Batched Predecessor and Sorting with Size-Priced Information in External Memory

Michael A. Bender[1]*, Mayank Goswami✉[2]**, Dzejla Medjedovic[3], Pablo Montes[4], and Kostas Tsichlas[5]

[1] Dept. of Computer Science, Stony Brook University, Stony Brook, NY 11794, USA
bender@cs.stonybrook.edu
[2] Dept. of Computer Science, Queens College, CUNY, New York 11367, USA
mayank.goswami@qc.cuny.edu
[3] International University of Sarajevo, Sarajevo 71000, Bosnia-Herzegovina
dzmedjedovic@ius.edu.ba
[4] Google Inc, Mountain View, California 94043, USA
pabmont@gmail.com
[5] Aristotle University of Thessaloniki, Thessaloniki 54124, Greece
tsichlas@csd.auth.gr

Abstract. The fundamental problems of sorting and searching, traditionally studied in the unit-cost comparison model, have been generalized to include priced information, where different pairs of items have different comparison costs. These costs can be arbitrary (Charikar et al. STOC 2000), structured (Gupta et al. FOCS 2001), or stochastic (Angelov et al. LATIN 2008). Motivated by the database setting where the comparison cost depends on the sizes of the records, we consider the problems of sorting and batched predecessor where two non-uniform sets of items A and B are given as input. In the RAM model, pairwise comparisons (A-A, A-B and B-B) have respective comparison costs a, b and c. We give upper and lower bounds for the case $a \leq b \leq c$, which serves as a warmup for the generalization to the external-memory model. In the Disk-Access Model (DAM), where transferring elements between disk and RAM is the main bottleneck, we consider the scenario where elements in B are larger than elements in A. All items are required in their entirety for comparisons in RAM. A key observation is that the complexity of sorting depends on the interleaving of the small and large items in the final sorted order, and with a high degree of interleaving, the lower bound is dominated by an associated batched predecessor problem. We give output-sensitive bounds on the batched predecessor and sorting; our bounds are tight in most cases. Our lower bounds require novel generalizations of lower bound techniques in external memory to accommodate non-uniform keys.

Keywords: Priced information · Sorting · Batched predecessor · External memory · Output-sensitive algorithms

* This work was supported in part by NSF grants CCF-1725543, CSR-1763680, CCF-1716252, CCF-1617618, CNS-1938709, and by Sandia National Laboratories.
** Supported by NSF grants CRII-1755791 and CCF-1910873.

1 Introduction

In most published literature on sorting and other comparison-based problems (e.g., searching and selection), the traditional assumption is that a comparison between any two elements costs one unit, and the efficiency of an algorithm depends on the total number of comparisons required to solve the problem. In this paper, we study a natural extension to sorting, where the cost of a comparison between a pair of elements can vary, and the comparison cost is the function of the elements being compared. We derive worst-case lower and upper bounds in the random-access-machine (RAM) and the disk-access-machine (DAM) [1] models for sorting and the batched predecessor problems.

In the RAM model, we assume that comparisons between a pair of keys have an associated cost that depends on the type of keys involved. As a toy problem, consider n red and n blue keys, where a comparison between a pair of red keys costs a, between a red key and a blue key costs b, and between a pair of blue keys costs c. Without loss of generality we can assume $a < c$, which gives rise to three cases to be considered, $a < b < c$, $a < c < b$, and $b < a < c$ (when $b = 1$ but $a = c = \infty$ corresponds to the well-known nuts and bolts problem [2].) In this paper we consider the setting of [16], where the comparison cost depends on the length of the keys being compared. However, our analysis considers the worst-case cost parameterized by the specific distribution (or the "interleaving") of the elements in the final sorted order.

Then we turn to the *disk access machine* (*DAM*) model (also called the *external-memory model* or the *I/O model*) [1], designed to capture the key aspect of data-intensive applications, where transferring data is the main bottleneck, as oppose to CPU computation. In this simplified model of modern memory hierarchy, data is transferred from an external disk of infinite capacity to the main memory of size M in blocks of size B, where $M > B$ and input size $N >> M$; the cost of the algorithm is measured by the number of block transfers (I/Os) that it needs; once data is in memory, all computation comes for free.

