a file of N items (each with distinct secondary information) having a total of $K \leq N$ unique key values (Section 5.5).
Stripe	the D blocks, one on each of the D disks, located at the same relative position on each disk (Chapter 3).
RAM	random access memory, used for internal memory.
DRAM	dynamic random access memory, frequently used for internal memory.

PDM	parallel disk model (Chapter 2).
SRM	simple randomized merge sort (Section 5.2.1).
RCD	randomized cycling distribution sort (Section 5.1.3).
TPIE	Transparent Parallel I/O Environment (Chapter 17).
$f(k) \approx g(k)$	$f(k)$ is approximately equal to $g(k)$.
$f(k) \sim g(k)$	$f(k)$ is asymptotically equal to $g(k)$, as $k \to \infty$: $\lim_{k\to\infty} \frac{f(k)}{g(k)} = 1.$
$f(k) = O\big(g(k)\big)$	$f(k)$ is big-oh of $g(k)$, as $k \to \infty$: there exist constants $c > 0$ and $K > 0$ such that $\big\lvert f(k)\big\rvert \leq c\big\lvert g(k)\big\rvert$, for all $k \geq K$.
$f(k) = \Omega\big(g(k)\big)$	$f(k)$ is big-omega of $g(k)$, as $k \to \infty$: $g(k) = O\big(f(k)\big)$.
$f(k) = \Theta\big(g(k)\big)$	$f(k)$ is big-theta of $g(k)$, as $k \to \infty$: $f(k) = O\big(g(k)\big)$ and $f(k) = \Omega\big(g(k)\big)$.
$f(k) = o\big(g(k)\big)$	$f(k)$ is little-oh of $g(k)$, as $k \to \infty$: $\lim_{k\to\infty} \frac{f(k)}{g(k)} = 0.$
$f(k) = \omega\big(g(k)\big)$	$f(k)$ is little-omega of $g(k)$, as $k \to \infty$: $g(k) = o\big(f(k)\big)$.
$\lceil x \rceil$	ceiling of x: the smallest integer k satisfying $k \geq x$.
$\lfloor x \rfloor$	floor of x: the largest integer k satisfying $k \leq x$.
$\min\{x, y\}$	minimum of x and y.
$\max\{x, y\}$	maximum of x and y.
$\text{Prob}\{R\}$	probability that relation R is true.
$\log_b x$	base-b logarithm of x; if b is not specified, we use $b = 2$.
$\ln x$	natural logarithm of x: $\log_e x$.

$\binom{k}{\ell}$ binomial coefficient: if $\ell = 0$, it is 1; else, it is

$$\frac{k(k-1)\dots(k-\ell+1)}{\ell(\ell-1)\dots 1}.$$

References

[1] D. J. Abel, "A B^+-tree structure for large quadtrees," *Computer Vision, Graphics, and Image Processing*, vol. 27, pp. 19–31, July 1984.

[2] J. Abello, A. L. Buchsbaum, and J. Westbrook, "A functional approach to external graph algorithms," *Algorithmica*, vol. 32, no. 3, pp. 437–458, 2002.

[3] J. Abello, P. M. Pardalos, and M. G. Resende, eds., *Handbook of Massive Data Sets*. Norwell, Mass.: Kluwer Academic Publishers, 2002.

[4] A. Acharya, M. Uysal, and J. Saltz, "Active disks: Programming model, algorithms and evaluation," *ACM SIGPLAN Notices*, vol. 33, pp. 81–91, November 1998.

[5] M. Adler, "New coding techniques for improved bandwidth utilization," in *Proceedings of the IEEE Symposium on Foundations of Computer Science*, (Burlington, VT), pp. 173–182, October 1996.

[6] P. K. Agarwal, L. Arge, G. S. Brodal, and J. S. Vitter, "I/O-efficient dynamic point location in monotone planar subdivisions," in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pp. 11–20, ACM Press, 1999.

[7] P. K. Agarwal, L. Arge, and A. Danner, "From LIDAR to GRID DEM: A scalable approach," in *Proceedings of the International Symposium on Spatial Data Handling*, 2006.

[8] P. K. Agarwal, L. Arge, and J. Erickson, "Indexing moving points," *Journal of Computer and System Sciences*, vol. 66, no. 1, pp. 207–243, 2003.

[9] P. K. Agarwal, L. Arge, J. Erickson, P. G. Franciosa, and J. S. Vitter, "Efficient searching with linear constraints," *Journal of Computer and System Sciences*, vol. 61, pp. 194–216, October 2000.

[10] P. K. Agarwal, L. Arge, T. M. Murali, K. Varadarajan, and J. S. Vitter, "I/O-efficient algorithms for contour line extraction and planar graph blocking," in

Proceedings of the ACM-SIAM Symposium on Discrete Algorithms, pp. 117–126, ACM Press, 1998.

[11] P. K. Agarwal, L. Arge, O. Procopiuc, and J. S. Vitter, "A framework for index bulk loading and dynamization," in *Proceedings of the International Colloquium on Automata, Languages, and Programming*, pp. 115–127, Crete, Greece: Springer-Verlag, 2001.

[12] P. K. Agarwal, L. Arge, and J. Vahrenhold, "Time responsive external data structures for moving points," in *Proceedings of the Workshop on Algorithms and Data Structures*, pp. 50–61, 2001.

[13] P. K. Agarwal, L. Arge, J. Yang, and K. Yi, "I/O-efficient structures for orthogonal range-max and stabbing-max queries," in *Proceedings of the European Symposium on Algorithms*, pp. 7–18, Springer-Verlag, 2003.

[14] P. K. Agarwal, L. Arge, and K. Yi, "I/O-efficient construction of constrained delaunay triangulations," in *Proceedings of the European Symposium on Algorithms*, pp. 355–366, Springer-Verlag, 2005.

[15] P. K. Agarwal, L. Arge, and K. Yi, "An optimal dynamic interval stabbing-max data structure?," in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pp. 803–812, ACM Press, 2005.

[16] P. K. Agarwal, L. Arge, and K. Yi, "I/O-efficient batched union-find and its applications to terrain analysis," in *Proceedings of the ACM Symposium on Computational Geometry*, ACM Press, 2006.

[17] P. K. Agarwal, M. de Berg, J. Gudmundsson, M. Hammar, and H. J. Haverkort, "Box-trees and R-trees with near-optimal query time," *Discrete and Computational Geometry*, vol. 28, no. 3, pp. 291–312, 2002.

[18] P. K. Agarwal and J. Erickson, "Geometric range searching and its relatives," in *Advances in Discrete and Computational Geometry*, (B. Chazelle, J. E. Goodman, and R. Pollack, eds.), pp. 1–56, Providence, Rhode Island: American Mathematical Society Press, 1999.

[19] P. K. Agarwal and S. Har-Peled, "Maintaining the approximate extent measures of moving points," in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pp. 148–157, Washington, DC: ACM Press, January 2001.

[20] A. Aggarwal, B. Alpern, A. K. Chandra, and M. Snir, "A model for hierarchical memory," in *Proceedings of the ACM Symposium on Theory of Computing*, pp. 305–314, New York: ACM Press, 1987.

[21] A. Aggarwal, A. Chandra, and M. Snir, "Hierarchical memory with block transfer," in *Proceedings of the IEEE Symposium on Foundations of Computer Science*, pp. 204–216, Los Angeles, 1987.

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

[23] A. Aggarwal and J. S. Vitter, "The Input/Output complexity of sorting and related problems," *Communications of the ACM*, vol. 31, no. 9, pp. 1116–1127, 1988.

[24] C. Aggarwal, *Data Streams: Models and Algorithms*. Springer-Verlag, 2007.

[25] M. Ajtai, M. Fredman, and J. Komlós, "Hash functions for priority queues," *Information and Control*, vol. 63, no. 3, pp. 217–225, 1984.

[26] D. Ajwani, I. Malinger, U. Meyer, and S. Toledo, "Characterizing the performance of flash memory storage devices and its impact on algorithm design," in *Proceedings of the International Workshop on Experimental Algorithmics*, (Provincetown, Mass.), pp. 208–219, Springer-Verlag, 2008.

[27] D. Ajwani, U. Meyer, and V. Osipov, "Improved external memory BFS implementation," in *Proceedings of the Workshop on Algorithm Engineering and Experiments*, (New Orleans), pp. 3–12, January 2007.

[28] S. Albers and M. Büttner, "Integrated prefetching and caching in single and parallel disk systems," *Information and Computation*, vol. 198, no. 1, pp. 24–39, 2005.

[29] B. Alpern, L. Carter, E. Feig, and T. Selker, "The uniform memory hierarchy model of computation," *Algorithmica*, vol. 12, no. 2–3, pp. 72–109, 1994.

