# On Sorting Strings in External Memory

(extended Abstract)

Lars Arge* Paolo Ferragina† Roberto Grossi‡ Jeffrey Scott Vitter§

**Abstract.** *In this paper we address for the first time the I/O complexity of the problem of sorting strings in external memory, which is a fundamental component of many large-scale text applications. In the standard unit-cost RAM comparison model, the complexity of sorting $K$ strings of total length $N$ is $\Theta(K \log_2 K + N)$. By analogy, in the external memory (or I/O) model, where the internal memory has size $M$ and the block transfer size is $B$, it would be natural to guess that the I/O complexity of sorting strings is $\Theta(\frac{K}{B} \log_{M/B} \frac{K}{B} + \frac{N}{B})$, but the known algorithms do not come even close to achieving this bound. Our results show, somewhat counterintuitively, that the I/O complexity of string sorting depends upon the length of the strings relative to the block size. We first consider a simple comparison I/O model, where one is not allowed to break the strings into their characters, and we show that the I/O complexity of string sorting in this model is $\Theta(\frac{N_1}{B} \log_{M/B} \frac{N_1}{B} + K_2 \log_{M/B} K_2 + \frac{N}{B})$, where $N_1$ is the total length of all strings shorter than $B$ and $K_2$ is the number of strings longer than $B$. We then consider two more general I/O comparison models in which string breaking is allowed. We obtain improved algorithms and in several cases lower bounds that match their I/O bounds. Finally, we develop more practical algorithms without assuming the comparison model.*

* Department of Computer Science, Duke University, Durham, NC 27708–0129, USA. Email: large@cs.duke.edu. Supported in part by the U.S. Army Research Office under grant DAAH04–96–1–0013 and by the ESPRIT Long Term Research Programme under project 20244 (ALCOM–IT). Part of this work was done while at BRICS, Dept. of Computer Science, University of Aarhus, Denmark, and while visiting Università di Firenze.

† Dipartimento di Informatica, Università di Pisa, Pisa, Italy. Email: ferragin@di.unipi.it. Supported in part by MURST of Italy.

‡ Dipartimento di Sistemi e Informatica, Università di Firenze, Firenze, Italy. Email: grossi@dsi2.dsi.unifi.it. Part of this work was done while visiting BRICS, University of Aarhus, Denmark.

§ Department of Computer Science, Duke University, Durham, NC 27708–0129, USA. Email: jsv@cs.duke.edu. Supported in part by the U.S. Army Research Office under grants DAAH04–93–G–0076 and DAAH04–96–1–0013 and by the National Science Foundation under grant CCR–9522047.

## 1 Introduction

The problem of lexicographically sorting a set of strings in the internal memory of computers has received much attention in the past (see e.g. [3, 34]). This problem is the most general formulation of sorting because it comprises integer sorting (i.e., strings of length one), multikey sorting (i.e., equal-length strings) and variable-length key sorting (i.e., arbitrarily long strings). Very recently several authors have considered the importance of this problem from different points of view: multikey sorting (e.g. [11, 14, 39]), integer sorting and its variants (e.g. [4, 5, 25, 43]), and parallel string sorting (e.g. [29, 30, 32]), just to cite a few.

Today's applications, however, tend to manipulate textual data sets of amazingly large size that do not fit into the internal memory of computers. Examples include textual databases [26], digital libraries [24], and relational databases [37]. In these cases, the data need to be stored on external storage devices like disks or CD ROMs, which are roughly one million times slower than internal caches in terms of access time (or latency). This disparity in latency has given rise in many applications to an input/output (or *I/O*) bottleneck, in which the time spent on moving data between internal and external memory dominates the overall execution time [27]. I/O bottlenecks are increasing in significance since the relative speed of disks versus internal memories is decreasing, especially with more and more use of parallel computers [42].

The performance of currently used algorithms for sorting strings can seriously degrade when the space occupied by the strings is larger than the available internal memory. In this paper, we address for the first time the important problem of how to sort strings when they are stored in external memory. We derive new lower and upper bounds as well as practical algorithms.

### 1.1 The I/O Model

The slow part of disk access is positioning the read-write head and waiting for the disk to rotate into position; once done, data in subsequent locations on disk can be accessed very quickly. To amortize (or hide) disk latency, each input/output operation (or *I/O*) transfers a large block of contiguous data. We use the standard model of I/O complex-

ity [2, 46] and define the following parameters:

$$\begin{aligned} K &= \#\text{ of strings to sort}, \\ N &= \text{total }\#\text{ of characters in the } K \text{ strings}, \\ M &= \#\text{ of characters fitting in internal memory}, \\ B &= \#\text{ of characters per disk block (or track)}, \end{aligned}$$

where $M < N$ and $1 \leq B \leq M/2$. We measure the performance of an algorithm in terms of the number of I/Os it performs. The quantity $\Theta(N/B)$ corresponds to "linear time" in the I/O model, since $N/B$ I/O operations are needed to read or write the whole set of $K$ strings.

We shall see that the I/O complexity of string sorting depends upon the lengths of the strings relative to the block size $B$. We use the term *short string* to denote any string with fewer than $B$ characters, and we define a *long string* to be a string of length $B$ or greater. This distinction naturally introduces four additional input parameters:

$$\begin{aligned} K_1 &= \#\text{ of short strings}, \\ K_2 &= \#\text{ of long strings}, \\ N_1 &= \text{total }\#\text{ of characters in the short strings}, \\ N_2 &= \text{total }\#\text{ of characters in the long strings}, \end{aligned}$$

where $K = K_1 + K_2$ is the total number of strings, and $N = N_1 + N_2$ is their total number of characters. Note that $N_1 < K_1 B$ and $N_2 \geq K_2 B$.

We assume that the $K$ input strings are initially stored on disk in contiguous locations; the long strings are stored in contiguous blocks. We assume without loss of generality that the length of each long string is a multiple of $B$. Every block on disk has a unique integer-valued address, and when we talk about a pointer to a (long) string we mean the address of its first block. We can locate the block containing the $i$th character of a long string by using simple arithmetic operations on the string's address. We assume that $\Theta(B)$ pointers or integers fit in a block.

