SIAM J. COMPUT.
Vol. 36, No. 2, pp. 394–410

EFFICIENT BUNDLE SORTING*

YOSSI MATIAS†, ERAN SEGAL‡, AND JEFFREY SCOTT VITTER§

Abstract. Many data sets to be sorted consist of a limited number of distinct keys. Sorting such data sets can be thought of as bundling together identical keys and having the bundles placed in order; we therefore denote this as *bundle sorting*. We describe an efficient algorithm for bundle sorting in external memory, which requires at most $c(N/B)\log_{M/B} k$ disk accesses, where N is the number of keys, M is the size of internal memory, k is the number of distinct keys, B is the transfer block size, and $2 < c < 4$. For moderately sized k, this bound circumvents the $\Theta((N/B)\log_{M/B}(N/B))$ I/O lower bound known for general sorting. We show that our algorithm is optimal by proving a matching lower bound for bundle sorting. The improved running time of bundle sorting over general sorting can be significant in practice, as demonstrated by experimentation. An important feature of the new algorithm is that it is executed "in-place," requiring no additional disk space.

Key words. sorting, external memory, bundle sorting, algorithms

AMS subject classification. 68W01

DOI. 10.1137/S0097539704446554

1. Introduction.

Sorting is a frequent operation in many applications. It is used not only to produce sorted output, but also in many sort-based algorithms such as grouping with aggregation, duplicate removal, and sort-merge join, as well as set operations including union, intersect, and except [Gra93, IBM95]. In this paper, we identify a common external memory sorting problem, present an algorithm to solve it while circumventing the lower bound for general sorting for this problem, prove a matching lower bound for our algorithm, and demonstrate the improved performance through experiments.

External merge sort is the most commonly used algorithm for large-scale sorting. It has a run formation phase, which produces sorted runs, and a merge phase, which merges the runs into sorted output. Its running time, as in most external memory algorithms, is dominated by the number of input/outputs (I/Os) performed, which is $O((N/B)\log_{M/B}(N/B))$, where N is the number of keys, M is the size of internal memory, and B is the transfer block size. It was shown in [AV88] (see also [Vit99]) that there is a matching lower bound within a constant factor.

The number of passes over the sequence performed by sorting algorithms is $\lceil \log_{M/B}(N/B) \rceil$ in the worst case. When the available memory is large enough compared to the size of the sequence, the sorting can be performed in one or two passes over the sequence (see [ADADC+97] and references therein). However, there are many

*Received by the editors November 22, 2004; accepted for publication (in revised form) February 24, 2006; published electronically June 23, 2006. A preliminary version of this paper was presented at the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms [MSV00].

http://www.siam.org/journals/sicomp/36-2/44655.html

†School of Computer Science, Tel-Aviv University, Tel-Aviv 69978 Israel (matias@cs.tau.ac.il). This author's work was supported in part by an Alon Fellowship, by the Israel Science Foundation founded by the Academy of Sciences and Humanities, and by the Israeli Ministry of Science.

‡Department of Computer Science, Stanford University, Stanford, CA 94305 (eran@cs.stanford.edu). Much of this author's work was done while the author was at Tel-Aviv University.

§Department of Computer Science, Purdue University, West Lafayette, IN 47907-2066 (jsv@purdue.edu). Much of this author's work was done while the author was on sabbatical at I.N.R.I.A. in Sophia Antipolis, France, and was supported in part by Army Research Office MURI grant DAAH04–96–1–0013 and DAAD19–01–1–0725, and by the National Science Foundation research grant CCR–9522047.

settings in which the available memory is moderate, at best. For instance, in multithreading and multiuser environments, an application, process, or thread which may execute a sorting program might be allocated only a small fraction of the machine memory. Such settings may be relevant to anything from low-end servers to high-end decision support systems. For moderate size memory, $\log_{M/B}(N/B)$ may become large enough to imply a significant number of passes over the data. As an example, consider the setting $N = 256$ GB, $B = 128$ kB, and $M = 16$ MB. Then we have $\log_{M/B}(N/B) = 3$, and the number of I/Os per disk block required by merge sort is at least 6. For smaller memory allocations, the I/O costs will be even greater.

Our contributions. Data sets that are to be sorted often consist of keys taken from a bounded universe. This fact is well exploited in main memory algorithms such as counting sort and radix sort, which are substantially more efficient than general sort. In this paper we consider the extent to which a limit, k, on the number of distinct keys can be exploited to obtain more effective sorting algorithms in external memory on massive data sets, where the attention is primarily given to the number of I/Os. Sorting such data sets can be thought of as bundling together identical keys and having the bundles placed in order; we therefore denote this as *bundle sorting*. It is similar to partial sorting, which was identified by Knuth [Knu73] as an important problem. While many algorithms are given for partial sorting in main memory, to the best of our knowledge, there exist no efficient algorithms for solving the problem in external memory. As we shall see, bundle sorting can be substantially more efficient than general sorting.

A key feature of bundle sorting is that the number of I/Os performed per disk block depends solely on the number k of distinct keys. Hence, in sorting applications in which the number of distinct keys is constant, the number of I/Os performed per disk block remains constant for any data set size. In contrast, merge sort or other general sorting algorithms will perform more I/Os per disk block as the size of the data set increases. In settings in which the size of the data set is large this can be significant. In the example given earlier, six I/Os per data block are needed to sort in the worst case. For some constant $k < 100$, bundle sorting performs only two I/Os per disk block, and for some constant $k < 10000$, only four I/Os per disk block, regardless of the size of the data set.

The algorithm that we present requires at most $3 \log_{M/B} k$ passes over the sequence. It performs the sorting in-place, meaning that the input data set can be permuted as needed without using any additional working space in external memory. When the number k of distinct keys is less than N/B, our bundle sorting algorithm circumvents the lower bound for general sorting. The lower bound for general sorting is derived by a lower bound for permuting the input sequence, which is an easier problem than general sorting. In contrast to general sorting, bundle sorting is not harder than permuting; rather than requiring that a particular key be moved to a specific location, it is required that the key be moved to a location within a specified range, which belongs to its bundle. This so-called bundle-permutation consists of a set of permutations, and implementing bundle-permutation can be done more efficiently than implementing a particular permutation.

For cases in which $k \ll N/B$, the improvement in the running time of bundle sorting over general sorting algorithms can be significant in practical sorting settings, as supported by our experimentation done on U.S. Census data and on synthetic data. In fact, the number of passes over the sequence executed by our algorithm does not depend at all on the size of the sequence, in contrast to general sorting algorithms.