[30] L. Arge, "The I/O-complexity of ordered binary-decision diagram manipulation," in *Proceedings of the International Symposium on Algorithms and Computation*, pp. 82–91, Springer-Verlag, 1995.

[31] L. Arge, "External memory data structures," in *Handbook of Massive Data Sets*, (J. Abello, P. M. Pardalos, and M. G. Resende, eds.), ch. 9, pp. 313–358, Norwell, Mass.: Kluwer Academic Publishers, 2002.

[32] L. Arge, "The buffer tree: A technique for designing batched external data structures," *Algorithmica*, vol. 37, no. 1, pp. 1–24, 2003.

[33] L. Arge, G. Brodal, and R. Fagerberg, "Cache-oblivious data structures," in *Handbook on Data Structures and Applications*, (D. Mehta and S. Sahni, eds.), CRC Press, 2005.

[34] L. Arge, G. S. Brodal, and L. Toma, "On external-memory MST, SSSP, and multi-way planar graph separation," *Journal of Algorithms*, vol. 53, no. 2, pp. 186–206, 2004.

[35] L. Arge, J. S. Chase, P. Halpin, L. Toma, J. S. Vitter, D. Urban, and R. Wickremesinghe, "Efficient flow computation on massive grid datasets," *GeoInformatica*, vol. 7, pp. 283–313, December 2003.

[36] L. Arge, A. Danner, H. J. Haverkort, and N. Zeh, "I/O-efficient hierarchical watershed decomposition of grid terrains models," in *Proceedings of the International Symposium on Spatial Data Handling*, 2006.

[37] L. Arge, A. Danner, and S.-H. Teh, "I/O-efficient point location using persistent B-trees," in *Workshop on Algorithm Engineering and Experimentation*, 2003.

[38] L. Arge, M. de Berg, H. J. Haverkort, and K. Yi, "The priority R-tree: A practically efficient and worst-case optimal R-tree," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 347–358, ACM Press, 2004.

[39] L. Arge, P. Ferragina, R. Grossi, and J. S. Vitter, "On sorting strings in external memory," in *Proceedings of the ACM Symposium on Theory of Computing*, pp. 540–548, ACM Press, 1997.

[40] L. Arge, M. T. Goodrich, M. Nelson, and N. Sitchinava, "Fundamental parallel algorithms for private-cache chip multiprocessors," in *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, Munich: ACM Press, June 2008.

[41] L. Arge, K. H. Hinrichs, J. Vahrenhold, and J. S. Vitter, "Efficient bulk operations on dynamic R-trees," *Algorithmica*, vol. 33, pp. 104–128, January 2002.

[42] L. Arge, M. Knudsen, and K. Larsen, "A general lower bound on the I/O-complexity of comparison-based algorithms," in *Proceedings of the Workshop on Algorithms and Data Structures*, pp. 83–94, Springer-Verlag, 1993.

[43] L. Arge, U. Meyer, and L. Toma, "External memory algorithms for diameter and all-pairs shortest-paths on sparse graphs," in *Proceedings of the International Colloquium on Automata, Languages, and Programming*, pp. 146–157, Springer-Verlag, 2004.

[44] L. Arge, U. Meyer, L. Toma, and N. Zeh, "On external-memory planar depth first search," *Journal of Graph Algorithms and Applications*, vol. 7, no. 2, pp. 105–129, 2003.

[45] L. Arge and P. Miltersen, "On showing lower bounds for external-memory computational geometry problems," in *External Memory Algorithms and Visualization*, (J. Abello and J. S. Vitter, eds.), pp. 139–159, Providence, Rhode Island: American Mathematical Society Press, 1999.

[46] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, J. Vahrenhold, and J. S. Vitter, "A unified approach for indexed and non-indexed spatial joins," in *Proceedings of the International Conference on Extending Database Technology*, Konstanz, Germany: ACM Press, March 2000.

[47] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter, "Scalable sweeping-based spatial join," in *Proceedings of the International Conference on Very Large Databases*, pp. 570–581, New York: Morgan Kaufmann, August 1998.

[48] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter, "Theory and practice of I/O-efficient algorithms for multidimensional batched searching problems," in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pp. 685–694, ACM Press, January 1998.

[49] L. Arge, O. Procopiuc, and J. S. Vitter, "Implementing I/O-efficient data structures using TPIE," in *Proceedings of the European Symposium on Algorithms*, pp. 88–100, Springer-Verlag, 2002.

[50] L. Arge, V. Samoladas, and J. S. Vitter, "Two-dimensional indexability and optimal range search indexing," in *Proceedings of the ACM Conference on Principles of Database Systems*, pp. 346–357, Philadelphia: ACM Press, May–June 1999.

[51] L. Arge, V. Samoladas, and K. Yi, "Optimal external memory planar point enclosure," in *Proceedings of the European Symposium on Algorithms*, pp. 40–52, Springer-Verlag, 2004.

[52] L. Arge and L. Toma, "Simplified external memory algorithms for planar DAGs," in *Proceedings of the Scandinavian Workshop on Algorithmic Theory*, pp. 493–503, 2004.

[53] L. Arge and L. Toma, "External data structures for shortest path queries on planar digraphs," in *Proceedings of the International Symposium on Algorithms and Computation*, pp. 328–338, Springer-Verlag, 2005.

[54] L. Arge, L. Toma, and J. S. Vitter, "I/O-efficient algorithms for problems on grid-based terrains," in *Workshop on Algorithm Engineering and Experimentation*, San Francisco: Springer-Verlag, January 2000.

[55] L. Arge, L. Toma, and N. Zeh, "I/O-efficient topological sorting of planar DAGs," in *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pp. 85–93, ACM Press, 2003.
[56] L. Arge and J. Vahrenhold, "I/O-efficient dynamic planar point location," *Computational Geometry*, vol. 29, no. 2, pp. 147–162, 2004.
[57] L. Arge, D. E. Vengroff, and J. S. Vitter, "External-memory algorithms for processing line segments in geographic information systems," *Algorithmica*, vol. 47, pp. 1–25, January 2007.
[58] L. Arge and J. S. Vitter, "Optimal external memory interval management," *SIAM Journal on Computing*, vol. 32, no. 6, pp. 1488–1508, 2003.
[59] L. Arge and N. Zeh, "I/O-efficient strong connectivity and depth-first search for directed planar graphs," in *Proceedings of the IEEE Symposium on Foundations of Computer Science*, pp. 261–270, 2003.
[60] C. Armen, "Bounds on the separation of two parallel disk models," in *Proceedings of the Workshop on Input/Output in Parallel and Distributed Systems*, pp. 122–127, May 1996.
[61] D. Arroyuelo and G. Navarro, "A Lempel–Ziv text index on secondary storage," in *Proceedings of the Symposium on Combinatorial Pattern Matching*, pp. 83–94, Springer-Verlag, 2007.
[62] M. J. Atallah and S. Prabhakar, "(Almost) optimal parallel block access for range queries," *Information Sciences*, vol. 157, pp. 21–31, 2003.
[63] R. A. Baeza-Yates, "Expected behaviour of B^+-trees under random insertions," *Acta Informatica*, vol. 26, no. 5, pp. 439–472, 1989.
[64] R. A. Baeza-Yates, "Bounded disorder: The effect of the index," *Theoretical Computer Science*, vol. 168, pp. 21–38, 1996.
[65] R. A. Baeza-Yates and P.-A. Larson, "Performance of B^+-trees with partial expansions," *IEEE Transactions on Knowledge and Data Engineering*, vol. 1, pp. 248–257, June 1989.
[66] R. A. Baeza-Yates and B. Ribeiro-Neto, eds., *Modern Information Retrieval*. Addison Wesley Longman, 1999.
[67] R. A. Baeza-Yates and H. Soza-Pollman, "Analysis of linear hashing revisited," *Nordic Journal of Computing*, vol. 5, pp. 70–85, 1998.
[68] R. D. Barve, E. F. Grove, and J. S. Vitter, "Simple randomized mergesort on parallel disks," *Parallel Computing*, vol. 23, no. 4, pp. 601–631, 1997.
[69] R. D. Barve, M. Kallahalla, P. J. Varman, and J. S. Vitter, "Competitive analysis of buffer management algorithms," *Journal of Algorithms*, vol. 36, pp. 152–181, August 2000.
[70] R. D. Barve, E. A. M. Shriver, P. B. Gibbons, B. K. Hillyer, Y. Matias, and J. S. Vitter, "Modeling and optimizing I/O throughput of multiple disks on a bus," in *Procedings of ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, pp. 83–92, Atlanta: ACM Press, May 1999.
[71] R. D. Barve and J. S. Vitter, "A theoretical framework for memory-adaptive algorithms," in *Proceedings of the IEEE Symposium on Foundations of*

Computer Science, pp. 273–284, New York: IEEE Computer Society Press, October 1999.

