Permuting and Batched Geometric Lower Bounds in the I/O Model

Peyman Afshani[1] **and Ingo van Duijn**[2]

1 **MADALGO**[*]**, Department of Computer Science, Aarhus University, Aarhus, Denmark**
`peyman@cs.au.dk`

2 **MADALGO, Department of Computer Science, Aarhus University, Aarhus, Denmark**
`ivd@cs.au.dk`

Abstract

We study permuting and batched orthogonal geometric reporting problems in the External Memory Model (EM), assuming indivisibility of the input records. Our main results are two-fold. First, we prove a general simulation result that essentially shows that any permutation algorithm (resp. duplicate removal algorithm) that does $\alpha N/B$ I/Os (resp. to remove a fraction of the existing duplicates) can be simulated with an algorithm that does α phases where each phase reads and writes each element once, but using a factor α smaller block size.

Second, we prove two lower bounds for batched rectangle stabbing and batched orthogonal range reporting queries. Assuming a short cache, we prove very high lower bounds that currently are not possible with the existing techniques under the tall cache assumption.

1998 ACM Subject Classification F.2.2. Nonnumerical Algorithms and Problems, G.2.1. Combinatorics

Keywords and phrases I/O Model, Batched Geometric Queries, Lower Bounds, Permuting

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.2

1 Introduction

The I/O model [7] is the well-established model to design and analyze algorithms for massive data. In this model, the internal *memory* has size M and the input data is stored in a *disk* of infinite size that is divided into *blocks* of size B. The transfer of data between disk and the memory is done via *I/Os* where each I/O can read or write one block. We define $m = M/B$. All computation must take place in the internal memory. The goal is to minimize the total number of I/Os. This is an elegant model for problems where the size of the input data far exceeds the size of the available memory. Sometimes, algorithms require that $M \geq B^{1+\varepsilon}$ for a constant ε and this is known as the *tall cache* assumption (and the converse as the *short cache* assumption).

Batched Input with Constrained Output. The I/O model has been extensively studied [9, 8, 24]. In this paper, we will focus on proving lower bounds for batched geometric problems as well as engaging in a more in-depth study of the permutation algorithms. The two important batched problems that we study are the following.

[*] MADALGO Center for Massive Data Algorithmics is supported in part by the Danish National Research Foundation grant DNRF 84.

▶ **Problem 1** (Batched rectangle stabbing (BRS)). *The input comprises a set I of N axis-aligned rectangles and a query set Q of N points in $\mathbb{R}^d$.*

▶ **Problem 2** (Batched orthogonal range reporting (BORR)). *The input comprises a set P of N points and a query set R of N axis-aligned rectangles in $\mathbb{R}^d$.*

In a batched query problem, it is often required that the output should consist of all the pairs (e_i, q_j) where e_i is an input element that matches the query q_j. In this case, which we call the *paired output format*, the two problems stated above are equivalent; both output the set of incidence between an input set of points and rectangles.

In this paper, we consider a different query output format: for every query q_j, we require that all the input elements that match q_j must be placed consecutively in the output. In other words, the algorithm should list the answer to q_j fully before answering any other query. However, there is no restriction on the order in which the queries are answered nor on the order of elements reported for each query. We call this *query output format.* Thus, BRS and BORR are equivalent when we consider the paired output format but they could behave differently if we consider the query output format.

As we shall see shortly, a very connected research direction is in-depth study of algorithms that permute a given set of input elements in the I/O model. A major or interesting (in our opinion) rather open-ended unsolved questions are the following.

▶ **Question 3.** *Can one prove an $\omega(N/B)$ lower bound assuming $M > B^2$ for*
(i) *explicit permutations*
(ii) *or general permuting using any proof technique that is not based on counting?*

▶ **Question 4.** *Let $\mathcal{A}$ be an algorithm that can compute some permutation π of a given N input elements in $\alpha N/B$ I/Os, for some parameter α. Can we transform $\mathcal{A}$ into another algorithm $\mathcal{A}'$ that computes the same permutation π using $\mathcal{O}(\alpha N/B)$ I/Os, such that $\mathcal{A}'$ has a "usefully structured canonical" form, e.g., it uses simple permutation algorithms as building blocks?*

Previous work. Sorting and permuting are possibly the two most fundamental problems in the area of I/O algorithms, with permuting being one of the first problems studied in an I/O setting [19]. Sorting N elements requires $\mathcal{O}(\mathrm{Sort}(N)) = \mathcal{O}\big(\frac{N}{B}\log_m \frac{N}{B}\big)$ I/Os and this bound is tight [7]. The permutation problem is very similar to the sorting problem where the goal is to produce (possibly an implicitly defined) permutation of the input elements. It is also known that any permutation can be performed in $\mathcal{O}(\mathrm{Sort}(N))$ I/Os and there exists permutations that require asymptotically that many I/Os; however, the proof is existential and no such explicit permutation is known to this date [7]. This lower bound (as well as many of the lower bounds in the I/O model) are proved in the so-call *Indivisibility Model*: the data elements are assumed to be indivisible and atomic and each block can store B data elements and the only computation allowed on the atomic elements is to move, delete, or copy them (to or from memory). All other information or computation (unless explicitly mentioned) is free. In the rest of this article, we will only focus on algorithms that work in the indivisibility model. Within the context of permutations in the indivisibility model, there has been attempts to answer Question 3 (or alternatively, to study "easy" permutations) but all the known explicit permutations can be shown to be easier [7, 16, 21] and in particular, they all can be done in $\mathcal{O}(N/B)$ I/Os when we do not have a short cache.