To complement the algorithmic component, we prove a matching lower bound

for bundle sorting. In particular, we show that the number of I/Os required in the worst case to sort N keys consisting of k distinct key values is $\Omega((N/B)\log_{M/B} k)$. This lower bound is realized by proving lower bounds on two problems that are both easier than bundle sorting, and the combination of the lower bounds gives the desired result. The first special case is bundle-permutation, and the second is a type of matrix transposition. Bundle-permutation is the special case of bundle sorting in which we know the distribution of key values beforehand, and thus it is easier than bundle sorting for much the same reason that permuting is easier than general sorting. The other special case of bundle sorting is a type of matrix transposition, in which we transpose a $k \times N/k$ matrix, but the final order of the elements in each row is not important. This problem is a special case of bundle sorting of N keys consisting of exactly N/k records for each of k different keys and is thus easier than bundle sorting. Interestingly, these two problems, when combined, capture the difficulty of bundle sorting.

Our bundle sorting algorithm is based on a simple observation: If the available memory, M, is at least kB, then we can sort the data in three passes over the sequence, as follows. In the first pass, we count the size of each bundle. After this pass we know the range of blocks in which each bundle will reside upon termination of the bundle sorting. The first block from each such range is loaded to main memory. The loaded blocks are scanned concurrently, while swapping keys so that each block is filled only with keys belonging to its bundle. Whenever a block is fully scanned (i.e., it contains only keys belonging to its bundle), it is written back to disk, and the next block in its range is loaded. In this phase, each block is loaded exactly once (except for at most k blocks in which the ranges begin), and the total number of accesses over the input sequence in the entire algorithm is hence 3. Whenever memory is insubstantial to hold the k blocks in memory, we group bundles together into M/B superbundles, implementing the algorithm to sort the superbundles to M/B subsequences and reiterate within each subsequence, incurring a total of $\log_{M/B} k$ iterations over the sequence to complete the bundle sorting.

There are many applications and settings in which bundle sorting may be applied, resulting in a significant speed-up in performance. For instance, any application that requires partial sorting or partitioning of a data set into value independent buckets can take advantage of bundle sorting since the number of buckets (k in bundle sorting) is small, thus making bundle sorting very appealing. Another example would be *accelerating sort join computation for suitable data sets*: Consider a join operation between two large relations, each having a moderate number of distinct keys; then our bundle sorting algorithm can be used in a sort join computation, with performance improvement over the use of general sort algorithm.

Finally, we consider a more performance-sensitive model that, rather than just counting the number of I/Os as a measurement for performance, differentiates between a sequential I/O and a random I/O and assigns a reduced cost for sequential I/Os. We study the tradeoffs that occur when we apply bundle sorting in this model, and show a simple adaptation of bundle sorting that results in an optimal performance. In this sense, we also present a slightly different algorithm for bundle sorting, which is more suitable for sequential I/Os.

The rest of the paper is organized as follows. In section 2 we explore related work. In section 3 we describe the external memory model in which we will analyze our algorithm and prove the lower bound. Section 4 presents our algorithm for bundle sorting along with the performance analysis. In section 5 we prove the lower bound for external bundle sorting. In section 6 we consider a more performance-sensitive

model, which takes into account a reduced cost for sequential I/Os and shows the modifications in our bundle sorting algorithm required to achieve an optimal algorithm in that model. Section 7 describes the experiments we conducted, and section 8 is our conclusions.

2. Related work. External memory sorting is an extensively researched area. Many efficient in-memory sorting algorithms have been adapted for sorting in external memory, such as merge sort, and much of the recent research in external memory sorting has been dedicated to improving the run time performance. Over the years, numerous authors have reported the performance of their sorting algorithms and implementations (cf. [Aga96, BBW86, BGK90]). We note a recent paper [ADADC+97] that shows external sorting of 6 GB of data in under one minute on a network of workstations. For the problem of bundle sorting where $k < N/B$ we note that our algorithm will reduce the number of I/Os that all these algorithms perform and hence can be utilized in benchmarks. We also consider a more performance-sensitive model of external memory, in which rather than just counting the I/Os for determining the performance, there is a reduced cost for sequential I/Os compared to random access I/Os. We study the tradeoffs there, and show the adaptation in our bundle sorting algorithm to arrive at an optimal algorithm in that model. We also note that another recent paper [ZL98] shows in detail how to improve the merge phase of the external merge sort algorithm, a phase that is completely avoided by using our in-place algorithm.

In the general framework of external memory algorithms, Aggarwal and Vitter showed a lower bound of $\Omega((N/B)\log_{M/B}(N/B))$ on the number of I/Os needed in the worst case for sorting [AV88, Vit99]. In contrast, since our algorithm relies on the number k of distinct keys for its performance, we are able to circumvent this lower bound when $k \ll N/B$. Moreover, we prove a matching lower bound for bundle sorting, which shows that our algorithm is optimal.

Finally, sorting is used not only to produce sorted output, but also in many sort-based algorithms such as grouping with aggregation, duplicate removal, and sort-merge join, as well as set operations including union, intersect, and except [Gra93, IBM95]. In many of these cases the number of distinct keys is relatively small, and hence bundle sorting can be used for improved performance. We identify important applications for bundle sorting, but note that since sorting is such a common procedure, there are probably many more applications for bundle sorting that we did not consider.

3. External memory model. In our main bundle sorting algorithm and in the lower bound that we prove, we use the external memory model from Aggarwal and Vitter [AV88] (see also [Vit99]). The model is as follows. We assume that there is a single central processing unit, and we model secondary storage as a generalized random-access magnetic disk. (For completeness, the model is also extended to the case in which the disk has some parallel capabilities.) The parameters are

$$\begin{aligned} N &= \#\text{ records to sort,} \\ M &= \#\text{ records that can fit into internal memory,} \\ B &= \#\text{ records transferred in a single block,} \\ D &= \#\text{ blocks that can be transferred concurrently,} \end{aligned}$$

where $1 \le B \le M/2$, $M < N$, and $1 \le D \le \lfloor M/B \rfloor$. For brevity we consider only the case of $D = 1$, which corresponds to a single conventional disk.