[72] R. D. Barve and J. S. Vitter, "A simple and efficient parallel disk mergesort," *ACM Trans. Computer Systems*, vol. 35, pp. 189–215, March/April 2002.

[73] J. Basch, L. J. Guibas, and J. Hershberger, "Data structures for mobile data," *Journal of Algorithms*, vol. 31, pp. 1–28, 1999.

[74] R. Bayer and E. McCreight, "Organization of large ordered indexes," *Acta Informatica*, vol. 1, pp. 173–189, 1972.

[75] R. Bayer and K. Unterauer, "Prefix B-trees," *ACM Transactions on Database Systems*, vol. 2, pp. 11–26, March 1977.

[76] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer, "An asymptotically optimal multiversion B-tree," *VLDB Journal*, vol. 5, pp. 264–275, December 1996.

[77] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: An efficient and robust access method for points and rectangles," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 322–331, ACM Press, 1990.

[78] A. L. Belady, "A study of replacement algorithms for virtual storage computers," *IBM Systems Journal*, vol. 5, pp. 78–101, 1966.

[79] M. A. Bender, E. D. Demaine, and M. Farach-Colton, "Cache-oblivious B-trees," *SIAM Journal on Computing*, vol. 35, no. 2, pp. 341–358, 2005.

[80] M. A. Bender, M. Farach-Colton, and B. Kuszmaul, "Cache-oblivious string B-trees," in *Proceedings of the ACM Conference on Principles of Database Systems*, pp. 233–242, ACM Press, 2006.

[81] M. A. Bender, J. T. Fineman, S. Gilbert, and B. C. Kuszmaul, "Concurrent cache-oblivious B-trees," in *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pp. 228–237, ACM Press, 2005.

[82] J. L. Bentley, "Multidimensional divide and conquer," *Communications of the ACM*, vol. 23, no. 6, pp. 214–229, 1980.

[83] J. L. Bentley and J. B. Saxe, "Decomposable searching problems I: Static-to-dynamic transformations," *Journal of Algorithms*, vol. 1, pp. 301–358, December 1980.

[84] S. Berchtold, C. Böhm, and H.-P. Kriegel, "Improving the query performance of high-dimensional index structures by bulk load operations," in *Proceedings of the International Conference on Extending Database Technology*, pp. 216–230, Springer-Verlag, 1998.

[85] M. Berger, E. R. Hansen, R. Pagh, M. Pătraşcu, M. Ružić, and P. Tiedemann, "Deterministic load balancing and dictionaries in the parallel disk model," in *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, ACM Press, 2006.

[86] R. Bhatia, R. K. Sinha, and C.-M. Chen, "A hierarchical technique for constructing efficient declustering schemes for range queries," *The Computer Journal*, vol. 46, no. 4, pp. 358–377, 2003.

[87] N. Blum and K. Mehlhorn, "On the average number of rebalancing operations in weight-balanced trees," *Theoretical Computer Science*, vol. 11, pp. 303–320, July 1980.

[88] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Communications of the ACM*, vol. 20, pp. 762–772, October 1977.

[89] C. Breimann and J. Vahrenhold, "External memory computational geometry revisited," in *Algorithms for Memory Hierarchies*, pp. 110–148, 2002.

[90] K. Brengel, A. Crauser, P. Ferragina, and U. Meyer, "An experimental study of priority queues in external memory," *ACM Journal of Experimental Algorithmics*, vol. 5, p. 17, 2000.

[91] G. S. Brodal and R. Fagerberg, "Lower bounds for external memory dictionaries," in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pp. 546–554, ACM Press, 2003.

[92] G. S. Brodal and J. Katajainen, "Worst-case efficient external-memory priority queues," in *Proceedings of the Scandinavian Workshop on Algorithmic Theory*, pp. 107–118, Stockholm: Springer-Verlag, July 1998.

[93] A. L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. R. Westbrook, "On external memory graph traversal," in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pp. 859–860, ACM Press, January 2000.

[94] P. Callahan, M. T. Goodrich, and K. Ramaiyer, "Topology B-trees and their applications," in *Proceedings of the Workshop on Algorithms and Data Structures*, pp. 381–392, Springer-Verlag, 1995.

[95] P. Cao, E. W. Felten, A. R. Karlin, and K. Li, "Implementation and performance of integrated application-controlled file caching, prefetching and disk scheduling," *ACM Transactions on Computer Systems*, vol. 14, pp. 311–343, November 1996.

[96] L. Carter and K. S. Gatlin, "Towards an optimal bit-reversal permutation program," in *Proceedings of the IEEE Symposium on Foundations of Computer Science*, pp. 544–553, November 1998.

[97] G. Chaudhry and T. H. Cormen, "Oblivious vs. distribution-based sorting: An experimental evaluation," in *Proceedings of the European Symposium on Algorithms*, pp. 317–328, Springer-Verlag, 2005.

[98] B. Chazelle, "Filtering search: A new approach to query-answering," *SIAM Journal on Computing*, vol. 15, pp. 703–724, 1986.

[99] B. Chazelle, "Lower bounds for orthogonal range searching: I. The reporting case," *Journal of the ACM*, vol. 37, pp. 200–212, April 1990.

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

[101] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, "RAID: High-performance, reliable secondary storage," *ACM Computing Surveys*, vol. 26, pp. 145–185, June 1994.

[102] R. Cheng, Y. Xia, S. Prabhakar, R. Shah, and J. S. Vitter, "Efficient indexing methods for probabilistic threshold queries over uncertain data," in *Proceedings of the International Conference on Very Large Databases*, Toronto: Morgan Kaufmann, August 2004.

[103] R. Cheng, Y. Xia, S. Prabhakar, R. Shah, and J. S. Vitter, "Efficient join processing over uncertain-valued attributes," in *Proceedings of the International*

ACM Conference on Information and Knowledge Management, Arlington, Va.: ACM Press, November 2006.

[104] Y.-J. Chiang, "Experiments on the practical I/O efficiency of geometric algorithms: Distribution sweep vs. plane sweep," *Computational Geometry: Theory and Applications*, vol. 8, no. 4, pp. 211–236, 1998.

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

[106] Y.-J. Chiang and C. T. Silva, "External memory techniques for isosurface extraction in scientific visualization," in *External Memory Algorithms and Visualization*, (J. Abello and J. S. Vitter, eds.), pp. 247–277, Providence, Rhode Island: American Mathematical Society Press, 1999.

[107] Y.-F. Chien, W.-K. Hon, R. Shah, and J. S. Vitter, "Geometric Burrows–Wheeler transform: Linking range searching and text indexing," in *Proceedings of the Data Compression Conference*, Snowbird, Utah: IEEE Computer Society Press, March 2008.

[108] R. A. Chowdhury and V. Ramachandran, "External-memory exact and approximate all-pairs shortest-paths in undirected graphs," in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pp. 735–744, ACM Press, 2005.

[109] V. Ciriani, P. Ferragina, F. Luccio, and S. Muthukrishnan, "Static optimality theorem for external memory string access," in *Proceedings of the IEEE Symposium on Foundations of Computer Science*, pp. 219–227, 2002.

[110] D. R. Clark and J. I. Munro, "Efficient suffix trees on secondary storage," in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pp. 383–391, Atlanta: ACM Press, June 1996.

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

[112] F. Claude and G. Navarro, "A fast and compact Web graph representation," in *Proceedings of the International Symposium on String Processing and Information Retrieval*, pp. 105–116, Springer-Verlag, 2007.

[113] A. Colvin and T. H. Cormen, "ViC*: A compiler for virtual-memory C*," in *Proceedings of the International Workshop on High-Level Programming Models and Supportive Environments*, pp. 23–33, 1998.

[114] D. Comer, "The ubiquitous B-Tree," *ACM Computing Surveys*, vol. 11, no. 2, pp. 121–137, 1979.

[115] P. Corbett, D. Feitelson, S. Fineberg, Y. Hsu, B. Nitzberg, J.-P. Prost, M. Snir, B. Traversat, and P. Wong, "Overview of the MPI-IO parallel I/O interface," in *Input/Output in Parallel and Distributed Computer Systems*, (R. Jain, J. Werth, and J. C. Browne, eds.), ch. 5, pp. 127–146, Kluwer Academic Publishers, 1996.

[116] P. F. Corbett and D. G. Feitelson, "The Vesta parallel file system," *ACM Transactions on Computer Systems*, vol. 14, pp. 225–264, August 1996.

[117] T. H. Cormen and E. R. Davidson, "FG: A framework generator for hiding latency in parallel programs running on clusters," in *Proceedings of the 17th International Conference on Parallel and Distributed Computing Systems*, pp. 137–144, Sep. 2004.

[118] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. Cambridge, Mass.: MIT Press, 2nd ed., September 2001.