Additionally, there has been a lot of interest in batched problems. For example, in a survey Vitter [24] cites 12 different problems that can be answered in $\mathcal{O}(\mathrm{Sort}(N) + K/B)$

I/Os where N is the total input size and K is the total output size. See also [10, 13, 17, 18, 20]. In particular Arge et al. [11] show that a slightly less restrictive version of Problem 2 can be solved in $\mathcal{O}\Big(N/B \log_m^{d-1} N/B + K/B\Big)$ I/Os. These results produce paired output format.

For the lower bounds, the permutation and sorting lower bounds as well as a problem known as "proximate neighbors" [15], provide a basis of $\Omega(\mathrm{Sort}(N))$ lower bounds for a lot of problems, including problems with batches of N input elements and N queries. Showing a lower bound of roughly $\Omega(\mathrm{Sort}(N))$ for smaller batches is more difficult but some such results are also known [4, 6] (although not explicitly stated in these papers). Lower bounds for dynamic batched queries have also been proved [5]. In general, $\Omega(\mathrm{Sort}(N))$ is the only lower bound available for all of these problems, in particular because in the indivisibility model we can consider any algorithm that solves a batched problem as an algorithm that computes an implicitly defined permutation of the input elements (possibly with duplicates).

Our results. In relation to Question 4, we prove a simulation result that shows any algorithm in the indivisibility model that performs αn I/Os such that it reads and writes each element $\mathcal{O}(\alpha)$ times, can be "simplified" into an algorithm that performs $\mathcal{O}(\alpha)$ rounds where in each round each element is read and written once, using α factor smaller blocks.

In relation to the batched problems and assuming query output format, we prove that if a data structure answers BRS queries in $f(N) + c_0K/B$ I/Os, for a constant c_0, then $f(N) = \frac{N}{\log B + \log\log\frac{N}{B}} \cdot \left(\frac{\log N}{m^{O(\alpha)}}\right)^{d-1}$, assuming $m \le B^{\varepsilon}$ for a small enough constant ε. For the BORR problem, then we prove $f(N) = \Omega\Big(\frac{N}{B}\log_m^{d-1}(N)\Big)$. Interestingly, this might mean that BRS is a more difficult problem than BORR in the query output format.

1.1 Preliminaries

Technical barriers. The indivisibility model has been extremely successful in proving lower bounds for algorithmic and data structure problems. However, despite the considerable attention, there are still some very natural questions left open. For instance, we consider Question 3 as a major open question. The situation becomes more exasperating when one considers that the known existential proof in fact shows that almost all permutations should require $\Omega(\mathrm{Sort}(N))$ I/Os to permute but yet, we do not know of a single permutation that even requires $\omega(N/B)$ I/Os. Furthermore, the existential proof (as well as the comparison-based lower bounds for sorting) only can show a $\Omega(\log_m(N!)) = \Omega(\mathrm{Sort}(N))$ lower bound for *any* reasonably defined batched problem. For example, we can only obtain a $\Omega(\mathrm{Sort}(N))$ lower bound for the d-dimensional BORR problem (for a constant d) since the total number of "combinatorially" different point sets of size N in $\mathbb{R}^d$ is at most $N!^d$ and $\log_m(N!^d) = \Theta(\mathrm{Sort}(N))$ for a constant d. Obviously, it is extremely unlikely that this bound is tight and that the d-dimensional BORR problem can be solved in $\mathcal{O}(\mathrm{Sort}(N))$ I/Os.

However, if we assume a short cache, then both of these obstacles go away: we can in fact show lower bounds for explicit permutations such as the matrix transpose permutation and using a different proof strategy [7]. So the natural question becomes, can we actually prove meaningful lower bounds for batched geometric queries under the short cache assumption? Apart from the above considerations, this is also motivated by the desire to understand the effects of short cache on the performance of the algorithms.

Hong-Kung's rounds. While trying to prove a lower bound for the complexity of fast Fourier transform, Hong and Kung [23] presented a general transformation of any I/O algorithm into a more standard form that works in rounds. While their transformation is originally presented for $B = 1$, it is easily generalizable to larger block sizes. We can thus present their transformation as follow.

▶ **Theorem 5.** *An I/O algorithm $\mathcal{A}$ that runs in a machine with memory size M can be transformed into an equivalent algorithm $\mathcal{A}'$ with the same asymptotic running time on a machine with memory size $2M$ and the same block size such that $\mathcal{A}'$ runs in rounds and during each round, $\mathcal{A}'$ first reads $2M/B$ blocks, performs some computation and then writes $2M/B$ blocks and clears the memory.*

The increase in the block size of the machine in the above theorem is not consequential. It is easy to show that two machines where the block sizes and memory sizes differ only by a constant amount are equivalent, up to constant factors.