In the DAM model, the notion of the comparison cost naturally comes into play when elements have different *sizes* (or *lengths*) because the larger the elements are, the fewer of them can fit in a block transfer. Specifically, if a key has length w, where $w \leq B$, then up to B/w keys can be fetched with one I/O; similarly, if $w \geq B$, then it takes w/B I/Os to bring that key into memory. Moreover, a long element, when brought into RAM, will displace a larger volume of keys than a short element, thus reducing the parallelism that many external-memory algorithms such as external merge sort benefit from. [6] In the DAM model, we are given two sets of keys, S keys of unit size (short keys) and L/w (long) keys of size w each (total volume L). We express our results parameterized by the interleaving of the elements in their final sorted order. Let the interleaving parameter

[6] DAM model can represent any two levels of memory, which is related to the record size in our problem. If the two levels are the disk and the main memory, elements could be larger than B but are much smaller than M. If the levels are cache and main memory, then the elements could have a length that is a nontrivial fraction of M.

k denote the number of consecutive runs of large keys (i.e., *stripes*) in the final sorted order. Our goal is to express the performance of the sorting algorithm, as a function of S, L, w, and k.

Sorting with two key lengths illustrates a special connection between sorting and the batched searching problem that we call *PLE* (Placement of Large Elements). Given S keys of unit size (short keys) in the sorted order, and L/w (long) keys of size w each, the objective in the PLE is to find the short key that is the immediate predecessor of each long key. The PLE problem is a lower bound on sorting because it starts off with more information than the original sorting and asks to do less; in many cases, it dominates the sorting cost.

However, obtaining lower bounds on the PLE presents several challenges. First, having records of different sizes implies that standard information-theoretic lower bounds on the unit-sized case do not apply, because now different types of I/Os can contribute different amounts of information. Second, depending on the interleaving of the small and large elements, expressed in our bounds with the interleaving parameter k, and the size of the large element w, the PLE bounds substantially change. And lastly, PLE is a batched searching problem with a nontrivial preprocessing-query tradeoff [9], an interesting problem on its own.

Related Work. In RAM, algorithms for inputs with priced information have been studied in the context of competitive analysis [11,12,16,3]. Another example of varying comparison costs is the well-known nuts-and-bolts problem [2]. Interleaving-sensitive lower bounds and batched searching are related to lower bounds for sorting multisets [19] and distribution-sensitive set-partitioning [14].

Aggarwal and Vitter [1] introduced the external-memory (DAM) model. The fundamental lower bounds were further generalized in [6] and [15] to the external algebraic decision tree model. Prominent examples studying lower bounds on batched and predecessor searching are found in [4,7,9].

Most previous work that considers variable-length keys does so in the context of B-trees [18,13,17,20,8], and string sorting [5], where in the latter, authors derive upper and lower bounds under different models of key divisibility. Strings are divisible, and that brings down the complexity of sorting, but the lower bounds in our paper also imply the worst-case bounds for the string scenario where the tie is broken at the last character. Indivisible keys are found in practical settings, where sorting and searching libraries such as GNU Sort [21] or Oracle Berkeley DB [10] allow the developer to pass in a comparison function as a parameter.

Organization. In Section 2 we present the RAM version of our problem. We present the sorting problem in external-memory in Section 3 and relate it to the batched predecessor problem. We then derive lower and upper bounds on the batched predecessor problem in sections 5 and end with open problems in Section 6.

2 Warmup: the RAM Version

Two types, RAM version (2RAMSORT**).** The input is n red and n blue keys, and the output is the sorted sequence of all keys. A comparison between a

pair of red keys costs a, between a red key and a blue key costs b, and between a pair of blue keys costs c. Without loss of generality we can assume that $a < c$.

The optimal sorting cost in RAM depends on the final interleaving of the elements in the final sorted order. If in the final sorted order all red keys come before all blue keys, then $\Theta(an \log n + cn \log n)$ is the optimal total comparison cost, and the algorithm that separately sorts the two sets, and uses only one red-blue comparison to concatenate is optimal. However, if the red and blue keys alternate in the final sorted order, then no blue-blue comparisons are ever required to sort, rendering the previous algorithm suboptimal.

Stripes and the interleaving parameter k. A consecutive run of red or blue keys in the final sorted order is called a *stripe*. For simplicity we assume we have as many red stripes as blue. Define k to be the number of stripes, and let ℓ_i (respectively s_i) be the number of blue (respectively red) keys in stripe i. The notation ℓ and s are chosen to correspond with the later sections when red elements will be small and blue elements will be large.

Below we prove the tight bounds for the case $a < b < c$: this is the most natural case to serve as a warmup for the I/O-model, because if we consider the red elements small, blue elements large, then the comparisons involving red elements cost less than those involving blue elements, thus $a < b < c$.