The parameters N, M, and B are referred to as the *file size*, *memory size*, and *transfer block size*, respectively. Each block transfer is allowed to access any contiguous group of B records on the disk. We will consider the case where $D = 1$, meaning that there is no disk parallelism. Performance in this model is measured by the number of I/O accesses performed where the cost of all I/Os is identical. In section 6 we consider a more performance-sensitive model in which we differentiate between costs of sequential and random-access I/Os and assign a reduced cost for sequential I/Os.

4. External bundle sorting algorithm. In this section we present our bundle sorting algorithm, which sorts in-place a sequence that resides on disk and contains k distinct keys. We start by defining the bundle sorting problem:

Input: *A sequence of keys* $\{a_1, a_2, \ldots, a_n\}$ *from an ordered universe* U *of size* k.

Output: *A permutation* $\{a'_1, a'_2, \ldots, a'_n\}$ *of the input sequence such that* $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

In our algorithm, it will be easy, and with negligible overhead, to compute and use an order-preserving mapping from U to $\{1, \ldots, k\}$; we discuss the implementation details of this function in section 4.2. This enables us to consider the problem at hand as an integer sorting problem in which the keys are taken from $\{1, \ldots, k\}$. Hence, we assume that $U = \{1, \ldots, k\}$.

We use the external memory model from section 3, where performance is determined by the number of I/Os performed. Our goal is to minimize the number of disk I/Os. In section 6 we consider a more performance-sensitive model in which rather than simply counting I/Os as a measurement of performance, we differentiate between a sequential I/O and a random I/O and assign a reduced cost to sequential I/Os. We show the necessary modifications to the bundle sorting presented in this section required to achieve an optimum in that model.

4.1. $\{1, \ldots, k\}$ integer sorting. We start by presenting "one-pass sorting"—a procedure that sorts a sequence into $\mu = \lfloor M/B \rfloor$ distinct keys. It will be used by our bundle sorting algorithm to perform one iteration that sorts a chunk of data blocks into μ ranges of keys.

The general idea is this: Initially we perform one pass on the sequence, loading one block of size B at a time, in which we count the number of appearances of each of the μ distinct keys in the sequence. Next, we keep in memory μ blocks and a pointer for each block, where each block is of size B. Using the count pass, we initialize the μ blocks, where the ith block is loaded from the exact location in the sequence where keys of type i will start residing in the sorted sequence. We set each block pointer to point to the first key in its block. When the algorithm runs, the ith block pointer is advanced as long as it encounters keys of type i. When a block pointer is "stuck" on a key of type j, it waits for the jth block pointer until it too is stuck (this will happen since a block pointer yields only to keys of its block), in which case a swap is performed and at least one of the two block pointers may continue to advance. When any of the μ block pointers reaches the end of its block, we write that block back to disk to the exact location from which it was loaded, and load the next contiguous block from disk into memory (and of course set its block pointer again to the first key in the block). We finish with each of the μ blocks upon crossing the boundaries of the next adjacent block. The algorithm terminates when all blocks are done with. The following is a pseudocode of the algorithm. See also Figure 1.

ONE-PASS SORTING ALGORITHM.

procedure one-pass-sort $(sequence, k, M, B)$

for $i = 0$ to $\lfloor M/B \rfloor$

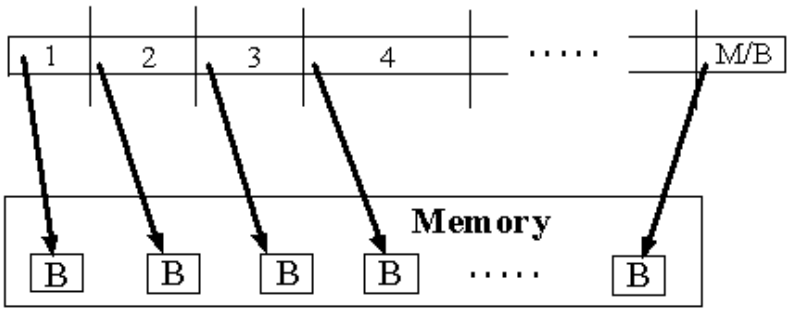

FIG. 1. *Initialization of the M/B blocks in the One-pass sorting algorithm. After the counting pass, we know where the sorted blocks reside, and load blocks from these locations. Swaps are performed in memory. When any of the blocks is full, we write it to disk to the location from which it was loaded, and load the next block from disk.*

```
        current_block = ith block of sequence
        for j = 0 to B of current_block
            Count[sequence[i · B + j]]++
    Count[k + 1] = M
    for i = 1 to k
        μ[i] = block at position Count[i] from sequence
        μ_block_pointer[i] = 0
        μ_global_pointer[i] = Count[i]
    blocks_done = 0
    while blocks_done < k
        for i = 1 to k
            if μ_global_pointer[i] < Count[i + 1]
                while μ_block_pointer[i] < B and μ[i][μ_block_pointer[i]] = i
                    μ_block_pointer[i]++
                    μ_global_pointer[i]++
                stuck[i] = (μ_block_pointer[i] < B and μ_global_pointer[i] < Count[i + 1])
                stuck_value = μ[i][μ_block_pointer[i]]
                if stuck[i] and stuck[stuck_value] then
                    swap μ[i][μ_block_pointer[i]] with
                    μ[stuck_value][μ_block_pointer[stuck_value]]
                if μ_block_pointer[i] = B
                    Write μ[i] as a block to the location from which it was loaded
                    μ[i] = block at position μ_global_pointer[i] from sequence
                    μ_block_pointer[i] = 0
                if μ_global_pointer[i] = Count[i + 1]
                    blocks_done++;
```

LEMMA 4.1. *Let S be a sequence of N keys from $\{1, \ldots, \mu\}$, let B be the transfer block size, and let M be the available memory such that $M \geq \mu B$. Then the sequence can be sorted in-place using the procedure "one-pass sorting" with a total of $\lceil 3N/B + 2M/B \rceil$ I/Os.*

Proof. We first show that the algorithm indeed sorts the input sequence. The algorithm allocates one pointer in memory for each of the μ distinct keys, and the ith

such pointer writes only contiguous blocks of records whose keys consist solely of the ith key. Thus, to show that the sequence is sorted by "one-pass sorting," it suffices to show that the algorithm terminates and that, upon termination, the ith pointer writes its blocks in a physical location that precedes the blocks written by any j pointer for $j > i$. The ordering between the pointers is ensured by setting the contiguous block of the ith pointer to write to the exact location where keys of its type should reside in the sorted sequence. This location is derived from the first pass in which we count the number of appearances of each of the μ distinct keys. Termination is guaranteed, since at each step at least one of the pointers encounters keys of its type, or a swap will be performed and at least one of the pointers can proceed. Note that such a swap will always be possible, since if the ith pointer is "stuck" on a key of type j, then the jth pointer will necessarily get stuck at some step. Since at each step one of the keys is written and there are N keys, the algorithm will terminate.