When discussing string sorting algorithms we shall discuss short and long strings separately. Given a set of strings we can easily separate it into a set of short strings and a set of long strings using $O(N/B)$ I/Os; similarly, two sorted sets of strings can be merged together in $O(N/B)$ I/Os. Our algorithms for sorting long strings first produce the sorted sequence of pointers to the strings (usually called *rank sorting*); the final sorted sequence can be obtained with $O(K_2 + N_2/B) = O(N_2/B)$ extra I/Os by moving the strings into their final position one at a time.

## 1.2 Previous Results in I/O-efficient Computation

Early work on I/O algorithms concentrated on sorting and permutation related problems [2, 9, 18, 41, 40, 46]. Work has also been done on matrix algebra and related problems arising in scientific computation [2, 45, 46]. More recently, researchers have designed I/O algorithms for a number of problems in different areas, such as in computational geometry [6, 10, 28], graph theoretic computation [6, 7, 16] and string matching [11, 17, 20, 22, 23].

Aggarwal and Vitter [2] proved that the number of I/Os needed to sort $N$ *indivisible* elements is $\Omega(\frac{N}{B}\log_{M/B}\frac{N}{B})$ in the *comparison I/O model* (where the order between two elements can be inferred only by their comparison or by transitivity).[1] They also proved that rearranging a set of $N$ indivisible elements according to a given permutation requires $\Omega(\min\{N, \frac{N}{B}\log_{M/B}\frac{N}{B}\})$ I/Os. Thus in general, permuting is harder in external memory than in internal memory where it takes linear time. Aggarwal and Vitter also developed optimal sorting algorithms based upon merge and distribution sort.

In internal memory a balanced search tree can be used in the context of insertion sort to sort optimally. If we use the standard external memory search tree, the B-tree [12, 19], we get an $O(N\log_B N)$-I/O insertion sort. This algorithm's I/O efficiency is a multiplicative factor of $B\log_B(M/B)$ larger than optimal. In [6, 8] a so-called *buffer* technique for lazy updates in external data structures is developed, and using this technique the *buffer tree* is designed. Using this structure in the external insertion sort yields an optimal sorting algorithm.

As far as the general string sorting problem is concerned, there are a number of data structures like prefix B-trees [13], SB-trees [20], compacted tries [38], suffix trees [17, 36], and suffix arrays [35] that can be used to sort arbitrarily long strings in external memory. Among them, the SB-tree is the most I/O-efficient in the worst case; it can be employed to sort a set of strings in $O(K\log_B K + \frac{N}{B})$ I/Os. This sorting algorithm can be converted into an optimal $\Theta(K\log_2 K + N)$-time algorithm for the internal comparison model by fixing $B$ to a constant [21]. Very recently, Bentley and Sedgwick [14] emphasized the practical importance of string sorting and presented a version of quicksort that also achieves the optimal $\Theta(K\log_2 K + N)$ comparison bound. Analyzed in the I/O model, however, the algorithm uses $O(K\log_2 K + N)$ I/Os, which is worse than the bound obtained using the SB-tree.

## 1.3 Our Results

Since algorithms exist that match the $\Omega(K\log_2 K + N)$ lower bound for sorting strings in the internal comparison model, it seems reasonable to expect that the complexity of sorting strings in the comparison I/O model is $\Theta(\frac{K}{B}\log_{M/B}\frac{K}{B} + \frac{N}{B})$. However, the existing sorting algorithms do not even come close to meeting this bound. In this paper we show as far as optimal I/O bounds are concerned that sorting strings in external memory is not nearly as simple as it is in internal memory. We analyze the problem in the following three variants of the *I/O comparison model*, which

[1] We define for convenience $\log_{M/B}\frac{N}{B} = \max\{1, (\log\frac{N}{B})/\log\frac{M}{B}\}$.

differ from one another in how strings can be processed:

***Model A*:** Strings are considered *indivisible* (i.e., they are moved in their entirety and cannot be broken into characters), except that long strings can be divided into blocks.

***Model B*:** We relax the indivisibility assumption of Model A by allowing strings to be divided into single characters *in internal memory only*.

***Model C*:** We waive the indivisibility assumption completely and allow division of strings *in both internal and external memory*.

The algorithm by Bentley and Sedgwick [14] can easily be modified to work in Model A; the SB-tree algorithm [20] requires the power of Model C.

One of our results is counterintuitive in the sense that we prove that the conjectured $\Theta(\frac{K}{B}\log_{M/B}\frac{K}{B}+\frac{N}{B})$ I/O bound is too low. We also formally confirm the intuition that breaking up strings into their individual characters can reduce the I/O complexity of string sorting. Specifically, Models A and B have different string sorting complexities. One of our main results is that the I/O complexity of string sorting depends upon the *number of characters* for small strings ($N_1$) and the *number of strings* for long strings ($K_2$). In addition, our theoretical study gives some crucial insights into the design of practical string sorting algorithms. Our specific results are summarized in the remainder of this section.

**Model A.** In Section 2 we prove the following optimal sorting bound for Model A:

**Theorem 1** *In Model A, the I/O complexity of sorting a set of $K_1$ short strings and $K_2$ long strings of total lengths $N_1$ and $N_2$, respectively, is $\Theta(\frac{N_1}{B}\log_{M/B}\frac{N_1}{B}+K_2\log_{M/B}K_2+\frac{N_1+N_2}{B})$.*

The first term in the bound is the cost of sorting the short strings, the second term is the cost of sorting the long strings, and the last term accounts for the cost of reading the whole input. The result shows that sorting short strings is as difficult as sorting their individual characters, while sorting long strings is as difficult as sorting their first $B$ characters.

The lower bound for small strings in Theorem 1 is proved by extending the technique in [2] and considering the special case where all $K_1$ small strings have the same length $N_1/K_1$. The lower bound for the long strings then follows by considering the $K_2$ small strings obtained by looking at their first $B$ characters. The upper bounds in Theorem 1 are obtained by using a novel external merge sort approach that takes advantage of a lazy trie in internal memory to guide the merge passes. Our upper bound represents a significant improvement over the previously known algorithms, since $\frac{N_1}{B}<K_1$ and the base of the logarithm terms is $M/B$.

**Model B.** In Section 3 we discuss the more complex situation of Model B, in which we need to handle long and short strings separately.