▶ **Corollary 6.** *Let $\mathcal{A}$ be an algorithm that works in Hong and Kung's rounds that creates a permutation π of a set of N input elements using $\alpha N/B$ I/Os. At least half of the elements are written at most $\mathcal{O}(\alpha)$ times. Therefore, every such element occurring in an output block can be traced back to one of $m^{\mathcal{O}(\alpha)}$ possible input blocks.*

Proof. By an averaging argument, not more than half of the elements can be written more than 2α times, thus at least half of the elements are written at most 2α times. Since $\mathcal{A}$ works in Hong-Kung rounds, we can trace the elements in an output block to $2m$ other blocks written previously by the algorithm. Those elements, subsequently can be traced back to $(2m)^2$ other blocks. For the elements that are written $\mathcal{O}(\alpha)$ times, the output block is traced back to $m^{O(\alpha)}$ input blocks. ◀

2 Universal External Permuting Algorithm

To study the hardness of permuting, we need to consider arbitrary algorithms that perform a specific permutation. That is, the hardness of a permutation is determined by the optimal algorithm performing it. Often, one admirable goal towards this end is to reduce any permuting algorithm into a "canonical" permuting algorithm that is simpler and easier to study. In fact, Hong and Kung's rounds is one such attempt. However, we would like to probe much deeper. Our basic building block is the following.

2.1 Blocked Shuffle Exchange

To simplify analysing external memory permuting lower bounds, we only consider a single type of algorithm that we call a Blocked Shuffle Exchange (BSE).

▶ **Definition 7.** In a machine with block size B and memory size M, *a blocked shuffle exchange with α phases* is an algorithm with the following structure.
(i) it runs in α phases
(ii) in each phase, the algorithm does the following until
all elements are read and written once: read at most $m = M/B$ blocks into the memory, write some permutation of the read elements to the disk, and then clear the memory.

The goal is to show that we can (partially) simulate any permuting algorithm with a BSE. In particular, the goal is to simulate an algorithm $\mathcal{A}$ that uses at most $\alpha N/B$ I/Os, with a BSE containing $O(\alpha)$ phases. We can in fact do this but under two caveats. The first caveat is that we only simulate the permutation of the elements that are written $O(\alpha)$

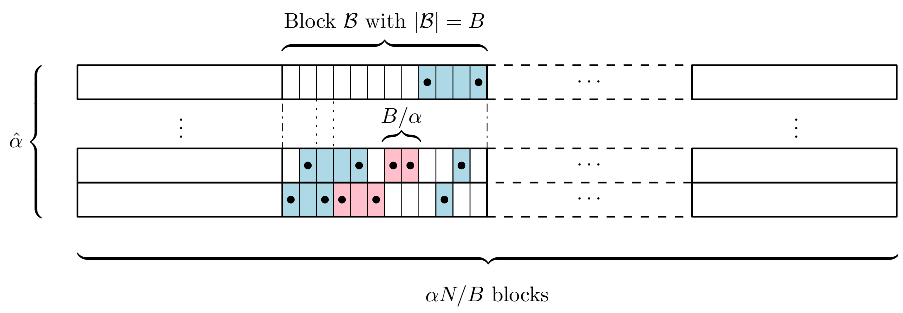

Figure 1 The write history of $\mathcal{A}$ consists of $\alpha N/B$ blocks of size B. Sparse layer blocks can be compactified using blocks of size B/α (2 in this example). Note that all columns contain at most one element.

times. This is necessary because of some (rather uninteresting) bad examples: an algorithm that sorts the first $O(N/\log N)$ elements of an input using N/B I/Os, obviously cannot be simulated with $O(1)$ phases of a BSE. However, the algorithm reads and writes a small portion of the elements many times while not touching the rest. So it is only meaningful to demand a simulation on the subset of the input elements that are not read or written many times. This is what we demand with the first caveat. For the second caveat (that we do not know if it is necessary or not) we define work as block size times number of I/Os performed; for an optimal simulation in terms of work, we need to run our simulation BSE on a smaller block size. The exact formulation of our result is the following.

▶ **Theorem 8.** *Let $\mathcal{A}$ be an algorithm that creates a permutation π of a set of N input elements using $\alpha N/B$ I/Os. Furthermore, we assume that $\mathcal{A}$ writes any element $O(\alpha)$ times.*

Then, we can create a BSE *that creates π using $O(\alpha)$ phases and* either *(i) uses $\alpha^2 N/B$ I/Os* or *(ii) uses the same amount of work but using blocks of size B/α .*

Proof. Observe that we can assume $\mathcal{A}$ writes every element exactly $\hat{\alpha} := c\alpha$ times for a constant c; if some elements are written fewer times, we can just read them and perform dummy writes.

To describe a BSE, we model the *sequential write history* of $\mathcal{A}$. That is, all the writes that $\mathcal{A}$ makes laid out sequentially in the order in which they are written. Now conceptually imagine having $\hat{\alpha}$ copies of this array stacked on top of each other, where each copy forms a *layer*. Every write performed by $\mathcal{A}$ thus corresponds to a column that is composed of $\hat{\alpha}$ *layer-blocks* stacked on top of each other. Assume the layer-blocks in one block are numbered from one to $\hat{\alpha}$, so that the ith block the kth column contains all elements written for the ith time at the kth write. Thus, simulating an I/O by algorithm $\mathcal{A}$ corresponds to reading or writing in the corresponding column.

The observation is that in the simulation, we can compute the $i+1$st layer by only reading from the ith layer. This follows from the fact that every read $\mathcal{A}$ makes is from a previously written block (or input block), and to produce all the $i+1$st writes only requires reading elements written i times. Thus, to compute the next layer, we run $\mathcal{A}$ but replace every read with a read to the corresponding block in the ith layer (and similar for writing to the $i+1$st layer). Since $\mathcal{A}$ uses $\alpha N/B$ I/Os, computing the next layer also takes at most that many I/Os. Since layer-blocks can be very *sparse*, this gives a work of $\hat{\alpha}\alpha N = \mathcal{O}(\alpha^2 N)$.