Theorem 1. 2RAMSORT *has the following comparison cost complexity for the comparison-cost case $a < b < c$:*

$$\Theta(an \log n + b(k \log n + n \log k) + c \sum_{i=1}^{k} \ell_i \log \ell_i)$$

Proof. The lower bounds follow by counting the number of permutations needed to solve the following subproblems of 2RAMSORT:

1. The total number of permutations any algorithm for 2RAMSORT must achieve is at least $n!$. A binary comparison reduces the number of permutations by a factor of at most 2, and the cheapest comparison cost is a, thus giving a lower bound $a \log n! = \Omega(an \log n)$, our first term.

2. Consider the following instance of the problem, where the red elements are given as sorted, and the locations and the contents of stripes of blue elements are also provided but stripes are unsorted inside. This gives us a lower bound of $\Omega(c \sum_{i=1}^{k} \ell_i \log \ell_i)$, our third term.

3. Proving the second term as a lower bound involves addressing the following instance: the red elements are given sorted, and the algorithm is just required to discover the contents and the locations of k blue stripes. Once the algorithm solves this batched predecessor problem, the stripes are individually sorted for free.

There are at least $P = \binom{n}{k} S(n, k)$ permutations to consider, first term corresponding to finding k positions for stripes, and the second term, $S(n, k)$ is the *Stirling number of the second kind*, i.e., the number of ways to partition a set of size n into k non-empty, disjoint subsets, which corresponds to distributing contents among the k blue stripes. Considering that $S(n, k) \geq k^{n-k}$, we obtain

that $P \geq (n/k)^k k^{n-k}$. Since red elements are already sorted, comparisons of cost a are useless, and the cheapest available comparisons are those costing b. Thus we get a lower bound of $\Omega(b(k\log(n/k) + (n-k)\log k))$, which equals $\Omega(b(k\log n + n\log k - 2k\log k))$. One can then show by a case analysis that the last $-2k\log k$ term can be removed from the above expression, a detail we allocate to the full version.

Since we derived the lower bound on a constant number of instances of 2RAMSORT, we can claim a lower bound of the maximum complexity of these instances, that in turn equals the sum of the complexities in $\Omega(.)$ notation.

Due to space constraints, we refer the reader to the full version of the paper for the description of the upper bound. The steps in the upper bound match the respective lower-bound terms, by 1) sorting the red elements, 2) using two balanced binary trees, one for discovering new stripes of blue elements, and the other for assigning blue elements to existing stripes, and lastly, 3) sorting the stripes of blue elements.

3 Sorting and Batched Predecessor in External Memory with Size-Priced Information

The input to the two-sized sorting and batched predecessor problems are $\mathcal{S} = \{s_*\}$ (the small records) and $\mathcal{L} = \{\ell_*\}$ (the large records, each of size $1 < w \leq M/2$). A set of large elements forms a *stripe* if for each pair of large elements ℓ_i and ℓ_j in the stripe, there does not exist a small element between ℓ_i and ℓ_j in the final sorted order. Let k be the number of large-element stripes, and let the large-element stripes be $\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_k$, as they are encountered in the ascending sorted order. The parameters in the complexity analysis of sorting and batched predecessor are thus $S = |\mathcal{S}|$, $L = |\mathcal{L}|$, w, k, and $\{L_i\}_{i=1}^k$, where $L_i = |\mathcal{L}_i|$.

Definition 1 (*Two-Sized Sorting* $\mathrm{Sort}\,(S, L)$). *The input is an (unsorted) set of elements $N = S \cup L$. Set S consists of S unit-size elements, and L consists of L/w elements, each of size w.*[7] *The output comprises the elements in N, sorted and stored contiguously in external memory.*

Definition 2 (PLE-*Placement of Large Elements:*)**.** *The input is the* sorted *set of small elements $\mathcal{S} = \{s_1, s_2, \ldots, s_S\}$, and the* unsorted *set of large elements $\mathcal{L} = \{\ell_1, \ell_2, \ldots, \ell_{L/w}\}$. In the output, elements in $\mathcal{S}$ are sorted, and elements in L are sorted according to which stripe they belong to, but arbitrarily ordered within their stripe.*

Theorem 2 (Sorting complexity). *Denote by* $\mathrm{PLE}\,(S, L)$ *the complexity of the* PLE *problem. Then the I/O complexity of Two-Sized Sorting* $\mathrm{Sort}\,(S, L)$ *is*

[7] We overload notation for convenience of presentation. We assume $w \geq B$ also for the convenience of presentation. Our bounds hold for any $1 < w \leq M/2$ and are presented in the full version of the paper.

$$\Theta\left(\frac{S}{B}\log_{M/B}\frac{S}{B} + \mathrm{PLE}\,(S,L) + \left(\sum_{i=1}^{k}\left(\frac{L_i}{B}\log_{M/w}\frac{L_i}{w}\right) + \frac{L}{B}\right)\right).$$

The first term refers to the sorting the short elements, and the third term refers to sorting the individual stripes of large elements. The first term is identical to the conventional $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ bound by Aggarwal and Vitter [1]. The third term requires a slight generalization of that bound, because large elements are larger than a block, a proof we include in our full version.