**Theorem 2** *In Model B, the I/O complexity of sorting $K_2$ long strings of total length $N_2$ is $\Theta(K_2\log_M K_2+\frac{N_2}{B})$.*

The optimal bound in Theorem 2 differs from the corresponding bound in Theorem 1 in terms of the base of the logarithm; the base is $M$ rather than $M/B$. This shows that breaking up long strings in internal memory is provably helpful in external string sorting. We again prove the lower bound by a generalization of the technique used in [2]. We obtain the upper bound by means of a novel combination of the SB-tree data structure [20] and the buffer tree technique [6]. This allows us to get a $\Theta(M)$ SB-tree node fanout rather than a $\Theta(B)$ fanout. Our algorithm is based upon a type of insertion sort with a new batched insertion procedure for SB-trees.

For short strings, we can prove the following upper bound:

**Theorem 3** *In Model B, $K_1$ short strings of total length $N_1$ can be sorted in $O(\min\{K_1\log_M K_1, \frac{N_1}{B}\log_{M/B}\frac{N_1}{B}\})$ I/Os.*

The bound in Theorem 3 is the same as the cost of sorting all the characters in the strings (i.e., $O(\frac{N_1}{B}\log_{M/B}\frac{N_1}{B})$) when the average length $N_1/K_1$ of the short strings is $O(B/\log_{M/B}M)$. For the (in practice) narrow range $B/\log_{M/B}M<N_1/K_1<B$, the bound in Theorem 3 becomes $O(K_1\log_M K_1)$. In this range, the sorting complexity for Model B is lower than the one for Model A, which shows that breaking up short strings in internal memory is provably helpful. We derive Theorem 3 by using either the merge sort algorithm proposed for Model A or by executing the sorting algorithm for long strings in Model B.

As far as the lower bound is concerned, we prove the following result:

**Theorem 4** *In order to sort $K_1$ short strings in Model B, $\Omega(\max\{\frac{N_1}{B}\log_{MN_1/BK_1}\frac{N_1}{B}, \min\{\frac{N_1}{B}\log_{M/B}\frac{N_1}{B}, K_1\}\})$ I/Os are needed, where $N_1$ is the total length of the strings.*

The lower bound in Theorem 4 is the maximum of two terms. The first term is derived using the same technique as in the proofs of Theorems 1 and 2. The second term accounts for the cost of permuting $K_1$ short strings once their ranks in the final sorted sequence are given. When the average (short) string length $N_1/K_1$ is $O(B/\log_{M/B}\frac{N_1}{B})$, the lower bound becomes $\Omega(\frac{N_1}{B}\log_{M/B}\frac{N_1}{B})$. In this case, Theorem 3 gives a matching upper bound. For the narrow range $B/\log_{M/B}\frac{N_1}{B}<N_1/K_1<B$, in which there are very few short strings per block, our bounds are only sometimes tight; the full characterization will appear in the full paper.

**Model C.** Although the original SB-tree data structure [20] fits in Model C (it keeps a variant of tries in some blocks of

the external memory and thus it divides strings into characters), it does not improve the bounds that we can immediately derive from Model B (Theorem 2 and 3). As a consequence, the algorithms designed for Model B also turn out to be the fastest known for Model C. In Section 3, we discuss more in detail the relation between Model B and Model C.

**Practical Algorithms.** Waiving the comparison model assumption and using the insights we have gained during the theoretical considerations, we have designed a number of practical algorithms. These algorithms exploit the limited size of the alphabet $\Sigma$ from which the characters are chosen in practice. By compressing the input strings in three different ways and using a variant of the *doubling technique* [33], we have obtained three different algorithms. Their I/O bounds are $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$, $O(\frac{N}{FB}\log_{M/B}\frac{N}{F}+\frac{N}{B})$ (where $F$ is a positive integer such that $F|\Sigma|^F \leq M$), and $O(\frac{K}{B}\log_{M/B}\frac{K}{B}+\frac{N}{B}\log_{M/B}|\Sigma|)$ (if there is a positive integer $F$ such that $|\Sigma|^F \leq N$), respectively. In Section 4 we sketch the idea behind the second algorithm. Note that in the third algorithm, compression techniques and a constant-size alphabet $\Sigma$ allow us to beat the theoretical lower bounds we proved for the I/O-comparison models above. An important property of the three algorithms is that their I/O complexities are typically linear in practice.

An increasingly popular approach to increase the throughput of I/O systems is to use a number $D$ of disks in parallel, such that one can read (or write) $D$ blocks in one I/O provided that they come from $D$ distinct disks [15, 46]. An important property of our three practical algorithms is that they are all able to take full advantage of multiple disks; that is, they obtain *linear speedup* with respect to the number of available disks. In contrast, we can make our theoretical algorithms work on $D$ disks but without obtaining a linear speedup. To do so we use the *disk striping* technique [46], which in terms of performance has the effect of using a single large disk with block size $B' = DB$. Even though disk striping does not in theory achieve asymptotic optimality when $D$ is large, it is often the method of choice in practice [44].

## 2 Model A—Indivisible Strings

In this section we prove Theorem 1 in two parts. In Section 2.1 we consider the upper bound and in Section 2.2 the lower bound.

### 2.1 Upper Bounds

We use variants of external merge sort to sort both short and long strings. For the small strings the algorithm is almost identical to the standard single-character external merge sort algorithm [2]: We first produce $\Theta(N_1/M)$ sorted "runs" using $O(N_1/B)$ I/O by repeatedly loading as many strings as can fit in internal memory, sorting them, and writing them back to disk. Next we continually merge $\Theta(M/B)$ sorted runs into a longer sorted run, until we end up with one sorted run containing all the strings. The crucial property is that we can merge $\Theta(M/B)$ sorted runs of small strings together in a linear number of I/Os in terms of their total length. Since there are $\log_{M/B}\frac{N_1}{M} = \log_{M/B}\frac{N_1}{B} - 1$ levels in the merge process, each requiring $O(N_1/B)$ I/Os, we obtain the following result:

**Lemma 1** *In Model A, $K_1$ short strings of total length $N_1$ can be sorted in $O(\frac{N_1}{B}\log_{M/B}\frac{N_1}{B})$ I/Os.*

We now focus on merging long strings, where we have $K_2 \leq N_2/B$. The main difficulty is that we cannot compare strings character by character, since we could end up loading every block of every string during each of the $\log_{M/B} K_2$ levels of the merge process, resulting in a total of $O(\frac{N_2}{B}\log_{M/B} K_2)$ I/Os. We instead obtain an $O(K_2 \log_{M/B} K_2 + \frac{N_2}{B})$ I/O algorithm by modifying the merging process in a novel way.