[119] T. H. Cormen and D. M. Nicol, "Performing out-of-core FFTs on parallel disk systems," *Parallel Computing*, vol. 24, pp. 5–20, January 1998.

[120] T. H. Cormen, T. Sundquist, and L. F. Wisniewski, "Asymptotically tight bounds for performing BMMC permutations on parallel disk systems," *SIAM Journal on Computing*, vol. 28, no. 1, pp. 105–136, 1999.

[121] A. Crauser and P. Ferragina, "A theoretical and experimental study on the construction of suffix arrays in external memory," *Algorithmica*, vol. 32, no. 1, pp. 1–35, 2002.

[122] A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, and E. A. Ramos, "I/O-optimal computation of segment intersections," in *External Memory Algorithms and Visualization*, (J. Abello and J. S. Vitter, eds.), pp. 131–138, Providence, Rhode Island: American Mathematical Society Press, 1999.

[123] A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, and E. A. Ramos, "Randomized external-memory algorithms for line segment intersection and other geometric problems," *International Journal of Computational Geometry and Applications*, vol. 11, no. 3, pp. 305–337, 2001.

[124] A. Crauser and K. Mehlhorn, "LEDA-SM: Extending LEDA to secondary memory," in *Proceedings of the Workshop on Algorithm Engineering*, (J. S. Vitter and C. Zaroliagis, eds.), pp. 228–242, London: Springer-Verlag, July 1999.

[125] K. Curewitz, P. Krishnan, and J. S. Vitter, "Practical Prefetching Via Data Compression," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 257–266, Washington, DC: ACM Press, May 1993.

[126] R. Cypher and G. Plaxton, "Deterministic sorting in nearly logarithmic time on the hypercube and related computers," *Journal of Computer and System Sciences*, vol. 47, no. 3, pp. 501–548, 1993.

[127] E. R. Davidson, *FG: Improving Parallel Programs and Parallel Programming Since 2003*. PhD thesis, Dartmouth College Department of Computer Science, Aug. 2006.

[128] M. de Berg, J. Gudmundsson, M. Hammar, and M. H. Overmars, "On R-trees with low query complexity," *Computational Geometry*, vol. 24, no. 3, pp. 179–195, 2003.

[129] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, *Computational Geometry Algorithms and Applications*. Berlin: Springer-Verlag, 1997.

[130] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proceedings of the Symposium on Operating Systems Design and Implementation*, pp. 137–150, USENIX, December 2004.

[131] F. K. H. A. Dehne, W. Dittrich, and D. A. Hutchinson, "Efficient External Memory Algorithms by Simulating Coarse-Grained Parallel Algorithms," *Algorithmica*, vol. 36, no. 2, pp. 97–122, 2003.

[132] F. K. H. A. Dehne, W. Dittrich, D. A. Hutchinson, and A. Maheshwari, "Bulk synchronous parallel algorithms for the external memory model," *Theory of Computing Systems*, vol. 35, no. 6, pp. 567–597, 2002.

[133] R. Dementiev, *Algorithm Engineering for Large Data Sets*. PhD thesis, Saarland University, 2006.

[134] R. Dementiev, J. Kärkkäinen, J. Mehnert, and P. Sanders, "Better external memory suffix array construction," *ACM Journal of Experimental Algorithmics*, in press.

[135] R. Dementiev, L. Kettner, and P. Sanders, "STXXL: Standard Template Library for XXL Data Sets," *Software — Practice and Experience*, vol. 38, no. 6, pp. 589–637, 2008.

[136] R. Dementiev and P. Sanders, "Asynchronous parallel disk sorting," in *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pp. 138–148, ACM Press, 2003.

[137] R. Dementiev, P. Sanders, D. Schultes, and J. Sibeyn, "Engineering an external memory minimum spanning tree algorithm," in *Proceedings of IFIP International Conference on Theoretical Computer Science*, Toulouse: Kluwer Academic Publishers, 2004.

[138] H. B. Demuth, *Electronic data sorting*. PhD thesis, Stanford University, 1956.

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

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

[141] W. Dittrich, D. A. Hutchinson, and A. Maheshwari, "Blocking in parallel multisearch problems," *Theory of Computing Systems*, vol. 34, no. 2, pp. 145–189, 2001.

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

[143] M. C. Easton, "Key-sequence data sets on indelible storage," *IBM Journal of Research and Development*, vol. 30, pp. 230–241, 1986.

[144] H. Edelsbrunner, "A new approach to rectangle intersections, Part I," *International Journal of Computer Mathematics*, vol. 13, pp. 209–219, 1983.

[145] H. Edelsbrunner, "A New approach to rectangle intersections, Part II," *International Journal of Computer Mathematics*, vol. 13, pp. 221–229, 1983.

[146] M. Y. Eltabakh, W.-K. Hon, R. Shah, W. Aref, and J. S. Vitter, "The SBC-tree: An index for run-length compressed sequences," in *Proceedings of the International Conference on Extending Database Technology*, Nantes, France: Springer-Verlag, March 2008.

[147] R. J. Enbody and H. C. Du, "Dynamic hashing schemes," *ACM Computing Surveys*, vol. 20, pp. 85–113, June 1988.

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

[149] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig, "Sparsification — a technique for speeding up dynamic graph algorithms," *Journal of the ACM*, vol. 44, no. 5, pp. 669–696, 1997.

[150] J. Erickson, "Lower bounds for external algebraic decision trees," in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pp. 755–761, ACM Press, 2005.

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

[152] R. Fagerberg, A. Pagh, and R. Pagh, "External string sorting: Faster and cache oblivious," in *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, pp. 68–79, Springer-Verlag, 2006.

[153] R. Fagin, J. Nievergelt, N. Pippinger, and H. R. Strong, "Extendible hashing — a fast access method for dynamic files," *ACM Transactions on Database Systems*, vol. 4, no. 3, pp. 315–344, 1979.

[154] M. Farach-Colton, P. Ferragina, and S. Muthukrishnan, "On the sorting-complexity of suffix tree construction," *Journal of the ACM*, vol. 47, no. 6, pp. 987–1011, 2000.

[155] J. Feigenbaum, S. Kannan, M. Strauss, and M. Viswanathan, "An approximate L1-difference algorithm for massive data streams," *SIAM Journal on Computing*, vol. 32, no. 1, pp. 131–151, 2002.

[156] W. Feller, *An Introduction to Probability Theory and its Applications*. Vol. 1, New York: John Wiley & Sons, 3rd ed., 1968.

[157] P. Ferragina and R. Grossi, "Fast string searching in secondary storage: Theoretical developments and experimental results," in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pp. 373–382, Atlanta: ACM Press, June 1996.

[158] P. Ferragina and R. Grossi, "The String B-tree: A new data structure for string search in external memory and its applications," *Journal of the ACM*, vol. 46, pp. 236–280, March 1999.

[159] P. Ferragina, R. Grossi, A. Gupta, R. Shah, and J. S. Vitter, "On searching compressed string collections cache-obliviously," in *Proceedings of the ACM Conference on Principles of Database Systems*, Vancouver: ACM Press, June 2008.

[160] P. Ferragina, N. Koudas, S. Muthukrishnan, and D. Srivastava, "Two-dimensional substring indexing," *Journal of Computer and System Sciences*, vol. 66, no. 4, pp. 763–774, 2003.

[161] P. Ferragina and F. Luccio, "Dynamic dictionary matching in external memory," *Information and Computation*, vol. 146, pp. 85–99, November 1998.

[162] P. Ferragina and G. Manzini, "Indexing compressed texts," *Journal of the ACM*, vol. 52, no. 4, pp. 552–581, 2005.

[163] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro, "Compressed representations of sequences and full-text indexes," *ACM Transaction on Algorithms*, vol. 3, p. 20, May 2007.

[164] P. Flajolet, "On the performance evaluation of extendible hashing and trie searching," *Acta Informatica*, vol. 20, no. 4, pp. 345–369, 1983.

[165] R. W. Floyd, "Permuting information in idealized two-level storage," in *Complexity of Computer Computations*, (R. Miller and J. Thatcher, eds.), pp. 105–109, Plenum, 1972.

[166] W. Frakes and R. A. Baeza-Yates, eds., *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, 1992.

[167] G. Franceschini, R. Grossi, J. I. Munro, and L. Pagli, "Implicit B-Trees: A new data structure for the dictionary problem," *Journal of Computer and System Sciences*, vol. 68, no. 4, pp. 788–807, 2004.

[168] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *Proceedings of the IEEE Symposium on Foundations of Computer Science*, pp. 285–298, 1999.

[169] T. A. Funkhouser, C. H. Sequin, and S. J. Teller, "Management of large amounts of data in interactive building walkthroughs," in *Proceedings of the ACM Symposium on Interactive 3D Graphics*, pp. 11–20, Boston: ACM Press, March 1992.