To achieve $\mathcal{O}(\alpha N)$ work, the layer blocks are tightly packed in smaller B/α-sized blocks. Every simulated I/O is now a sequence of densely filled B/α-sized blocks and one additional sparse block. Since every element occurs exactly once per layer, there are at most $N/(B/\alpha) =$

$\alpha N/B$ dense I/Os per layer. The same bound holds for sparse I/Os, since there are $\hat{\alpha}N/B$ columns, and at most one sparse I/O per column. Together, this yields $\hat{\alpha}\alpha N/B$ I/Os and $(\hat{\alpha}\alpha N/B)(B/\alpha) = \hat{\alpha}N$ work. ◀

The factor α reduction in block size in this result might not be optimal. For small block size, it might happen that $\alpha = \Omega(B)$, and thus the simulation essentially becomes an internal memory simulation. However, for simulations where α is a constant, the theorem is particularly useful.

▶ **Corollary 9.** *To prove that an explicit permutation π requires $\omega(N/B)$ I/Os (and thus $\omega(N)$ work), it is sufficient to prove that permuting π with a* BSE *requires $\omega(N)$ work.*

2.2 Abstract Duplicate Removal

As we show in Section 3, creating the output of a batched problem is not modelled as a permutation problem, but as a duplicate removal problem. Essentially, we can think of the algorithm as an algorithm that runs "backwards" and given the output of the batched problem, it is trying to remove all the duplicates and produce the input set of elements. Because of this, we prove a different simulation result that shows a duplicate removal algorithm can be manipulated to produce a particular permutation of a subset of the elements with some nice properties. Before stating our simulation result, we need to introduce some definitions pertaining to duplicate removal.

▶ **Definition 10.** Consider a set S of K atomic elements together with an equivalence relation $\equiv$ defined on S. An element e_1 is a *duplicate* of an element e_2 if and only if $e_1 \equiv e_2$. The duplicate removal problem is the problem of finding the quotient set or specifically, it is the problem of finding a subset $S' \subset S$ of N elements such that no two elements in S' are equivalent but for every element in S there is an equivalent element in S'.

The duplicate removal problem is trivial if the algorithm has full knowledge of which elements are duplicates and if we only care about the movement of the elements. However, such an algorithm is highly unrealistic. To tie up the algorithm into a more realistic behavior, we force the algorithm into *duplicate elimination framework (DEF).*

1: The algorithm starts with an input of K atomic elements, but with no knowledge of the equivalent relation $\equiv$.
2: At cost of one I/O, the algorithm can read or write a block.
3: The algorithm can move or delete elements in the main memory.
4: The algorithm works in the Hong-Kung's rounds.
5: The algorithm can detect all elements e_1, e_2 in the main memory s.t., $e_1 \equiv e_2$. From now on, the algorithm remembers this for free, for all copies of e_1 and e_2.

Crucially, an algorithm $\mathcal{A}$ can actually delete all copies of an element, if it detects that it is a duplicate of another element. This is a problem for showing lower bounds for the batched problem since this operation can shrink the input size of the duplicate removal algorithm, leaving an easier instance of the problem. In the following theorem, we overcome this difficulty.

▶ **Theorem 11.** *Consider an algorithm $\mathcal{A}$ that works in the DEF and given an input S of size K it detects a subset $S' \subset S$ of $K/2$ duplicate pairs in $\alpha K/B$ I/Os.*

Then, using $O(\alpha K/B)$ I/Os, and using a machine with $M' = M + B$ memory, we can create a permutation of a subset $S'' \subset S$ such that S'' contains $K/4$ pairs of elements (e, e') of S where e is a duplicate of e' and e and e' are placed in the same block.

Proof. Our overall proof strategy is as follows. We allocate a special buffer of size B in the memory where we collect pairs of elements (e, e') such that e is a duplicate of e'. Once the special buffer is full, we write them to the disk. To fill the special buffer we simulate $\mathcal{A}$ two times: once forward and once backwards. During the backwards execution of $\mathcal{A}$ we make some modifications where instead of writing an element e into a block $\mathcal{B}$, we may write an element e' instead. This means that, in the future, when we read the block $\mathcal{B}$, we will have the element e' instead of e. We continue the backwards execution of $\mathcal{A}$ while treating e' the same way e was treated; this is possible since $\mathcal{A}$ only moves or copies the element e and both can be applied obliviously to e' instead. It is important to observe that $\mathcal{A}$ might copy elements (consider it bookkeeping), even though it is a duplicate removal algorithm. Ultimately, what we want to show is that by using the sequence of I/Os that $\mathcal{A}$ performs, we can create an algorithm that produces a permutation of the input such that at least half of the elements reside in a block with at least one equivalent element. In order to show this, we define two notions.

Consider the original execution of $\mathcal{A}$. First, every element e defines a *copy tree* $C(e)$, which is a rooted tree, as follows. There is a node in $C(e)$ for every time e was loaded into a Hong-Kung round. The root of $C(e)$ is the first time the element e is loaded into memory. More than one copy of e could be in memory in a specific round. Therefore, pick one arbitrarily to be the representative that round. Two nodes u and v in $C(e)$ are connected if the block u is loaded from was written in the round where v was the representative. Note that this implies that $C(e)$ is a path if e is only ever moved around and never duplicated.