For computing the number of I/Os, note that the first counting pass reads each block once and thus requires $\lceil N/B \rceil$ I/Os. All the μ pointers combined read and write each block once, adding another $\lceil 2N/B \rceil$ I/Os. Finally, if the number of appearances of each distinct key is not an exact multiple of B, then every pair of consecutive pointers may overlap by one block at the boundaries, thus requiring an additional $\lceil 2M/B \rceil$ I/Os. □

We now present the complete integer sorting algorithm. We assume that the sequence contains keys in the range $1, \ldots, k$, where k is the number of distinct keys. In section 4.2 we discuss the adaptation needed if the k distinct keys are not from this integer range. We use the above one-pass sorting procedure. The general idea is this: We initially perform one sorting iteration in which we sort the sequence into $k' = \lfloor M/B \rfloor$ keys. We select a mapping function f such that for all $1 \le i \le k$ we have $f(i) = \lceil ik'/k \rceil$, and we apply f to every key when the key is examined. This ensures that we are actually in the range of $1, \ldots, k'$. Moreover, it will create sorted buckets on disk such that the number of distinct keys in each of the buckets is roughly k/k'. We repeat this procedure recursively for each of the sorted blocks obtained in this iteration until the whole sequence is sorted. Each sorting iteration is done by calling the procedure for one-pass sorting. We give a pseudocode of the algorithm below, followed by an analysis of its performance.

THE INTEGER SORTING ALGORITHM.

```
procedure sort (sequence, k, M, B)
    k' = max(⌊M/B⌋, 2)      // compute k'
    if (k > 2) then
        call one-pass sorting (sequence, k', M, B)
        for i = 1 to k'
            bucket = the ith bucket sorted
            call sort (bucket, ⌈k/k'⌉, M, B)
```

THEOREM 4.1. *Let S be a sequence of N keys from $\{1, \ldots, k\}$, let M be the available memory, and let B be the transfer block size. A sequence residing on disk can be sorted in-place using the bundle sorting algorithm, while the number of I/Os is at most*

$$\left\lceil \frac{3N}{B} \log_{\lfloor M/B \rfloor} k \right\rceil + 4k \left\lfloor \frac{M}{B} \right\rfloor.$$

Proof. We first show that bundle sorting results in a sorting of the input sequence. Since we map each key i to $\lceil ik'/k \rceil$, it follows from the correctness of the one-pass

sorting that, after the first call to one-pass sorting, the sequence will be sorted such that for all i, keys in the range $\{\lceil (i-1)k'/k \rceil + 1, \ldots, \lceil ik'/k \rceil\}$ precede all keys greater than $\lceil ik'/k \rceil$. Each of the resulting range of keys is then recursively sorted. After at most $\log_{\lfloor M/B \rfloor} k$ recursive iterations, the number of distinct keys will be less than k', in which case the one-pass sorting will result in a full sorting of the sequence.

For the number of I/Os, we can view the bundle sorting algorithm as proceeding in levels of recursion, where at the first level of recursion bundle sorting is applied once, at the second level it is applied k' times, and at the ith level it is applied k'^{i-1} times. The total number of levels of recursion is $\log_{\lfloor M/B \rfloor} k$. Even though at the ith recursive level bundle sorting is applied k'^{i-1} times, each application is given a disjoint sequence shorter than N as input, and all applications of bundle sorting at the same recursive level cover the N input sequence exactly once. Thus, the counting pass of all applications at the same recursive level will still require $\lceil N/B \rceil$ I/Os, and all such applications will result in a read and write of each block, incurring an additional $\lceil 2N/B \rceil$ I/Os. Finally, since in general the number of distinct keys will not be a multiple of B, there might be an overlap of at most one block between every pair of consecutive pointers in one-pass sorting. Thus, we require an additional $2\lfloor M/B \rfloor$ I/Os for each application of one-pass sorting. One-pass sorting is called once for the first level of recursion, k' for the second level, and k'^{i-1} for the ith level, and thus the total number of times that one-pass sorting is called is $\frac{k'^{1+log_{k'} k}-1}{k'-1} = \frac{k'k-1}{k'-1} \leq 2k$. Hence, we add an additional $4k\lfloor M/B \rfloor$ I/Os, which results in the desired bound on the number of I/Os. ∎

4.2. General bundle sorting. In section 4.1 we assumed that the input was in the range $1, \ldots, k$, where k is the number of distinct keys in the sequence. We now discuss how to construct a mapping function when the input is not in this range.

In the simple case where the input is from a universe that is not ordered (i.e., the sorting is done just to cluster keys together), we can simply select any universal hash function as our mapping function. This ensures that the number of distinct keys that will be distributed to each bucket is fairly equal and that our algorithm performs without any loss of performance.

For the general case we assume that the input is from an ordered universe U and consists of k distinct keys. We show how to construct a mapping function from U to $1, \ldots, k$. More specifically, we need a way to map the keys into the range $[1, M/B]$ at every application of the one-pass sorting procedure. A solution to this mapping is to build an M/B-ary tree, whose leaves are the k distinct keys in sorted order and each internal node stores the minimum and the maximum values of its M/B children. Each application of one-pass sorting in integer sorting corresponds to an internal node in the tree (starting from the root) along with its children, and so the tree provides the appropriate mapping. This is because in each run of one-pass sorting the keys are within the range of the minimum and maximum values stored in the corresponding internal node, and the mapping into $1, \ldots, M/B$ is done according to the ranges of the internal node's children.

Constructing the sorted leaves can be done via count sort, in which we are given a sequence of size N with k distinct keys and we need to produce a sorted list of the k distinct keys and their counts. An easy way to do count sort is via merge sort, in which identical keys are combined together (and their counts summed) whenever they appear together. In each merge sort pass, the output run will never be longer than k/B blocks. Initially, the runs contain at most M/B blocks. After $\log_{M/B}(k/B)$ passes, the runs will be of length at most k/B blocks, and after that point the number

of runs decreases geometrically, and the running time is thus linear in the number of I/Os. The rest of the tree can be computed in at most one extra scan of the leaves-array and lower order postprocessing. We can show the following.