[170] V. Gaede and O. Günther, "Multidimensional access methods," *ACM Computing Surveys*, vol. 30, pp. 170–231, June 1998.

[171] G. R. Ganger, "Generating representative synthetic workloads: An unsolved problem," in *Proceedings of the Computer Measurement Group Conference*, pp. 1263–1269, December 1995.

[172] M. Gardner, ch. 7, *Magic Show*. New York: Knopf, 1977.

[173] I. Gargantini, "An effective way to represent quadtrees," *Communications of the ACM*, vol. 25, pp. 905–910, December 1982.

[174] T. M. Ghanem, R. Shah, M. F. Mokbel, W. G. Aref, and J. S. Vitter, "Bulk operations for space-partitioning trees," in *Proceedings of IEEE International Conference on Data Engineering*, Boston: IEEE Computer Society Press, April 2004.

[175] G. A. Gibson, J. S. Vitter, and J. Wilkes, "Report of the working group on storage I/O issues in large-scale computing," *ACM Computing Surveys*, vol. 28, pp. 779–793, December 1996.

[176] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," in *Proceedings of the International Conference on Very Large Databases*, pp. 78–89, Edinburgh, Scotland: Morgan Kaufmann, 1999.

[177] R. Goldman, N. Shivakumar, S. Venkatasubramanian, and H. Garcia-Molina, "Proximity search in databases," in *Proceedings of the International Conference on Very Large Databases*, pp. 26–37, August 1998.

[178] R. González and G. Navarro, "A compressed text index on secondary memory," in *Proceedings of the International Workshop on Combinatorial Algorithms*, (Newcastle, Australia), pp. 80–91, College Publications, 2007.

[179] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter, "External-memory computational geometry," in *Proceedings of the IEEE Symposium on Foundations of Computer Science*, pp. 714–723, Palo Alto: IEEE Computer Society Press, November 1993.

[180] "Google Earth online database of satellite images," Available on the World-Wide Web at http://earth.google.com/.

[181] S. Govindarajan, P. K. Agarwal, and L. Arge, "CRB-tree: An efficient indexing scheme for range-aggregate queries," in *Proceedings of the International Conference on Database Theory*, pp. 143–157, Springer-Verlag, 2003.

[182] S. Govindarajan, T. Lukovszki, A. Maheshwari, and N. Zeh, "I/O-efficient well-separated pair decomposition and its applications," *Algorithmica*, vol. 45, pp. 385–614, August 2006.

[183] D. Greene, "An implementation and performance analysis of spatial data access methods," in *Proceedings of IEEE International Conference on Data Engineering*, pp. 606–615, 1989.

[184] J. L. Griffin, S. W. Schlosser, G. R. Ganger, and D. F. Nagle, "Modeling and performance of MEMS-based storage devices," in *Procedings of ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, pp. 56–65, Santa Clara, Cal.: ACM Press, June 2000.

[185] R. Grossi, A. Gupta, and J. S. Vitter, "High-order entropy-compressed text indexes," in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, ACM Press, January 2003.

[186] R. Grossi and G. F. Italiano, "Efficient cross-trees for external memory," in *External Memory Algorithms and Visualization*, (J. Abello and J. S. Vitter, eds.), pp. 87–106, Providence, Rhode Island: American Mathematical Society Press, 1999.

[187] R. Grossi and G. F. Italiano, "Efficient splitting and merging algorithms for order decomposable problems," *Information and Computation*, vol. 154, no. 1, pp. 1–33, 1999.

[188] S. K. S. Gupta, Z. Li, and J. H. Reif, "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*, pp. 510–513, October 1995.

[189] D. Gusfield, *Algorithms on Strings, Trees, and Sequences*. Cambridge, UK: Cambridge University Press, 1997.

[190] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 47–57, ACM Press, 1984.

[191] H. J. Haverkort and L. Toma, "I/O-efficient algorithms on near-planar graphs," in *Proceedings of the Latin American Theoretical Informatics Symposium*, pp. 580–591, 2006.

[192] T. Hazel, L. Toma, R. Wickremesinghe, and J. Vahrenhold, "Terracost: A versatile and scalable approach to computing least-cost-path surfaces for massive grid-based terrains," in *Proceedings of the ACM Symposium on Applied Computing*, pp. 52–57, ACM Press, 2006.

[193] J. M. Hellerstein, E. Koutsoupias, and C. H. Papadimitriou, "On the analysis of indexing schemes," in *Proceedings of the ACM Symposium on Principles of Database Systems*, pp. 249–256, Tucson: ACM Press, May 1997.

[194] L. Hellerstein, G. Gibson, R. M. Karp, R. H. Katz, and D. A. Patterson, "Coding techniques for handling failures in large disk arrays," *Algorithmica*, vol. 12, no. 2–3, pp. 182–208, 1994.

[195] M. R. Henzinger, P. Raghavan, and S. Rajagopalan, "Computing on data streams," in *External Memory Algorithms and Visualization*, (J. Abello and J. S. Vitter, eds.), pp. 107–118, Providence, Rhode Island: American Mathematical Society Press, 1999.

[196] K. Hinrichs, "Implementation of the grid file: Design concepts and experience," *BIT*, vol. 25, no. 4, pp. 569–592, 1985.

[197] W.-K. Hon, T.-W. Lam, R. Shah, S.-L. Tam, and J. S. Vitter, "Cache-oblivious index for approximate string matching," in *Proceedings of the Symposium on Combinatorial Pattern Matching*, pp. 40–51, London, Ontario, Canada: Springer-Verlag, July 2007.

[198] W.-K. Hon, R. Shah, P. J. Varman, and J. S. Vitter, "Tight competitive ratios for parallel disk prefetching and caching," in *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, Munich: ACM Press, June 2008.

[199] J. W. Hong and H. T. Kung, "I/O complexity: The red-blue pebble game," in *Proceedings of the ACM Symposium on Theory of Computing*, pp. 326–333, ACM Press, May 1981.

[200] D. Hutchinson, A. Maheshwari, J.-R. Sack, and R. Velicescu, "Early experiences in implementing the buffer tree," in *Proceedings of the Workshop on Algorithm Engineering*, Springer-Verlag, 1997.

[201] D. A. Hutchinson, A. Maheshwari, and N. Zeh, "An external memory data structure for shortest path queries," *Discrete Applied Mathematics*, vol. 126, no. 1, pp. 55–82, 2003.

[202] D. A. Hutchinson, P. Sanders, and J. S. Vitter, "Duality between prefetching and queued writing with parallel disks," *SIAM Journal on Computing*, vol. 34, no. 6, pp. 1443–1463, 2005.

[203] P. Indyk, R. Motwani, P. Raghavan, and S. Vempala, "Locality-preserving hashing in multidimensional spaces," in *Proceedings of the ACM Symposium on Theory of Computing*, pp. 618–625, El Paso: ACM Press, May 1997.

[204] M. Kallahalla and P. J. Varman, "Optimal prefetching and caching for parallel I/O systems," in *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, Crete, Greece: ACM Press, July 2001.

[205] M. Kallahalla and P. J. Varman, "Optimal read-once parallel disk scheduling," *Algorithmica*, vol. 43, no. 4, pp. 309–343, 2005.

[206] I. Kamel and C. Faloutsos, "On packing R-trees," in *Proceedings of the International ACM Conference on Information and Knowledge Management*, pp. 490–499, 1993.

[207] I. Kamel and C. Faloutsos, "Hilbert R-tree: An improved R-tree using fractals," in *Proceedings of the International Conference on Very Large Databases*, pp. 500–509, 1994.

[208] I. Kamel, M. Khalil, and V. Kouramajian, "Bulk insertion in dynamic R-trees," in *Proceedings of the International Symposium on Spatial Data Handling*, pp. 3B, 31–42, 1996.

[209] P. C. Kanellakis, G. M. Kuper, and P. Z. Revesz, "Constraint query languages," *Journal of Computer and System Sciences*, vol. 51, no. 1, pp. 26–52, 1995.

[210] P. C. Kanellakis, S. Ramaswamy, D. E. Vengroff, and J. S. Vitter, "Indexing for data models with constraints and classes," *Journal of Computer and System Sciences*, vol. 52, no. 3, pp. 589–612, 1996.

[211] K. V. R. Kanth and A. K. Singh, "Optimal dynamic range searching in non-replicating index structures," in *Proceedings of the International Conference on Database Theory*, pp. 257–276, Springer-Verlag, January 1999.

[212] J. Kärkkäinen and S. S. Rao, "Full-text indexes in external memory," in *Algorithms for Memory Hierarchies*, (U. Meyer, P. Sanders, and J. Sibeyn, eds.), ch. 7, pp. 149–170, Berlin: Springer-Verlag, 2003.