Second, for every equivalence class $\mathcal{E}$ we define an *equivalence tree* $T(\mathcal{E})$. Two elements e_i and e_j in $T(\mathcal{E})$ are connected if the algorithm discovered their equivalence. This implies in particular that (some copies of) e_i and e_j were in the memory at the same time. It is therefore easy to write all edges of $T(\mathcal{E})$ to a special buffer during the execution of $\mathcal{A}$. However, this would not be a permutation since every element of the equivalence tree is written as many times as its degree.

The basic idea is to output siblings of $T(\mathcal{E})$ in pairs, so that in the at least half of the elements of $T(\mathcal{E})$ are output as disjoint pairs. There are two obstacles with this approach. The most important is that siblings might not reside in memory at the same time. The other obstacle is that nodes might not have an even number of children.

First we show how to handle the second obstacle with the following grouping scheme (the algorithm does not actually perform these operations, but it is considered known by the algorithm). Consider the deepest internal node e. If it has an even number of children they are marked to belong to the same *group*, meaning that they will be paired up later. If e has an odd number of children, then e is grouped together with its children. In either case, e and its children are discarded and the scheme is repeated until there are no nodes left to group. Note that some elements are not grouped (i.e. those that had an even number of children), but at least half of the elements will be grouped. The goal now is to pair each element with exactly one member of its group.

We first run $\mathcal{A}$ with a memory of size $M+B$. The special buffer is used to write discovered pairs to disk, and it is written to the output section on disk when it is full. If an element meets a member of its group in the memory, we write the pair to the buffer. The two elements are now disregarded for the rest of the procedure, meaning they will not be paired up with other elements anymore. Note that all nodes that were grouped with their children have been written to disk. What is left is to show how to pair up the remaining unpaired siblings.

We do this by performing the I/Os of $\mathcal{A}$ backwards by considering writes as reads and vice versa. Elements that have already been paired up will not be used to form new pairs, but are still moved around in the backwards run. Consider the situation where the algorithm

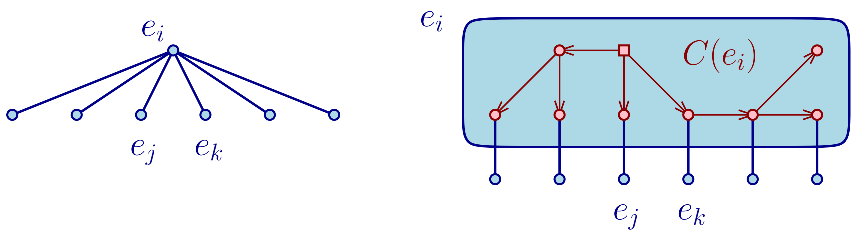

Figure 2 An element e_i and its children in the equivalence tree (left), and e_i as a supernode showing the underlying copy tree $C(e_i)$ (right).

"discovers" an equivalence between e_i and e_j, where e_i is the parent in the equivalence tree and e_j is still unpaired. Since e_i is either already paired in the forward run, or will not be paired at all, we can safely substitute it with e_j. That means that for the rest of the backwards run, every copy of this e_i upwards in the copy tree is now replaced by e_j. The claim is that all unpaired siblings will be paired up in this way.

Figure 2 depicts part of the equivalence tree of e_i. The two children e_j and e_k were never in memory at the same time. However, if we replace the copy of e_i (call it a) that discovers e_j by e_j (and similarly for the copy b that discovers e_k) the following happens. The substituted elements will end up in the memory at the same time, namely at the round where some copy of e_i wrote copies a and b to disk. At this point the elements are written to the buffer and ignored for the rest of the backwards run as usual. If more than two substituted elements meet in memory, then at most one (namely the one that was not written to the buffer) will propagate up the copy tree of e_i. By construction, an even number of siblings were left, so all of them will eventually be paired and written to the buffer.

Thus, all elements that were grouped are written to disk exactly once, and always paired. Since at least half of the elements were grouped, the proof is complete. ◀

3 Batched Lower Bounds for Short Cache

In this section, we describe our lower bounds for offline problems under the short cache assumption. As discussed earlier, a major open problem is to obtain some non-trivial lower bound of $\omega(\text{Sort}(N))$ for some offline problem without the short cache assumption and unfortunately, none of the known techniques seem capable of doing that.

In general, proving lower bounds for geometric problems involves first building a "difficult" input set and then proving that the input is indeed difficult. For our problems, this first part is now considered standard since there have been plenty of lower bounds that have been using similar set of basic constructions of points and rectangles [2, 3, 12, 14, 22].

These standard constructions have been summarized in the following theorems.