As in the RAM setting, since we have three subproblems, their maximum complexity, and hence the complexity of their sum, is a lower bound on Sort (S, L). We have thus reduced the sorting problem to the batched predecessor problem, which will occupy the rest of this article.

3.1 Main Challenges in the Batched Predecessor Problem

To solve the PLE (S, L), we need the notion of a **fan-out**, which measures the efficiency of an I/O. Before an I/O, a large element has a set of candidate positions where it might land, and this interval gets reduced by a certain factor (possibly 1) after an I/O. The fan-out of an I/O is defined to be the product of all such factors for all large elements that reduce their search intervals during this I/O. Some of the challenges involved in adapting the RAM results (and classical DAM results) to the DAM model include:

1.Non-uniformity of record sizes: In the unit-sized setting, the transfer of a block to main memory can decrease the number of permutations by a factor of at most[8] $B!\binom{M}{B}$. In our setting, the number of comparisons performed by an I/O varies depending on whether the block transfer carries large records or small records, and what the contents of RAM are at the time of the I/O, which yields following possibilities:

- The transfer of a large element into main memory full of large elements gives only $\binom{M/w}{1}$ per w/B I/Os as a large-element transfer costs w/B.
- The transfer of B small records into main memory filled with small records gives $\binom{M}{B}$.
- The transfer of a large element into a memory full of M small elements gives a fan-out of $M + 1$.
- While the above three cases are tight, the main issue is in getting *an upper bound on how much a small block I/O can achieve*. The main memory can hold $p = (M - B)/w$ large elements, and an incoming small block has B

[8] The proof of the lower bound for sorting N unit-sized keys in [15] proceeds in the following fashion: assuming that all blocks are sorted (using a linear scan costing N/B I/Os), there are $N!/(B!)^{(N/B)}$ permutations required to achieve, and the transfer of a block of B sorted elements into the main memory containing $M - B$ sorted elements reduce the number of permutations by at most a factor $\binom{M}{B}$ (the "fan-out," since this is the degree of the node in the decision tree). Standard algebra gives a lower bound of $\Omega(\frac{N}{B}\log_{M/B}\frac{N}{B})$.

small elements. Naively the maximum fan-out can be upper bounded by B^p, which is not tight. Our main aim is to get a better bound on this fan-out.

Both our upper and lower bounds are a minimum of two terms, where one dominates the other depending on how large the large elements are.

2. Requiring output-sensitive lower bound limits adversarial arguments: Lower bounds on the unit-sized batched predecessor problem in external memory were recently obtained in [9]. In our setting, a more complicated adversarial analysis is required, that forms exactly k stripes at the end. Using this, we can argue a fan-out of at most 2^B on *most* small block I/Os.

4 Complexity of the Batched Predecessor Problem: Lower Bounds

The following theorem presents the $\mathrm{PLE}\,(S, L)$ lower bound:

Theorem 3 (PLE Lower Bound).

$$\mathrm{PLE}\,(S, L) = \Omega\left(\min\left\{\frac{kw}{B}\log_M S + \frac{L}{B}\log_M k, \frac{k}{B}\log S + \frac{L}{wB}\log k + \frac{L}{B}\right\}\right).$$

In order to prove Theorem 3, we first divide the $\mathrm{PLE}\,(S, L)$ problem further into three subproblems for which we develop a common adversary argument framework:

1. S-k: An instance with only *one large element* in each large-element stripe.
   - Input: Set S of unit-sized elements $s_1, \ldots, s_S$ *(sorted)*, where $s_1 = -\infty$ and $s_S = \infty$, and large elements $\ell_1, \ldots, \ell_k$ (volume kw) *unsorted*.
   - Output: For each ℓ_i output s_j such that $s_j \le \ell_i \le s_{j+1}$. It is guaranteed that no other ℓ_k satisfies $s_j \le \ell_k \le s_{j+1}$ (one large element per stripe).
2. k-$\tilde{k}$: An instance with only *one small element* in each small-element stripe.
   - Input: Unit-sized elements $s_1, \ldots, s_{k+1}$ *sorted*, where $s_1 = -\infty$ and $s_{k+1} = \infty$, and large elements $\ell_1, \ldots, \ell_{\tilde{k}}$ (volume $\tilde{k}w$) *unsorted*.
   - Output: For each ℓ_i, output its predecessor and successor in S.
3. k-k: An instance with only *one element in each stripe*, large or small.
   - Input: Unit-sized elements $s_1, \ldots, s_{k+1}$ *sorted*, where $s_1 = -\infty$ and $s_{k+1} = \infty$, and large elements $\ell_1, \ldots, \ell_k$ (volume kw) *unsorted*.
   - Output: The entire set in the sorted order.

The format of lower bounds for S-k, k-$\tilde{k}$ and k-k is as follows: let X be the logarithm of the total number of permutations that an algorithm needs to achieve in order to solve the problem. As is easily observed, the values of X (modulo constant factors) for these three subproblems are $k\log(S/k)$, $\tilde{k}\log k$, and $k\log k$, respectively. Lemma 1 below is the most technical part of this paper, and it helps us quantify the behavior of the adversary during small-block and large-element inputs for all three subproblems. We use this lemma to prove the lower bounds for the individual three subproblems (found in Lemma 2, Lemma 3, and Lemma 4). Then we put the three lemmas together to obtain the expression from Theorem 3.

Lemma 1. *Consider any algorithm for the S-k, k-$\tilde{k}$, or the k-k problem. There exists an adversary such that:*

- *On the input of any block of B short elements, the adversary answers comparisons between all elements in main memory such that the fan-out of this I/O is at most 2^B. In other words, the number of permutations the algorithm needs to check is reduced by a factor at most 2^B.*
- *On the input of any large element (costing w/B I/Os), the adversary answers comparisons between all elements in main memory such that the fan-out of this I/O is at most $O(M)$. In other words, the number of permutations the algorithm needs to check is reduced by a factor at most $O(M)$.*

Proof of Lemma 1: We prove this lemma by describing the adversary. We capture the information learned at every point of the algorithm by assigning a *search interval* $R(\ell) = (s_i, s_j)$ for a large element ℓ at step t as the narrowest interval of small elements where ℓ can possibly land in the final sorted order, given the information the algorithm has learned so far. *Bits* of information is the logarithm of the search interval (e.g., halving the search interval means learning one bit of information.)

It will be useful to consider the binary tree $\mathcal{T}$ on the set $\mathcal{S}$. The search interval of any large element at any point during the execution of the algorithm is a contiguous collection of leaves in $\mathcal{T}$.

For simplicity we will assume that the size of $\mathcal{S}$ is a power of 2, and hence $\mathcal{T}$ is perfectly balanced. Also, if $R(\ell) = (s_i, s_j)$ is the range of a large element, we will make sure the adversary "rounds off" the search space so that the new range corresponds exactly to a subtree of some node in $\mathcal{T}$. This is accomplished by first finding the least common ancestor *lca* of s_i and s_j, and then shrinking the search space of ℓ to either the search space in the left subtree of *lca* or to the search space in the right subtree of *lca*, whichever is larger. Thus each large element ℓ at any time has an associated node in $\mathcal{T}$, which we denote by $v(\ell)$. We also denote the interval corresponding to $v(\ell)$ (this is just the interval of its subtree) as $I(v(\ell))$.

Mechanics of the adversary's strategy: Our adversary will try to maintain the following invariant at all times during the execution of the algorithm.

Invariant: The search intervals of large elements in main memory are disjoint.

We denote by $\{\ell_i^{p-1}\}_{i=1}^{M/w}$ the set of at most M/w large elements in memory before the pth I/O. By hypothesis, the nodes in $\mathcal{T}$ belonging to the set $\{v(\ell_i^{p-1})\}_{i=1}^{M/w}$ have no ancestor-descendant relationships between them. We write S_i^{p-1} to denote $I(v(\ell_i^{p-1}))$, the search interval of large element ℓ_i^{p-1} at step $p-1$.

Small-block input. Consider the incoming block. We denote $n_{p,i}$ as the number of incoming small elements that belong to S_i^{p-1}. These elements divide S_i^{p-1} into $n_{p,i}+1$ parts $\{P_1, \ldots, P_{n_{pi}+1}\}$, some of them possibly empty. The largest of these parts (say P_j) is of size at least $1/(n_{p,i}+1)$ times the size of S_i^{p-1}. The new search interval of ℓ_i^p is defined to be the highest node in $\mathcal{T}$ such that $I(v) \subset P_j$.