LEMMA 4.2 (see [WVI98]). *A sequence of size N consisting of k distinct keys can be count-sorted, using a memory of size M and block transfer size B, within an I/O bound of*

$$\frac{2N}{B}\log_{M/B}\frac{k}{B}.$$

An interesting observation is that by adding a count to each leaf representing its frequency in the sequence, and a count to each internal node which is the sum of the counts of its children, we can eliminate the count phase of the one-pass sorting procedure in the integer sorting algorithm. Thus, the general bundle sorting algorithm is as follows. Initially, we use count sort and produce the tree. We now traverse the tree, and on each internal node we call one-pass sorting, where the mapping function is simply the ranges of values of the node's M/B children. By combining Theorem 4.1 and Lemma 4.2 we can prove the bound for general bundle sorting.

THEOREM 4.2. *Let S be a sequence of size N, which consists of k distinct keys; let M be the available memory; and let B be the transfer block size. Then we can in-place sort S using the bundle sorting algorithm, while the number of I/Os is at most*

$$\frac{2N}{B}\left(\log_{\lfloor M/B\rfloor} k + \log_{\lfloor M/B\rfloor}\frac{k}{B}\right).$$

For all $k < B^2$, this bound would be better than the bound for integer sorting. Note that we can traverse the tree in either BFS (breadth first search) or DFS (depth first search). If we choose BFS, the sorting will be done concurrently, and we get an algorithm that gradually refines the sort. If we choose DFS, we get fully sorted items quickly, while the rest of the items are left completely unsorted. The overhead we incur by using the mapping will be in memory, where we now have to perform a search over the M/B children of the internal node that we are traversing in order to determine the mapping of each key into the range $1,\ldots,M/B$. Using a simple binary search over the ranges, the overhead will be an additional $\log_2(M/B)$ memory operations per key.

5. Lower bound for external bundle sorting. In this section we present a lower bound for the I/O complexity of bundle sorting. We let k be the number of distinct keys, M be the available memory, N be the size of the sequence, and B be the transfer block size. We then differentiate between the following two cases:

1. $k/B = B^{\Omega(1)}$ or $M/B = B^{\Omega(1)}$. We prove the lower bound for this case by proving a lower bound on bundle permutation, which is an easier problem than bundle sorting.
2. $k/B = B^{o(1)}$ and $M/B = B^{o(1)}$. We prove the lower bound for this case by proving a lower bound on a special case of matrix transposition, which is easier than bundle sorting.

Lower bound using bundle-permutation. We assume that $k/B = B^{\Omega(1)}$ or $M/B = B^{\Omega(1)}$ and use a similar approach as in the lower bound for general sorting of Aggarwal and Vitter [AV88] (see also [Vit99]). They proved the lower bound on the problem of computing an arbitrary permutation, which is easier than sorting. Bundle sorting is not necessarily harder than computing an arbitrary permutation, since

the output sequence may consist of one out of a set of permutations, denoted as a bundle-permutation. A *bundle-permutation* is an equivalence class of permutations, where two permutations can be in the same class if one can be obtained from the other by permuting within bundles. Computing a permutation from an arbitrary bundle-permutation, which we will refer to as the bundle-permutation problem, is easier than bundle sorting.

LEMMA 5.1. *Under the assumption that $k/B = B^{\Omega(1)}$ or $M/B = B^{\Omega(1)}$, the number of I/Os required in the worst case for sorting N data items of k distinct keys, using a memory of size M and block transfer size B, is*

$$\Omega\left(\frac{N}{B}\log_{M/B} k\right).$$

Proof. Given a sequence of N data items consisting of k bundles of sizes $\alpha_1, \alpha_2, \ldots, \alpha_k$, the number of distinct bundle-permutations is

$$\frac{N!}{\alpha_1! \cdot \alpha_2! \cdot \cdots \cdot \alpha_k!} \geq \frac{N!}{\left(\left(\frac{N}{k}\right)!\right)^k};$$

the inequality is obtained using a convexity argument.

For the bundle-permutation problem, for each $t \geq 0$ we measure the number of distinct orderings that are realizable by at least one sequence of t I/Os. The value of t for which the number of distinct orderings first exceeds the minimum orderings needed to be considered is a lower bound on the worst-case number of I/Os needed for the bundle-permutation problem and thus on the bundle sorting on disks.

Initially, the number of different permutations defined is 1. We consider the effect of an output operation. There can be at most $N/B + t - 1$ full blocks before the tth output, and hence the tth output changes the number of permutations generated by at most a multiplicative factor of $N/B+t$, which can be bounded trivially by $N \log N$.

For an input operation, we consider a block of B records input from a specific block on disk. The B data keys in the block can intersperse among the M keys in the internal memory in at most $\binom{M}{B}$ ways, so that the number of realizable orderings increases by a factor of $\binom{M}{B}$. If the block has never before resided in internal memory, the number of realizable orderings increases by an extra factor of $B!$, since the keys in the block can be permuted among themselves. This extra contribution can occur only once for each of the N/B original blocks. Hence, the number of distinct orderings that can be realized by some sequence of t I/Os is at most

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

We want to find the minimum t for which the number of realizable orderings exceeds the minimum orderings required. Hence we have

$$(B!)^{N/B}\left(N \log N \binom{M}{B}\right)^t \geq \frac{N!}{\left(\left(\frac{N}{k}\right)!\right)^k}.$$

Taking the logarithm and applying Stirling's formula, with some algebraic manipulations, we get

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

By solving for t, we get

$$\text{number of IOs} = \Omega\left(\frac{N}{B}\log_{M/B}\frac{k}{B}\right).$$

Recall that we assume either $k/B = B^{\Omega(1)}$ or $M/B = B^{\Omega(1)}$. In either case, it is easy to see that $\log_{M/B}(k/B) = \Theta(\log_{M/B} k)$, which gives us the desired bound. □

Lower bound using a special case of matrix transposition. We now assume that $k/B = B^{o(1)}$ and $M/B = B^{o(1)}$ (the case not handled earlier) and prove a lower bound on a special case of matrix transposition, which is easier than bundle sorting. Our proof proceeds under the normal assumption that the records are treated indivisibly and that no compression of any sort is utilized.