▶ **Theorem 12.** *[2, 3, 14] For any parameter n, we can place a set P of n points inside the unit cube $\mathcal{U}$ in $\mathbb{R}^d$ such that for any two points $p, q \in P$, the rectangle created by p and q has volume $\Omega(1/n)$. Furthermore, any rectangle of volume v contains $\Omega(vn - O(1))$ points.*

▶ **Theorem 13.** *[3, 14, 12] For any two parameters n and ℓ, $2 \le \ell \le n^{1/3}$, we can place a set R of n rectangles inside the unit cube $\mathcal{U}$ in $\mathbb{R}^d$ with the following properties. There are $t := c_t(\log_\ell n)^{d-1}$ types of rectangles, for a constant c_t, with each type having the dimensions $\left(\frac{1}{\ell}\right)^{i_1} \times \left(\frac{1}{\ell}\right)^{i_2} \times \cdots \left(\frac{1}{\ell}\right)^{i_{d-1}} \times \frac{t\ell^{i_1+i_2+\cdots i_{d-1}}}{n}$, for some integers $i_x \in \{0, \ldots, \log_\ell(n/t)\}$. The set R has the following properties:*

(i) *each rectangle has volume* $\frac{t}{n}$*,*
(ii) $\Theta(\frac{n}{t})$ *rectangles of each type are sufficient to tile* $\mathcal{U}$*,*
(iii) *every two rectangles of same type are disjoint, and*
(iv) r *rectangles that have distinct types intersect at a volume at most* $\frac{t}{n\ell^r}$*.*

3.1 Batched rectangle stabbing problem

In BRS we are given an input set I of N rectangles and a query set Q of N points. The goal is to find for every point q in Q, the set of rectangles that contain q. For every query point q, the algorithm is required to output the set of rectangles that contain q in contiguous blocks. However, the algorithm is given freedom to choose the order in which to report the queries, and within each query, the order of the rectangles that contain q.

▶ **Theorem 14.** *Let $\mathcal{A}$ be an algorithm that given the input sets I and Q for the BRS problem, answers the queries in query order format and in $f(N) + c_0K/B$ I/Os for a constant c_0. We prove that $f(N) = \frac{N}{\log B + \log\log n} \cdot \left(\frac{\log N}{m^{O(c_0)}}\right)^{d-1}$, assuming $B = \Omega(\log\log N)$.*

The first step is to construct the difficult input sets. First, we create a set Q of N points using *Theorem* 12. Then, using Theorem 13, we create a set I_1 of n initial rectangles for a parameter n. Next, we "clone" each initial rectangle β times, where β is a parameter. This is inspired by a data structure lower bound of [1]. Specifically, we create β copies of each initial rectangle and then place the copies in the input set I. Thus, we can construct a set I of $N = n\beta$ rectangles. One should think of the clones as slightly perturbed copies of the original rectangles, meaning, the cloned rectangles are distinct atomic rectangles. However, for simplicity we consider them to cover the same area. If an initial rectangle is stabbed by k query points, all its clones are said to have multiplicity k.

Thus, we have a set Q of query points, and a set I of rectangles. Assume that the algorithm decides to answer the set of queries in the order $< q_1, \cdots, q_N >$. For each q_i, let I_{q_i} refer to the subset of I that contains the point q_i. By Theorem 13 and because of our cloning, I_{q_i} contains $t\beta$ rectangles where $t = (\log_\ell n)^{d-1}$. The output of the algorithm can therefore be described as $O := I_{q_1}, ..., I_{q_N}$. Thus, O is a sequence of atomic elements, where each atomic element is a rectangle from I. Let K be the total length of O. With the input and output formalised, we have the necessary tools to prove the theorem.

Proof of Theorem ??. Consider the input (Q, I) and the output O as described above. O is generated from the sequence in which I is presented to the algorithm. Multiple query points might stab the same rectangle, so O can contain many duplicates. Since the operations of the algorithm are reversible in the indivisibility model, we can consider the algorithm in reverse. In this setting, the sequence O is the input and the goal is to remove duplicates. Observe that we have many duplicates; by Theorem 13, each rectangle $r \in I$ has volume t/n, and therefore by Theorem 12 it contains $\Theta(\frac{t}{n}N) = \Theta(t\beta)$ points. This means r appears $\Theta(t\beta)$ times among the query answers, and thus it is duplicated $\Theta(t\beta)$ times. By assumption the algorithm spends at most $f(N) + c_0K/B$ I/Os to remove all the duplicates. We claim it is enough to prove that this duplicate removal requires more than $(c_0 + 1)K/B$ I/Os. Most of this proof is devoted to proving this *main claim*, and in the end we show how it implies that $f(N) \geq K/B$.

By contradiction, assume the duplicate removal can be done in $\alpha K/B$ I/Os, for $\alpha = c_0 + 1$. If there are more than $K/10$ elements of O that are written more than 10α times, then it follows that the algorithm has spent more than $(10\alpha \cdot K/10)/B = \alpha K/B$ I/Os, which is not possible. Thus, let O' be the subset of O where each element of O' is written at most

10α times and now we know that O' contains at least $9K/10$ elements. Since O contains N unique elements, it follows that O' contains $9K/10 - N \geq 8K/10$ duplicates. Now ignore any element that is not in O' (or assume the algorithm can just remove the duplicates for free outside O'). This means, the algorithm has an input of size at least $9K/10$ and it remove at least $8K/10$ duplicates.

By Theorem 11, we can do a simulation of the duplicate removal with an α (i.e. constant) factor overhead on the number of I/Os. Let $\hat{O}$ be the sequence of elements produced by the simulation, and consider a block $\mathcal{B}$ in $\hat{O}$. By Theorem 11, we know $\mathcal{B}$ is filled with pairs of elements that are duplicates and by Corollary 6, $\mathcal{B}$ can be traced back to $w = m^{O(\alpha)}$ blocks of size B in the sequence O' and in particular to blocks in the sequence $I_{q_1}, ..., I_{q_N}$. Each block of size B can store answers for $\max\{1, \frac{B}{t\beta}\}$ queries and thus w blocks of size B correspond to $u = w \cdot \max\{1, \frac{B}{t\beta}\}$ queries. Since every rectangle is stored in the same block with at most β of its clones, there are at least $\frac{B}{c\beta}$ un-cloned (initial) rectangles in I_1 with multiplicity > 1. That means that there exists a subset $S \subseteq Q$ of queries (where $|S| \leq u$), so that in the set of rectangles in I_1 stabbed by S, there are at least $\frac{B}{c\beta}$ rectangles stabbed by at least two query points.