Let us assume that we are given $M/B$ string sequences $\mathcal{S}_1, \ldots, \mathcal{S}_{M/B}$ as input in a merging process. All string sequences have the same number of strings but are possibly of different total length. Each string in the sequence is represented by a block of characters (initially its first block) and the length of its longest common prefix with the previous string in the sequence. We denote by $X_i$ the next nonmerged string in $\mathcal{S}_i$. Since we want to merge the $M/B$ sequences into a new ordered string sequence $\mathcal{S}$, we maintain a heap-like data structure $H$ on $X_1, \ldots, X_{M/B}$ in internal memory. At the beginning of the merging process, we initialize $H$ with the first string of each sequence $\mathcal{S}_i$. At the generic step, we use $H$ to obtain the minimum string $X_r$ among the $M/B$ strings in $H$, and we append (a block of) $X_r$ and its longest common prefix length with the previous output string to the output sequence $\mathcal{S}$. We then insert the string following $X_r$ in $\mathcal{S}_r$ into $H$. We call this sequence of operations extract_min.

The $H$ data structure, which is always stored in internal memory, is a lazy and slightly modified version of a compacted search trie [34, 38] storing $X_1, \ldots, X_{M/B}$ in lexicographical order. Only the leftmost path of the trie is maintained (so $H$ is actually in the form of a linear list), and no characters are explicitly stored with the arcs, but for conceptual purposes its function is that of a trie. The leftmost leaf in the trie stores the current minimum string $X_r$; the rest of the strings are stored in lists associated with the internal nodes of $H$. We label each node $u$ on the leftmost path in $H$ by the length $len(u)$ of the string $W(u)$ spelled out by the path from the root to $u$. Node $u$ has associated with it a list of all the strings whose longest common prefix with $X_r$ has length $len(u)$; the block from each string containing character number $len(u)$ is also stored in internal memory. The number of nodes in $H$ is bounded by $M/B$ and the structure occupies $O(M/B)$ blocks in internal memory even though the $M/B$ strings can collectively be much longer.

We now describe how to implement the extract_min operation. Let $X_r$ be the (minimum) string stored in the leftmost

leaf $f \in H$, and let $X'_r$ be its succeeding string in the sequence $\mathcal{S}_r$ (stored in external memory). Associated with the header information for $X'_r$ is the length of the longest common prefix between $X_r$ and $X'_r$, denoted by $lcp(X_r, X'_r)$. We load the block of string $X'_r$ containing character number $lcp(X_r, X'_r)$ and insert $X'_r$ into $H$ in two phases. In the first phase, $X'_r$ is inserted into an existing or new node on the leftmost path based upon the value of $lcp(X_r, X'_r)$. Then leaf $f$ is deleted from $H$, and (a block of) $X_r$ is appended to the merged output (along with the length of the longest common prefix between $X_r$ and the last output string). In the second phase, a new leftmost leaf node is constructed by expanding the currently lowest internal node $u$ in $H$. Several strings—each represented by one block of characters in internal memory, including character number $len(u)$—may be associated with $u$. Succeeding blocks from these strings are read until a unique leftmost leaf (i.e., minimum string) can be determined. In this process the strings stored in node $u$ are "pushed" down the tree. Several new internal nodes may be formed immediately above the new leaf. The longest common prefix between this new minimum string and the previous minimum string $X_r$ is the value previously stored as $len(u)$.

**Lemma 2** *In Model A, $K_2$ long strings of total length $N_2$ can be sorted in $O(K_2 \log_{M/B} K_2 + \frac{N_2}{B})$ I/Os.*

*Proof (sketch)*: In each merge pass, disregarding for now the initialization of $H$, each string requires no more than one I/O plus some extra I/Os needed to push the string down the $H$ structure. The extra I/Os are to blocks of the string that are not reloaded in later merge passes. Summed over the $\log_{M/B} K_2$ merge passes, the total number of I/Os done for each string is $\sum(1 + \#\text{ extra I/Os}) = \log_{M/B} K_2 + (\text{length of string})/B$. Hence, over all the $K_2$ strings, the total number of I/O operations done is $K_2 \log_{M/B} K_2 + N_2/B$.

All that remains is to analyze the I/O cost of all the initializations of $H$. At the beginning of each merge pass, we initialize $H$ at a cost bounded by the number of blocks in the $M/B$ strings initially stored in $H$. Except for the minimal string $X_r$, we charge the I/O cost of examining the blocks of a string to the string itself. For $X_r$, we charge its I/O cost to the string among the other $M/B - 1$ strings that has the longest prefix with $X_r$; the number of blocks of this string is at least the I/O cost for $X_r$. The string $X_r$ is the only string from this merge pass that will participate in the initialization of $H$ in a later merge pass. By the above charging scheme, no string is charged in more than one merge pass, and no string is charged more than double the number of blocks that it contains. Hence, the total number of I/Os needed to initialize the lazy tries $H$ among all the merge passes is bounded by $2N_2/B$. □

## 2.2 Lower Bounds

**Lemma 3** *In Model A, the number of I/Os needed to sort $K_1$ short strings of total length $N_1$ is $\Omega(\frac{N_1}{B} \log_{M/B} \frac{N_1}{B})$.*

*Proof (sketch)*: Given the input parameters $K_1$ and $N_1$, we consider a special instance in which all short strings have the same length $N_1/K_1$ (hence, we have $X = BK_1/N_1$ strings per block), and apply a modified version of the argument used in [2]. Initially there are $K_1!$ possibilities for the correct ordering of the $K_1$ equal-length short strings; every input operation and the string comparisons done after it decrease this number. Consider an input operation loading $X$ short strings from one block and assume that their relative order is not known (this may happen $N_1/B$ times). After the input operation, there are at most $\binom{(M/B)X}{X} X!$ sets of possible outcomes to the comparisons between the strings in internal memory. Conversely, if the order of the $X$ strings is known, there are at most $\binom{(M/B)X}{X}$ possible outcomes. An adversary always chooses the outcome that maximizes the number of total orderings consistent with the comparisons done so far. After $t$ I/Os, the number of consistent orderings is at least $K_1!/(X!)^{N_1/B}\binom{(M/B)X}{X}^t$. Setting this expression to 1, we get t = $\Omega(\frac{N_1}{B} \log_{M/B} \frac{N_1}{B})$. □

We now have all the ingredients to prove the sorting lower bound for long strings. It suffices to observe that sorting $K_2$ long strings is at least as difficult as sorting their prefixes of length $B$. Consequently, by using Lemma 3 and the fact that we need to read all the $N_2$ characters (which takes $\frac{N_2}{B}$ I/Os), we obtain the following corollary, which completes the proof of Theorem 1.