Large element input. On an input of a large element ℓ_{new}^p (with search interval S_{new}^{p-1}), the adversary uses a strategy similar to that one on a small-block input

to compare ℓ^p_{new} with the (at most) M small elements present in memory. These M small elements divide S^{p-1}_{new} into at most M parts, and the new search interval of ℓ^p_{new} corresponds to the highest node in $\mathcal{T}$ that contains the largest part. This is the temporary search interval S_{new}, with the corresponding node v_{new}. S_{new} can be related to the search intervals of large elements in memory in three ways:

- Case 1: The element ℓ^p_{new} shares a node with another large element ℓ^p_i. The conflict is resolved by sending ℓ^p_{new} and ℓ^p_i to the left and right children of v_{new}, respectively.
- Case 2: The element ℓ^p_{new} has an ancestor in memory. The ancestor is sent one level down, to the child that does not contain v_{new} in its subtree. Thus the conflict is resolved while giving at most $O(1)$ bit.
- Case 3: The element ℓ^p_{new} has descendants in memory. Denote the nodes that are descendants of v_{new} in $\mathcal{T}$ as $v_1, \ldots, v_{M/w}$. Let the corresponding search intervals be $S^{p-1}_1, \ldots, S^{p-1}_{M/w}$, respectively. Let $X = \cup_{i=1}^{M/w} S^{p-1}_i$ and $Y = S_{\text{new}} \setminus X$. The set Y is a union of at most $M/w + 1$ intervals, each of which we denote by Y_i. Let Z be the largest interval from the set $\{S^{p-1}_1, \ldots, S^{p-1}_{M/w}, Y_1, \ldots, Y_{M/w}\}$. Hence, $|Z| \geq |S_{\text{new}}|/(2M/w)$.

  There are two cases to consider. The first case is when $Z = S^{p-1}_i$ for some i. In this case, $S_{\text{new}} = S^{p-1}_i$. In doing this we have given at most $O(\log M)$ bits. Now we proceed as in Case 1 to resolve the conflict with at most $O(1)$ extra bits. Otherwise, if $Z = Y_i$ for some i, then the adversary allots ℓ^p_{new} to the highest node v in $\mathcal{T}$ such that $I(v) \subseteq Z$.

We show that the above strategy produces the following guarantees (proof in full version):

1. On a small-block input, the adversary gives at most $O(\log(n_{p,i} + 1))$ bits to ℓ^p_i.
2. On a small-block input, the adversary gives at most $O(B)$ bits.
3. On the input of a large element, the adversary gives at most $O(\log M)$ bits.

4.1 Putting It All Together: Lower Bounds for S-k, k-$\tilde{k}$ and k-k

1) S-k Lower Bound. The proof rests on the following action of the adversary: in the very beginning, the adversary gives the algorithm the extra information that the ith largest large element lies somewhere between $s_{(i-1)\alpha}$ and $s_{i\alpha}$, where $\alpha = S/k$. In other words, the adversary tells the algorithm that the large elements are equally distributed across $\mathcal{S}$, one in each chunk of size S/k in $\mathcal{S}$. This deems the invariant of large elements in main memory having disjoint search intervals automatically satisfied. Since any algorithm that solves S-k must achieve $\Omega(k \log(S/k))$ bits of information, we have that

Lemma 2. S–k= $\Omega\left(\min\left(\frac{kw}{B}\log_M \frac{S}{k}, \frac{k}{B}\log\frac{S}{k} + \frac{kw}{B}\right)\right)$.

2) k-$\tilde{k}$ Lower Bound. To solve k-$\tilde{k}$, an algorithm needs to learn $k \log \tilde{k}$ bits of information. Using the adversary strategy we described, we observe:

Lemma 3. k-$\tilde{k}$ $= \Omega\left(\min\left(\frac{\tilde{k}w}{B}\log_M k, \frac{\tilde{k}}{B}\log k + \frac{\tilde{k}w}{B}\right)\right)$.

3) k-k Lower Bound. To solve k-k, an algorithm needs to learn $k \log k$ bits of information. In the k-k problem, we expect to produce the perfect interleaving of the small and large elements in the final sorted order. That is, *each element lands in its own leaf of* $\mathcal{T}$. Therefore, the adversary does not posses the freedom to route elements down the tree at all times using the strategy we described. Instead, the strategy is used for a fraction of total bits the algorithm learns, and the remaining fraction is used to make up for the potential imbalance created by sending more elements to the left or to the right. We call these *type one* and *type two* bits, respectively. Type two bits are effectively given away for free by the adversary.