[213] I. Katriel and U. Meyer, "Elementary graph algorithms in external memory," in *Algorithms for Memory Hierarchies*, (U. Meyer, P. Sanders, and J. Sibeyn, eds.), ch. 4, pp. 62–84, Berlin: Springer-Verlag, 2003.

[214] R. Khandekar and V. Pandit, "Offline Sorting Buffers On Line," in *Proceedings of the International Symposium on Algorithms and Computation*, pp. 81–89, Springer-Verlag, December 2006.

[215] S. Khuller, Y. A. Kim, and Y.-C. J. Wan, "Algorithms for data migration with cloning," *SIAM Journal on Computing*, vol. 33, no. 2, pp. 448–461, 2004.

[216] M. Y. Kim, "Synchronized disk interleaving," *IEEE Transactions on Computers*, vol. 35, pp. 978–988, November 1986.

[217] T. Kimbrel and A. R. Karlin, "Near-optimal parallel prefetching and caching," *SIAM Journal on Computing*, vol. 29, no. 4, pp. 1051–1082, 2000.

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

[219] S. T. Klein and D. Shapira, "Searching in compressed dictionaries," in *Proceedings of the Data Compression Conference*, Snowbird, Utah: IEEE Computer Society Press, 2002.

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

[221] D. E. Knuth, *MMIXware*. Berlin: Springer-Verlag, 1999.

[222] D. E. Knuth, J. H. Morris, and V. R. Pratt, "Fast pattern matching in strings," *SIAM Journal on Computing*, vol. 6, pp. 323–350, 1977.

[223] G. Kollios, D. Gunopulos, and V. J. Tsotras, "On indexing mobile objects," in *Proceedings of the ACM Symposium on Principles of Database Systems*, pp. 261–272, ACM Press, 1999.

[224] E. Koutsoupias and D. S. Taylor, "Tight bounds for 2-dimensional indexing schemes," in *Proceedings of the ACM Symposium on Principles of Database Systems*, pp. 52–58, Seattle: ACM Press, June 1998.

[225] M. Kowarschik and C. Weiß, "An overview of cache optimizaiton techniques and cache-aware numerical algorithms," in *Algorithms for Memory Hierarchies*, (U. Meyer, P. Sanders, and J. Sibeyn, eds.), ch. 10, pp. 213–232, Berlin: Springer-Verlag, 2003.

[226] P. Krishnan and J. S. Vitter, "Optimal prediction for prefetching in the worst case," *SIAM Journal on Computing*, vol. 27, pp. 1617–1636, December 1998.

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

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

[229] P.-A. Larson, "Performance analysis of linear hashing with partial expansions," *ACM Transactions on Database Systems*, vol. 7, pp. 566–587, December 1982.

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

[231] P. L. Lehman and S. B. Yao, "Efficient locking for concurrent operations on B-trees," *ACM Transactions on Database Systems*, vol. 6, pp. 650–570, December 1981.

[232] F. T. Leighton, "Tight bounds on the complexity of parallel sorting," *IEEE Transactions on Computers*, vol. C-34, pp. 344–354, Special issue on sorting, E. E. Lindstrom, C. K. Wong, and J. S. Vitter, eds., April 1985.

[233] C. E. Leiserson, S. Rao, and S. Toledo, "Efficient out-of-core algorithms for linear relaxation using blocking covers," *Journal of Computer and System Sciences*, vol. 54, no. 2, pp. 332–344, 1997.

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

[235] W. Litwin, "Linear hashing: A new tool for files and tables addressing," in *Proceedings of the International Conference on Very Large Databases*, pp. 212–223, October 1980.

[236] W. Litwin and D. Lomet, "A new method for fast data searches with keys," *IEEE Software*, vol. 4, pp. 16–24, March 1987.

[237] D. Lomet, "A simple bounded disorder file organization with good performance," *ACM Transactions on Database Systems*, vol. 13, no. 4, pp. 525–551, 1988.

[238] D. B. Lomet and B. Salzberg, "The hB-tree: A multiattribute indexing method with good guaranteed performance," *ACM Transactions on Database Systems*, vol. 15, no. 4, pp. 625–658, 1990.

[239] D. B. Lomet and B. Salzberg, "Concurrency and recovery for index trees," *VLDB Journal*, vol. 6, no. 3, pp. 224–240, 1997.

[240] T. Lukovszki, A. Maheshwari, and N. Zeh, "I/O-efficient batched range counting and its applications to proximity problems," *Foundations of Software Technology and Theoretical Computer Science*, pp. 244–255, 2001.

[241] A. Maheshwari and N. Zeh, "I/O-efficient algorithms for graphs of bounded treewidth," in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pp. 89–90, Washington, DC: ACM Press, January 2001.

[242] A. Maheshwari and N. Zeh, "I/O-Optimal Algorithms for Planar Graphs Using Separators," in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pp. 372–381, ACM Press, 2002.

[243] A. Maheshwari and N. Zeh, "A survey of techniques for designing I/O-efficient algorithms," in *Algorithms for Memory Hierarchies*, (U. Meyer, P. Sanders, and J. Sibeyn, eds.), ch. 3, pp. 36–61, Berlin: Springer-Verlag, 2003.

[244] A. Maheshwari and N. Zeh, "I/O-optimal algorithms for outerplanar graphs," *Journal of Graph Algorithms and Applications*, vol. 8, pp. 47–87, 2004.

[245] V. Mäkinen, G. Navarro, and K. Sadakane, "Advantages of backward searching — efficient secondary memory and distributed implementation of compressed suffix arrays," in *Proceedings of the International Symposium on Algorithms and Computation*, pp. 681–692, Springer-Verlag, 2004.

[246] U. Manber and G. Myers, "Suffix arrays: A new method for on-line string searches," *SIAM Journal on Computing*, vol. 22, pp. 935–948, October 1993.

[247] U. Manber and S. Wu, "GLIMPSE: A tool to search through entire file systems," in *Proceedings of the Winter USENIX Conference*, (USENIX Association, ed.), pp. 23–32, San Francisco: USENIX, January 1994.

[248] G. N. N. Martin, "Spiral storage: Incrementally augmentable hash addressed storage," Technical Report CS-RR-027, University of Warwick, March 1979.

[249] Y. Matias, E. Segal, and J. S. Vitter, "Efficient bundle sorting," in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pp. 839–848, San Francisco: ACM Press, January 2000.

[250] E. M. McCreight, "A space-economical suffix tree construction algorithm," *Journal of the ACM*, vol. 23, no. 2, pp. 262–272, 1976.

[251] E. M. McCreight, "Priority Search Trees," *SIAM Journal on Computing*, vol. 14, pp. 257–276, May 1985.

[252] K. Mehlhorn and U. Meyer, "External-memory breadth-first search with sublinear I/O," in *Proceedings of the European Symposium on Algorithms*, pp. 723–735, Springer-Verlag, 2002.

[253] H. Mendelson, "Analysis of extendible hashing," *IEEE Transactions on Software Engineering*, vol. SE-8, pp. 611–619, November 1982.

[254] U. Meyer, "External memory BFS on undirected graphs with bounded degree," in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pp. 87–88, Washington, DC: ACM Press, January 2001.

[255] U. Meyer, "On dynamic breadth-first search in external-memory," in *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, (Schloss Dagstuhl, Germany), pp. 551–560, Internationales Begegnungs- und Forschungszentrum für Informatik, 2008.

[256] U. Meyer, "On trade-offs in external-memory diameter approximation," in *Proceedings of the Scandinavian Workshop on Algorithm Theory*, (Gothenburg, Sweden), Springer-Verlag, July 2008.

[257] U. Meyer, P. Sanders, and J. Sibeyn, eds., *Algorithms for Memory Hierarchies*. Berlin: Springer-Verlag, 2003.

[258] U. Meyer and N. Zeh, "I/O-efficient undirected shortest paths," in *Proceedings of the European Symposium on Algorithms*, pp. 435–445, Springer-Verlag, 2003.

[259] U. Meyer and N. Zeh, "I/O-efficient undirected shortest paths with unbounded weights," in *Proceedings of the European Symposium on Algorithms*, Springer-Verlag, 2006.

[260] C. Mohan, "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*, pp. 392–405, August 1990.

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

[262] S. A. Moyer and V. Sunderam, "Characterizing concurrency control performance for the PIOUS parallel file system," *Journal of Parallel and Distributed Computing*, vol. 38, pp. 81–91, October 1996.

[263] J. K. Mullin, "Spiral storage: Efficient dynamic hashing with constant performance," *The Computer Journal*, vol. 28, pp. 330–334, July 1985.

[264] K. Munagala and A. Ranade, "I/O-complexity of graph algorithms," in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pp. 687–694, Baltimore: ACM Press, January 1999.

[265] S. Muthukrishnan, *Data Streams: Algorithms and Applications*. Vol. 1, issue 2 of *Foundations and Trends in Theoretical Computer Science*, Hanover, Mass.: now Publishers, 2005.