**Corollary 1** *In Model A, $\Omega(K_2 \log_{M/B} K_2 + \frac{N_2}{B})$ I/Os are needed to sort a set of $K_2$ long strings of total length $N_2$.*

# 3 Model B—Strings Indivisible in External Memory

In this section we prove Theorems 2, 3 and 4. The upper bounds (Theorems 2 and 3) are discussed in Section 3.1 and the lower bounds (Theorems 2 and 4) are discussed in Section 3.2.

## 3.1 Upper Bounds

In this section we describe an algorithm for sorting long strings, and this proves the upper bound in Theorem 2. As mentioned in the introduction the algorithm for short strings is obtained by applying either this long string algorithm or the short string algorithm for Model A. Our solution for long strings will be based upon the SB-tree and the buffer tree data structures which we briefly review below (we refer the reader to [6, 8, 20] for more details). We then describe our improved algorithm which is a combination of these two data structures.

**The Buffer Tree.** The basic buffer tree on $R$ integer keys is a B-tree (or rather an $(a, b)$-tree [31]) with fanout $\Theta(M/B)$ and with blocks of elements in the leaves; thus the tree has height $O(\log_{M/B} \frac{R}{B})$. A *buffer* of size $\Theta(M/B)$ blocks is

assigned to each internal node and the operations on the structure are done in a "lazy" manner. For example, when inserting a new key, one does not search all the way down the tree to find the place among the leaves to insert the element. Instead, one waits until a block of insertions has been collected, and then this block is inserted into the buffer of the root. When a buffer "runs full," its elements are "pushed" one level down to the buffers on the next level. This *buffer emptying process* is performed by loading the elements (and the splitting/routing elements) into internal memory, sorting them, and writing them back to the appropriate buffers on the next level. If the buffer of any of the nodes on the next level becomes full, the buffer emptying process is applied recursively. The main point is that a buffer emptying process can be performed in $O(M/B)$ I/Os, and since it pushes $\Theta(M/B)$ blocks of elements one level down, a constant number of I/Os is used on each block (as opposed to each element) on each of the $O(\log_{M/B} \frac{R}{B})$ levels of the structure. In [6] it is thus shown that the total number of I/Os used to insert $R$ elements in a buffer tree, including I/Os used for rebalancing, is $O(\frac{R}{B} \log_{M/B} \frac{R}{B})$.

**The SB-Tree.** From a high level point of view an SB-tree storing a set of $R$ strings is a B-tree [12, 19] built on the set of pointers to the strings: The pointers are stored in the leaves of the tree ($\Theta(B)$ pointers per leaf) and are ordered according to the lexicographical order of the strings they point to. Each internal node contains $\Theta(B)$ string pointers, namely, a copy of the leftmost and rightmost pointer contained in each of its $\Theta(B)$ children. The SB-tree has height $O(\log_B R)$ and rebalancing after an update is done by splitting or fusing nodes (or by sharing sons).

In order to allow I/O-efficient search and update operations, each internal node $\pi$ contains the so-called *blind trie* data structure $BT_\pi$, which is built on the set of strings $\mathcal{S}_\pi$ pointed to by the pointers in $\pi$. The blind trie is a variant of the compacted trie [34, 38] where each node $u$ is labeled with the length $len(u)$ of the string $W(u)$ spelled out by the path from the root to $u$, just like in the $H$ structure in Section 2.1. The substring normally labeling an arc is replaced by its first character, called the *branching character*. (In the $H$ structure the substring was instead completely removed.) $BT_\pi$ occupies $O(|\mathcal{S}_\pi|)$ space (it contains $O(|\mathcal{S}_\pi|)$ characters and pointers), even though the total length of the strings in $\mathcal{S}_\pi$ can be much larger. Consequently, $BT_\pi$ fits in one block. This is one of the important properties used in [20], where it is shown how the blind tries can be used to efficiently guide the search for the lexicographical position of a string $P$ among the strings (i.e., pointers) stored in the leaves. This can then be used to design an $O(\log_B R + \frac{|P|}{B})$ I/O insertion procedure, which in turn can be used to obtain an external sorting algorithm using $O(R \log_B R + \frac{N}{B})$ I/Os, where $N$ is the total length of the $R$ strings. Note that the SB-tree uses the power of Model C because single characters from many strings are used to form the blind tries.

**The Buffered SB-Tree.** We obtain the improved long string sorting algorithm by means of a novel combination of the the buffer tree and the SB-tree which we call the *buffered SB-tree*.

We modify the SB-tree by increasing the fanout (and thus the size of both $BT_\pi$ and $S_\pi$) to $\Theta(M)$ in order to obtain height $O(\log_M K_2)$. Furthermore, in order to make the SB-tree work in Model B (where strings are considered indivisible in external memory), we do not explicitly store the blind trie branching characters but rather *pointers* to them. Each SB-tree node $\pi$ can then be stored in $\Theta(M/B)$ blocks in external memory. However, when we need to use the blind trie $BT_\pi$ of a node $\pi$ in internal memory, we have to perform $O(M/B)$ I/Os to load $\pi$ and $O(M)$ extra I/Os to load the branching characters of $BT_\pi$ from the $O(M)$ strings belonging to $S_\pi$.

If we were to insert strings in the above structure on an individual basis, we would get an $O(M \log_M K_2 + \frac{|P|}{B})$ insertion I/O bound for a string $P$, because we would use $O(M)$ I/Os on each level of the structure to load a blind trie. The main idea to go around this problem is (as in the buffer tree) not to insert strings on an individual basis, but to collect $\Theta(M)$ strings (pointers) in a buffer $\mathcal{B}_\pi$ associated with the node $\pi$ and to push the strings one level down the structure once $\mathcal{B}_\pi$ gets full. In this way the $O(M)$ I/Os used to load $BT_\pi$ can be charged to the strings in $\mathcal{B}_\pi$ so that only a constant number of I/Os is charged to each of them. As a consequence, we get an amortized $O(\log_M K_2 + \frac{|P|}{B})$ insertion I/O bound for each string $P$, and thus an $O(K_2 \log_M K_2 + \frac{N_2}{B})$ I/O algorithm for sorting $K_2$ long strings of total length $N_2$.