More formally, we define the *node capacity* ($c^T(v)$) as the number of large elements that pass through v during the execution of an algorithm. If the k-k algorithm runs in T I/Os, then the node capacity of v at a level h of $\mathcal{T}$ is designated by $c^T(v) = k/2^h$.

Definition 3 (*type one* and *type two bits*). *A bit gained by a large element ℓ is an* type one *bit if, when ℓ moves from v to one of v's children, at most $c^T(v)/4 - 1$ other large elements have already passed through v. The remainder of the bits are* type two *bits.*

Because a small-block input gives $O(B)$ bits and a large-element input gives $O(\log M)$ bits, and we need to achieve all type one bits to solve the problem (there are $(k \log k)/4$ of them), we obtain the following lower bound:

Lemma 4. k–k= $\Omega\left(\min\left(\frac{kw}{B}\log_M k, \frac{k}{B}\log k + \frac{kw}{B}\right)\right)$.

Proof of Theorem 3 The lower bounds for k-k, k-$\tilde{k}$ and S-k are each a minimum of two terms; it is safe to add the respective terms as the transition between which term dominates occurs at exactly the same value of w for each of the subproblems. Adding the terms for the lower bounds of k-k and S-k provides the $\frac{k}{B}\log S$ and $\frac{kw}{B}\log_M S$ terms in Theorem 3. Adding the terms for the lower bounds of k-k and k-$\tilde{k}$, and using that $k + \tilde{k} = L/w$ provides the $\frac{L}{wB}\log S$ and $\frac{L}{B}\log_M S$ terms in Theorem 3. The theorem also holds for other item sizes ($w_1 \leq w_2 \leq B$) and we prove the generalized result in the full version.

5 Upper Bounds

Our algorithm for Sort (S, L) works in three steps:1) sort the short elements using traditional multi-way external memory merge-sort [1], 2) solve the associated PLE (S, L) problem, and 3) sort the long stripes obtained again using multi-way mergesort. The first and third steps give the first and third terms in the sorting complexity in Theorem 2.

We give two algorithms to solve PLE (S, L): PLE-DFS and PLE-BFS. The final upper bound is the minimum of the two terms, as presented in Theorem 4.
PLE-DFS: PLE-DFS builds a static B-tree T on S, and searches for large elements in T one by one. This approach is preferred in the case of really large elements, and it is better to input them fewer times.

We dynamically maintain a smaller B-tree T' that contains only *border elements* (the two small elements sandwiching each large element in the final sorted order) and has depth at most $\log_B k$. All large elements first travel down T' to locate their stripe. Only those elements for which their stripe has not yet been discovered need to travel down T. After a new stripe is discovered in T, it is then added to T'. The total cost becomes

$$O\left(\frac{L}{w}\log_B k + k\log_B S + \frac{L}{B} + \frac{S}{B}\right). \tag{1}$$

PLE-BFS: Our second algorithm for PLE uses a batch-searching tree with fanout $\Theta(M)$. When a node of the tree is brought into memory, we route all large elements via the node to the next level. We process the nodes of the M-tree level by level so all large elements proceed at an equal pace from the root to leaves. This technique is helpful when large elements are sufficiently small so that bringing them many times into memory does not hurt while they benefit from a large fanout.

The analysis is as follows: at each level of M-tree, the algorithm spends $\Theta(L/B)$ I/Os in large-element inputs. Every node of the tree is brought in at most once, which results in total $O(S/B)$ I/Os in small-element inputs. The total number of memory transfers for PLE-BFS then becomes $O\left(\frac{L}{B}\log_M S + \frac{S}{B}\right)$.

Our final upper bound is the better of the two algorithms:

Theorem 4 (PLE Upper Bound).

$$\mathrm{PLE}(S,L) = O\left(\min\left(\frac{L}{B}\log_M S + \frac{S}{B}, \frac{L}{w}\log_B k + k\log_B S + \frac{L}{B} + \frac{S}{B}\right)\right).$$

Substituting the lower and upper bounds of the batched predecessor problem ($\mathrm{PLE}(S,L)$) derived in Theorems 3 and 4 into the complexity of sorting in Theorem 2 gives us lower and upper bounds on the sorting problem $\mathrm{Sort}(S,L)$.
Remark 1: One observes that in Theorem 4 ($\mathrm{PLE}(S,L)$ lower bound), the transition between the two terms in the minimum happens at $w = B\log M$. This is because when large elements are very large, the bound obtained by algorithms that do not input the large elements too often (PLE-DFS) is smaller than algorithms that input large elements multiple times (e.g., PLE-BFS).
Remark 2: The upper and lower bounds on $\mathrm{PLE}(S,L)$ are tight for a wide range of parameters. Moreover, if the first and third terms in the complexity of sorting (Theorem 2) dominate the complexity of the associated $\mathrm{PLE}(S,L)$ problem, our sorting algorithms are tight.