LEMMA 5.2. *Under the assumption that $k/B = B^{o(1)}$ and $M/B = B^{o(1)}$, the number of I/Os required in the worst case for sorting N data items of k distinct keys, using a memory of size M block transfer size B, is*

$$\Omega\left(\frac{N}{B}\log_{M/B} k\right).$$

Proof. Consider the problem of transposing a $k \times N/k$ matrix, in which the final order of the elements in each row is not important. More specifically, let us assume that the elements of the matrix are originally in column-major order. The problem is to convert the matrix into row-major order, but the place in a row to which the element goes can be arbitrary, as long as it is transferred to the proper row. Each element that ends up in row i can be thought of as having the same key i. This problem is a special case of sorting N keys consisting of exactly N/k records for each of the k distinct keys. Hence, this problem is easier than bundle sorting. We now prove a lower bound for this problem of

$$\Omega\left(\frac{N}{B}\log_{M/B}\min(k, B)\right)$$

I/Os. Under our assumption that $k/B = B^{o(1)}$, this proves the desired bound for bundle sorting.

We can assume that $k \le N/B$, since otherwise bundle sorting can be executed by using any general sorting algorithm. We assume, without loss of generality, by the assumption of the indivisibility of records, that there is always exactly one copy of each record, and it is either on disk or in memory but not in both. At time t, let X_{ij} for $1 \le i \le k$ and $1 \le j \le N/B$ be the number of elements in the jth block on disk that need to end up on the ith row of the transposed matrix. At time t, let Y_i be the number of elements currently in internal memory that need to go on the ith row in the transposed matrix. We use the potential function $f(x) = x \log x$ for all $x \ge 0$. Its value at $x = 0$ is $f(0) = 0$. We define the overall potential function POT to be

$$POT = \sum_{i,j} f(X_{ij}) + \sum_i f(Y_i).$$

When the algorithm terminates, we have $Y_i = 0$ for all i, and the final value of potential POT is

$$\frac{N}{B}(B \log B) + 0 = N \log B.$$

If $k < B$, the initial potential is

$$\frac{N}{B} k \left(\frac{B}{k} \log \frac{B}{k} \right) = N \log \frac{B}{k},$$

and the initial potential is 0 otherwise (if $k \geq B$).

Note that our potential function satisfies

$$f(a+b) = (a+b)\log(a+b) \geq f(a) + f(b)$$

for all $a, b \geq 0$. Consider an output operation that writes a complete block of size B from memory to disk. If we write x_i records that need to go to the ith row and there are y_i such records in memory, then the change in potential is $\sum_i \big(f(x_i) + f(y_i) - f(x_i + y_i)\big) \leq 0$. Hence, output operations can only decrease the potential, and thus we need to consider only how much an input operation increases the potential.

If we read during an input operation a complete block of B records that contains x_i records that need to go to the ith row and there are y_i such records already in memory, then the change in the potential is

$$\sum_{1 \leq i \leq k} \left(f(x_i + y_i) - f(x_i) - f(y_i) \right).$$

By a convexity argument, this quantity is maximized when $x_i = B/k$ and $y_i = (M-B)/k$ for each $1 \leq i \leq k$, in which case the change in potential is bounded by $B \log(M/B)$.

We get a lower bound on the number of read operations by dividing the difference of the initial and final potentials by the bound on the maximum change in potential per read. For $k < B$, we get the I/O bound

$$\frac{N \log B - N \log \frac{B}{k}}{B \log \frac{M}{B}} = \frac{N}{B} \log_{M/B} k.$$

For $k \geq B$, we get the I/O bound

$$\frac{N \log B - 0}{B \log \frac{M}{B}} = \frac{N}{B} \log_{M/B} B.$$

We have thus proved a lower bound of $\Omega((N/B) \log_{M/B} \min(k, B))$ I/Os. Under our assumption that $k/B = B^{o(1)}$, this gives us an I/O lower bound for this case of bundle sorting of

$$\Omega \left(\frac{N}{B} \log_{M/B} k \right). \qquad \square$$

Theorem 5.1 for the lower bound of bundle sorting follows from Lemmas 5.1 and 5.2, since together they cover all possibilities for k, M, and B.

THEOREM 5.1. *The number of I/Os required in the worst case for sorting N data items of k distinct keys, using a memory of size M and block transfer size B, is*

$$\Omega \left(\frac{N}{B} \log_{M/B} k \right).$$

6. The disk latency model. In this section we consider the necessary modifications in the external bundle sorting algorithm in order to achieve an optimum number of I/Os in a more performance-sensitive model, as in [FFM98]. In this model, we differentiate between two types of I/Os: sequential I/Os and random I/Os, where there is a reduced cost for sequential I/Os. We start by presenting the model, followed by the modifications necessary in the bundle sorting, as presented in section 4.2. We also provide an additional, slightly different integer sorting algorithm that, depending on the setting, may enhance performance by up to 33% in this model for the integer sorting problem.

6.1. The model. The only difference between this model and the external memory model presented in section 3 is that we now differentiate between costs of two types of I/O: sequential and random I/Os. We define ℓ to be the latency to move the disk read/write head to a new position during a random seek. We define r to be the cost of reading a block of size B into internal memory once the read/write head is positioned at the start of the block.

The parameters N, M, and B, as before, are referred to as the *file size*, *memory size*, and *transfer block size*, respectively, and they satisfy $1 \leq B \leq M/2$ and $M < N$. We will consider the case where $D = 1$, meaning that there is no disk parallelism. It should be clear, from the above parameters, that the cost of a random I/O that loads one transfer block into memory is $\ell + r$, and the cost of a sequential I/O is simply r.

6.2. Optimal bundle sorting in the disk latency model. The modification for bundle sorting is based on the observation that in the worst-case scenario of the algorithm as described in section 4.2, every I/O in the sorting pass can be a random I/O. This is because we are loading $\lfloor M/B \rfloor$ blocks from disk into $\lfloor M/B \rfloor$ buckets, and in the worst case they may be written back in a round robin fashion resulting solely in random I/Os. However, if we decide to read more blocks into each bucket, we will increase the total number of I/Os, which will result in the worst case with sequential I/Os in addition to random I/Os.

Let α be the number of blocks that we load into each bucket, where clearly, $1 \leq \alpha \leq (M/2B)$. Thus, in each call to one-pass sorting of bundle sorting we sort into $\lfloor M/(\alpha B) \rfloor$ distinct keys, resulting in a total of $\log_{M/(\alpha B)} k$ passes over the sequence. However, we are now sure that at least $(\alpha - 1)/\alpha$ of the I/Os are sequential. We differentiate between the I/Os required in the external count sort, in which we perform only sequential I/Os, and the sorting pass, in which we also have random I/Os. Using Theorem 4.2, the performance is now

$$\frac{2N}{B}\left(\frac{1}{\alpha}(\ell + \alpha r)\log_{M/\alpha B} k + r \log_{M/B} \frac{k}{B}\right)$$

I/Os, and the optimal value of α can be determined via an optimization procedure. In section 7 we show experimentally how the execution time varies in this model as we change α.