We show that for the right choice of parameters, this is impossible which would in turn prove our main claim above. To do this, it is enough to show that two query points q_i and q_j cannot stab $r = \frac{B}{c\beta u^2}$ common initial rectangles; if that holds, then the total of common initial rectangles over all u^2 pairs in S cannot amount to $\frac{B}{c\beta}$.

By an area argument, we show that two query points q_i and q_j cannot stab $r = \frac{B}{c\beta u^2} + 1$ initial rectangles. If the area of the intersection of r initial rectangles, which is $\frac{t}{n\ell^{r-1}}$, is smaller than the area spanned by q_i and q_j, which is at least $\Omega(\frac{1}{N}) = \Omega(\frac{1}{n\beta})$ by Theorem 12, then we are done. Thus, we must ensure that $\frac{t}{n\ell^{r-1}} < \frac{1}{c'n\beta}$, (i.e. $c'\beta t < \ell^{r-1}$) for some constant c'. By substituting $t = c_t(\log_\ell n)^{d-1}$ and r we get:

$$c' c_t \beta (\log_\ell n)^{d-1} < \ell^{\frac{B}{c\beta u^2}} \tag{1}$$

$$(d-1)\log\log_\ell n + \log\beta + O(1) < \frac{B\log\ell}{c\beta u^2} \tag{2}$$

$$\beta((d-1)\log\log_\ell n + \log\beta + O(1)) < \frac{B\log\ell}{cu^2} \tag{3}$$

We set $\beta = B/(\log B + \log\log n)$ and thus we get $u = w$ since $\beta t \geq B$. We assume $B = \Omega(\log\log N)$ and thus this implies $\beta \geq 1$ and thus it is a valid choice for β. Then, we observe that setting the value $\log\ell = m^{O(\alpha)}$ satisfies the inequality 3. We thus we obtain a lower bound of $f(N) = \Omega(K/B)$. Since each query point hits exactly $t\beta$ rectangles, the output size is $K = Nt\beta$. For our choice of ℓ, we have $t = \left(\frac{\log N}{m^{O(\alpha)}}\right)^{d-1}$. Using the notation $f \gg g$ for $f = \Omega(g)$, we get

$$f(N) \gg \frac{K}{B} = N \cdot \left(\frac{\log N}{m^{O(\alpha)}}\right)^{d-1} \frac{1}{\log B + \log\log n} \tag{4}$$

◀

This lower bound is higher than the upper bound shown in [11] for the paired output format. One trivial approach to achieve the query output format is sorting the paired output format. This yields $\mathcal{O}\Big(N/B \log_m^{d-1} N/B + K/B \log_m K/B\Big)$ I/Os, which of course does not match our lower bound. Besides, our theorem only applies to algorithms that use $\mathcal{O}(f(N) + \alpha K/B)$ I/Os. It would be interesting to see if our lower bounds can be matched by a specialised algorithm tailored for the query output format.

3.2 Batched Orthogonal Range Reporting

▶ **Theorem 15.** *Let $\mathcal{A}$ be an algorithm that given the input sets P and R for the BORR problem, answers the queries in $f(N) + c_0K/B$ I/Os. And assume $B^\varepsilon > m^{c_0}$ for some small enough constant ε. We prove that*

$$f(N) = \Omega\left(\frac{N}{B}\log_m^{d-1}(N) + K/B\right).$$

Proof. The proof of this theorem follows the same reasoning as the proof of Theorem 14, but the input objects (points) are not cloned. Consider an input (P, R) where P is a set of N points as in Theorem 12 and R is set of n query ranges as in Theorem 13. The value of n is determined by a parameter β so that $\beta = N/n$. Note that we create fewer rectangles than points.

As before, we look at the problem as a duplicate removal problem, define the sequence O of size K and observe that it is sufficient to prove that removal of duplicates from O requires at least $\alpha K/B$ I/Os where $\alpha = c_0 + 1$. As before, we assume otherwise, meaning, we assume that the algorithm can remove duplicates in $\alpha K/B$ I/Os. We then define the sequence O' and use Theorem 11 to define the sequence $\hat{O}$. We know that every block in $\hat{O}$ contains B/c duplicates for some constant c. However, here the role of the rectangles and points are swapped and proofs start to diverge.

The volume of a rectangle is $\frac{t}{n}$, so by Theorem 12 it contains $\Theta(N\frac{t}{n}) = \Theta(t\beta)$ points. Since the points contained in a rectangle are reported consecutively, a block of size B can store answers to $\max\{1, \frac{B}{t\beta}\}$ queries. Thus, setting β to be $\max\{1, \Theta(\frac{B}{t})\}$, every block can store the answers to $\mathcal{O}(1)$ queries. As before, every block in $\hat{O}$ can be traced back to only $m^{O(\alpha)}$ blocks in the sequence O and since very block in the sequence O stores answers of at most $O(1)$ queries, it follows that every block in $\hat{O}$ can be traced back to $w = m^{O(\alpha)}$ rectangles that contains B/c points that are contained in at least two of the rectangles. This means, for some pair of rectangles q_1, q_2, we must have $B/(cu^2)$ common points. Observe that the area of $q_1 \cap q_2$ is at most $O(\frac{t}{n\ell})$ and thus by Theorem 12, $q_1 \cap q_2$ can contain at most $1 + N \cdot O(\frac{t}{n\ell}) = 1 + c'\frac{t\beta}{\ell}$ points, for some constant c'.