[266] G. Navarro, "Indexing text using the Ziv–Lempel trie," *Journal of Discrete Algorithms*, vol. 2, no. 1, pp. 87–114, 2004.

[267] G. Navarro and V. Mäkinen, "Compressed full-text indexes," *ACM Computing Surveys*, vol. 39, no. 1, p. 2, 2007.

[268] J. Nievergelt, H. Hinterberger, and K. C. Sevcik, "The grid file: An adaptable, symmetric multi-key file structure," *ACM Transactions on Database Systems*, vol. 9, pp. 38–71, 1984.

[269] J. Nievergelt and E. M. Reingold, "Binary search tree of bounded balance," *SIAM Journal on Computing*, vol. 2, pp. 33–43, March 1973.

[270] J. Nievergelt and P. Widmayer, "Spatial data structures: Concepts and design choices," in *Algorithmic Foundations of GIS*, (M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer, eds.), pp. 153–197, Springer-Verlag, 1997.

[271] M. H. Nodine, M. T. Goodrich, and J. S. Vitter, "Blocking for external graph searching," *Algorithmica*, vol. 16, pp. 181–214, August 1996.

[272] M. H. Nodine, D. P. Lopresti, and J. S. Vitter, "I/O overhead and parallel VLSI architectures for lattice computations," *IEEE Transactions on Communications*, vol. 40, pp. 843–852, July 1991.

[273] M. H. Nodine and J. S. Vitter, "Deterministic distribution sort in shared and distributed memory multiprocessors," in *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pp. 120–129, Velen, Germany: ACM Press, June–July 1993.

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

[275] P. E. O'Neil, "The SB-tree. An index-sequential structure for high-performance sequential access," *Acta Informatica*, vol. 29, pp. 241–265, June 1992.

[276] J. A. Orenstein, "Redundancy in spatial databases," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 294–305, Portland: ACM Press, June 1989.

[277] J. A. Orenstein and T. H. Merrett, "A class of data structures for associative searching," in *Proceedings of the ACM Conference on Principles of Database Systems*, pp. 181–190, ACM Press, 1984.

[278] M. H. Overmars, *The design of dynamic data structures*. 1983. Springer-Verlag.

[279] H. Pang, M. Carey, and M. Livny, "Memory-adaptive external sorts," in *Proceedings of the International Conference on Very Large Databases*, pp. 618–629, 1993.

[280] H. Pang, M. J. Carey, and M. Livny, "Partially preemptive hash joins," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, (P. Buneman and S. Jajodia, eds.), pp. 59–68, Washington, DC: ACM Press, May 1993.

[281] I. Parsons, R. Unrau, J. Schaeffer, and D. Szafron, "PI/OT: Parallel I/O templates," *Parallel Computing*, vol. 23, pp. 543–570, June 1997.

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

[283] D. Pfoser, C. S. Jensen, and Y. Theodoridis, "Novel approaches to the indexing of moving object trajectories," in *Proceedings of the International Conference on Very Large Databases*, pp. 395–406, 2000.

[284] F. P. Preparata and M. I. Shamos, *Computational Geometry*. Berlin: Springer-Verlag, 1985.

[285] O. Procopiuc, P. K. Agarwal, L. Arge, and J. S. Vitter, "Bkd-tree: A dynamic scalable kd-tree," in *Proceedings of the International Symposium on Spatial and Temporal Databases*, Santorini, Greece: Springer-Verlag, July 2003.

[286] S. J. Puglisi, W. F. Smyth, and A. Turpin, "Inverted files versus suffix arrays for locating patterns in primary memory," in *Proceedings of the International Symposium on String Processing Information Retrieval*, pp. 122–133, Springer-Verlag, 2006.

[287] N. Rahman and R. Raman, "Adapting radix sort to the memory hierarchy," in *Workshop on Algorithm Engineering and Experimentation*, Springer-Verlag, January 2000.

[288] R. Raman, V. Raman, and S. S. Rao, "Succinct indexable dictionaries with applications to encoding k-ary trees and multisets," in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pp. 233–242, ACM Press, 2002.

[289] S. Ramaswamy and S. Subramanian, "Path caching: A technique for optimal external searching," in *Proceedings of the ACM Conference on Principles of Database Systems*, pp. 25–35, Minneapolis: ACM Press, 1994.

[290] J. Rao and K. Ross, "Cache conscious indexing for decision-support in main memory," in *Proceedings of the International Conference on Very Large Databases*, (M. Atkinson *et al.*, eds.), pp. 78–89, Los Altos, Cal.: Morgan Kaufmann, 1999.

[291] J. Rao and K. A. Ross, "Making B^+-trees cache conscious in main memory," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, (W. Chen, J. Naughton, and P. A. Bernstein, eds.), pp. 475–486, Dallas: ACM Press, 2000.

[292] E. Riedel, G. A. Gibson, and C. Faloutsos, "Active storage for large-scale data mining and multimedia," in *Proceedings of the International Conference on Very Large Databases*, pp. 62–73, August 1998.

[293] J. T. Robinson, "The k-d-b-tree: A search structure for large multidimensional dynamic indexes," in *Proceedings of the ACM Conference on Principles of Database Systems*, pp. 10–18, ACM Press, 1981.

[294] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod, "Using the SimOS machine simulator to study complex computer systems," *ACM Transactions on Modeling and Computer Simulation*, vol. 7, pp. 78–103, January 1997.

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

[296] K. Salem and H. Garcia-Molina, "Disk striping," in *Proceedings of IEEE International Conference on Data Engineering*, pp. 336–242, Los Angeles, 1986.

[297] S. Šaltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez, "Indexing the positions of continuously moving objects," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, (W. Chen, J. Naughton, and P. A. Bernstein, eds.), pp. 331–342, Dallas: ACM Press, 2000.

[298] B. Salzberg and V. J. Tsotras, "Comparison of access methods for time-evolving data," *ACM Computing Surveys*, vol. 31, pp. 158–221, June 1999.

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

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

[301] V. Samoladas and D. Miranker, "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*, pp. 44–51, Seattle: ACM Press, June 1998.

[302] P. Sanders, "Fast priority queues for cached memory," *ACM Journal of Experimental Algorithmics*, vol. 5, no. 7, pp. 1–25, 2000.

[303] P. Sanders, "Reconciling simplicity and realism in parallel disk models," *Parallel Computing*, vol. 28, no. 5, pp. 705–723, 2002.

[304] P. Sanders, S. Egner, and J. Korst, "Fast concurrent access to parallel disks," *Algorithmica*, vol. 35, no. 1, pp. 21–55, 2002.

[305] J. E. Savage, "Extending the Hong-Kung model to memory hierarchies," in *Proceedings of the International Conference on Computing and Combinatorics*, pp. 270–281, Springer-Verlag, August 1995.

[306] J. E. Savage and J. S. Vitter, "Parallelism in space-time tradeoffs," in *Advances in Computing Research*, (F. P. Preparata, ed.), pp. 117–146, JAI Press, 1987.

[307] S. W. Schlosser, J. L. Griffin, D. F. Nagle, and G. R. Ganger, "Designing computer systems with MEMS-based storage," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 1–12, November 2000.

[308] K. E. Seamons and M. Winslett, "Multidimensional array I/O in Panda 1.0," *Journal of Supercomputing*, vol. 10, no. 2, pp. 191–211, 1996.

[309] B. Seeger and H.-P. Kriegel, "The buddy-tree: An efficient and robust access method for spatial data base systems," in *Proceedings of the International Conference on Very Large Databases*, pp. 590–601, 1990.

[310] M. Seltzer, K. A. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan, "File system logging versus clustering: A performance comparison," in *Proceedings of the Annual USENIX Technical Conference*, pp. 249–264, New Orleans, 1995.

[311] S. Sen, S. Chatterjee, and N. Dumir, "Towards a theory of cache-efficient algorithms," *Journal of the ACM*, vol. 49, no. 6, pp. 828–858, 2002.

[312] R. Shah, P. J. Varman, and J. S. Vitter, "Online algorithms for prefetching and caching on parallel disks," in *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pp. 255–264, ACM Press, 2004.

[313] R. Shah, P. J. Varman, and J. S. Vitter, "On competitive online read-many parallel disks scheduling," in *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, p. 217, ACM Press, 2005.

[314] E. A. M. Shriver, A. Merchant, and J. Wilkes, "An analytic behavior model for disk drives with readahead caches and request reordering," in *Procedings of ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, pp. 182–191, Madison, Wisc.: ACM Press, June 1998.

[315] E. A. M. Shriver and M. H. Nodine, "An introduction to parallel I/O models and algorithms," in *Input/Output in Parallel and Distributed Computer Systems*, (R. Jain, J. Werth, and J. C. Browne, eds.), ch. 2, pp. 31–68, Kluwer Academic Publishers, 1996.