7. Experiments. We conducted several experiments with various data sets and settings, while changing the size of the data sets N, the available memory M, the transfer block size B, and the number of distinct items k. The data sets were generated by the IBM test data generator (http://www.almaden.ibm.com/cs/quest), or taken from the U.S. Census data, and the following experiments were executed on both data sources. In all our experiments, the records consisted of 10-byte keys in 100-byte records. All experiments were run on a Pentium2, 300 MHz, 128 MB RAM machine.

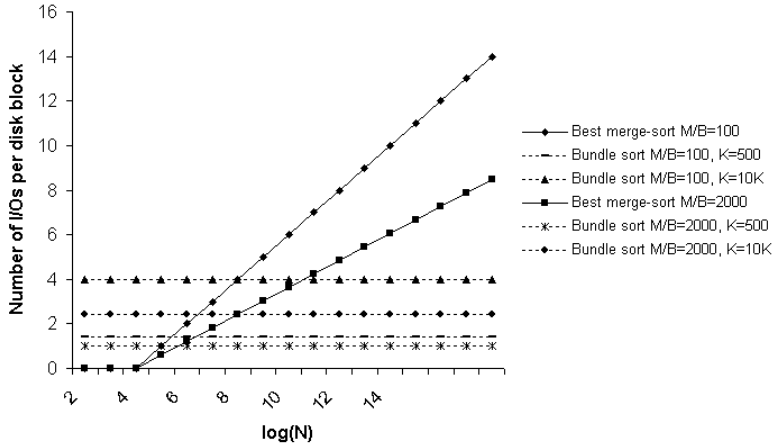

FIG. 2. *Bundle sorting versus regular sorting (best merge sort, for instance). The x-axis is the size of the data set drawn on a log scale. The y-axis is the number of I/Os performed per block of input. As can be seen, in contrast to merge sort, the number of I/Os per block in bundle sorting remains the same for a constant k as N increases.*

We first demonstrate an important feature of bundle sorting: As long as the number k of distinct keys remains constant, it performs the same number of I/O accesses per disk block with no dependence on the size of the data set. This is in contrast to general sort algorithms such as merge sort, which require more I/Os per disk block as the size of the data set increases. See Figure 2. The parameter B was set to 10 kB, and we tested for a memory of 1 MB and a memory of 20 MB. In both these cases merge sort, as expected, increased the number of I/Os per disk block as the size of the data set increased. In contrast, bundle sort performed a constant number of I/O accesses per disk block. As N increases, the improvement in performance becomes significant, demonstrating the advantages of bundle sorting. For instance, even when $k = 10000$ and the available memory is 20 MB, the break-even point occurs at $N = 1$ GB. As N increases, bundle sorting will perform better. If $k \leq 500$, then in the setting above, the break-even point occurs at $N = 10$ MB, making bundle sorting most appealing.

The next experiments demonstrate the performance of bundle sort as a function of k. See Figure 3. We set N at a fixed size of 1 GB and B at 10 kB. We ran the tests with a memory of 1 MB and 20 MB and counted the number of I/Os. We let k vary over a wide range of values from 2 to 10^9 ($k \leq N$ is always true). Since merge sort does not depend on the number of distinct keys, it performed the same number of I/O accesses per disk block in all these settings. In all these runs, as long as $k \leq N/B$, bundle sort performed better. When k is small the difference in performance is significant.

As for the disk-latency model, we show the optimal α values for various settings. Recall that in this model we attribute different costs to sequential and random I/Os. See Figure 4. We measured α for different ratios between ℓ, the cost of moving the disk reader to a random location (the latency), and r, the cost of reading a transfer block of size B. Parameter α also depends on the relation between M and B, so we plot M/B on the x-axis of the graph. As can be seen, when the ratio is 1, the optimal algorithm is exactly our bundle sorting algorithm, which counts only I/Os (hence it

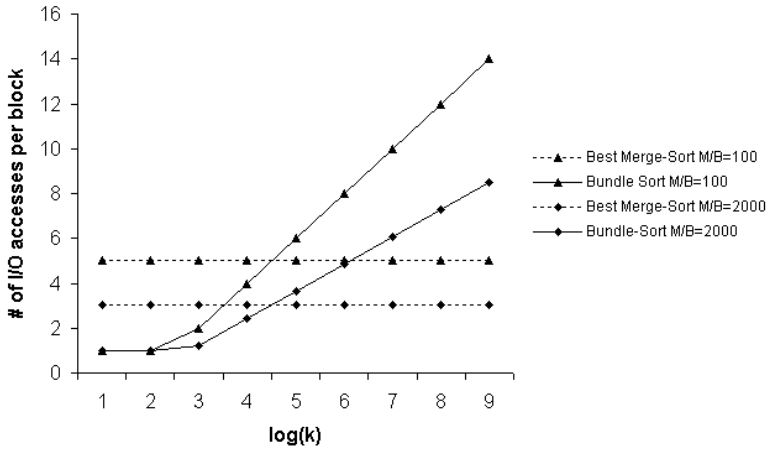

FIG. 3. *Bundle sorting versus regular sorting (best merge sort, for instance). The x-axis is the number of distinct keys (k) in the sequence drawn on a log scale. The y-axis is the number of I/Os per disk block. As can be seen, for $k \leq N/B$, bundle sorting performs better than merge sort, and the difference is large as k is smaller.*

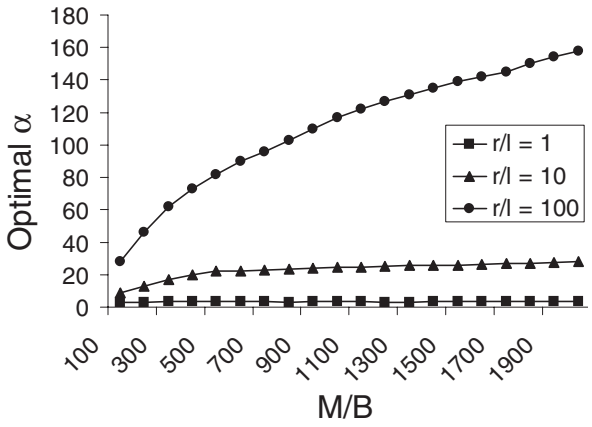

FIG. 4. *Optimum bundle sorting in the disk latency model—resolving α as a function of r, ℓ, and M/B.*

assumes that the cost of a random and a sequential I/O are equivalent). As this ratio increases, α increases, calling for a larger adaptation of our algorithm. Also affecting α, but in a more moderate way, is M/B. As this ratio increases, the optimum is achieved for a larger α.