It is worth noting that the insertion procedure developed in [20] needs to be modified to work in the setting described above. We describe here the new batched insertion procedure, leaving a lot of details to the full paper: When a buffer $\mathcal{B}_\pi$ contains $\Theta(M)$ elements, we perform a buffer emptying process on node $\pi$ as follows. We first load blind trie $BT_\pi$ into internal memory using $O(M)$ I/Os, and then we route each string pointer in $\mathcal{B}_\pi$ through $\pi$'s children by executing the procedure BT_Search described below. In order to do the routing efficiently, we inductively maintain the invariant that each element in buffer $\mathcal{B}_\pi$ is actually a triple $\langle P, \ell, X \rangle$, such that $P$ is the string pointer and $X$ is a (pointer to a) string in $\mathcal{S}_\pi$ that shares the first $\ell$ characters with $P$ (i.e., $lcp(P, X) \geq \ell$).

Procedure BT_Search($\langle P, \ell, X \rangle$, $\mathcal{S}_\pi$) searches for the lexicographic position of $P$ among the strings in $\mathcal{S}_\pi$ and thus determines the child where $P$ has to be routed. The procedure works in two phases. In the *first phase*, we identify the leaf $v$ in $BT_\pi$ associated with $X$ (i.e., $W(v) = X$). We define the *hit node* for a pair $(v, \ell)$ to be $v$'s ancestor $u$ satisfying $len(u) \geq \ell > len(parent(u))$; if $\ell = 0$, the hit node is the root. We then begin a downward traversal of $BT_\pi$ from the hit node $u$ by matching $P$'s characters with the branching characters of the traversed arcs, until either a leaf $f$ is reached or no further branching is possible. In the latter case, $f$ is chosen to be one of the leaves descend-

ing from the last traversed node. Note that $f$ stores one of the strings in $\mathcal{S}_\pi$ that share the longest common prefix with $P$ [20]. The downward traversal of $BT_\pi$ cannot be executed without I/Os because $P$ is stored in external memory. We therefore perform it as follows: We first load the block containing the character $P[\ell+1]$ into internal memory; assume that this block contains the substring $P[i, i+B-1]$. After that, we traverse $BT_\pi$ downwards using $P[i, i+B-1]$ until we end up in a node $w$ in which either no further branching is possible (and thus we can determine $f$), or a character of $P[i+B, |P|]$ is needed to continue the branching. In the latter case, we take a leaf $t$ descending from $w$ and compare $P[i, i+B-1]$ with the corresponding substring of $W(t)$ using only one I/O. If they are different, $t$ is the leaf $f$ we are looking for, and we have found $lcp(P, W(f))$ and the mismatching character of $P$ and $W(f)$, and thus we can go on to the second phase described below. Otherwise, we load the block for $P[i+B, i+2B-1]$ and resume the downward traversal of $BT_\pi$ from node $w$. This approach guarantees that we do not rescan $P$. If $lcp = lcp(P, W(f))$ the number of I/Os used in the first phase is $O(\frac{lcp-\ell}{B}+1)$.

The *second phase* begins when leaf $f$ has been identified. Since $f$ stores one of the strings in $\mathcal{S}_\pi$ that shares the longest common prefix with $P$, the mismatching character of $P$ and $W(f)$ can be used along with the hit node for the pair $(f, lcp(P, W(f)))$ to find $P$'s position in $\mathcal{S}_\pi$ without any further I/Os. In [20], it has been shown that one of the strings, say $Y$, adjacent to $P$'s position in $\mathcal{S}_\pi$ shares $lcp(P, W(f))$ characters with $P$ (i.e., $lcp(P, W(f)) = lcp(P, Y)$). We therefore maintain the invariant for $P$ by using one I/O to store the triple $\langle P, lcp, Y\rangle$ in the buffer of the child of $\pi$ that contains $Y$. (Details will appear in the full version of the paper.)

From the above discussion we get that the routing cost for string $P$ is $O(\frac{lcp-\ell}{B}+1)$. The $O(M)$ I/Os used to load $BT_\pi$ are divided among the $\Theta(M)$ strings as described above, such that $O(1)$ I/Os are charged to $P$. It follows that routing $P$ contributes with $O(\sum_{i=1}^{H}\frac{\ell_i-\ell_{i-1}}{B}+1)$ I/Os to the overall cost of the buffer emptying processes, where $H = O(\log_M K_2)$ is the height of the SB-tree and $\ell_i$ is the value of $\ell$ at level $i$, for all $1 \le i \le H$. This sums up to $O(\log_M K_2 + \frac{|P|}{B})$ because $0 \le \ell_{i-1} \le \ell_i \le |P|$, and hence we need a total of $O(K_2 \log_M K_2 + \frac{N_2}{B})$ I/Os for all the strings.

The only I/Os we have not accounted for so far are the ones used to rebalance the buffered SB-tree structure. Following the argument in [6], it can be argued that inserting $K_2$ strings results in $\Theta(K_2/M)$ splits in total. Since one split can be performed in $O(M)$ I/Os, the rebalancing cost adds up to $O(K_2)$. Details will appear in the full version of this paper. This ends our proof of the following:

**Lemma 4** *In Model B, $K_2$ long strings of total length $N_2$ can be sorted in $O(K_2 \log_M K_2 + \frac{N_2}{B})$ I/Os using the buffered SB-tree.*

An interesting question is what happens if we waive the indivisibility assumption and use Model C. In the above algorithm for Model B, we replaced the branching characters by their pointers to avoid string breaking in external memory. Consequently, we needed $O(M)$ I/Os to load a blind trie into internal memory. If we waive the indivisibility assumption also in external memory, we can store the blind trie directly in $\Theta(M/B)$ blocks and thus decrease the cost of loading it to $\Theta(M/B)$ I/Os. However, we still obtain the same overall I/O bound. In general, it seems that we cannot use the buffered SB-tree technique to obtain a better bound in Model C, because we have already obtained the maximal $\Theta(M)$ fanout on every node.