6 Conclusion and Open Problems

We derived upper and lower bounds on sorting and batched predecessor in the RAM and DAM models, when comparison or I/O costs depend on the length of the items being compared. In many settings, we show that the optimal sorting algorithm involves the optimal batched predecessor problem as a subroutine, and develop algorithms for the batched predecessor problem.

While our results are for the two-size setting, we would like to point out that our algorithms generalize to the multiple-sizes setting. However, generalizing our lower bound techniques to the multiple-size setting requires more ideas.

References

1. Aggarwal, A., Vitter, J.: The input/output complexity of sorting and related problems. Commun. ACM **31**, 1116–1127 (1988)
2. Alon, N., Blum, M., Fiat, A., Kannan, S., Naor, M., Ostrovsky, R.: Matching nuts and bolts. In: Proc. SODA. pp. 690–696 (1994)
3. Angelov, S., Kunal, K., McGregor, A.: Sorting and selection with random costs. In: Latin American Symposium on Theoretical Informatics. pp. 48–59. Springer (2008)
4. Arge, L.: The buffer tree: A technique for designing batched external data structures. Algorithmica **37**(1), 1–24 (2003)
5. Arge, L., Ferragina, P., Grossi, R., Vitter, J.: On sorting strings in external memory (extended abstract). In: Proc. STOC. pp. 540–548 (1997)
6. Arge, L., Knudsen, M., Larsen, K.: A general lower bound on the I/O-complexity of comparison-based algorithms. In: Proc. WADS. pp. 83–94 (1993)
7. Arge, L., Procopiuc, O., Ramaswamy, S., Suel, T., Vitter, J.: Theory and practice of I/O-efficient algorithms for multidimensional batched searching problems. In: Proc. SODA (1998)
8. Bender, M.A., Hu, H., Kuszmaul, B.C.: Performance guarantees for B-trees with different-sized atomic keys. In: Proc. PODS. pp. 305–316 (2010)
9. Bender, M.A., Farach-Colton, M., Goswami, M., Medjedovic, D., Montes, P., Tsai, M.: The batched predecessor problem in external memory. In: Proc. ESA. pp. 112–124 (2014)
10. Berkeley DB C API Reference: `set_bt_compare`, `http://www.berkeleydb.com/`
11. Charikar, M., Fagin, R., Guruswami, V., Kleinberg, J., P. Raghavan, A.S.: Query strategies for priced information. In: Proc. STOC. pp. 582–591 (2000)
12. Cicalese, F., Laber, E.: A new strategy for querying priced information. In: Proc. STOC. pp. 674–683 (2005)
13. Diehrand, G., Faaland, B.: Optimal pagination of B-trees with variable-length items. Commun. ACM **27**(3), 241–247 (1984)
14. Elmasry, A.: Distribution-sensitive set multi-partitioning. In: 1st International Conference on the Analysis of Algorithms (2005)
15. Erickson, J.: Lower bounds for external algebraic decision trees. In: Proc. SODA. pp. 755–761 (2005)
16. Gupta, A., Kumar, A.: Sorting and selection with structured costs. In: Proceedings 42nd IEEE Symposium on Foundations of Computer Science. pp. 416–425. IEEE (2001)
17. Larmore, L., Hirschberg, D.: Efficient optimal pagination of scrolls. Commun. ACM **28**(8), 854–856 (1985)
18. McCreight, E.: Pagination of B*-trees with variable-length records. Commun. ACM **20**(9), 670–674 (1977)
19. Munro, J., Spira, P.: Sorting and searching in multisets. SIAM J. Comput. **5**(1), 1–8 (1976)
20. Pinchuk, A., Shvachko, K.: Maintaining dictionaries: Space-saving modifications of b-trees. In: Database Theory ICDT '92, vol. 646, pp. 421–435. Springer Berlin Heidelberg (1992)
21. The GNU C Library: qsort, `http://www.gnu.org/software/libc/manual/`