Thus, we can get a contradiction by satisfying the inequality

$$1 + c'\frac{t\beta}{\ell} < \frac{B}{w^2}.$$

Observe that if $\frac{B}{w^2} > 1$, then we can pick ℓ large enough such that it satisfies the inequality. In particular, we set $\ell = \Omega(w^2)$. The assumption of $\frac{B}{w^2} > 1$ translates to $B^\varepsilon > m^{c_0}$ which is satisfied by our short cache assumption. Thus, we get a lower bound

$$f(N) = \Omega\left(\frac{K}{B}\right) = \Omega\left(\frac{t\beta n}{B}\right) = \Omega\left(\frac{tN}{B}\right) = \Omega\left(\frac{N\log_{w^2}^{d-1} N}{B}\right) = \Omega\left(\frac{N}{B}\log_m^{d-1} N\right). \quad ◀$$

References

1 Peyman Afshani. Improved pointer machine and I/O lower bounds for simplex range reporting and related problems. In *Symposium on Computational Geometry (SoCG)*, pages 339–346, 2012.

2 Peyman Afshani, Lars Arge, and Kasper Dalgaard Larsen. Orthogonal range reporting in three and higher dimensions. In *Proceedings of Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 149–158, 2009.

3 Peyman Afshani, Lars Arge, and Kasper Dalgaard Larsen. Orthogonal range reporting: query lower bounds, optimal structures in 3-d, and higher-dimensional improvements. In *Symposium on Computational Geometry (SoCG)*, pages 240–246, 2010.

4 Peyman Afshani, Gerth Stolting Brodal, and Norbert Zeh. Ordered and unordered top-k range reporting in large data sets. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 390–400, 2011.

5 Peyman Afshani and Nodari Sitchinava. I/O-efficient range minima queries. In *Scandinavian Workshop on Algorithms Theory*, pages 1–12, 2014.

6 Peyman Afshani and Norbert Zeh. Lower bounds for sorted geometric queries in the I/O model. In *ESA 12: Proceedings of the 20th Annual European Symposium*, pages 48–59, 2012.

7 Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM (CACM)*, 31(9):1116–1127, 1988.

8 L. Arge. External memory data structures. In J. Abello, P. M. Pardalos, and M. G. C. Resende, editors, *Handbook of Massive Data Sets*, pages 313–358. Kluwer Academic Publishers, 2002.

9 Lars Arge. *Efficient external-memory data structures and applications*. PhD thesis, Aarhus University, 1996.

10 Lars Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.

11 Lars Arge, Octavian Procopiuc, Sridhar Ramaswamy, Torsten Suel, and Jeffrey Scott Vitter. Theory and practice of I/O efficient algorithms for multidimensional batched searching problems. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 685–694, 1998.

12 Lars Arge, Vasilis Samoladas, and Ke Yi. Optimal external-memory planar point enclosure. In *Proceedings of European Symposium on Algorithms (ESA)*, pages 40–52, 2004.

13 Lars Arge, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory algorithms for processing line segments in geographic information systems. In *Proceedings of European Symposium on Algorithms (ESA)*, pages 295–310. Springer, 1995.

14 Bernard Chazelle. Lower bounds for orthogonal range searching: I. the reporting case. *Journal of the ACM (JACM)*, 37(2):200–212, 1990.

15 Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory graph algorithms. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 139–149, 1995.

16 T. H. Cormen. Fast permuting on disk arrays. *Journal of Parallel and Distributed Computing*, 17(1):41–57, 1993.

17 Andreas Crauser, Paolo Ferragina, Kurt Mehlhorn, Ulrich Meyer, and Edgar A Ramos. I/O-optimal computation of segment intersections. *External Memory Algorithms and Visualization*, pages 131–138, 1999.

18 Andreas Crauser, Paolo Ferragina, Kurt Mehlhorn, Ulrich Meyer, and Edgar A. Ramos. Randomized external-memory algorithms for line segment intersection and other geometric problems. *International Journal of Computational Geometry & Applications*, 11(03):305–337, 2001.

19 Robert W. Floyd. Permuting information in idealized two-level storage. In *Complexity of computer computations*, pages 105–109. Springer, 1972.

20 Michael T. Goodrich, Jyh-Jong Tsay, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory computational geometry. In *Proceedings of Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 714–723, 1993.

21 Gero Griener. *Sparse Matrix Computations and their I/O Complexity*. PhD thesis, Technische Universität München, 2012.

22 Joseph M. Hellerstein, Elias Koutsoupias, Daniel P. Miranker, Christos H. Papadimitriou, and Vasilis Samoladas. On a model of indexability and its bounds for range queries. *Journal of the ACM (JACM)*, 49(1):35–55, 2002.

23 Hong T. Kung. Computational models for parallel computers. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 326(1591):357–371, 1988.

24 J. S. Vitter. Algorithms and data structures for external memory. *Foundations and Trends in Theoretical Computer Science*, 2(4):305–474, 2008. `doi:10.1561/0400000014`.