## 3.2 Lower Bounds

**Lemma 5** *In Model B, the number of I/Os needed to sort $K_1$ short strings of total length $N_1$ is $\Omega(\frac{N_1}{B}\log_{MN_1/BK_1}\frac{N_1}{B})$.*

*Proof (sketch)*: We use a technique similar to the one in the proof of Lemma 3 except that we bound the number of possible outcomes of the comparisons among the strings in internal memory by $\binom{M}{X}X!$, as we can have characters from as many as $M$ different strings in internal memory. Setting $K_1!/(X!)^{N_1/B}\binom{M}{X}^t \le 1$ and simplifying, we get the desired lower bound on $t$. □

Analogously to what we was done in Section 2.2, we can now obtain the following for long strings:

**Corollary 2** *In Model B, $\Omega(K_2 \log_M K_2 + \frac{N_2}{B})$ I/Os are required to sort $K_2$ long strings of total length $N_2$.*

We have now characterized the asymptotic I/O complexity of sorting long strings (i.e., proved Theorem 2). As discussed in the introduction, Lemma 5 and Lemma 6 (below) allow us to characterize the complexity of sorting short strings except for a narrow set of inputs:

**Lemma 6** *In Model B, $\Omega(\min(\frac{N_1}{B}\log_{M/B}\frac{N_1}{B}, K_1))$ I/Os are needed to sort $K_1$ short strings of total length $N_1$.*

*Proof (sketch)*: We again consider the equal length case where we have $X = BK_1/N_1$ strings per block. First note that from a *permutation* point of view, Models A and B are equivalent because strings are moved in and out of internal memory in their entirety. Thus a permutation bound in Model A also holds in Model B. The permutation lower bound proved in [2] shows that there exists a permutation $\Pi$ of $N'$ indivisible elements requiring $\Omega(\min\{N', \frac{N'}{B'}\log_{M'/B'}\frac{N'}{B'}\})$ I/Os on a machine with block size $B'$ and memory size $M'$, respectively. An algorithm that sorts $K_1 = N'$ short strings in Model A must be capable of permuting the strings according to $\Pi$. Thus by regarding strings as single elements, we can obtain a sequence of I/Os performing the permutation $\Pi$ on a machine with $B' = X$ and $M' = \frac{M}{B}X$ from a sequence of I/Os permuting the strings in Model A. Since the two sequences perform the

same number of I/Os, we obtain the lower bound by substituting $N' = K_1$, $B' = X$, and $M' = \frac{M}{B}X$ into the above bound. □

## 4 Practical Algorithms

The basic tool we use to design practical sorting algorithms is an external memory version of the Karp-Miller-Rosenberg labeling technique [33], also called the *doubling algorithm*, which uses $O(\frac{N}{DB}\log_{M/B}\frac{N}{B})$ I/Os. We defer the description of the external doubling algorithm to the full paper. Our three algorithms share a similar overall structure: In a preprocessing step, the input string sequence $\mathcal{S}$ (of total length $N$) is compressed into a sequence $\mathcal{S}'$ by shrinking each string in $\mathcal{S}$ by a factor $F > 1$, while preserving their relative lexicographic order. Then the doubling algorithm is applied to $\mathcal{S}'$.

In this extended abstract, we just sketch one of our three algorithms. In this algorithm, the preprocessing step consists of compressing the original strings as follows: We "break" the strings into a set $\mathcal{C}$ of $O(\frac{N}{F} + K)$ substrings of length $F$ (strings shorter than $F$ form a single substring). Each string of length $s$ is thus a sequence of $O(s/F)$ "supercharacters", each one being a string in $\mathcal{C}$. The parameter $F > 1$ is chosen to be the maximum integer (if any) such that the whole set $\mathcal{C}$ can be kept simultaneously in internal memory; it suffices that $F|\Sigma|^F \leq M$. Since we can keep $\mathcal{C}$ in internal memory (by means of a ternary search tree [14]), we can sort $\mathcal{C}$'s strings by one single scan of the whole input. This clearly gives the final rank of $\mathcal{C}$'s strings. Subsequently, we create the new string sequence $\mathcal{S}'$ in which each string in $\mathcal{S}$ is compressed by replacing its supercharacters (i.e., length-$F$ substrings) with their ranks computed previously. The main observation is that for any two substrings of $\mathcal{C}$ one is lexicographically smaller than the other if and only if the rank of the former is smaller than the rank of the latter. Therefore, sorting the strings in $\mathcal{S}'$ is equivalent to sorting the strings in $\mathcal{S}$. We finally execute the doubling algorithm on $\mathcal{S}'$ and determine the sorted string sequence. The overall cost is $O(\frac{N}{FDB}\log_{M/B}\frac{N}{F} + \frac{N}{DB})$, which becomes linear when $F \geq \log_{M/B}(N/B)$ (i.e., in many practical cases).

## 5 Conclusions and Open Problems

Our goal in this paper was to explore the I/O complexity of sorting strings from both a theoretical and practical perspective. For various I/O models, we showed the relationship between string length and sorting complexity and derived upper and lower bounds. All our algorithms have better I/O performance than all previously known algorithms.

Several interesting open questions remain. One primary open problem is to close the remaining gap between the upper and lower bounds in Model B. Another is to prove lower bounds in Model C. Proving such new or better lower bounds would probably require a fundamentally new approach.

Our strongest model (Model C) removes the indivisibility assumption on strings, but still assumes that the individual characters are indivisible. A natural question to ask is whether compression and coding techniques can be used to get better algorithms when, for example, bit operations are allowed on the characters, akin to the recent result on a type of matrix transposition [1]. The practical algorithms in Section 4 already exploit some of these techniques.

Another natural question to ask is how to sort strings optimally using $D$ disks. As discussed, the practical algorithms in Section 4 can achieve optimal (linear) speedup in the $D$-disk model in many real situations. We are currently implementing several of our algorithms and hope to report full experimental results soon.

**Acknowledgments.** The first author thanks Dany Breslauer for many string algorithm discussions and Peter Bro Miltersen for many great external-memory lower bound discussions. The second author thanks his supervisor Fabrizio Luccio for numerous interesting discussions. The third author wishes to thank Dany Breslauer for inviting him at BRICS and for hours of delighting discussions.