[316] E. A. M. Shriver and L. F. Wisniewski, "An API for choreographing data accesses," Tech. Rep. PCS-TR95-267, Dept. of Computer Science, Dartmouth College, November 1995.

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

[318] J. F. Sibeyn, "External selection," *Journal of Algorithms*, vol. 58, no. 2, pp. 104–117, 2006.

[319] J. F. Sibeyn and M. Kaufmann, "BSP-like external-memory computation," in *Proceedings of the Italian Conference on Algorithms and Complexity*, pp. 229–240, 1997.

[320] R. Sinha, S. Puglisi, A. Moffat, and A. Turpin, "Improving suffix array locality for fast pattern matching on disk," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Vancouver: ACM Press, June 2008.

[321] B. Srinivasan, "An adaptive overflow technique to defer splitting in B-trees," *The Computer Journal*, vol. 34, no. 5, pp. 397–405, 1991.

[322] A. Srivastava and A. Eustace, "ATOM: A system for building customized program analysis tools," *ACM SIGPLAN Notices*, vol. 29, pp. 196–205, June 1994.

[323] S. Subramanian and S. Ramaswamy, "The P-range tree: A new data structure for range searching in secondary memory," in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pp. 378–387, ACM Press, 1995.

[324] R. Tamassia and J. S. Vitter, "Optimal cooperative search in fractional cascaded data structures," *Algorithmica*, vol. 15, pp. 154–171, February 1996.

[325] "TerraServer-USA: Microsoft's online database of satellite images," Available on the World-Wide Web at http://terraserver.microsoft.com/.

[326] R. Thakur, A. Choudhary, R. Bordawekar, S. More, and S. Kuditipudi, "Passion: Optimized I/O for parallel applications," *IEEE Computer*, vol. 29, pp. 70–78, June 1996.

[327] "Topologically Integrated Geographic Encoding and Referencing system, TIGER/Line 1992 datafiles," Available on the World-Wide Web at http://www.census.gov/geo/www/tiger/, 1992.

[328] S. Toledo, "A survey of out-of-core algorithms in numerical linear algebra," in *External Memory Algorithms and Visualization*, (J. Abello and J. S. Vitter, eds.), pp. 161–179, Providence, Rhode Island: American Mathematical Society Press, 1999.

[329] L. Toma and N. Zeh, "I/O-efficient algorithms for sparse graphs," in *Algorithms for Memory Hierarchies*, (U. Meyer, P. Sanders, and J. Sibeyn, eds.), ch. 5, pp. 85–109, Berlin: Springer-Verlag, 2003.

[330] TPIE User Manual and Reference, "The manual and software distribution," available on the web at http://www.cs.duke.edu/TPIE/, 1999.

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

[332] J. Vahrenhold and K. Hinrichs, "Planar point location for large data sets: To seek or not to seek," *ACM Journal of Experimental Algorithmics*, vol. 7, p. 8, August 2002.

[333] J. van den Bercken, B. Seeger, and P. Widmayer, "A generic approach to bulk loading multidimensional index structures," in *Proceedings of the International Conference on Very Large Databases*, pp. 406–415, 1997.

[334] 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, 1997.

[335] P. J. Varman and R. M. Verma, "An efficient multiversion access structure," *IEEE Transactions on Knowledge and Data Engineering*, vol. 9, pp. 391–409, May–June 1997.

[336] D. E. Vengroff and J. S. Vitter, "Efficient 3-D range searching in external memory," in *Proceedings of the ACM Symposium on Theory of Computing*, pp. 192–201, Philadelphia: ACM Press, May 1996.

[337] D. E. Vengroff and J. S. Vitter, "I/O-efficient scientific computation using TPIE," in *Proceedings of NASA Goddard Conference on Mass Storage Systems*, pp. II, 553–570, September 1996.

[338] P. Vettiger, M. Despont, U. Drechsler, U. Dürig, W. Häberle, M. I. Lutwyche, E. Rothuizen, R. Stutz, R. Widmer, and G. K. Binnig, "The 'Millipede' — more than one thousand tips for future AFM data storage," *IBM Journal of Research and Development*, vol. 44, no. 3, pp. 323–340, 2000.

[339] J. S. Vitter, "Efficient memory access in large-scale computation," in *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, pp. 26–41, Springer-Verlag, 1991. Invited paper.

[340] J. S. Vitter, *Notes*. 1999.

[341] J. S. Vitter and P. Flajolet, "Average-case analysis of algorithms and data structures," in *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, (J. van Leeuwen, ed.), ch. 9, pp. 431–524, Elsevier and MIT Press, 1990.

[342] J. S. Vitter and D. A. Hutchinson, "Distribution sort with randomized cycling," *Journal of the ACM*, vol. 53, pp. 656–680, July 2006.

[343] J. S. Vitter and P. Krishnan, "Optimal prefetching via data compression," *Journal of the ACM*, vol. 43, pp. 771–793, September 1996.

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

[345] J. S. Vitter and E. A. M. Shriver, "Algorithms for parallel memory I: Two-level memories," *Algorithmica*, vol. 12, no. 2–3, pp. 110–147, 1994.

[346] J. S. Vitter and E. A. M. Shriver, "Algorithms for parallel memory II: Hierarchical multilevel memories," *Algorithmica*, vol. 12, no. 2–3, pp. 148–169, 1994.

[347] J. S. Vitter and M. Wang, "Approximate computation of multidimensional aggregates of sparse data using wavelets," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 193–204, Philadelphia: ACM Press, June 1999.

[348] J. S. Vitter, M. Wang, and B. Iyer, "Data cube approximation and histograms via wavelets," in *Proceedings of the International ACM Conference on Information and Knowledge Management*, pp. 96–104, Washington, DC: ACM Press, November 1998.

[349] M. Wang, B. Iyer, and J. S. Vitter, "Scalable mining for classification rules in relational databases," in *Herman Rubin Festschrift*, Hayward, CA: Institute of Mathematical Statistics, Fall 2004.

[350] M. Wang, J. S. Vitter, L. Lim, and S. Padmanabhan, "Wavelet-based cost estimation for spatial queries," in *Proceedings of the International Symposium on Spatial and Temporal Databases*, pp. 175–196, Redondo Beach, Cal.: Springer-Verlag, July 2001.

[351] R. W. Watson and R. A. Coyne, "The parallel I/O architecture of the high-performance storage system (HPSS)," in *Proceedings of the IEEE Symposium on Mass Storage Systems*, pp. 27–44, September 1995.

[352] P. Weiner, "Linear pattern matching algorithm," in *Proceedings of the IEEE Symposium on Switching and Automata Theory*, pp. 1–11, 1973.

[353] K.-Y. Whang and R. Krishnamurthy, "Multilevel grid files — a dynamic hierarchical multidimensional file structure," in *Proceedings of the International Symposium on Database Systems for Advanced Applications*, pp. 449–459, World Scientific Press, 1992.

[354] D. E. Willard and G. S. Lueker, "Adding range restriction capability to dynamic data structures," *Journal of the ACM*, vol. 32, no. 3, pp. 597–617, 1985.

[355] I. H. Witten, A. Moffat, and T. C. Bell, *Managing Gigabytes: Compressing and Indexing Documents and Images*. Los Altos, Cal.: Morgan Kaufmann, 2nd ed., 1999.

[356] O. Wolfson, P. Sistla, B. Xu, J. Zhou, and S. Chamberlain, "DOMINO: Databases for moving objects tracking," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 547–549, Philadelphia: ACM Press, June 1999.

[357] D. Womble, D. Greenberg, S. Wheat, and R. Riesen, "Beyond core: Making parallel computer I/O practical," in *Proceedings of the DAGS Symposium on Parallel Computation*, pp. 56–63, June 1993.

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

[359] Y. Xia, S. Prabhakar, S. Lei, R. Cheng, and R. Shah, "Indexing continuously changing data with mean-variance tree," in *Proceedings of the ACM Symposium on Applied Computing*, pp. 52–57, ACM Press, March 2005.

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

[361] S. B. Zdonik and D. Maier, eds., *Readings in object-oriented database systems*. Morgan Kauffman, 1990.

[362] N. Zeh, *I/O-Efficient Algorithms for Shortest Path Related Problems*. PhD thesis, School of Computer Science, Carleton University, 2002.

[363] W. Zhang and P.-A. Larson, "Dynamic memory adjustment for external mergesort," in *Proceedings of the International Conference on Very Large Databases*, pp. 376–385, 1997.

[364] B. Zhu, "Further computational geometry in secondary memory," in *Proceedings of the International Symposium on Algorithms and Computation*, pp. 514–522, Springer-Verlag, 1994.

[365] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *IEEE Transactions on Information Theory*, vol. 24, pp. 530–536, September 1978.