8. Conclusions. We considered the sorting problem for large data sets with a moderate number of distinct keys, which we denote as bundle sorting, and identified it as a problem that is inherently easier than general sorting. We presented a simple in-place sorting algorithm for external memory, which may provide significant improvement over current sorting techniques. We also provided a matching lower bound, indicating that our solution is optimal.

Sorting is a fundamental problem, and any improvement in its solution may have

many applications. For instance, consider the sort join algorithm that computes join queries by first sorting the two relations that are to be joined, after which the join can be done efficiently in only one pass on both relations. Clearly, if the relations are large and their keys are taken from a universe of moderate size, then bundle sorting could provide more efficient execution than general sort.

It is interesting to note that the nature of the sorting algorithm is such that after the ith pass over the data set, the sequence is fully sorted into $(\lfloor M/B \rfloor)^i$ keys. In effect, the sequence is gradually sorted, where after each pass a further refinement is achieved, until finally the sequence is sorted. We can take advantage of this feature and use it in applications that benefit from quick, rough estimates that are gradually refined as we perform additional passes over the sequence. For instance, we could use it to produce intermediate join estimates, while refining the estimates by additional passes over the sequence. We can estimate the join after each iteration over the data set, improving the estimate after each such pass, and arrive at the final join after bundle sorting has completely finished.

The bundle sorting algorithm can be adapted efficiently and in a most straightforward way in the parallel disk model (PDM) described in [Vit99]. We now assume, in the external memory model, that $D > 1$, meaning that we can transfer D blocks into memory concurrently. This is like having D independent parallel disks. Assume that the data to be stored is initially located on one of the disks. In the first step we sort the data into exactly D buckets, writing each bucket into a distinct disk. Next, we sort, in parallel on each of the disks, the data set that was partitioned into each of the disks. Except for the initial partitioning step, we make full utilization of the parallel disks, thus enhancing performance by a factor of nearly D over all the bounds given in this paper. Note that extending bundle sorting to fit the PDM model was straightforward because of its top-down nature. Bundle sorting can also be utilized to enhance the performance of general sorting when the available working space is substantially smaller than the input set.

Bundle sorting is a fully in-place algorithm, which in effect causes the available memory to be doubled as compared to non–in-place algorithms. The performance gain from this feature can be significant. For instance, even if $M/B = 1000$, the performance gain is 10% and can be much higher for a smaller ratio. In some cases, an in-place sorting algorithm can avoid the use of high cost memory such as virtual memory.

We considered the disk latency model, which is a more performance-sensitive model where we differentiate between two types of I/Os—sequential and random I/Os—with a reduced cost for sequential I/Os. This model can be more realistic for performance analysis, and we have shown the necessary adaptation in the bundle sorting algorithm to arrive at an optimal solution in this model.

We have shown experimentation with real and synthetic data sets, which demonstrates that the theoretical analysis gives an accurate prediction of the actual performance.

REFERENCES

[ADADC+97] A. C. Arpaci-Dussaeu, R. H. Arpaci-Dussaeu, D. E. Culler, J. M. Hellerstein, and D. A. Patterson, *High-performance sorting on networks of workstations*, in Proceedings of the ACM SIGMOD International Conference on Management of Data, 1997, ACM, New York, 1997, pp. 243–254.

[Aga96] R. C. Agarwal, *A super scalar sort algorithm for RISC processors*, in Proceed-

ings of the ACM SIGMOD International Conference on Management of Data, Montral, QC, 1996, ACM, New York, 1996, pp. 240–246.

[AV88] A. Aggarwal and J. S. Vitter, *The input/output complexity of sorting and related problems*, Comm. ACM, 31 (1988), pp. 1116–1127.

[BBW86] M. Beck, D. Bitton, and W. K. Wilkinson, *Sorting Large Files on a Backend Multiprocessor*, Technical Report 86-741, Department of Computer Science, Cornell University, Ithaca, NY, 1986.

[BGK90] B. Baugsto, J. Greipsland, and J. Kamerbeek, *Sorting large data files on POMA*, in Proceedings of CONPAR-90: Proceedings of the Joint International Conference on Vector and Parallel Processing, Zurich, Switzerland, 1990, Springer-Verlag, pp. 536–547.

[FFM98] M. Farach, P. Ferragina, and S. Muthukrishnan, *Overcoming the memory bottleneck in suffix tree construction*, in Proceedings of the 39th IEEE Annual Symposium on Foundations of Computer Science, Palo Alto, CA, 1998, IEEE Press, Piscataway, NJ, 1998, pp. 174–183.

[Gra93] G. Graefe, *Query evaluation techniques for large databases*, ACM Comput. Surveys, 25 (1993), pp. 73–170.

[IBM95] IBM, *Database* 2*, Administration Guide for Common Servers*, Version 2, 1995.

[Knu73] D. E. Knuth, *The Art of Computer Programming. Vol.* 3. *Sorting and Searching*, Addison Wesley Longman Publishing, Redwood City, CA, 1973.

[MSV00] Y. Matias, E. Segal, and J. S. Vitter, *Efficient bundle sorting*, in Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms, San Francisco, 2000, SIAM, Philadelphia, 2000, pp. 839–848.

[Vit99] J. S. Vitter, *External memory algorithms and data structures*, in External Memory Algorithms and Visualization, J. Abello and J. S. Vitter, eds., AMS, Providence, RI, 1999; updated version available online at http://www.cs.duke.edu/~jsv/.

[WVI98] M. Wang, J. S. Vitter, and B. R. Iyer, *Scalable mining for classification rules in relational databases*, in Proceedings of the International Database Engineering & Application Symposium, Cardiff, Wales, 1998, IEEE Computer Society Press, Washington, DC, pp. 58–67.

[ZL98] W. Zhang and P.-A. Larson, *Buffering and read-ahead strategies for external mergesort*, in Proceedings of the International Conference on Very Large Data Bases (VLDB), 1998, Morgan Kaufmann, San Francisco, CA, pp. 523–533.