## References

[1] M. Adler. New coding techniques for improved bandwidth utilization. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, pages 173–182, 1996.

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

[3] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

[4] A. Andersson, T. Hagerup, S. Nilsson, and R. Raman. Sorting in linear time? In *Proc. ACM Symp. on Theory of Computation*, pages 427–436, 1995.

[5] A. Andersson and S. Nilsson. A new efficient radix sort. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, pages 714–721, 1994.

[6] L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Proc. Workshop on Algorithms and Data Structures, LNCS 955*, pages 334–345, 1995. A complete version appears as BRICS technical report RS-96-28, University of Aarhus.

[7] L. Arge. The I/O-complexity of ordered binary-decision diagram manipulation. In *Proc. Int. Symp. on Algorithms and Computation, LNCS 1004*, pages 82–91, 1995. A complete version appears as BRICS technical report RS-96-29, University of Aarhus.

[8] L. Arge. *Efficient External-Memory Data Structures and Applications*. PhD thesis, University of Aarhus, February/August 1996.

[9] L. Arge, M. Knudsen, and K. Larsen. A general lower bound on the I/O-complexity of comparison-based algorithms. In *Proc. Workshop on Algorithms and Data Structures, LNCS 709*, pages 83–94, 1993.

[10] L. Arge, D. E. Vengroff, and J. S. Vitter. External-memory algorithms for processing line segments in geographic information systems. In *Proc. Annual European Symposium on Algorithms, LNCS 979*, pages 295–310, 1995. A complete version (to appear in special issue of Algorithmica) appears as BRICS technical report RS-96-12, University of Aarhus.

[11] R. A. Baeza-Yates and G. H. Gonnet. *Handbook of Algorithms and Data Structures*. Addison-Wesley, 1991.

[12] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[13] R. Bayer and K. Unterauer. Prefix B-trees. *ACM Trans. Database Syst.*, 2(1):11–26, 1977.

[14] J. L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 360–369, 1996.

[15] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.

[16] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 139–149, 1995.

[17] D. R. Clark and J. I. Munro. Efficient suffix trees on secondary storage. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 383–391, 1996.

[18] Thomas H. Cormen, Thomas Sundquist, and Leonard F. Wisniewski. Asymptotically tight bounds for performing BMMC permutations on parallel disk systems. Technical Report PCS-TR94-223, Dartmouth College Dept. of Computer Science, July 1994.

[19] D. Cormer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[20] P. Ferragina and R. Grossi. A fully-dynamic data structure for external substring search. In *Proc. ACM Symp. on Theory of Computation*, pages 693–702, 1995.

[21] P. Ferragina and R. Grossi. An external-memory indexing data structure with applications. Full version of STOC'95 paper "A fully-Dynamic data structure for external substring search", 1996.

[22] P. Ferragina and R. Grossi. Fast string searching in secondary storage: Theoretical developments and experimental results. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 373–382, 1996.

[23] P. Ferragina and F. Luccio. On the parallel dictionary matching problem: New results with applications. In *Proc. Annual European Symposium on Algorithms, LNCS 1136*, pages 261–275, 1996.

[24] E. A. Fox, R. M. Akscyn, R. K. Furuta, and J. J. Leggett. Special issue: Digital libraries: Introduction. *Communications of the ACM*, 38(4), April 1995.

[25] M.L. Fredman and D.E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47:424–436, 1993.

[26] K. A. Frenkel. The human genome project and informatics. *Communications of the ACM*, 34:41–51, 1991.

[27] G. A. Gibson, J. S. Vitter, and J. Wilkes. Report of the working group on storage I/O issues in large-scale computing. *ACM Computing Surveys*, 28(4), December 1996.

[28] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, pages 714–723, 1993.

[29] T. Hagerup. Optimal parallel string algorithms: Merging, sorting and computing the minimum. In *Proc. ACM Symp. on Theory of Computation*, pages 382–391, 1994.

[30] T. Hagerup and O. Petersson. Merging and sorting strings in parallel. In *Proc. International Symp. on Mathematical Foundations of Computer Science*, pages 298–306, 1992.

[31] S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982.

[32] J. F. JáJá, K. W. Ryu, and U. Vishkin. Sorting strings and constructing digital search trees in parallel. *Theoretical Computer Science*, 154(2):225–245, 1996.

[33] R. Karp, R. Miller, and A. Rosenberg. Rapid identification of repeated patterns in strings, arrays and trees. In *Proc. ACM Symp. on Theory of Computation*, pages 125–136, 1992.

[34] D. Knuth. *The Art of Computer Programming, Vol. 3 Sorting and Searching*. Addison-Wesley, 1973.

[35] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal of Computing*, 25(5):935–948, 1993.

[36] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[37] T. H. Merrett. Why sort-merge gives the best implementation of the natural join. *ACM SIGMOD Record*, 13(2), 1983.

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

[39] J. I. Munro and V. Raman. Sorting multisets and vectors inplace. In *Proc. Workshop on Algorithms and Data Structures, LNCS 519*, pages 473–479, 1991.

[40] M. H. Nodine and J. S. Vitter. Deterministic distribution sort in shared and distributed memory multiprocessors. In *Proc. ACM Symp. on Parallel Algorithms and Architectures*, pages 120–129, 1993.

[41] M. H. Nodine and J. S. Vitter. Greed sort: An optimal sorting algorithm for multiple disks. *Journal of the ACM*, pages 919–933, 1995.

[42] Yale N. Patt. The I/O subsystem—a candidate for improvement. *Guest Editor's Introduction in IEEE Computer*, 27(3):15–16, 1994.

[43] M. Thorup. Randomized sorting in $o(n \log \log n)$ time and linear space using addition, shift, and bit-wise boolean operations. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 352–359, 1997.

[44] D. E. Vengroff and J. S. Vitter. Supporting I/O-efficient scientific computation in TPIE. In *Proc. IEEE Symp. on Parallel and Distributed Computing*, 1995. Appears also as Duke University Dept. of Computer Science technical report CS-1995-18.

[45] D. E. Vengroff and J. S. Vitter. I/O-efficient computation: The TPIE approach. In *Proceedings of the Goddard Conference on Mass Storage Systems and Technologies*, NASA Conference Publication 3340, Volume II, pages 553–570, College Park, MD, September 1996.

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